5

Indexes and Index-Organized Tables

Indexes are schema objects that can speed access to table rows. Index-organized tables are tables stored in an index structure.

Introduction to Indexes

An **index** is an optional structure, associated with a table or **table cluster**, that can sometimes speed data access.

Overview of B-Tree Indexes

B-trees, short for *balanced trees*, are the most common type of database index. A **B-tree index** is an ordered list of values divided into ranges. By associating a key with a row or range of rows, B-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches.

Overview of Bitmap Indexes

In a **bitmap index**, the database stores a bitmap for each index key. In a conventional B-tree index, one index entry points to a single row. In a bitmap index, each index key stores pointers to multiple rows.

Overview of Function-Based Indexes

A **function-based index** computes the value of a function or expression involving one or more columns and stores it in an index. A function-based index can be either a B-tree or a bitmap index.

Overview of Application Domain Indexes

An **application domain index** is a customized index specific to an application.

Overview of Index-Organized Tables

An **index-organized table** is a table stored in a variation of a B-tree index structure. In contrast, a **heap-organized table** inserts rows where they fit.

Introduction to Indexes

An **index** is an optional structure, associated with a table or **table cluster**, that can sometimes speed data access.

Indexes are schema objects that are logically and physically independent of the data in the objects with which they are associated. Thus, you can drop or create an index without physically affecting the indexed table.



If you drop an index, then applications still work. However, access of previously indexed data can be slower.

For an analogy, suppose an HR manager has a shelf of cardboard boxes. Folders containing employee information are inserted randomly in the boxes. The folder for employee Whalen (ID 200) is 10 folders up from the bottom of box 1, whereas the folder for King (ID 100) is at the bottom of box 3. To locate a folder, the manager looks at every folder in box 1 from bottom to

top, and then moves from box to box until the folder is found. To speed access, the manager could create an index that sequentially lists every employee ID with its folder location:

```
ID 100: Box 3, position 1 (bottom)
ID 101: Box 7, position 8
ID 200: Box 1, position 10
.
```

Similarly, the manager could create separate indexes for employee last names, department IDs, and so on.

This section contains the following topics:

- Advantages and Disadvantages of Indexes
- Index Usability and Visibility
- Keys and Columns
- · Composite Indexes
- Unique and Nonunique Indexes
- Types of Indexes
- How the Database Maintains Indexes
- Index Storage
- Advantages and Disadvantages of Indexes

The absence or presence of an index does not require a change in the wording of any SQL statement.

Index Usability and Visibility

Indexes are usable (default) or unusable, visible (default) or invisible.

Keys and Columns

A **key** is a set of columns or expressions on which you can build an index.

Composite Indexes

A **composite index**, also called a **concatenated index**, is an index on multiple columns in a table.

Unique and Nonunique Indexes

Indexes can be unique or nonunique. Unique indexes guarantee that no two rows of a table have duplicate values in the key column or columns.

Types of Indexes

Oracle Database provides several indexing schemes, which provide complementary performance functionality.

How the Database Maintains Indexes

The database automatically maintains and uses indexes after they are created.

Index Storage

Oracle Database stores index data in an index segment.

Advantages and Disadvantages of Indexes

The absence or presence of an index does not require a change in the wording of any SQL statement.

An index is a fast access path to a single row of data. It affects only the speed of execution. Given a data value that has been indexed, the index points directly to the location of the rows containing that value.

When an index exists on one or more columns of a table, the database can in some cases retrieve a small set of randomly distributed rows from the table. Indexes are one of many means of reducing disk I/O. If a heap-organized table has no indexes, then the database must perform a full table scan to find a value. For example, a query of location 2700 in the unindexed hr.departments table requires the database to search every row in every block. This approach does not scale well as data volumes increase.

Disadvantages of indexes include the following:

- Creating indexes manually often requires deep knowledge of the data model, application, and data distribution.
- As the data changes, you must revisit previous decisions about indexes. An index might stop being useful, or new indexes might be required.
- Indexes occupy disk space.
- The database must update the index when DML occurs on the indexed data, which creates performance overhead.

Note:

Starting in Oracle Database 19c, Oracle Database can constantly monitor the application workload, creating and managing indexes automatically. Automated indexing is implemented as a database task that runs at a fixed interval.

Consider creating an index in the following situations:

- The indexed columns are queried frequently and return a small percentage of the total number of rows in the table.
- A referential integrity constraint exists on the indexed column or columns. The index is a means to avoid a full table lock that would otherwise be required if you update the parent table primary key, merge into the parent table, or delete from the parent table.
- A unique key constraint will be placed on the table and you want to manually specify the index and all index options.

See Also:

- Oracle Database Administrator's Guide to learn more about automated indexing
- Oracle Database Licensing Information User Manual for details on which features are supported for different editions and services

Index Usability and Visibility

Indexes are usable (default) or unusable, visible (default) or invisible.

These properties are defined as follows:



Usability

An unusable index, which is ignored by the optimizer, is not maintained by DML operations. An unusable index can improve the performance of bulk loads. Instead of dropping an index and later re-creating it, you can make the index unusable and then rebuild it. Unusable indexes and index partitions do not consume space. When you make a usable index unusable, the database drops its index segment.

Visibility

An invisible index is maintained by DML operations, but is not used by default by the optimizer. Making an index invisible is an alternative to making it unusable or dropping it. Invisible indexes are especially useful for testing the removal of an index before dropping it or using indexes temporarily without affecting the overall application.



"Overview of the Optimizer" to learn about how the optimizer select execution plans

Keys and Columns

A **key** is a set of columns or expressions on which you can build an index.

Although the terms are often used interchangeably, indexes and keys are different. Indexes are structures stored in the database that users manage using SQL statements. Keys are strictly a logical concept.

The following statement creates an index on the <code>customer_id</code> column of the sample table <code>oe.orders</code>:

```
CREATE INDEX ord customer ix ON orders (customer id);
```

In the preceding statement, the <code>customer_id</code> column is the index key. The index itself is named ord <code>customer_ix</code>.

Note:

Primary and unique keys automatically have indexes, but you might want to create an index on a foreign key.

See Also:

- "Data Integrity"
- Oracle Database SQL Language Reference CREATE INDEX syntax and semantics



Composite Indexes

A **composite index**, also called a **concatenated index**, is an index on multiple columns in a table.

Place columns in a composite index in the order that makes the most sense for the queries that will retrieve data. The columns need not be adjacent in the table.

Composite indexes can speed retrieval of data for SELECT statements in which the WHERE clause references all or the leading portion of the columns in the composite index. Therefore, the order of the columns used in the definition is important. In general, the most commonly accessed columns go first.

For example, suppose an application frequently queries the <code>last_name</code>, <code>job_id</code>, and <code>salary</code> columns in the <code>employees</code> table. Also assume that <code>last_name</code> has high cardinality, which means that the number of distinct values is large compared to the number of table rows. You create an index with the following column order:

```
CREATE INDEX employees_ix
   ON employees (last name, job id, salary);
```

Queries that access all three columns, only the $last_name$ column, or only the $last_name$ and job_id columns use this index. In this example, queries that do not access the $last_name$ column do not use the index.



In some cases, such as when the leading column has very low cardinality, the database may use a skip scan of this index (see "Index Skip Scan").

Multiple indexes can exist on the same table with the same column order when they meet any of the following conditions:

The indexes are of different types.

For example, you can create bitmap and B-tree indexes on the same columns.

The indexes use different partitioning schemes.

For example, you can create indexes that are locally partitioned and indexes that are globally partitioned.

The indexes have different uniqueness properties.

For example, you can create both a unique and a non-unique index on the same set of columns.

For example, a nonpartitioned index, global partitioned index, and locally partitioned index can exist for the same table columns in the same order. Only one index with the same number of columns in the same order can be visible at any one time.

This capability enables you to migrate applications without the need to drop an existing index and re-create it with different attributes. Also, this capability is useful in an OLTP database when an index key keeps increasing, causing the database to insert new entries into the same set of index blocks. To alleviate such "hot spots," you could evolve the index from a nonpartitioned index into a global partitioned index.



If indexes on the same set of columns do not differ in type or partitioning scheme, then these indexes must use different column permutations. For example, the following SQL statements specify valid column permutations:

```
CREATE INDEX employee_idx1 ON employees (last_name, job_id); CREATE INDEX employee idx2 ON employees (job id, last name);
```

See Also:

Oracle Database SQL Tuning Guide to learn more about creating multiple indexes on the same set of columns

Unique and Nonunique Indexes

Indexes can be unique or nonunique. Unique indexes guarantee that no two rows of a table have duplicate values in the key column or columns.

For example, your application may require that no two employees have the same employee ID. In a unique index, one rowid exists for each data value. The data in the leaf blocks is sorted only by key.

Nonunique indexes permit duplicates values in the indexed column or columns. For example, the first_name column of the employees table may contain multiple Mike values. For a nonunique index, the rowid is included in the key in sorted order, so nonunique indexes are sorted by the index key and rowid (ascending).

Oracle Database does not index table rows in which all key columns are null, except for bitmap indexes or when the cluster key column value is null.

Types of Indexes

Oracle Database provides several indexing schemes, which provide complementary performance functionality.

B-tree indexes are the standard index type. They are excellent for highly selective indexes (few rows correspond to each index entry) and primary key indexes. Used as concatenated indexes, a B-tree index can retrieve data sorted by the indexed columns. B-tree indexes have the subtypes shown in the following table.

Table 5-1 B-Tree Index Subtypes

| B-Tree Index Subtype | Description | To Learn More |
|------------------------|---|---|
| Index-organized tables | An index-organized table differs from a heap-organized because the data is itself the index. | "Overview of Index-Organized Tables" |
| Reverse key indexes | In this type of index, the bytes of the index key are reversed, for example, 103 is stored as 301. The reversal of bytes spreads out inserts into the index over many blocks. | "Reverse Key Indexes" |



Table 5-1 (Cont.) B-Tree Index Subtypes

| B-Tree Index Subtype | Description | To Learn More |
|------------------------|---|------------------------------------|
| Descending indexes | This type of index stores data on a particular column or columns in descending order. | "Ascending and Descending Indexes" |
| B-tree cluster indexes | This type of index stores data on a particular column or columns in descending order. | "Ascending and Descending Indexes" |

The following table shows types of indexes that do not use a B-tree structure.

Table 5-2 Indexes Not Using a B-Tree Structure

| Туре | Description | To Learn More |
|--------------------------------|--|---|
| Bitmap and bitmap join indexes | In a bitmap index, an index entry uses a bitmap to point to multiple rows. In contrast, a B-tree index entry points to a single row. A bitmap join index is a bitmap index for the join of two or more tables. | "Overview of Bitmap Indexes" |
| Function-based indexes | This type of index includes columns that are either transformed by a function, such as the UPPER function, or included in an expression. B-tree or bitmap indexes can be function-based. | "Overview of Function-Based Indexes" |
| Application domain indexes | A user creates this type of index for data in an application-specific domain. The physical index need not use a traditional index structure and can be stored either in the Oracle database as tables or externally as a file. | "Overview of Application Domain Indexes" |

See Also:

- Oracle Database Administrator's Guide to learn how to manage indexes
- Oracle Database SQL Tuning Guide to learn about different index access paths

How the Database Maintains Indexes

The database automatically maintains and uses indexes after they are created.

Indexes automatically reflect data changes to their underlying tables. Examples of changes include adding, updating, and deleting rows. No user actions are required.

Retrieval performance of indexed data remains almost constant, even as rows are inserted. However, the presence of many indexes on a table degrades DML performance because the database must also update the indexes.

See Also:

- Oracle Database Administrator's Guide to learn more about automated indexing
- Oracle Database Licensing Information User Manual for details on which features are supported for different editions and services

Index Storage

Oracle Database stores index data in an index segment.

Space available for index data in a data block is the data block size minus block overhead, entry overhead, rowid, and one length byte for each value indexed.

The tablespace of an index segment is either the default tablespace of the owner or a tablespace specifically named in the CREATE INDEX statement. For ease of administration you can store an index in a separate tablespace from its table. For example, you may choose not to back up tablespaces containing only indexes, which can be rebuilt, and so decrease the time and storage required for backups.



"Overview of Index Blocks" to learn about types of index block (root, branch, and leaf), and how index entries are stored within a block

Overview of B-Tree Indexes

B-trees, short for *balanced trees*, are the most common type of database index. A **B-tree index** is an ordered list of values divided into ranges. By associating a key with a row or range of rows, B-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches.

The following figure illustrates the structure of a B-tree index. The example shows an index on the department id column, which is a foreign key column in the employees table.



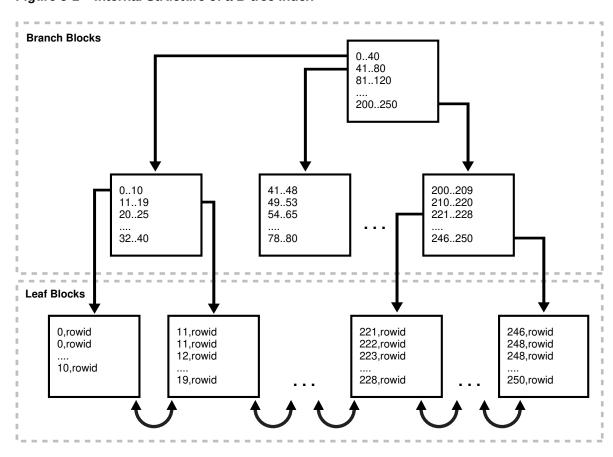


Figure 5-1 Internal Structure of a B-tree Index

This section contains the following topics:

- Branch Blocks and Leaf Blocks
- Index Scans
- Reverse Key Indexes
- Ascending and Descending Indexes
- Index Compression
- Branch Blocks and Leaf Blocks

A B-tree index has two types of blocks: the **branch block** for searching, and the **leaf block** for storing key values. The upper-level branch blocks of a B-tree index contain index data that points to lower-level index blocks.

Index Scans

In an **index scan**, the database retrieves a row by traversing the index, using the indexed column values specified by the statement. If the database scans the index for a value, then it will find this value in n I/Os where n is the height of the B-tree index. This is the basic principle behind Oracle Database indexes.

Reverse Key Indexes

A **reverse key index** is a type of B-tree index that physically reverses the bytes of each index key while keeping the column order.

Ascending and Descending Indexes

In an **ascending index**, Oracle Database stores data in ascending order. By default, character data is ordered by the binary values contained in each byte of the value, numeric data from smallest to largest number, and date from earliest to latest value.

Index Compression

To reduce space in indexes, Oracle Database can employ different compression algorithms.

Branch Blocks and Leaf Blocks

A B-tree index has two types of blocks: the **branch block** for searching, and the **leaf block** for storing key values. The upper-level branch blocks of a B-tree index contain index data that points to lower-level index blocks.

In Figure 5-1, the root branch block has an entry 0-40, which points to the leftmost block in the next branch level. This branch block contains entries such as 0-10 and 11-19. Each of these entries points to a leaf block that contains key values that fall in the range.

A B-tree index is balanced because all leaf blocks automatically stay at the same depth. Thus, retrieval of any record from anywhere in the index takes approximately the same amount of time. The height of the index is the number of blocks required to go from the root block to a leaf block. The branch level is the height minus 1. In Figure 5-1, the index has a height of 3 and a branch level of 2.

Branch blocks store the minimum key prefix needed to make a branching decision between two keys. This technique enables the database to fit as much data as possible on each branch block. The branch blocks contain a pointer to the child block containing the key. The number of keys and pointers is limited by the block size.

The leaf blocks contain every indexed data value and a corresponding rowid used to locate the actual row. Each entry is sorted by (key, rowid). Within a leaf block, a key and rowid is linked to its left and right sibling entries. The leaf blocks themselves are also doubly linked. In Figure 5-1 the leftmost leaf block (0-10) is linked to the second leaf block (11-19).



Indexes in columns with character data are based on the binary values of the characters in the database character set.

Index Scans

In an **index scan**, the database retrieves a row by traversing the index, using the indexed column values specified by the statement. If the database scans the index for a value, then it will find this value in n I/Os where n is the height of the B-tree index. This is the basic principle behind Oracle Database indexes.

If a SQL statement accesses only indexed columns, then the database reads values directly from the index rather than from the table. If the statement accesses nonindexed columns in addition to the indexed columns, then the database uses rowids to find the rows in the table. Typically, the database retrieves table data by alternately reading an index block and then a table block.



Full Index Scan

In a **full index scan**, the database reads the entire index in order. A full index scan is available if a **predicate** (WHERE clause) in the SQL statement references a column in the index, and in some circumstances when no predicate is specified. A full scan can eliminate sorting because the data is ordered by index key.

Fast Full Index Scan

A **fast full index scan** is a full index scan in which the database accesses the data in the index itself without accessing the table, and the database reads the index blocks in no particular order.

Index Range Scan

An **index range scan** is an ordered scan of an index in which one or more leading columns of an index are specified in conditions, and 0, 1, or more values are possible for an index key.

Index Unique Scan

In contrast to an index range scan, an **index unique scan** must have either 0 or 1 rowid associated with an index key.

Index Skip Scan

An **index skip scan** uses logical subindexes of a composite index. The database "skips" through a single index as if it were searching separate indexes.

Index Clustering Factor

The **index clustering factor** measures row order in relation to an indexed value such as employee last name. As the degree of order increases, the clustering factor decreases.



Oracle Database SQL Tuning Guide for detailed information about index scans

Full Index Scan

In a **full index scan**, the database reads the entire index in order. A full index scan is available if a **predicate** (WHERE clause) in the SQL statement references a column in the index, and in some circumstances when no predicate is specified. A full scan can eliminate sorting because the data is ordered by index key.

Example 5-1 Full Index Scan

Suppose that an application runs the following query:

```
SELECT department_id, last_name, salary FROM employees
WHERE salary > 5000
ORDER BY department id, last name;
```

In this example, the <code>department_id</code>, <code>last_name</code>, and <code>salary</code> are a composite key in an index. Oracle Database performs a full scan of the index, reading it in sorted order (ordered by department ID and last name) and filtering on the salary attribute. In this way, the database scans a set of data smaller than the <code>employees</code> table, which contains more columns than are included in the query, and avoids sorting the data.

The full scan could read the index entries as follows:

```
50, Atkinson, 2800, rowid 60, Austin, 4800, rowid
```



```
70, Baer, 10000, rowid
80, Abel, 11000, rowid
80, Ande, 6400, rowid
110, Austin, 7200, rowid
.
```

Fast Full Index Scan

A **fast full index scan** is a full index scan in which the database accesses the data in the index itself without accessing the table, and the database reads the index blocks in no particular order.

Fast full index scans are an alternative to a full table scan when both of the following conditions are met:

- The index must contain all columns needed for the query.
- A row containing all nulls must not appear in the query result set. For this result to be guaranteed, at least one column in the index must have either:
 - A NOT NULL constraint
 - A predicate applied to the column that prevents nulls from being considered in the query result set

Example 5-2 Fast Full Index Scan

Assume that an application issues the following query, which does not include an ORDER BY clause:

```
SELECT last_name, salary
FROM employees;
```

The last_name column has a not null constraint. If the last name and salary are a composite key in an index, then a fast full index scan can read the index entries to obtain the requested information:

```
Baida,2900,rowid
Atkinson,2800,rowid
Zlotkey,10500,rowid
Austin,7200,rowid
Baer,10000,rowid
Austin,4800,rowid
```

Index Range Scan

An **index range scan** is an ordered scan of an index in which one or more leading columns of an index are specified in conditions, and 0, 1, or more values are possible for an index key.

A condition specifies a combination of one or more expressions and logical (Boolean) operators. It returns a value of TRUE, FALSE, or UNKNOWN.

The database commonly uses an index range scan to access selective data. The selectivity is the percentage of rows in the table that the query selects, with 0 meaning no rows and 1 meaning all rows. Selectivity is tied to a query predicate, such as WHERE last name LIKE 'A%',

or a combination of predicates. A predicate becomes more selective as the value approaches 0 and less selective (or more unselective) as the value approaches 1.

For example, a user queries employees whose last names begin with A. Assume that the last name column is indexed, with entries as follows:

```
Abel, rowid
Ande, rowid
Atkinson, rowid
Austin, rowid
Austin, rowid
Baer, rowid
.
```

The database could use a range scan because the <code>last_name</code> column is specified in the predicate and multiples rowids are possible for each index key. For example, two employees are named Austin, so two rowids are associated with the key <code>Austin</code>.

An index range scan can be bounded on both sides, as in a query for departments with IDs between 10 and 40, or bounded on only one side, as in a query for IDs over 40. To scan the index, the database moves backward or forward through the leaf blocks. For example, a scan for IDs between 10 and 40 locates the first index leaf block that contains the lowest key value that is 10 or greater. The scan then proceeds horizontally through the linked list of leaf nodes until it locates a value greater than 40.

Index Unique Scan

In contrast to an index range scan, an **index unique scan** must have either 0 or 1 rowid associated with an index key.

The database performs a unique scan when a predicate references all of the columns in the key of a UNIQUE index using an equality operator. An index unique scan stops processing as soon as it finds the first record because no second record is possible.

As an illustration, suppose that a user runs the following query:

```
SELECT *
FROM employees
WHERE employee_id = 5;
```

Assume that the employee id column is the primary key and is indexed with entries as follows:

```
1, rowid
2, rowid
4, rowid
5, rowid
6, rowid
.
```

In this case, the database can use an index unique scan to locate the rowid for the employee whose ID is 5.

Index Skip Scan

An **index skip scan** uses logical subindexes of a composite index. The database "skips" through a single index as if it were searching separate indexes.

Skip scanning is beneficial if there are few distinct values in the leading column of a composite index and many distinct values in the nonleading key of the index. The database may choose an index skip scan when the leading column of the composite index is not specified in a query predicate.

Example 5-3 Skip Scan of a Composite Index

Assume that you run the following query for a customer in the sh.customers table:

```
SELECT * FROM sh.customers WHERE cust_email = 'Abbey@company.example.com';
```

The customers table has a column cust_gender whose values are either ${\tt M}$ or ${\tt F}$. Assume that a composite index exists on the columns (cust_gender, cust_email). The following example shows a portion of the index entries:

```
F, Wolf@company.example.com, rowid
F, Wood@company.example.com, rowid
F, Wood@company.example.com, rowid
F, Woodman@company.example.com, rowid
F, Yang@company.example.com, rowid
F, Zimmerman@company.example.com, rowid
M, Abbassi@company.example.com, rowid
M, Abbey@company.example.com, rowid
```

The database can use a skip scan of this index even though <code>cust_gender</code> is not specified in the <code>WHERE</code> clause.

In a skip scan, the number of logical subindexes is determined by the number of distinct values in the leading column. In the preceding example, the leading column has two possible values. The database logically splits the index into one subindex with the key $\mathbb F$ and a second subindex with the key $\mathbb F$.

When searching for the record for the customer whose email is <code>Abbey@company.example.com</code>, the database searches the subindex with the value <code>F</code> first and then searches the subindex with the value <code>M</code>. Conceptually, the database processes the query as follows:

```
SELECT * FROM sh.customers WHERE cust_gender = 'F'
AND cust_email = 'Abbey@company.example.com'
UNION ALL
SELECT * FROM sh.customers WHERE cust_gender = 'M'
AND cust_email = 'Abbey@company.example.com';
```

See Also:

Oracle Database SQL Tuning Guide to learn more about skip scans

Index Clustering Factor

The **index clustering factor** measures row order in relation to an indexed value such as employee last name. As the degree of order increases, the clustering factor decreases.

The clustering factor is useful as a rough measure of the number of I/Os required to read an entire table using an index:

- If the clustering factor is high, then Oracle Database performs a relatively high number of I/Os during a large index range scan. The index entries point to random table blocks, so the database may have to read and reread the same blocks over and over again to retrieve the data pointed to by the index.
- If the clustering factor is low, then Oracle Database performs a relatively low number of I/Os during a large index range scan. The index keys in a range tend to point to the same data block, so the database does not have to read and reread the same blocks over and over.

The clustering factor is relevant for index scans because it can show:

- Whether the database will use an index for large range scans
- The degree of table organization in relation to the index key
- Whether you should consider using an index-organized table, partitioning, or table cluster if rows must be ordered by the index key

Example 5-4 Clustering Factor

Assume that the employees table fits into two data blocks. Table 5-3 depicts the rows in the two data blocks (the ellipses indicate data that is not shown).

Table 5-3 Contents of Two Data Blocks in the Employees Table

| Data Block 1 | | | Data | Block 2 | | | |
|---|-------|---------------------------------------|----------------|----------------------------|---------------------------|---------------------|--|
| 100 Steven 156 Janette 115 Alexander | • | SKING JKING AKHOO | | | | | |
| 116 Shelli E 204 Hermann E | | SBAIDA HBAER | | | _ | EZLOTKEY JWHALEN | |
| 105 David A 130 Mozhe A 166 Sundar A | ustin | DAUSTIN MATKINSO SANDE EABEL | 137 173 | Renske Sundita Neena | Ladwig Kumar Kochar | | |

Rows are stored in the blocks in order of last name (shown in bold). For example, the bottom row in data block 1 describes Abel, the next row up describes Ande, and so on alphabetically until the top row in block 1 for Steven King. The bottom row in block 2 describes Kochar, the next row up describes Kumar, and so on alphabetically until the last row in the block for Zlotkey.



Assume that an index exists on the last name column. Each name entry corresponds to a rowid. Conceptually, the index entries would look as follows:

```
Abel,block1row1
Ande,block1row2
Atkinson,block1row3
Austin,block1row4
Baer,block1row5
.
```

Assume that a separate index exists on the employee ID column. Conceptually, the index entries might look as follows, with employee IDs distributed in almost random locations throughout the two blocks:

```
100,block1row50
101,block2row1
102,block1row9
103,block2row19
104,block2row39
105,block1row4
```

The following statement queries the ALL_INDEXES view for the clustering factor for these two indexes:

```
SQL> SELECT INDEX_NAME, CLUSTERING_FACTOR

2 FROM ALL_INDEXES

3 WHERE INDEX_NAME IN ('EMP_NAME_IX', 'EMP_EMP_ID_PK');

INDEX_NAME CLUSTERING_FACTOR

EMP_EMP_ID_PK 19

EMP_NAME_IX 2
```

The clustering factor for EMP_NAME_IX is low, which means that adjacent index entries in a single leaf block tend to point to rows in the same data blocks. The clustering factor for $EMP_EMP_ID_PK$ is high, which means that adjacent index entries in the same leaf block are much less likely to point to rows in the same data blocks.

```
See Also:
```

Oracle Database Reference to learn about ALL_INDEXES

Reverse Key Indexes

A **reverse key index** is a type of B-tree index that physically reverses the bytes of each index key while keeping the column order.

For example, if the index key is 20, and if the two bytes stored for this key in hexadecimal are C1,15 in a standard B-tree index, then a reverse key index stores the bytes as 15,C1.

Reversing the key solves the problem of contention for leaf blocks in the right side of a B-tree index. This problem can be especially acute in an Oracle Real Application Clusters (Oracle RAC) database in which multiple instances repeatedly modify the same block. For example, in an orders table the primary keys for orders are sequential. One instance in the cluster adds order 20, while another adds 21, with each instance writing its key to the same leaf block on the right-hand side of the index.

In a reverse key index, the reversal of the byte order distributes inserts across all leaf keys in the index. For example, keys such as 20 and 21 that would have been adjacent in a standard key index are now stored far apart in separate blocks. Thus, I/O for insertions of sequential keys is more evenly distributed.

Because the data in the index is not sorted by column key when it is stored, the reverse key arrangement eliminates the ability to run an index range scanning query in some cases. For example, if a user issues a query for order IDs greater than 20, then the database cannot start with the block containing this ID and proceed horizontally through the leaf blocks.

Ascending and Descending Indexes

In an **ascending index**, Oracle Database stores data in ascending order. By default, character data is ordered by the binary values contained in each byte of the value, numeric data from smallest to largest number, and date from earliest to latest value.

For an example of an ascending index, consider the following SQL statement:

```
CREATE INDEX emp_deptid_ix ON hr.employees(department_id);
```

Oracle Database sorts the hr.employees table on the department_id column. It loads the ascending index with the department_id and corresponding rowid values in ascending order, starting with 0. When it uses the index, Oracle Database searches the sorted department_id values and uses the associated rowids to locate rows having the requested department_id value.

By specifying the DESC keyword in the CREATE INDEX statement, you can create a descending index. In this case, the index stores data on a specified column or columns in descending order. If the index in Table 5-3 on the employees.department_id column were descending, then the leaf blocking containing 250 would be on the left side of the tree and block with 0 on the right. The default search through a descending index is from highest to lowest value.

Descending indexes are useful when a query sorts some columns ascending and others descending. For an example, assume that you create a composite index on the <code>last_name</code> and <code>department id columns</code> as follows:

```
CREATE INDEX emp_name_dpt_ix ON hr.employees(last_name ASC, department_id
DESC);
```



If a user queries hr.employees for last names in ascending order (A to Z) and department IDs in descending order (high to low), then the database can use this index to retrieve the data and avoid the extra step of sorting it.

See Also:

- Oracle Database SQL Tuning Guide to learn more about ascending and descending index searches
- Oracle Database SQL Language Reference for descriptions of the ASC and DESC options of CREATE INDEX

Index Compression

To reduce space in indexes, Oracle Database can employ different compression algorithms.

Prefix Compression

Oracle Database can use **prefix compression**, also known as **key compression**, to compress portions of the primary key column values in a B-tree index or an indexorganized table. Prefix compression can greatly reduce the space consumed by the index.

Advanced Index Compression
 Starting with Oracle Database 12c Release 1 (12.1.0.2), advanced index compression improves on traditional prefix compression for supported indexes on heap-organized

Prefix Compression

Oracle Database can use **prefix compression**, also known as **key compression**, to compress portions of the primary key column values in a B-tree index or an index-organized table. Prefix compression can greatly reduce the space consumed by the index.

An uncompressed index entry has one piece. An index entry using prefix compression has two pieces: a prefix entry, which is the grouping piece, and a suffix entry, which is the unique or nearly unique piece. The database achieves compression by sharing the prefix entries among the suffix entries in an index block.

Note:

If a key is not defined to have a unique piece, then the database provides one by appending a rowid to the grouping piece.

By default, the prefix of a unique index consists of all key columns excluding the last one, whereas the prefix of a nonunique index consists of all key columns. Suppose you create a composite, unique index on two columns of the oe.orders table as follows:

```
CREATE UNIQUE INDEX orders_mod_stat_ix ON orders ( order_mode, order_status );
```

In the preceding example, an index key might be online, 0. The rowid is stored in the key data portion of the entry, and is not part of the key itself.

```
Note:
```

If you create a unique index on a single column, then Oracle Database cannot use prefix key compression because no common prefixes exist.

Alternatively, suppose you create a nonunique index on the same columns:

```
CREATE INDEX orders mod stat ix ON orders ( order mode, order status );
```

Also assume that repeated values occur in the <code>order_mode</code> and <code>order_status</code> columns. An index block could have entries as shown in the follow example:

```
online, 0, AAAPvCAAFAAAAFaAAa online, 0, AAAPvCAAFAAAAFaAAAg online, 0, AAAPvCAAFAAAAFaAAl online, 2, AAAPvCAAFAAAAFaAAm online, 3, AAAPvCAAFAAAAFaAAA online, 3, AAAPvCAAFAAAAFaAAA
```

In the preceding example, the key prefix would consist of a concatenation of the order_mode and order_status values, as in online, 0. The suffix consists in the rowid, as in AAAPvCAAFAAAAFaAAA. The rowid makes the whole index entry unique because a rowid is itself unique in the database.

If the index in the preceding example were created with default prefix compression (specified by the COMPRESS keyword), then duplicate key prefixes such as online,0 and online,3 would be compressed. Conceptually, the database achieves compression as follows:

```
online, 0
AAAPvCAAFAAAAFaAAa
AAAPvCAAFAAAAFaAAg
AAAPvCAAFAAAAFaAAl
online, 2
AAAPvCAAFAAAAFaAAm
online, 3
AAAPvCAAFAAAAFaAAq
AAAPvCAAFAAAAFaAAq
```

Suffix entries (the rowids) form the compressed version of index rows. Each suffix entry references a prefix entry, which is stored in the same index block as the suffix.

Alternatively, you could specify a prefix length when creating an index that uses prefix compression. For example, if you specified COMPRESS 1, then the prefix would be order_mode and the suffix would be order_status, rowid. For the values in the index block example, the index would factor out duplicate occurrences of the prefix online, which can be represented conceptually as follows:

```
online
0,AAAPvCAAFAAAAFaAAa
0,AAAPvCAAFAAAAFaAAg
0,AAAPvCAAFAAAAFaAAl
2,AAAPvCAAFAAAAFaAAm
3,AAAPvCAAFAAAAFaAAq
3,AAAPvCAAFAAAAFaAAt
```

The index stores a specific prefix once per leaf block at most. Only keys in the leaf blocks of a B-tree index are compressed. In the branch blocks the key suffix can be truncated, but the key is not compressed.

See Also:

- Oracle Database Administrator's Guide to learn how to use compressed indexes
- Oracle Database VLDB and Partitioning Guide to learn how to use prefix compression for partitioned indexes
- Oracle Database SQL Language Reference for descriptions of the key compression clause of CREATE INDEX

Advanced Index Compression

Starting with Oracle Database 12c Release 1 (12.1.0.2), **advanced index compression** improves on traditional prefix compression for supported indexes on heap-organized tables.

Benefits of Advanced Index Compression

Prefix compression has limitations for types of indexes supported, compression ratio, and ease of use. Unlike prefix compression, which uses fixed duplicate key elimination for every block, advanced index compression uses adaptive duplicate key elimination on a per-block basis. The main advantages of advanced index compression are:

- The database automatically chooses the best compression for each block, using a number
 of internal algorithms such as intra-column level prefixes, duplicate key elimination, and
 rowid compression. Unlike in prefix compression, advanced index compression does not
 require the user to know data characteristics.
- Advanced compression works on both non-unique and unique indexes. Prefix compression
 works well on some non-unique indexes, but the ratios are lower on indexes whose leading
 columns do not have many repeats.
- The compressed index is usable in the same way as an uncompressed index. The index supports the same access paths: unique key lookups, range scans, and fast full scans.
- Indexes can inherit advanced compression from a parent table or containing tablespace.
- Index-organized tables and function-based indexes support advanced index compression.
 Index-organized tables support low compression. Function-based indexes support both low and high compression.

How Advanced Index Compression Works

Advanced index compression works at the block level to provide the best compression for each block. The database uses the following technique:

- During index creation, as a leaf block becomes full, the database automatically compresses the block to the optimal level.
- When reorganizing an index block as a result of DML, if the database can create sufficient space for the incoming index entry, then a block split does not occur. During DML without advanced index compression, however, an index block split always occurs when the block becomes full.

Advanced Index Compression HIGH

In releases previous to Oracle Database 12c Release 2 (12.2), the only form of advanced index compression was low compression (COMPRESS ADVANCED LOW). Now you can also specify



high compression (COMPRESS ADVANCED HIGH), which is the default. Advanced index compression with the HIGH option offers the following advantages:

- Gives higher compression ratios in most cases, while also improving performance for queries that access the index
- Employs more complex compression algorithms than advanced low
- Stores data in a compression unit, which is a special on-disk format



When you apply ${\tt HIGH}$ compression, all blocks have compression. When you apply ${\tt LOW}$ compression, the database may leave some blocks uncompressed. You can use statistics to determine how many blocks were left uncompressed.

Example 5-5 Creating an Index with Advanced High Compression

This example enables advanced index compression for an index on the hr.employees table:

```
CREATE INDEX hr.emp_mndp_ix
  ON hr.employees(manager_id, department_id)
  COMPRESS ADVANCED;
```

The following query shows the type of compression:

See Also:

- Oracle Database Administrator's Guide to learn how to enable advanced index compression
- Oracle Database SQL Language Referencefor descriptions of the key compression clause of CREATE INDEX
- Oracle Database Reference to learn about ALL INDEXES

Overview of Bitmap Indexes

In a **bitmap index**, the database stores a bitmap for each index key. In a conventional B-tree index, one index entry points to a single row. In a bitmap index, each index key stores pointers to multiple rows.

Bitmap indexes are primarily designed for data warehousing or environments in which queries reference many columns in an ad hoc fashion. Situations that may call for a bitmap index include:

- The indexed columns have low cardinality, that is, the number of distinct values is small compared to the number of table rows.
- The indexed table is either read-only or not subject to significant modification by DML statements.

For a data warehouse example, the sh.customers table has a cust_gender column with only two possible values: M and F. Suppose that queries for the number of customers of a particular gender are common. In this case, the customers.cust_gender column would be a candidate for a bitmap index.

Each bit in the bitmap corresponds to a possible rowid. If the bit is set, then the row with the corresponding rowid contains the key value. A mapping function converts the bit position to an actual rowid, so the bitmap index provides the same functionality as a B-tree index although it uses a different internal representation.

If the indexed column in a single row is updated, then the database locks the index key entry (for example, M or F) and not the individual bit mapped to the updated row. Because a key points to many rows, DML on indexed data typically locks all of these rows. For this reason, bitmap indexes are not appropriate for many OLTP applications.

- Example: Bitmap Indexes on a Single Table
 In this example, some columns of sh.customers table are candidates for a bitmap index.
- Bitmap Join Indexes
 A bitmap join index is a bitmap index for the join of two or more tables.
- Bitmap Storage Structure
 Oracle Database uses a B-tree index structure to store bitmaps for each indexed key.

See Also:

- Oracle Database SQL Tuning Guide to learn more about bitmap indexes
- Oracle Database Data Warehousing Guide to learn how to use bitmap indexes in a data warehouse

Example: Bitmap Indexes on a Single Table

In this example, some columns of sh.customers table are candidates for a bitmap index.

Consider the following query:



```
4 Hardy M
5 Gowen M
6 Charles single F
7 Ingram single F
```

7 rows selected.

The cust_marital_status and cust_gender columns have low cardinality, whereas cust_id and cust_last_name do not. Thus, bitmap indexes may be appropriate on cust_marital_status and cust_gender. A bitmap index is probably not useful for the other columns. Instead, a unique B-tree index on these columns would likely provide the most efficient representation and retrieval.

Table 5-4 illustrates the bitmap index for the <code>cust_gender</code> column output shown in the preceding example. It consists of two separate bitmaps, one for each gender.

Table 5-4 Sample Bitmap for One Column

| Value | Row 1 | Row 2 | Row 3 | Row 4 | Row 5 | Row 6 | Row 7 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| М | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| F | 0 | 1 | 0 | 0 | 0 | 1 | 1 |

A mapping function converts each bit in the bitmap to a rowid of the customers table. Each bit value depends on the values of the corresponding row in the table. For example, the bitmap for the $\mbox{\tt M}$ value contains a 1 as its first bit because the gender is $\mbox{\tt M}$ in the first row of the customers table. The bitmap cust_gender='M' has a 0 for the bits in rows 2, 6, and 7 because these rows do not contain $\mbox{\tt M}$ as their value.



Bitmap indexes can include keys that consist entirely of null values, unlike B-tree indexes. Indexing nulls can be useful for some SQL statements, such as queries with the aggregate function ${\tt COUNT}$.

An analyst investigating demographic trends of the customers may ask, "How many of our female customers are single or divorced?" This question corresponds to the following SQL query:

```
SELECT COUNT(*)
FROM customers
WHERE cust_gender = 'F'
AND cust marital status IN ('single', 'divorced');
```

Bitmap indexes can process this query efficiently by counting the number of 1 values in the resulting bitmap, as illustrated in Table 5-5. To identify the customers who satisfy the criteria, Oracle Database can use the resulting bitmap to access the table.

Table 5-5 Sample Bitmap for Two Columns

| Value | Row 1 | Row 2 | Row 3 | Row 4 | Row 5 | Row 6 | Row 7 |
|---------------------------------|-------|-------|-------|-------|-------|-------|-------|
| М | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| F | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| single | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| divorced | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| single or divorced, and F | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

Bitmap indexing efficiently merges indexes that correspond to several conditions in a WHERE clause. Rows that satisfy some, but not all, conditions are filtered out before the table itself is accessed. This technique improves response time, often dramatically.

Bitmap Join Indexes

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. In contrast, a standard bitmap index is created on a single table.

A bitmap join index is an efficient means of reducing the volume of data that must be joined by performing restrictions in advance. For an example of when a bitmap join index would be useful, assume that users often query the number of employees with a particular job type. A typical query might look as follows:

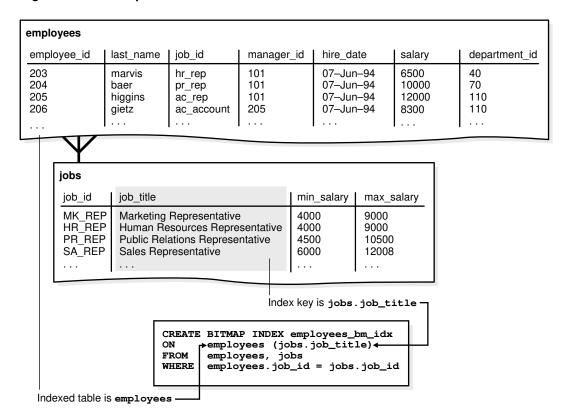
```
SELECT COUNT(*)
FROM employees, jobs
WHERE employees.job_id = jobs.job_id
AND jobs.job title = 'Accountant';
```

The preceding query would typically use an index on <code>jobs.job_title</code> to retrieve the rows for <code>Accountant</code> and then the job ID, and an index on <code>employees.job_id</code> to find the matching rows. To retrieve the data from the index itself rather than from a scan of the tables, you could create a bitmap join index as follows:

```
CREATE BITMAP INDEX employees_bm_idx
ON employees (jobs.job_title)
FROM employees, jobs
WHERE employees.job_id = jobs.job_id;
```

As illustrated in the following figure, the index key is jobs.job_title and the indexed table is employees.

Figure 5-2 Bitmap Join Index



Conceptually, <code>employees_bm_idx</code> is an index of the <code>jobs.title</code> column in the SQL query shown in the following query (sample output included). The <code>job_title</code> key in the index points to rows in the <code>employees</code> table. A query of the number of accountants can use the index to avoid accessing the <code>employees</code> and <code>jobs</code> tables because the index itself contains the requested information.

```
SELECT jobs.job title AS "jobs.job title", employees.rowid AS
"employees.rowid"
FROM
       employees, jobs
WHERE employees.job id = jobs.job id
ORDER BY job title;
jobs.job title
                                     employees.rowid
                                     AAAQNKAAFAAAABSAAL
Accountant
Accountant
                                     AAAQNKAAFAAAABSAAN
                                     AAAQNKAAFAAAABSAAM
Accountant
Accountant
                                     AAAQNKAAFAAAABSAAJ
                                     AAAQNKAAFAAAABSAAK
Accountant
Accounting Manager
                                     AAAQNKAAFAAAABTAAH
Administration Assistant
                                     AAAQNKAAFAAAABTAAC
Administration Vice President
                                     AAAQNKAAFAAAABSAAC
Administration Vice President
                                     AAAQNKAAFAAAABSAAB
```

In a data warehouse, the join condition is an equijoin (it uses the equality operator) between the primary key columns of the dimension tables and the foreign key columns in the fact table. Bitmap join indexes are sometimes much more efficient in storage than materialized join views, an alternative for materializing joins in advance.



Oracle Database Data Warehousing Guide for more information on bitmap join indexes

Bitmap Storage Structure

Oracle Database uses a B-tree index structure to store bitmaps for each indexed key.

For example, if <code>jobs.job_title</code> is the key column of a bitmap index, then one B-tree stores the index data. The leaf blocks store the individual bitmaps.

Example 5-6 Bitmap Storage Example

Assume that the jobs.job_title column has unique values Shipping Clerk, Stock Clerk, and several others. A bitmap index entry for this index has the following components:

- The job title as the index key
- A low rowid and high rowid for a range of rowids
- A bitmap for specific rowids in the range

Conceptually, an index leaf block in this index could contain entries as follows:

```
Shipping Clerk, AAAPzRAAFAAAABSABQ, AAAPzRAAFAAAABSABZ, 0010000100 Shipping Clerk, AAAPzRAAFAAAABSABa, AAAPzRAAFAAAABSABh, 010010 Stock Clerk, AAAPzRAAFAAAABSAAa, AAAPzRAAFAAAABSAAc, 1001001100 Stock Clerk, AAAPzRAAFAAAABSAAd, AAAPzRAAFAAAABSAAt, 0101001001 Stock Clerk, AAAPzRAAFAAAABSAAu, AAAPzRAAFAAAABSABz, 100001
```

.

The same job title appears in multiple entries because the rowid range differs.

A session updates the job ID of one employee from Shipping Clerk to Stock Clerk. In this case, the session requires exclusive access to the index key entry for the old value (Shipping Clerk) and the new value (Stock Clerk). Oracle Database locks the rows pointed to by these two entries—but not the rows pointed to by Accountant or any other key—until the UPDATE commits.

The data for a bitmap index is stored in one segment. Oracle Database stores each bitmap in one or more pieces. Each piece occupies part of a single data block.



See Also:

"User Segments" explains the different types of segments, and how segments are created

Overview of Function-Based Indexes

A **function-based index** computes the value of a function or expression involving one or more columns and stores it in an index. A function-based index can be either a B-tree or a bitmap index.

The indexed function can be an arithmetic expression or an expression that contains a SQL function, user-defined PL/SQL function, package function, or C callout. For example, a function could add the values in two columns.

Uses of Function-Based Indexes

Function-based indexes are efficient for evaluating statements that contain functions in their WHERE clauses. The database only uses the function-based index when the function is included in a query. When the database processes INSERT and UPDATE statements, however, it must still evaluate the function to process the statement.

Optimization with Function-Based Indexes

For queries with expressions in a WHERE clause, the optimizer might use an index range scan on a function-based index.

See Also:

- Oracle Database Administrator's Guide to learn how to create function-based indexes
- Oracle Database Development Guide for more information about using functionbased indexes
- Oracle Database SQL Language Reference for restrictions and usage notes for function-based indexes

Uses of Function-Based Indexes

Function-based indexes are efficient for evaluating statements that contain functions in their WHERE clauses. The database only uses the function-based index when the function is included in a query. When the database processes INSERT and UPDATE statements, however, it must still evaluate the function to process the statement.

Example 5-7 Index Based on Arithmetic Expression

For example, suppose you create the following function-based index:

```
CREATE INDEX emp_total_sal_idx
ON employees (12 * salary * commission pct, salary, commission pct);
```

The database can use the preceding index when processing queries such as the following (partial sample output included):

| EMPLOYEE_ID | LAST_NAME | FIRST_NAME | ANNUAL SAL |
|-------------|-----------|------------|------------|
| | | | |
| 159 | Smith | Lindsey | 28800 |
| 151 | Bernstein | David | 28500 |
| 152 | Hall | Peter | 27000 |
| 160 | Doran | Louise | 27000 |
| 175 | Hutton | Alyssa | 26400 |
| 149 | Zlotkey | Eleni | 25200 |
| 169 | Bloom | Harrison | 24000 |

Example 5-8 Index Based on an UPPER Function

Function-based indexes defined on the SQL functions <code>UPPER(column_name)</code> or <code>LOWER(column_name)</code> facilitate case-insensitive searches. For example, suppose that the <code>first_name</code> column in <code>employees</code> contains mixed-case characters. You create the following function-based index on the <code>hr.employees</code> table:

```
CREATE INDEX emp_fname_uppercase_idx
ON employees ( UPPER(first name) );
```

The emp fname uppercase idx index can facilitate queries such as the following:

```
SELECT *
FROM employees
WHERE UPPER(first name) = 'AUDREY';
```

Example 5-9 Indexing Specific Rows in a Table

A function-based index is also useful for indexing only specific rows in a table. For example, the <code>cust_valid</code> column in the <code>sh.customers</code> table has either <code>I</code> or <code>A</code> as a value. To index only the <code>A</code> rows, you could write a function that returns a null value for any rows other than the <code>A</code> rows. You could create the index as follows:

```
CREATE INDEX cust_valid_idx
ON customers ( CASE cust valid WHEN 'A' THEN 'A' END );
```

See Also:

- Oracle Database Globalization Support Guide for information about linguistic indexes
- Oracle Database SQL Language Reference to learn more about SQL functions

Optimization with Function-Based Indexes

For queries with expressions in a WHERE clause, the optimizer might use an index range scan on a function-based index.

The range scan access path is especially beneficial when the predicate is highly selective, that is, when it chooses relatively few rows. In Example 5-7, if an index is built on the expression 12*salary*commission pct, then the optimizer can use an index range scan.

A virtual column is also useful for speeding access to data derived from expressions. For example, you could define virtual column annual_sal as 12*salary*commission_pct and create a function-based index on annual sal.

The optimizer performs expression matching by parsing the expression in a SQL statement and then comparing the expression trees of the statement and the function-based index. This comparison is case-insensitive and ignores blank spaces.

Different queries may use different types of index scans.

See Also:

- "Overview of the Optimizer"
- Oracle Database SQL Tuning Guide to learn more about gathering statistics
- Oracle Database Administrator's Guide to learn how to add virtual columns to a table

Overview of Application Domain Indexes

An **application domain index** is a customized index specific to an application.

Extensive indexing can:

- Accommodate indexes on customized, complex data types such as documents, spatial data, images, and video clips (see)
- Make use of specialized indexing techniques

You can encapsulate application-specific index management routines as an indextype schema object, and then define a domain index on table columns or attributes of an object type. Extensible indexing can efficiently process application-specific operators.

The application software, called the *cartridge*, controls the structure and content of a domain index. The database interacts with the application to build, maintain, and search the domain index. The index structure itself can be stored in the database as an index-organized table or externally as a file.

See Also:

- Oracle Database Development Guide to learn more about complex data types
- Oracle Database Data Cartridge Developer's Guide for information about using data cartridges within the Oracle Database extensibility architecture



Overview of Index-Organized Tables

An **index-organized table** is a table stored in a variation of a B-tree index structure. In contrast, a **heap-organized table** inserts rows 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. Thus, the index is the data, and the data is the index. Applications manipulate index-organized tables just like heap-organized tables, using SQL statements.

For an analogy of an index-organized table, suppose a human resources manager has a book case of cardboard boxes. Each box is labeled with a number—1, 2, 3, 4, and so on—but the boxes do not sit on the shelves in sequential order. Instead, each box contains a pointer to the shelf location of the next box in the sequence.

Folders containing employee records are stored in each box. The folders are sorted by employee ID. Employee King has ID 100, which is the lowest ID, so his folder is at the bottom of box 1. The folder for employee 101 is on top of 100, 102 is on top of 101, and so on until box 1 is full. The next folder in the sequence is at the bottom of box 2.

In this analogy, ordering folders by employee ID makes it possible to search efficiently for folders without having to maintain a separate index. Suppose a user requests the records for employees 107, 120, and 122. Instead of searching an index in one step and retrieving the folders in a separate step, the manager can search the folders in sequential order and retrieve each folder as found.

Index-organized tables provide faster access to table rows by primary key or a valid prefix of the key. The presence of non-key columns of a row in the leaf block avoids an additional data block I/O. For example, the salary of employee 100 is stored in the index row itself. Also, because rows are stored in primary key order, range access by the primary key or prefix involves minimal block I/Os. Another benefit is the avoidance of the space overhead of a separate primary key index.

Index-organized tables are useful when related pieces of data must be stored together or data must be physically stored in a specific order. A typical use of this type of table is for information retrieval, spatial data, and OLAP applications.

- Index-Organized Table Characteristics
 The database system performs all operations on index-organized tables by manipulating the B-tree index structure.
- Index-Organized Tables with Row Overflow Area
 When creating an index-organized table, you can specify a separate segment as a row overflow area.
- Secondary Indexes on Index-Organized Tables
 A secondary index is an index on an index-organized table.



See Also:

- Oracle Spatial Developer's Guide for an overview of Oracle Spatial and Graph
- Oracle Database Administrator's Guide to learn how to manage index-organized tables
- Oracle Database SQL Language Reference for CREATE TABLE ...
 ORGANIZATION INDEX syntax and semantics

Index-Organized Table Characteristics

The database system performs all operations on index-organized tables by manipulating the B-tree index structure.

The following table summarizes the differences between index-organized tables and heaporganized tables.

Table 5-6 Comparison of Heap-Organized Tables with Index-Organized Tables

| Heap-Organized Table | Index-Organized Table |
|--|--|
| The rowid uniquely identifies a row. Primary key constraint may optionally be defined. | Primary key uniquely identifies a row. Primary key constraint must be defined. |
| Physical rowid in ROWID pseudocolumn allows building secondary indexes. | Logical rowid in ROWID pseudocolumn allows building secondary indexes. |
| Individual rows may be accessed directly by rowid. | Access to individual rows may be achieved indirectly by primary key. |
| Sequential full table scan returns all rows in some order. | A full index scan or fast full index scan returns all rows in some order. |
| Can be stored in a table cluster with other tables. | Cannot be stored in a table cluster. |
| Can contain a column of the LONG data type and columns of LOB data types. | Can contain LOB columns but not LONG columns. |
| Can contain virtual columns (only relational heap tables are supported). | Cannot contain virtual columns. |

Index-Organized Tables can be compressed using Advanced Low compression by adding the COMPRESS ADVANCED LOW clause to the CREATE TABLE statement. Advanced Low compression is an adaptive form of prefix compression that autonomously calculates the optimal number of prefix columns for every leaf block, leading to optimal compression at the block level without any user intervention. This includes the possibility of not compressing at all, if appropriate. This guarantees that negative compression is not possible. Greater compression ratios are observed with negligible overall overhead.

Scans on index-organized tables may be offloaded to Exadata Smart Scan where the optimizer has deemed the query can be executed as a fast full scan. This does not require any user intervention. Full cell side processing will be supported on index-organized table leaf blocks where all columns of interest reside in the leaf block. For cases where some columns reside in the overflow segment, smart scan will not process such columns and the database will complete the scan.

Figure 5-3 illustrates the structure of an index-organized departments table. The leaf blocks contain the rows of the table, ordered sequentially by primary key. For example, the first value

in the first leaf block shows a department ID of 20, department name of Marketing, manager ID of 201, and location ID of 1800.

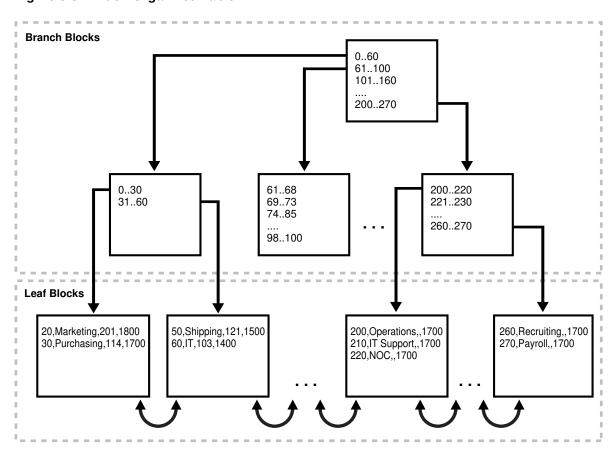


Figure 5-3 Index-Organized Table

Example 5-10 Scan of Index-Organized Table

An index-organized table stores all data in the same structure and does not need to store the rowid. As shown in Figure 5-3, leaf block 1 in an index-organized table might contain entries as follows, ordered by primary key:

```
20, Marketing, 201, 1800
30, Purchasing, 114, 1700
```

Leaf block 2 in an index-organized table might contain entries as follows:

```
50, Shipping, 121, 1500 60, IT, 103, 1400
```

A scan of the index-organized table rows in primary key order reads the blocks in the following sequence:

- Block 1
- 2. Block 2



Example 5-11 Scan of Heap-Organized Table

To contrast data access in a heap-organized table to an index-organized table, suppose block 1 of a heap-organized departments table segment contains rows as follows:

```
50, Shipping, 121, 1500 20, Marketing, 201, 1800
```

Block 2 contains rows for the same table as follows:

```
30, Purchasing, 114, 1700 60, IT, 103, 1400
```

A B-tree index leaf block for this heap-organized table contains the following entries, where the first value is the primary key and the second is the rowid:

```
20, AAAPeXAAFAAAAAyAAD
30, AAAPeXAAFAAAAAyAAA
50, AAAPeXAAFAAAAAyAAC
60, AAAPeXAAFAAAAAyAAB
```

A scan of the table rows in primary key order reads the table segment blocks in the following sequence:

- 1. Block 1
- Block 2
- 3. Block 1
- 4. Block 2

Thus, the number of block I/Os in this example is double the number in the index-organized example.

See Also:

- "Table Organization"
 - to learn more about heap-organized tables
- "Introduction to Logical Storage Structures"

to learn more about the relationship between segments and data blocks

Index-Organized Tables with Row Overflow Area

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.

See Also:

- Oracle Database Administrator's Guide to learn how to use the OVERFLOW clause
 of CREATE TABLE to set a row overflow area
- Oracle Database SQL Language Referencefor CREATE TABLE ... OVERFLOW syntax and semantics

Secondary Indexes on Index-Organized Tables

A **secondary index** is an index on an index-organized table.

In a sense, a secondary index is an index on an index. It is an independent schema object and is stored separately from the index-organized table.

Oracle Database uses row identifiers called logical rowids for index-organized tables. A logical rowid is a base64-encoded representation of the table primary key. The logical rowid length depends on the primary key length.

Rows in index leaf blocks can move within or between blocks because of insertions. Rows in index-organized tables do not migrate as heap-organized rows do. Because rows in index-organized tables do not have permanent physical addresses, the database uses logical rowids based on primary key.

For example, assume that the departments table is index-organized. The location_id column stores the ID of each department. The table stores rows as follows, with the last value as the location ID:

```
10, Administration, 200, 1700
20, Marketing, 201, 1800
30, Purchasing, 114, 1700
40, Human Resources, 203, 2400
```

A secondary index on the <code>location_id</code> column might have index entries as follows, where the value following the comma is the logical rowid:

```
1700, *BAFAJqoCwR/+
1700, *BAFAJqoCwQv+
1800, *BAFAJqoCwRX+
2400, *BAFAJqoCwSn+
```



Secondary indexes provide fast and efficient access to index-organized tables using columns that are neither the primary key nor a prefix of the primary key. For example, a query of the names of departments whose ID is greater than 1700 could use the secondary index to speed data access.

- Logical Rowids and Physical Guesses
 Secondary indexes use logical rowids to locate table rows.
- Bitmap Indexes on Index-Organized Tables
 A secondary index on an index-organized table can be a **bitmap index**. A bitmap index stores a bitmap for each index key.

See Also:

- "Rowid Data Types" to learn more about the use of rowids, and the ROWID pseudocolumn
- "Chained and Migrated Rows"

to learn why rows migrate, and why migration increases the number of I/Os

- Oracle Database Administrator's Guide to learn how to create secondary indexes on an index-organized table
- Oracle Database VLDB and Partitioning Guide to learn about creating secondary indexes on indexed-organized table partitions

Logical Rowids and Physical Guesses

Secondary indexes use 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. Oracle Database can use physical guesses to probe directly into the leaf block of the index-organized table, bypassing the primary key search. When the physical location of a row changes, the logical rowid remains valid even if it contains a physical guess that is stale.

For a heap-organized table, access by a secondary index involves a scan of the secondary index and an additional I/O to fetch the data block containing the row. For index-organized tables, access by a secondary index varies, depending on the use and accuracy of physical quesses:

- Without physical guesses, access involves two index scans: a scan of the secondary index followed by a scan of the primary key index.
- With physical guesses, access depends on their accuracy:
 - With accurate physical guesses, access involves a secondary index scan and an additional I/O to fetch the data block containing the row.
 - 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.



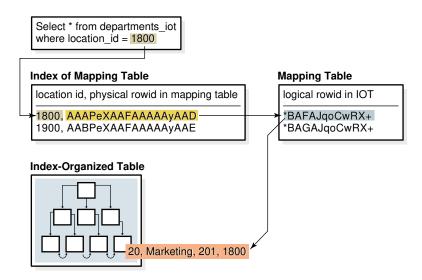
Bitmap Indexes on Index-Organized Tables

A secondary index on an index-organized table can be a **bitmap index**. A bitmap index stores a bitmap for each index key.

When bitmap indexes exist on an index-organized table, all the bitmap indexes use a heap-organized mapping table. The mapping table stores the logical rowids of the index-organized table. Each mapping table row stores one logical rowid for the corresponding index-organized table row.

The database accesses a bitmap index using a search key. If the database finds the key, then the bitmap entry is converted to a physical rowid. With heap-organized tables, the database uses the physical rowid to access the base table. With index-organized tables, the database uses the physical rowid to access the mapping table, which in turn yields a logical rowid that the database uses to access the index-organized table. The following figure illustrates index access for a query of the departments iot table.

Figure 5-4 Bitmap Index on Index-Organized Table





Movement of rows in an index-organized table does not leave the bitmap indexes built on that index-organized table unusable.

See Also:

"Rowids of Row Pieces" to learn about the differences between physical and logical rowids

