Database Concepts

3 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.

This chapter contains the following sections:

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.

Note:

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:

Benefits 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** (glossary.htm#GUID-BF9B54D6-892E-4C3B-8536-38958ACC069D) to find a value. For example, a **query** (glossary.htm#GUID-CCF91C9F-A98A-498F-A84B-58A0FA16CD6E) 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.

In general, consider creating an index on a column in any of 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** (glossary.htm#GUID-67F8FE8C-EBA5-4796-820A-8919982A1411) exists on the indexed column or columns. The index is a means to avoid a full table **lock** (glossary.htm#GUID-6D016291-A487-4F88-BE0B-ACF8FA2AE72C) that would otherwise be required if you update the parent table **primary key** (glossary.htm#GUID-8640EFA5-276C-4812-A078-1F21F55F4200) , 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.

Index Usability and Visibility

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

These properties are defined as follows:

- Usability

  An **unusable index** (glossary.htm#GUID-5EEB4F35-818F-4478-8BE3-F70CF22CD11F) , which is ignored by the **optimizer** (glossary.htm#GUID-54114749-0A81-41D7-8E16-7B76D93CEE2B) , 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** (glossary.htm#GUID-EC12AA68-8C89-43B3-B1F9-3AABF7CAEB9F) .

- Visibility

An **invisible index** (glossary.htm#GUID-B60609DA-2397-4715-B7E2-75AEC3FAD0BF) 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.

See Also:

"Overview of the Optimizer (sqllangu.htm#GUID-3F42B1AA-530A-4144-8179-F0074832AF81) "

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 `customer_id` column of the sample table `oe.orders`:

```
CREATE INDEX ord_customer_ix ON orders (customer_id);
```

In the preceding statement, the `customer_id` column is the index key. The index itself is named `ord_customer_ix`.

Note:

Primary and unique keys automatically have indexes, but you might want to create an index on a **foreign key** (glossary.htm#GUID-EEE8A67D-6B0E-468E-9554-48FEE552BC9A) .

See Also:

- Data Integrity (datainte.htm#GUID-6A89FF39-AD42-4399-BD1B-E51ECEE50B4E)
- *Oracle Database SQL Language Reference* (../SQLRF/statements_5013.htm#SQLRF01209) `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 `last_name`, `job_id`, and `salary` columns in the `employees` table. Also assume that `last_name` has high **cardinality** (glossary.htm#GUID-5CD22620-6D7A-40DC-BA09-EE3B5339C7F8) , 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.

> **Note:**
>
> 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 (indexiot.htm#GUID-BB75CC6E-90E5-457F-A4A0-0ABBEB755181) ").

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* (../TGSQL/tgsql_indc.htm#TGSQL855) for more information about using composite indexes

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** (glossary.htm#GUID-647822F1-EFF1-4E26-BE22-D54081BE1C7B) 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** (glossary.htm#GUID-8854502F-2B8F-4ABC-98FA-BBFC3695A964) , 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.

Indexes are categorized as follows:

- B-tree indexes

  These 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** (glossary.htm#GUID-8D6D0C64-6AC8-4B22-A9AF-1B62F61AE10B) can retrieve data sorted by the indexed columns. B-tree indexes have the following subtypes:

  - Index-organized tables

    An index-organized table differs from a heap-organized because the data is itself the index. See "Overview of Index-Organized Tables (indexiot.htm#GUID-DAEC075B-C16D-4A57-898C-70EBCB364F0C) ".

  - 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. See "Reverse Key Indexes (indexiot.htm#GUID-2646BDA9-F776-4C98-9487-C7EBC2EECF0B) ".

  - Descending indexes

    This type of index stores data on a particular column or columns in descending order. See "Ascending and Descending Indexes (indexiot.htm#GUID-8C2EA2EC-18E5-4E4A-BF74-D1DE86D7F24A) ".

  - B-tree cluster indexes

    This type indexes a table cluster key. Instead of pointing to a row, the key points to the block that contains rows related to the cluster key. See "Overview of Indexed Clusters (tablecls.htm#GUID-CC31365B-83B0-4E09-A047-BF1B79AC887A) ".

- 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. See "Overview of Bitmap Indexes (indexiot.htm#GUID-B15C4817-7748-456D-9740-8B9628AF9F47) ".

- 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. See "Overview of Function-Based Indexes (indexiot.htm#GUID-9AD7651D-0F0D-4FC6-A984-5845F0224EE6) ".

- 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. See "Overview of Application Domain Indexes (indexiot.htm#GUID-9586EB86-4B84-4A43-A66D-958776FE558B) ".

See Also:

- *Oracle Database Administrator's Guide* (../ADMIN/indexes.htm#ADMIN016) to learn how to manage indexes
- *Oracle Database SQL Tuning Guide* (../TGSQL/tgsql_indc.htm#TGSQL850) to learn about different index types

How the Database Maintains Indexes

The database automatically maintains and uses indexes after they are created. Indexes automatically reflect data changes, such as adding, updating, and deleting rows in their underlying tables, with no additional actions required by users.

Retrieval performance of indexed data remains almost constant, even as rows are inserted. However, the presence of many indexes on a table degrades **DML** (glossary.htm#GUID-B5F2F112-1B33-41B5-B63D-9DC8F99A369D) performance because the database must also update the indexes.

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** (glossary.htm#GUID-AA66891C-71B2-4D55-8F64-0E427AE24E88) 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.

See Also:

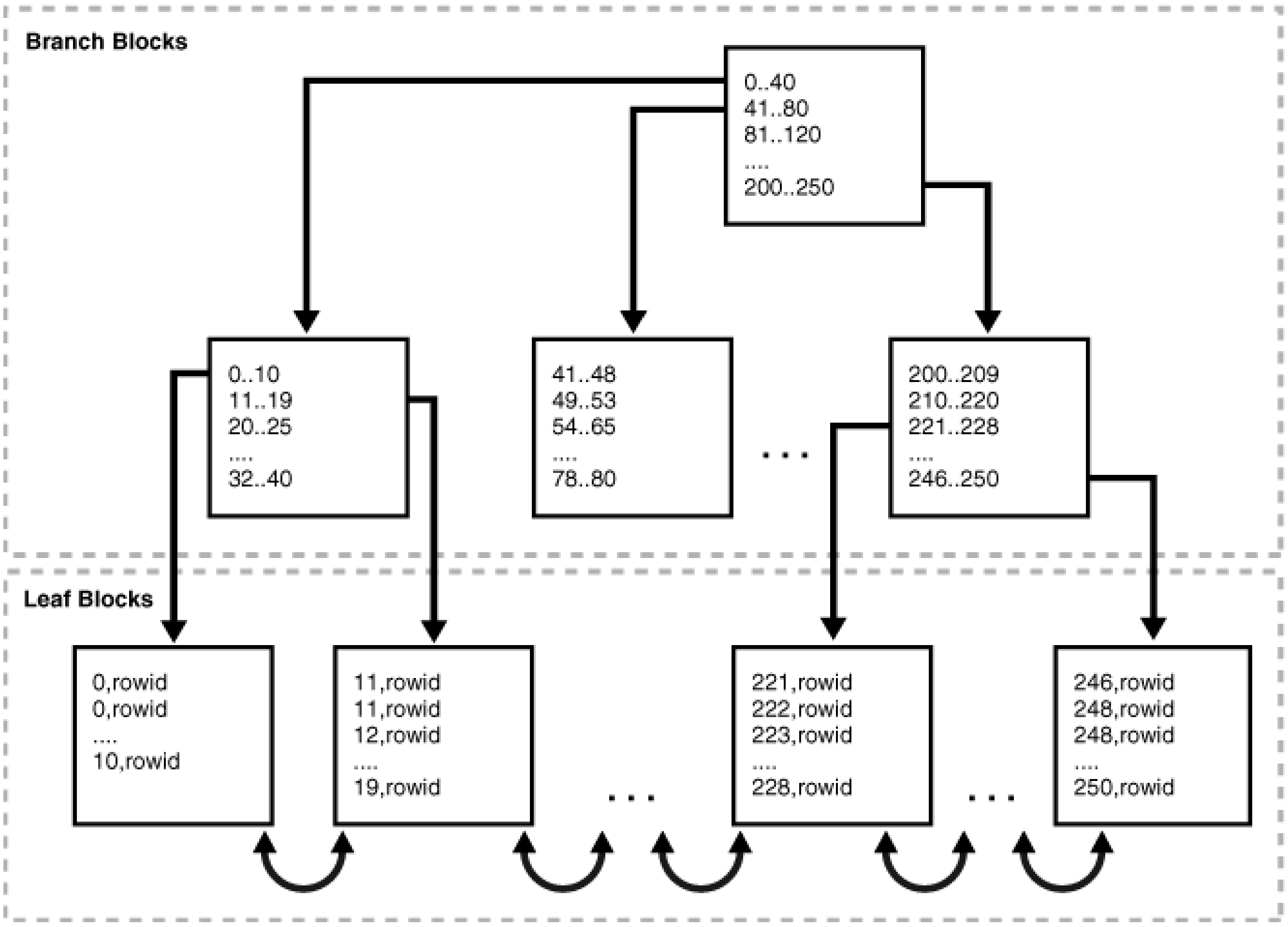
"Overview of Index Blocks (logical.htm#GUID-82531CDF-407E-4D70-AFD0-8E8929B72783) "

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 3-1 Internal Structure of a B-tree Index



Description of "Figure 3-1 Internal Structure of a B-tree Index" (img_text/GUID-83032A7C-64B8-4705-A1A4-9289C791DBE2-print.htm)

This section contains the following topics:

- Branch Blocks and Leaf Blocks (indexiot.htm#GUID-89A9F85F-BE0E-4596-AEC3-CAF0D821B1CA)
- Index Scans (indexiot.htm#GUID-D54BD14D-0065-4B61-B2F6-5567913B16CD)
- Reverse Key Indexes (indexiot.htm#GUID-2646BDA9-F776-4C98-9487-C7EBC2EECF0B)
- Ascending and Descending Indexes (indexiot.htm#GUID-8C2EA2EC-18E5-4E4A-BF74-D1DE86D7F24A)
- Index Compression (indexiot.htm#GUID-33AEA2E3-1355-4224-BB39-890A71784062)

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 3-1 (indexiot.htm#GUID-FC93A85B-C237-4249-AD1E-FF54576ED050__I5765) , 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 3-1 (indexiot.htm#GUID-FC93A85B-C237-4249-AD1E-FF54576ED050__I5765) , 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 3-1 (indexiot.htm#GUID-FC93A85B-C237-4249-AD1E-FF54576ED050__I5765) the leftmost leaf block (`0-10`) is linked to the second leaf block (`11-19`).

> **Note:**
>
> 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.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* (../TGSQL/tgsql_optop.htm#TGSQL234) 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 3-1 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 `department_id`, `last_name`, and `salary` 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 `employees` 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** (glossary.htm#GUID-BF9B54D6-892E-4C3B-8536-38958ACC069D) 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 3-2 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** (glossary.htm#GUID-B5AA8627-E7DC-487B-8D4B-2DE3F1497A83) specifies a combination of one or more expressions and logical (Boolean) operators and returns a value of `TRUE`, `FALSE`, or `UNKNOWN`.

The database commonly uses an index range scan to access selective data. The **selectivity** (glossary.htm#GUID-AA830B4F-E5E8-4CCC-A960-0FA0E2F8DE12) 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** (glossary.htm#GUID-891CF9E9-78CD-470C-9C4A-D65A101B2C38) , 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 `last_name` 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 `Austin`.

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 3-3 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 `M` or `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,Wolsey@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 `cust_gender` is not specified in the `WHERE` 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 `F` and a second subindex with the key `M`.

When searching for the record for the customer whose email is `Abbey@company.example.com`, the database searches the subindex with the value `F` first and then searches the subindex with the value `M`. 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 (../TGSQL/tgsql_optop.htm#TGSQL238) 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 3-4 Clustering Factor

Assume that the `employees` table fits into two data blocks. Table 3-1 (indexiot.htm#GUID-9F572458-50AF-4B7C-BC84-807FB815AC39__CHDEIDBA) depicts the rows in the two data blocks (the ellipses indicate data that is not shown).

Table 3-1 Contents of Two Data Blocks in the Employees Table

Data Block 1	Data Block 2
`100 Steven` **`King`** `SKING ... 156 Janette` **`King`** `JKING ... 115 Alexander` **`Khoo`** `AKHOO ... . . . 116 Shelli` **`Baida`** `SBAIDA ... 204 Hermann` **`Baer`** `HBAER ... 105 David` **`Austin`** `DAUSTIN ... 130 Mozhe` **`Atkinson`** `MATKINSO ... 166 Sundar` **`Ande`** `SANDE ... 174 Ellen` **`Abel`** `EABEL ...`	`149 Eleni` **`Zlotkey`** `EZLOTKEY ... 200 Jennifer` **`Whalen`** `JWHALEN ... . . . 137 Renske` **`Ladwig`** `RLADWIG ... 173 Sundita` **`Kumar`** `SKUMAR ... 101 Neena` **`Kochar`** `NKOCHHAR ...`

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* (../REFRN/GUID-E39825BA-70AC-45D8-AF30-C7FF561373B6.htm#REFRN20088) 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** (glossary.htm#GUID-77ADFA8C-BBA8-4E2D-B122-2D490FC35CFE) . In this case, the index stores data on a specified column or columns in descending order. If the index in Table 3-1 (indexiot.htm#GUID-9F572458-50AF-4B7C-BC84-807FB815AC39__CHDEIDBA) 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 `last_name` and `department_id` columns 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* (../TGSQL/tgsql_optop.htm#TGSQL236) to learn more about ascending and descending index searches
- *Oracle Database SQL Language Reference* (../SQLRF/statements_5013.htm#SQLRF53994) 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 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 `order_mode` and `order_status` columns. An index block could have entries as shown in the follow example:

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

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 AAAPvCAAFAAAAFaAAt
```

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* (../ADMIN/tables.htm#ADMIN11692) to learn how to use compressed indexes
> - *Oracle Database VLDB and Partitioning Guide* (../VLDBG/GUID-0C13E8A8-568B-4A60-8A46-F1120D193EFE.htm#VLDBG1111) to learn how to use prefix compression for partitioned indexes

- *Oracle Database SQL Language Reference* (../SQLRF/statements_5013.htm#SQLRF53999) for descriptions of the `key_compression` clause of `CREATE INDEX`

Advanced Index Compression

Starting with Oracle Database 12*c* Release 1 (12.1.0.2), **advanced index compression** improves on traditional prefix compression for indexes on heap-organized tables. Unlike prefix compression, which uses fixed duplicate key elimination for every block, advanced compression uses adaptive duplicate key elimination on a per-block basis.

Advanced index compression works at the block level in the following situations:

- 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.

The main advantage of this form of compression is that the database automatically chooses the best compression for each block, so that the user does not require knowledge of data characteristics.

Enable advanced index compression using the `COMPRESS ADVANCED LOW` clause, as in the following example:

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

See Also:

- *Oracle Database Administrator's Guide* (../ADMIN/indexes.htm#ADMIN14308) to learn how to enable advanced index compression
- *Oracle Database SQL Language Reference* (../SQLRF/statements_5013.htm#SQLRF53999) for descriptions of the `key_compression` clause of `CREATE INDEX`

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** (glossary.htm#GUID-5CD22620-6D7A-40DC-BA09-EE3B5339C7F8) , 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** (glossary.htm#GUID-709E943F-FF0E-4AA6-979A-C4CB2A7B0C29) applications.

See Also:

- *Oracle Database SQL Tuning Guide* (../TGSQL/tgsql_indc.htm#TGSQL867) to learn how to use bitmap indexes for performance
- *Oracle Database Data Warehousing Guide* (../DWHSG/schemas.htm#DWHSG8130) 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:

```
SQL> SELECT cust_id, cust_last_name, cust_marital_status, cust_gender 2 FROM
sh.customers 3 WHERE ROWNUM < 8 ORDER BY cust_id; CUST_ID CUST_LAST_ CUST_MAR C ------
---- ---------- -------- - 1 Kessel M 2 Koch F 3 Emmerson M 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 3-2 (indexiot.htm#GUID-4D0B3A22-51F7-4F6E-8F95-B6AD7F9A2729__CBBGCGFC) illustrates the bitmap index for the `cust_gender` column output shown in the preceding example. It consists of two separate bitmaps, one for each gender.

Table 3-2 Sample Bitmap for One Column

Value	Row 1	Row 2	Row 3	Row 4	Row 5	Row 6	Row 7
`M`	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 `M` value contains a `1` as its first bit because the gender is `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 `M` as their value.

Note:

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 `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 3-3 (indexiot.htm#GUID-4D0B3A22-51F7-4F6E-8F95-B6AD7F9A2729__CBBGEJDG) . To identify the customers who satisfy the criteria, Oracle Database can use the resulting bitmap to access the table.

Table 3-3 Sample Bitmap for Two Columns

Value	Row 1	Row 2	Row 3	Row 4	Row 5	Row 6	Row 7
`M`	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:

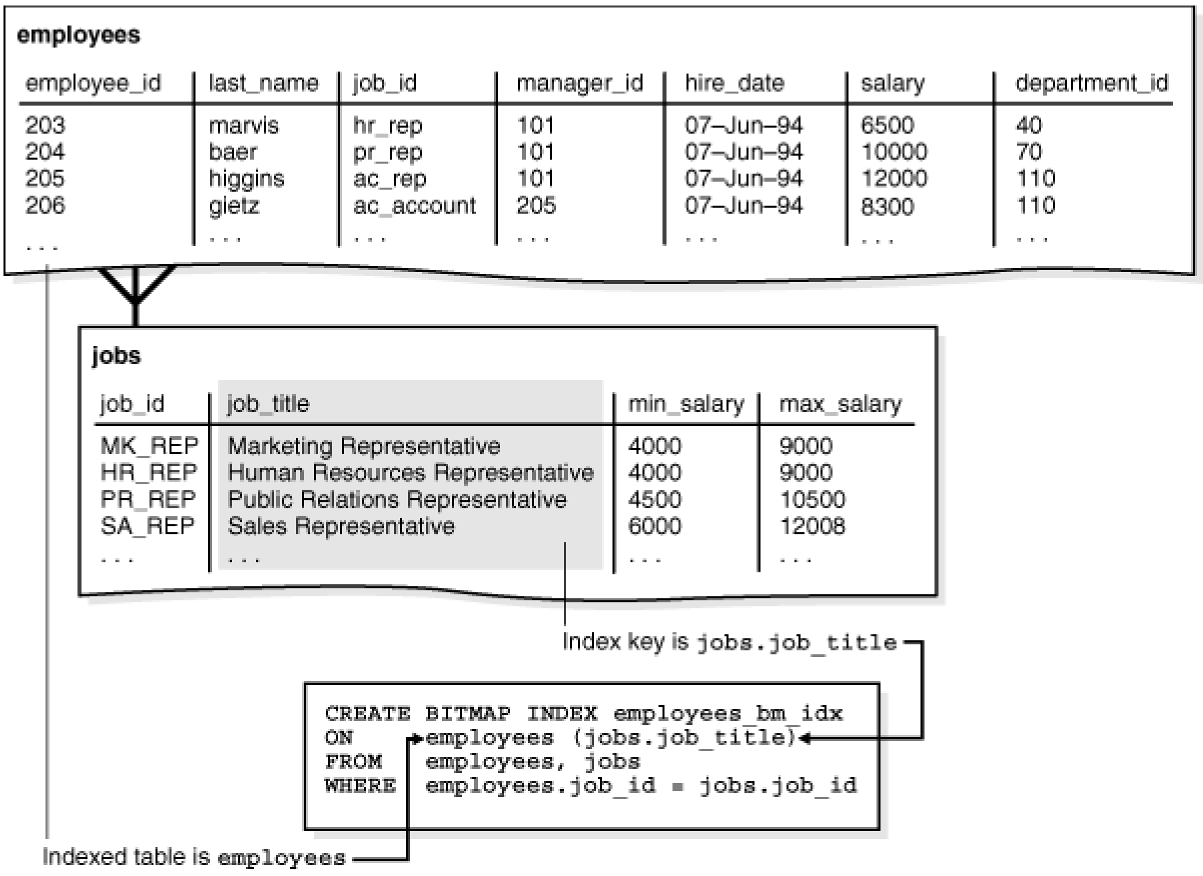
```
SELECT COUNT(*) FROM employees, jobs WHERE employees.job_id = jobs.job_id AND
jobs.job_title = 'Accountant';
```

The preceding query would typically use an index on `jobs.job_title` to retrieve the rows for `Accountant` and then the job ID, and an index on `employees.job_id` 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 3-2 Bitmap Join Index



employee_id	last_name	job_id	manager_id	hire_date	salary	department_id
203	marvis	hr_rep	101	07–Jun–94	6500	40
204	baer	pr_rep	101	07–Jun–94	10000	70
205	higgins	ac_rep	101	07–Jun–94	12000	110
206	gietz	ac_account	205	07–Jun–94	8300	110
. . .	. . .	. . .	. . .	. . .	. . .	. . .

job_id	job_title	min_salary	max_salary
MK_REP	Marketing Representative	4000	9000
HR_REP	Human Resources Representative	4000	9000
PR_REP	Public Relations Representative	4500	10500
SA_REP	Sales Representative	6000	12008
. . .	. . .	. . .	. . .

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

Description of "Figure 3-2 Bitmap Join Index" (img_text/GUID-3D4A9BDE-80EF-40BD-998C-3D34F82B5D78-print.htm)

Conceptually, `employees_bm_idx` is an index of the `jobs.title` column in the SQL query shown in the following query (sample output included). The `job_title` key in the index points to rows in the `employees` table. A query of the number of accountants can use the index to avoid accessing the `employees` and `jobs` 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 ----------------------------------- ------------------
Accountant AAAQNKAAFAAAABSAAL Accountant AAAQNKAAFAAAABSAAN Accountant
AAAQNKAAFAAAABSAAM Accountant AAAQNKAAFAAAABSAAJ Accountant AAAQNKAAFAAAABSAAK
Accounting Manager AAAQNKAAFAAAABTAAH Administration Assistant AAAQNKAAFAAAABTAAC
Administration Vice President AAAQNKAAFAAAABSAAC Administration Vice President
AAAQNKAAFAAAABSAAB . . .
```

In a data warehouse, the **join condition** (glossary.htm#GUID-6DF529B9-611D-4C00-BAF8-614E86BCB39E) is an **equijoin** (glossary.htm#GUID-2DD43111-489E-4F08-9949-AD0C21858DCB) (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.

See Also:

Oracle Database Data Warehousing Guide (../DWHSG/schemas.htm#DWHSG9047) 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 `jobs.job_title` is the key column of a bitmap index, then the index data is stored in one B-tree. The individual bitmaps are stored in the leaf blocks.

Example 3-5 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 (logical.htm#GUID-EFB292CB-87EA-42AA-808C-BD85E540BACC) "

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.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* (../ADMIN/indexes.htm#ADMIN11730) to learn how to create function-based indexes
> - *Oracle Database SQL Tuning Guide* (../TGSQL/tgsql_indc.htm#TGSQL864) for more information about using function-based indexes
> - *Oracle Database SQL Language Reference* (../SQLRF/statements_5013.htm#SQLRF53993) 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 3-6 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):

```
SELECT employee_id, last_name, first_name, 12*salary*commission_pct AS "ANNUAL SAL"
FROM employees WHERE (12 * salary * commission_pct) < 30000 ORDER BY "ANNUAL SAL"
DESC; 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 3-7 Index Based on an UPPER Function

Function-based indexes defined on the SQL functions `UPPER(`*`column_name`*`)` or `LOWER(`*`column_name`*`)` facilitate case-insensitive searches. For example, suppose that the `first_name` column in `employees` contains mixed-case characters. You create the following function-based index on the `hr.employees` 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 3-8 Indexing Specific Rows in a Table

A function-based index is also useful for indexing only specific rows in a table. For example, the `cust_valid` column in the `sh.customers` table has either `I` or `A` as a value. To index only the `A` rows, you could write a function that returns a null value for any rows other than the `A` 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* (../NLSPG/ch5lingsort.htm#NLSPG294) for information about linguistic indexes
> - *Oracle Database SQL Language Reference* (../SQLRF/functions001.htm#SQLRF51173) to learn more about SQL functions

Optimization with Function-Based Indexes

For queries with expressions in a `WHERE` clause, the optimizer can use an index range scan on a function-based index.
The range scan **access path** (glossary.htm#GUID-56F7FB03-3499-4D51-8D23-F86C45194F09) is especially beneficial when the predicate is highly selective, that is, when it chooses relatively few rows.
In Example 3-6 (indexiot.htm#GUID-9B3F9FB8-6A17-442B-A294-06CD8096CC10__GUID-66A7BA6D-17B4-4F72-A490-81D10C4A58D6) , if an index is built on the expression `12*salary*commission_pct`, then the **optimizer** (glossary.htm#GUID-54114749-0A81-41D7-8E16-7B76D93CEE2B) can use an index range scan.

A **virtual column** (glossary.htm#GUID-E95FC6AD-C932-4DE2-9D7B-B98D1168E7DA) 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.

> **See Also:**
>
> - "Overview of the Optimizer (sqllangu.htm#GUID-3F42B1AA-530A-4144-8179-F0074832AF81) "
> - *Oracle Database SQL Tuning Guide* (../TGSQL/tgsql_stats.htm#TGSQL389) to learn more about gathering statistics
> - *Oracle Database Administrator's Guide* (http://www.oracle.com/pls/topic/lookup?ctx=E50529-01&id=ADMIN12494) 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 "Unstructured Data (cncptdev.htm#GUID-B39E395E-99FF-4FF1-AC65-E581BD53EFA9) ")

- 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 Data Cartridge Developer's Guide* (../ADDCI/introduction.htm#ADDCI110) 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** (glossary.htm#GUID-A6734D1E-A45B-4BE3-ABF8-F6201A40F6B3) applications.

> **See Also:**
>
> - "Overview of Oracle Spatial and Graph (cncptdev.htm#GUID-F3C48565-B360-40F2-A201-239D614139B5) "
> - "OLAP (cmntopc.htm#GUID-8905A5A6-1546-47E8-A7D7-C83E9D7F4903) "

- *Oracle Database Administrator's Guide* (../ADMIN/tables.htm#ADMIN01506) to learn how to manage index-organized tables
- *Oracle Database SQL Tuning Guide* (../TGSQL/tgsql_indc.htm#TGSQL866) to learn how to use index-organized tables to improve performance
- *Oracle Database SQL Language Reference* (../SQLRF/statements_7002.htm#SQLRF54492) 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 heap-organized tables.

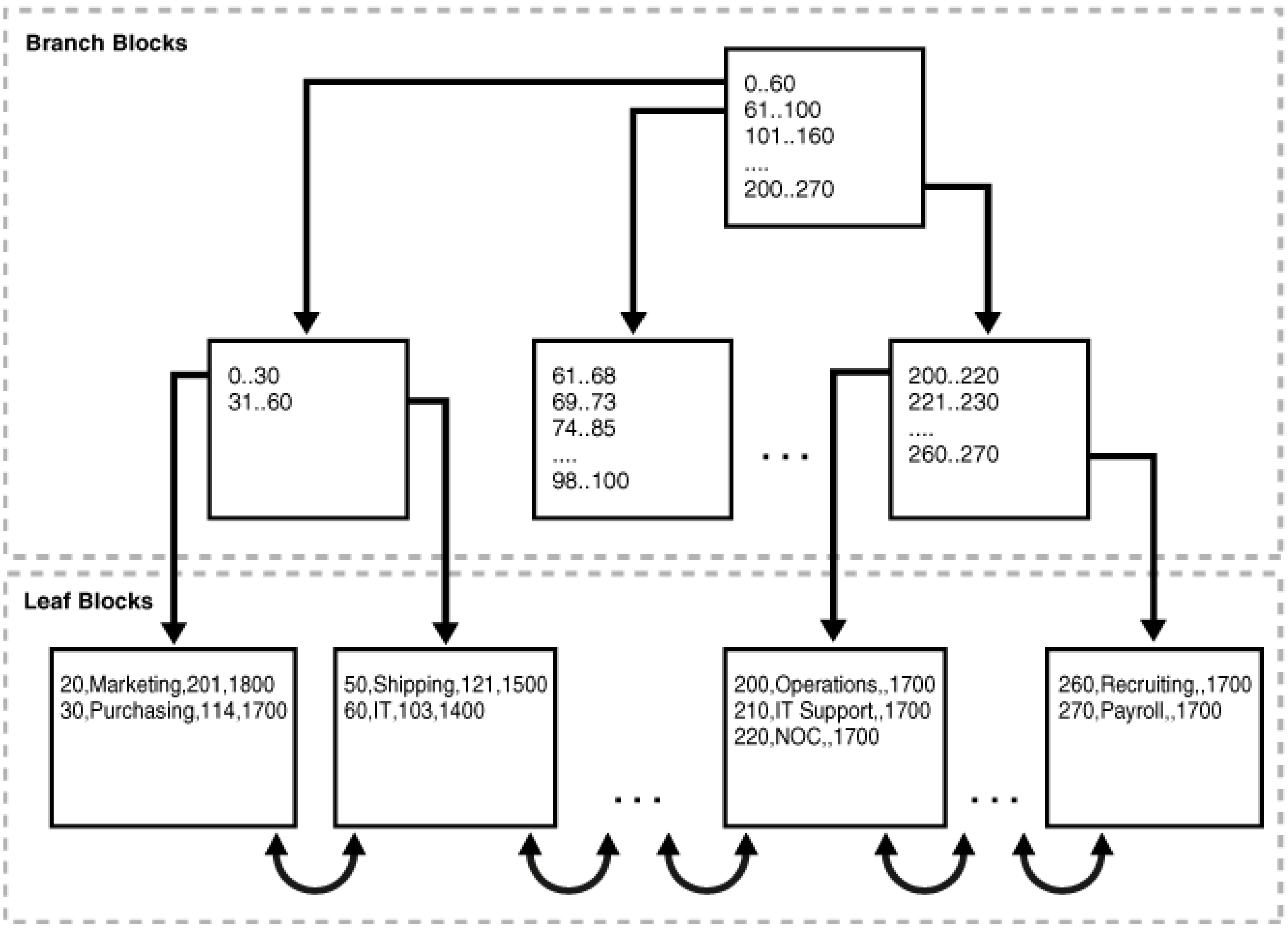
Table 3-4 Comparison of Heap-Organized Tables with Index-Organized Tables

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** (glossary.htm#GUID-175D4923-5C7E-4FF0-A69B-C4D8F3D93A3D) 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** (glossary.htm#GUID-BF9B54D6-892E-4C3B-8536-38958ACC069D) 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** (glossary.htm#GUID-1C56177E-6BEE-4FE7-B45E-38298CDB946D) with other tables.	Cannot be stored in a table cluster.
Can contain a column of the `LONG` data type and columns of **LOB** (glossary.htm#GUID-A85748CE-C4D4-43ED-BD49-29AFC4AD3A02) data types.	Can contain LOB columns but not `LONG` columns.
Can contain virtual columns (only relational heap tables are supported).	Cannot contain virtual columns.

Figure 3-3 (indexiot.htm#GUID-279309E0-3D53-47C5-8FA8-249BD835C88A__CHDJGJHG) 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 3-3 Index-Organized Table



Description of "Figure 3-3 Index-Organized Table" (img_text/GUID-BC216AA8-3A01-4CD3-8A50-BF94EF40B544-print.htm)

Example 3-9 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 3-3 (indexiot.htm#GUID-279309E0-3D53-47C5-8FA8-249BD835C88A__CHDJGJHG) , 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:

1. Block 1
2. Block 2

Example 3-10 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
2. 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 (tablecls.htm#GUID-4C718D65-D7AF-4596-9A31-C11938047224) "
- "Introduction to Logical Storage Structures (logical.htm#GUID-52FE1A8C-74EA-4B81-B1AC-69FD34252659) "

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* (../ADMIN/tables.htm#ADMIN11685) to learn how to use the `OVERFLOW` clause of `CREATE TABLE` to set a row overflow area
- *Oracle Database SQL Language Reference* (../SQLRF/statements_7002.htm#SQLRF54499) for `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, it is an index on an index. The secondary index is an independent schema object and is stored separately from the index-organized table.

As explained in "Rowid Data Types (tablecls.htm#GUID-0258C4C2-2BF2-445F-B1E1-F282A57A6859) ", Oracle Database uses row identifiers called logical rowids for index-organized tables. A **logical rowid** (glossary.htm#GUID-92AE2643-9DA4-4364-942F-81F006759302) 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 (see "Chained and Migrated Rows (logical.htm#GUID-49D4E586-57BF-4310-9EE9-2DD54108E651) "). 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 `location_id` 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.

See Also:

- *Oracle Database Administrator's Guide* (../ADMIN/tables.htm#ADMIN11696) to learn how to create secondary indexes on an index-organized table
- *Oracle Database VLDB and Partitioning Guide* (../VLDBG/GUID-F2C99CFA-1090-4A62-B452-D4E116A40A34.htm#VLDBG003) to learn about creating secondary indexes on indexed-organized table partitions

Logical Rowids and Physical Guesses

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

- 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.

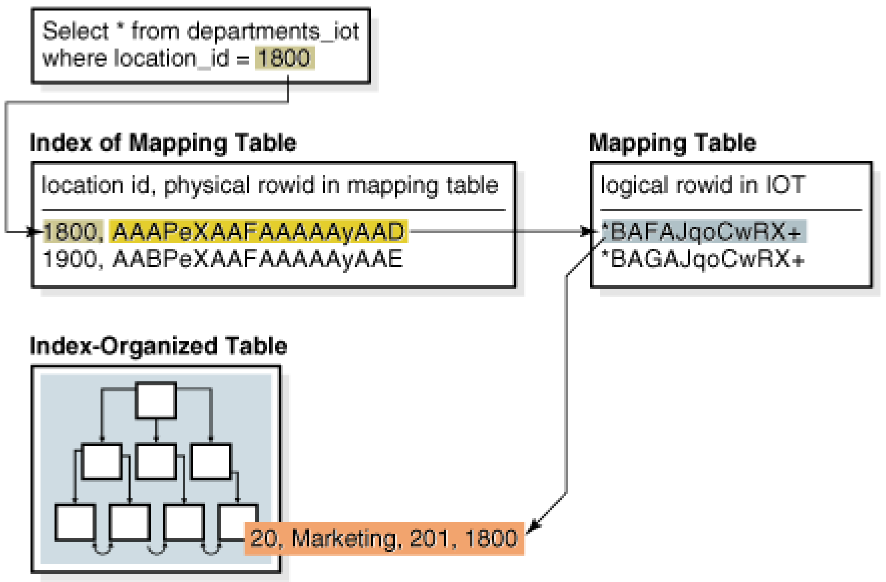
Bitmap Indexes on Index-Organized Tables

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 3-4 Bitmap Index on Index-Organized Table



Description of "Figure 3-4 Bitmap Index on Index-Organized Table" (img_text/GUID-96D91CD9-5032-417A-B9C6-E581131835E6-print.htm)

Note:

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 (tablecls.htm#GUID-83BDB6CC-8CE1-44FE-9BCB-B018AC316FFC) "