Oracle® Database

Concepts

10*g* Release 2 (10.2)

B14220-02

October 2005

ORACLE®

Oracle Database Concepts, 10*g* Release 2 (10.2)

B14220-02

Copyright © 1993, 2005, Oracle. All rights reserved.

Primary Author: Michele Cyran

Contributing Author: Paul Lane, JP Polk

Contributor: Omar Alonso, Penny Avril, Hermann Baer, Sandeepan Banerjee, Mark Bauer, Bill Bridge, Sandra Cheevers, Carol Colrain, Vira Goorah, Mike Hartstein, John Haydu, Wei Hu, Ramkumar Krishnan, Vasudha Krishnaswamy, Bill Lee, Bryn Llewellyn, Rich Long, Diana Lorentz, Paul Manning, Valarie Moore, Mughees Minhas, Gopal Mulagund, Muthu Olagappan, Jennifer Polk, Kathy Rich, John Russell, Viv Schupmann, Bob Thome, Randy Urbano, Michael Verheij, Ron Weiss, Steve Wertheimer

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software—Restricted Rights (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

Oracle, JD Edwards, PeopleSoft, and Retek are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

database administrator to hide the identity of a base table and reduce the complexity of SQL statements. Assume the following:

- A table called SALES_DATA is in the schema owned by the user JWARD.
- The SELECT privilege for the SALES_DATA table is granted to PUBLIC.

At this point, you have to query the table SALES_DATA with a SQL statement similar to the following:

```
SELECT * FROM jward.sales_data;
```

Notice how you must include both the schema that contains the table along with the table name to perform the query.

Assume that the database administrator creates a public synonym with the following SQL statement:

```
CREATE PUBLIC SYNONYM sales FOR jward.sales_data;
```

After the public synonym is created, you can query the table SALES_DATA with a simple SQL statement:

```
SELECT * FROM sales;
```

Notice that the public synonym SALES hides the name of the table SALES_DATA and the name of the schema that contains the table.

Overview of Indexes

Indexes are optional structures associated with tables and clusters. You can create indexes on one or more columns of a table to speed SQL statement execution on that table. Just as the index in this manual helps you locate information faster than if there were no index, an Oracle index provides a faster access path to table data. Indexes are the primary means of reducing disk I/O when properly used.

You can create many indexes for a table as long as the combination of columns differs for each index. You can create more than one index using the same columns if you specify distinctly different combinations of the columns. For example, the following statements specify valid combinations:

```
CREATE INDEX employees_idx1 ON employees (last_name, job_id);
CREATE INDEX employees_idx2 ON employees (job_id, last_name);
```

Oracle provides several indexing schemes, which provide complementary performance functionality:

- B-tree indexes
- B-tree cluster indexes
- Hash cluster indexes
- Reverse key indexes
- Bitmap indexes
- Bitmap join indexes

Oracle also provides support for function-based indexes and domain indexes specific to an application or cartridge.

The absence or presence of an index does not require a change in the wording of any SQL statement. An index is merely a fast access path to the data. It affects only the

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:** *Oracle Database Performance Tuning Guide* for more information about gathering statistics

Dependencies of Function-Based Indexes

Function-based indexes depend on the function used in the expression that defines the index. If the function is a PL/SQL function or package function, the index is disabled by any changes to the function specification.

To create a function-based index, the user must be granted `CREATE INDEX` or `CREATE ANY INDEX`.

To use a function-based index:

- The table must be analyzed after the index is created.
- The query must be guaranteed not to need any `NULL` values from the indexed expression, because `NULL` values are not stored in indexes.

The following sections describe additional requirements.

DETERMINISTIC Functions Any user-written function used in a function-based index must have been declared with the `DETERMINISTIC` keyword to indicate that the function will always return the same output return value for any given set of input argument values, now and in the future.

> **See Also:** *Oracle Database Performance Tuning Guide*

Privileges on the Defining Function The index owner needs the `EXECUTE` privilege on the function used to define a function-based index. If the `EXECUTE` privilege is revoked, Oracle marks the index `DISABLED`. The index owner does not need the `EXECUTE WITH GRANT OPTION` privilege on this function to grant `SELECT` privileges on the underlying table.

Resolve Dependencies of Function-Based Indexes A function-based index depends on any function that it is using. If the function or the specification of a package containing the function is redefined (or if the index owner's `EXECUTE` privilege is revoked), then the following conditions hold:

- The index is marked as `DISABLED`.
- Queries on a `DISABLED` index fail if the optimizer chooses to use the index.
- DML operations on a `DISABLED` index fail unless the index is also marked `UNUSABLE` and the initialization parameter `SKIP_UNUSABLE_INDEXES` is set to true.

To re-enable the index after a change to the function, use the `ALTER INDEX ... ENABLE` statement.

How Indexes Are Stored

When you create an index, Oracle automatically allocates an index segment to hold the index's data in a tablespace. You can control allocation of space for an index's segment and use of this reserved space in the following ways:

- Set the storage parameters for the index segment to control the allocation of the index segment's extents.
- Set the `PCTFREE` parameter for the index segment to control the free space in the data blocks that constitute the index segment's extents.

The tablespace of an index's segment is either the owner's default tablespace or a tablespace specifically named in the `CREATE INDEX` statement. You do not have to place an index in the same tablespace as its associated table. Furthermore, you can improve performance of queries that use an index by storing an index and its table in different tablespaces located on different disk drives, because Oracle can retrieve both index and table data in parallel.

See Also: "PCTFREE, PCTUSED, and Row Chaining" on page 2-6

Format of Index Blocks

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

When you create an index, Oracle fetches and sorts the columns to be indexed and stores the rowid along with the index value for each row. Then Oracle loads the index from the bottom up. For example, consider the statement:

```
CREATE INDEX employees_last_name ON employees(last_name);
```

Oracle sorts the `employees` table on the `last_name` column. It then loads the index with the `last_name` and corresponding rowid values in this sorted order. When it uses the index, Oracle does a quick search through the sorted `last_name` values and then uses the associated rowid values to locate the rows having the sought `last_name` value.

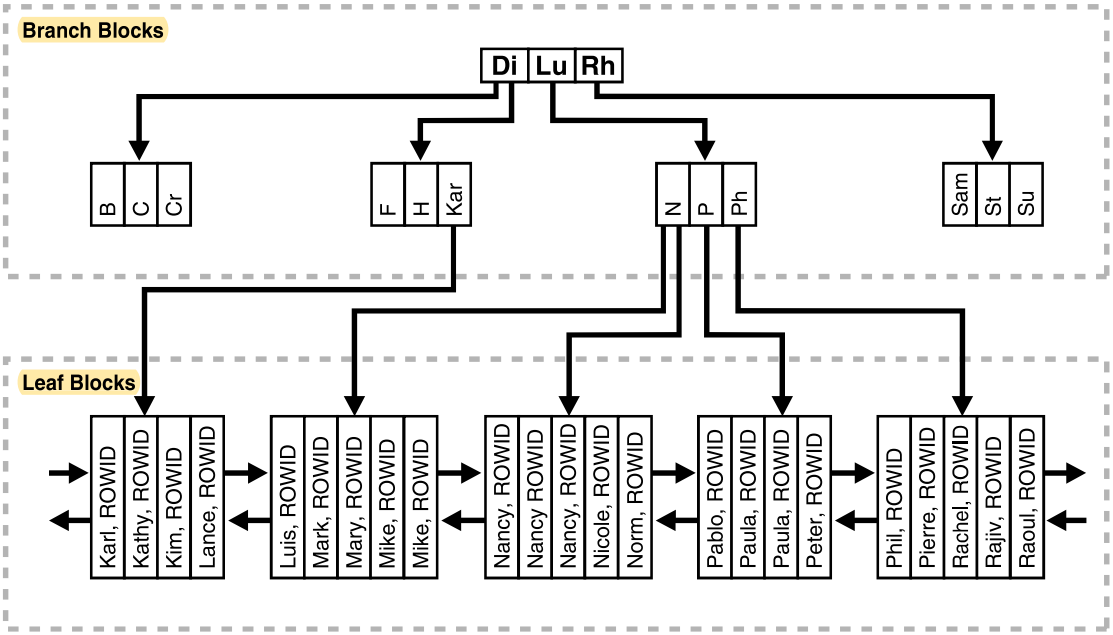
The Internal Structure of Indexes

Oracle uses B-trees to store indexes to speed up data access. With no indexes, you have to do a sequential scan on the data to find a value. For n rows, the average number of rows searched is $n/2$. This does not scale very well as data volumes increase.

Consider an ordered list of the values divided into block-wide ranges (leaf blocks). The end points of the ranges along with pointers to the blocks can be stored in a search tree and a value in $\log(n)$ time for n entries could be found. This is the basic principle behind Oracle indexes.

Figure 5–7 illustrates the structure of a B-tree index.

Figure 5–7 Internal Structure of a B-tree Index



The upper blocks (**branch blocks**) of a B-tree index contain index data that points to lower-level index blocks. The lowest level index blocks (**leaf blocks**) contain every indexed data value and a corresponding rowid used to locate the actual row. The leaf blocks are doubly linked. Indexes in columns containing character data are based on the binary values of the characters in the database character set.

For a unique index, one rowid exists for each data value. 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. Key values containing all nulls are not indexed, except for cluster indexes. Two rows can both contain all nulls without violating a unique index.

Index Properties

The two kinds of blocks:

- Branch blocks for searching
- Leaf blocks that store the values

Branch Blocks Branch blocks store the following:

- The minimum key prefix needed to make a branching decision between two keys
- The pointer to the child block containing the key

If the blocks have *n* keys then they have *n*+1 pointers. The number of keys and pointers is limited by the block size.

Leaf Blocks All leaf blocks are at the same depth from the root branch block. Leaf blocks store the following:

- The complete key value for every row
- `ROWID`s of the table rows

All key and `ROWID` pairs are linked to their left and right siblings. They are sorted by (key, `ROWID`).

Advantages of B-tree Structure

The B-tree structure has the following advantages:

- All leaf blocks of the tree are at the same depth, so retrieval of any record from anywhere in the index takes approximately the same amount of time.
- B-tree indexes automatically stay balanced.
- All blocks of the B-tree are three-quarters full on the average.
- B-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches.
- Inserts, updates, and deletes are efficient, maintaining key order for fast retrieval.
- B-tree performance is good for both small and large tables and does not degrade as the size of a table grows.

> **See Also:** Computer science texts for more information about B-tree indexes

Index Unique Scan

Index unique scan is one of the most efficient ways of accessing data. This access method is used for returning the data from B-tree indexes. The optimizer chooses a unique scan when all columns of a unique (B-tree) index are specified with equality conditions.

Index Range Scan

Index range scan is a common operation for accessing selective data. It can be bounded (bounded on both sides) or unbounded (on one or both sides). Data is returned in the ascending order of index columns. Multiple rows with identical values are sorted (in ascending order) by the `ROWID`s.

Key Compression

Key compression lets you compress portions of the primary key column values in an index or index-organized table, which reduces the storage overhead of repeated values.

Generally, keys in an index have two pieces, a grouping piece and a unique piece. If the key is not defined to have a unique piece, Oracle provides one in the form of a rowid appended to the grouping piece. Key compression is a method of breaking off the grouping piece and storing it so it can be shared by multiple unique pieces.

Prefix and Suffix Entries

Key compression breaks the index key into a prefix entry (the grouping piece) and a suffix entry (the unique piece). Compression is achieved by sharing the prefix entries among the suffix entries in an index block. 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.

Key compression is done within an index block but not across multiple index blocks. Suffix entries 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 entry.

By default, the prefix consists of all key columns excluding the last one. For example, in a key made up of three columns (column1, column2, column3) the default prefix is

(column1, column2). For a list of values (1,2,3), (1,2,4), (1,2,7), (1,3,5), (1,3,4), (1,4,4) the repeated occurrences of (1,2), (1,3) in the prefix are compressed.

Alternatively, you can specify the prefix length, which is the number of columns in the prefix. For example, if you specify prefix length 1, then the prefix is column1 and the suffix is (column2, column3). For the list of values (1,2,3), (1,2,4), (1,2,7), (1,3,5), (1,3,4), (1,4,4) the repeated occurrences of 1 in the prefix are compressed.

The maximum prefix length for a nonunique index is the number of key columns, and the maximum prefix length for a unique index is the number of key columns minus one.

Prefix entries are written to the index block only if the index block does not already contain a prefix entry whose value is equal to the present prefix entry. Prefix entries are available for sharing immediately after being written to the index block and remain available until the last deleted referencing suffix entry is cleaned out of the index block.

Performance and Storage Considerations

Key compression can lead to a huge saving in space, letting you store more keys in each index block, which can lead to less I/O and better performance.

Although key compression reduces the storage requirements of an index, it can increase the CPU time required to reconstruct the key column values during an index scan. It also incurs some additional storage overhead, because every prefix entry has an overhead of 4 bytes associated with it.

Uses of Key Compression

Key compression is useful in many different scenarios, such as:

- In a nonunique regular index, Oracle stores duplicate keys with the rowid appended to the key to break the duplicate rows. If key compression is used, Oracle stores the duplicate key as a prefix entry on the index block without the rowid. The rest of the rows are suffix entries that consist of only the rowid.
- This same behavior can be seen in a unique index that has a key of the form (**item**, **time stamp**), for example (`stock_ticker`, `transaction_time`). Thousands of rows can have the same `stock_ticker` value, with `transaction_time` preserving uniqueness. On a particular index block a `stock_ticker` value is stored only once as a prefix entry. Other entries on the index block are `transaction_time` values stored as suffix entries that reference the common `stock_ticker` prefix entry.
- In an index-organized table that contains a `VARRAY` or `NESTED TABLE` datatype, the object identifier is repeated for each element of the collection datatype. Key compression lets you compress the repeating object identifier values.

In some cases, however, key compression cannot be used. For example, in a unique index with a single attribute key, key compression is not possible, because even though there is a unique piece, there are no grouping pieces to share.

See Also: "Overview of Index-Organized Tables" on page 5-34

Reverse Key Indexes

Creating a **reverse key index**, compared to a standard index, reverses the bytes of each column indexed (except the rowid) while keeping the column order. Such an arrangement can help avoid performance degradation with Real Application Clusters

where modifications to the index are concentrated on a small set of leaf blocks. By reversing the keys of the index, the insertions become distributed across all leaf keys in the index.

Using the reverse key arrangement eliminates the ability to run an index range scanning query on the index. Because lexically adjacent keys are not stored next to each other in a reverse-key index, only fetch-by-key or full-index (table) scans can be performed.

Sometimes, using a reverse-key index can make an OLTP Real Application Clusters application faster. For example, keeping the index of mail messages in an e-mail application: some users keep old messages, and the index must maintain pointers to these as well as to the most recent.

The `REVERSE` keyword provides a simple mechanism for creating a reverse key index. You can specify the keyword `REVERSE` along with the optional index specifications in a `CREATE INDEX` statement:

```
CREATE INDEX i ON t (a,b,c) REVERSE;
```

You can specify the keyword `NOREVERSE` to `REBUILD` a reverse-key index into one that is not reverse keyed:

```
ALTER INDEX i REBUILD NOREVERSE;
```

Rebuilding a reverse-key index without the `NOREVERSE` keyword produces a rebuilt, reverse-key index.

Bitmap Indexes

The purpose of an index is to provide pointers to the rows in a table that contain a given key value. In a regular index, this is achieved by storing a list of rowids for each key corresponding to the rows with that key value. Oracle stores each key value repeatedly with each stored rowid. In a **bitmap index**, a bitmap for each key value is used instead of a list of rowids.

Each bit in the bitmap corresponds to a possible rowid. If the bit is set, then it means that 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 regular index even though it uses a different representation internally. If the number of different key values is small, then bitmap indexes are very space efficient.

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 improves response time, often dramatically.

Benefits for Data Warehousing Applications

Bitmap indexing benefits data warehousing applications which have large amounts of data and ad hoc queries but a low level of concurrent transactions. For such applications, bitmap indexing provides:

- Reduced response time for large classes of ad hoc queries
- A substantial reduction of space use compared to other indexing techniques
- Dramatic performance gains even on very low end hardware
- Very efficient parallel DML and loads

Fully indexing a large table with a traditional B-tree index can be prohibitively expensive in terms of space, because the index can be several times larger than the data in the table. Bitmap indexes are typically only a fraction of the size of the indexed data in the table.

Bitmap indexes are not suitable for OLTP applications with large numbers of concurrent transactions modifying the data. These indexes are primarily intended for decision support in data warehousing applications where users typically query the data rather than update it.

Bitmap indexes are also not suitable for columns that are primarily queried with less than or greater than comparisons. For example, a salary column that usually appears in `WHERE` clauses in a comparison to a certain value is better served with a B-tree index. Bitmapped indexes are only useful with equality queries, especially in combination with `AND`, `OR`, and `NOT` operators.

Bitmap indexes are integrated with the Oracle optimizer and execution engine. They can be used seamlessly in combination with other Oracle execution methods. For example, the optimizer can decide to perform a hash join between two tables using a bitmap index on one table and a regular B-tree index on the other. The optimizer considers bitmap indexes and other available access methods, such as regular B-tree indexes and full table scan, and chooses the most efficient method, taking parallelism into account where appropriate.

Parallel query and parallel DML work with bitmap indexes as with traditional indexes. Bitmap indexes on partitioned tables must be local indexes. Parallel create index and concatenated indexes are also supported.

Cardinality

The advantages of using bitmap indexes are greatest for low cardinality columns: that is, columns in which the number of distinct values is small compared to the number of rows in the table. If the number of distinct values of a column is less than 1% of the number of rows in the table, or if the values in a column are repeated more than 100 times, then the column is a candidate for a bitmap index. Even columns with a lower number of repetitions and thus higher cardinality can be candidates if they tend to be involved in complex conditions in the `WHERE` clauses of queries.

For example, on a table with 1 million rows, a column with 10,000 distinct values is a candidate for a bitmap index. A bitmap index on this column can out-perform a B-tree index, particularly when this column is often queried in conjunction with other columns.

B-tree indexes are most effective for high-cardinality data: that is, data with many possible values, such as `CUSTOMER_NAME` or `PHONE_NUMBER`. In some situations, a B-tree index can be larger than the indexed data. Used appropriately, bitmap indexes can be significantly smaller than a corresponding B-tree index.

In ad hoc queries and similar situations, bitmap indexes can dramatically improve query performance. `AND` and `OR` conditions in the `WHERE` clause of a query can be quickly resolved by performing the corresponding Boolean operations directly on the bitmaps before converting the resulting bitmap to rowids. If the resulting number of rows is small, the query can be answered very quickly without resorting to a full table scan of the table.

Bitmap Index Example

Table 5–1 shows a portion of a company's customer data.

Table 5–1 Bitmap Index Example

CUSTOMER #	MARITAL_ STATUS	REGION	GENDER	INCOME_ LEVEL
101	single	east	male	bracket_1
102	married	central	female	bracket_4
103	married	west	female	bracket_2
104	divorced	west	male	bracket_4
105	single	central	female	bracket_2
106	married	central	female	bracket_3

`MARITAL_STATUS`, `REGION`, `GENDER`, and `INCOME_LEVEL` are all low-cardinality columns. There are only three possible values for marital status and region, two possible values for gender, and four for income level. Therefore, it is appropriate to create bitmap indexes on these columns. A bitmap index should not be created on `CUSTOMER#` because this is a high-cardinality column. Instead, use a unique B-tree index on this column to provide the most efficient representation and retrieval.

Table 5–2 illustrates the bitmap index for the `REGION` column in this example. It consists of three separate bitmaps, one for each region.

Table 5–2 Sample Bitmap

REGION='east'	REGION='central'	REGION='west'
1	0	0
0	1	0
0	0	1
0	0	1
0	1	0
0	1	0

Each entry or bit in the bitmap corresponds to a single row of the `CUSTOMER` table. The value of each bit depends upon the values of the corresponding row in the table. For instance, the bitmap `REGION='east'` contains a one as its first bit. This is because the region is east in the first row of the `CUSTOMER` table. The bitmap `REGION='east'` has a zero for its other bits because none of the other rows of the table contain `east` as their value for `REGION`.

An analyst investigating demographic trends of the company's customers can ask, "How many of our married customers live in the central or west regions?" This corresponds to the following SQL query:

```
SELECT COUNT(*) FROM CUSTOMER
    WHERE MARITAL_STATUS = 'married' AND REGION IN ('central','west');
```

Bitmap indexes can process this query with great efficiency by counting the number of ones in the resulting bitmap, as illustrated in Figure 5–8. To identify the specific customers who satisfy the criteria, the resulting bitmap can be used to access the table.

Figure 5–8 Running a Query Using Bitmap Indexes

status = 'married'		region = 'central'		region = 'west'						
0		0		0		0		0		0
1		1		0		1		1		1
1	AND	0	OR	1	=	1	AND	1	=	1
0		0		1		0		1		0
0		1		0		0		1		0
1		1		0		1		1		1

Bitmap Indexes and Nulls

Bitmap indexes can include rows that have `NULL` values, unlike most other types of indexes. Indexing of nulls can be useful for some types of SQL statements, such as queries with the aggregate function `COUNT`.

Bitmap Indexes on Partitioned Tables

Like other indexes, you can create bitmap indexes on partitioned tables. The only restriction is that bitmap indexes must be local to the partitioned table—they cannot be global indexes. Global bitmap indexes are supported only on nonpartitioned tables.

See Also:

- Chapter 18, "Partitioned Tables and Indexes" for information about partitioned tables and descriptions of local and global indexes
- *Oracle Database Performance Tuning Guide* for more information about using bitmap indexes, including an example of indexing null values

Bitmap Join Indexes

In addition to a bitmap index on a single table, you can create a bitmap join index, which is a bitmap index for the join of two or more tables. A bitmap join index is a space efficient way of reducing the volume of data that must be joined by performing restrictions in advance. For each value in a column of a table, a bitmap join index stores the rowids of corresponding rows in one or more other tables. In a data warehousing environment, the join condition is an equi-inner join between the primary key column or columns of the dimension tables and the foreign key column or columns in the fact table.

Bitmap join indexes are much more efficient in storage than materialized join views, an alternative for materializing joins in advance. This is because the materialized join views do not compress the rowids of the fact tables.

See Also: *Oracle Database Data Warehousing Guide* for more information on bitmap join indexes

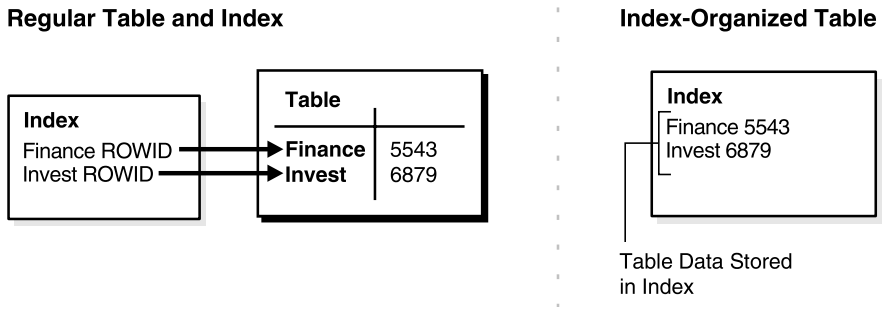
Overview of Index-Organized Tables

An **index-organized table** has a storage organization that is a variant of a primary B-tree. Unlike an ordinary (heap-organized) table whose data is stored as an

unordered collection (heap), data for an index-organized table is stored in a B-tree index structure in a primary key sorted manner. Besides storing the primary key column values of an index-organized table row, each index entry in the B-tree stores the nonkey column values as well.

As shown in Figure 5–9, the index-organized table is somewhat similar to a configuration consisting of an ordinary table and an index on one or more of the table columns, but instead of maintaining two separate storage structures, one for the table and one for the B-tree index, the database system maintains only a single B-tree index. Also, rather than having a row's rowid stored in the index entry, the nonkey column values are stored. Thus, each B-tree index entry contains `<primary_key_value, non_primary_key_column_values>`.

Figure 5–9 Structure of a Regular Table Compared with an Index-Organized Table



Applications manipulate the index-organized table just like an ordinary table, using SQL statements. However, the database system performs all operations by manipulating the corresponding B-tree index.

Table 5–3 summarizes the differences between index-organized tables and ordinary tables.

Table 5–3 Comparison of Index-Organized Tables with Ordinary Tables

Ordinary Table	Index-Organized Table
Rowid uniquely identifies a row. Primary key can be optionally specified	Primary key uniquely identifies a row. Primary key must be specified
Physical rowid in `ROWID` pseudocolumn allows building secondary indexes	Logical rowid in `ROWID` pseudocolumn allows building secondary indexes
Access is based on rowid	Access is based on logical rowid
Sequential scan returns all rows	Full-index scan returns all rows
Can be stored in a cluster with other tables	Cannot be stored in a cluster
Can contain a column of the `LONG` datatype and columns of LOB datatypes	Can contain LOB columns but not `LONG` columns

Benefits of Index-Organized Tables

Index-organized tables provide faster access to table rows by the primary key or any key that is a valid prefix of the primary key. Presence of nonkey columns of a row in the B-tree leaf block itself avoids an additional block access. Also, because rows are stored in primary key order, range access by the primary key (or a valid prefix) involves minimum block accesses.

In order to allow even faster access to frequently accessed columns, you can use a row overflow segment (as described later) to push out infrequently accessed nonkey

columns from the B-tree leaf block to an optional (heap-organized) overflow segment. This allows limiting the size and content of the portion of a row that is actually stored in the B-tree leaf block, which may lead to a higher number of rows in each leaf block and a smaller B-tree.

Unlike a configuration of heap-organized table with a primary key index where primary key columns are stored both in the table and in the index, there is no such duplication here because primary key column values are stored only in the B-tree index.

Because rows are stored in primary key order, a significant amount of additional storage space savings can be obtained through the use of key compression.

Use of primary-key based logical rowids, as opposed to physical rowids, in secondary indexes on index-organized tables allows high availability. This is because, due to the logical nature of the rowids, secondary indexes do not become unusable even after a table reorganization operation that causes movement of the base table rows. At the same time, through the use of physical guess in the logical rowid, it is possible to get secondary index based index-organized table access performance that is comparable to performance for secondary index based access to an ordinary table.

See Also:

- "Key Compression" on page 5-28
- "Secondary Indexes on Index-Organized Tables" on page 5-36
- *Oracle Database Administrator's Guide* for information about creating and maintaining index-organized tables

Index-Organized Tables with Row Overflow Area

B-tree index entries are usually quite small, because they only consist of the key value and a `ROWID`. In index-organized tables, however, the B-tree index entries can be large, because they consist of the entire row. This may destroy the dense clustering property of the B-tree index.

Oracle provides the `OVERFLOW` clause to handle this problem. You can specify an overflow tablespace so that, if necessary, a row can be divided into the following two parts that are then stored in the index and in the overflow storage area segment, respectively:

- The index entry, containing 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 nonkey columns
- The overflow part, containing column values for the remaining nonkey columns

With `OVERFLOW`, you can use two clauses, `PCTTHRESHOLD` and `INCLUDING`, to control how Oracle determines whether a row should be stored in two parts and if so, at which nonkey column to break the row. Using `PCTTHRESHOLD`, you can specify a threshold value as a percentage of the block size. If all the nonkey column values can be accommodated within the specified size limit, the row will not be broken into two parts. Otherwise, starting with the first nonkey column that cannot be accommodated, the rest of the nonkey columns are all stored in the row overflow segment for the table.

The `INCLUDING` clause lets you specify a column name so that any nonkey column, appearing in the `CREATE TABLE` statement after that specified column, is stored in the row overflow segment. Note that additional nonkey columns may sometimes need to be stored in the overflow due to `PCTTHRESHOLD`-based limits.

See Also: *Oracle Database Administrator's Guide* for examples of using the `OVERFLOW` clause

Secondary Indexes on Index-Organized Tables

Secondary index support on index-organized tables provides efficient access to index-organized table using columns that are not the primary key nor a prefix of the primary key.

Oracle constructs secondary indexes on index-organized tables using logical row identifiers (**logical rowids**) that are based on the table's primary key. A logical rowid includes a **physical guess**, which identifies the block location of the row. Oracle can use these physical guesses to probe directly into the leaf block of the index-organized table, bypassing the primary key search. Because rows in index-organized tables do not have permanent physical addresses, the physical guesses can become stale when rows are moved to new blocks.

For an ordinary 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 secondary index scan followed by a scan of the primary key index.
- 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 physical guess), followed by a scan of the primary key index.

See Also: "Logical Rowids" on page 26-16

Bitmap Indexes on Index-Organized Tables

Oracle supports bitmap indexes on partitioned and nonpartitioned index-organized tables. A mapping table is required for creating bitmap indexes on an index-organized table.

Mapping Table

The mapping table is a heap-organized table that stores logical rowids of the index-organized table. Specifically, each mapping table row stores one logical rowid for the corresponding index-organized table row. Thus, the mapping table provides one-to-one mapping between logical rowids of the index-organized table rows and physical rowids of the mapping table rows.

A bitmap index on an index-organized table is similar to that on a heap-organized table except that the rowids used in the bitmap index on an index-organized table are those of the mapping table as opposed to the base table. There is one mapping table for each index-organized table and it is used by all the bitmap indexes created on that index-organized table.

In both heap-organized and index-organized base tables, a bitmap index is accessed using a search key. If the key is found, the bitmap entry is converted to a physical rowid. In the case of heap-organized tables, this physical rowid is then used to access the base table. However, in the case of index-organized tables, the physical rowid is then used to access the mapping table. The access to the mapping table yields a logical rowid. This logical rowid is used to access the index-organized table.

Though a bitmap index on an index-organized table does not store logical rowids, it is still logical in nature.

> **Note:** Movement of rows in an index-organized table does not leave the bitmap indexes built on that index-organized table unusable. Movement of rows in the index-organized table does invalidate the physical guess in some of the mapping table's logical rowid entries. However, the index-organized table can still be accessed using the primary key.

Partitioned Index-Organized Tables

You can partition an index-organized table by `RANGE`, `HASH`, or `LIST` on column values. The partitioning columns must form a subset of the primary key columns. Just like ordinary tables, local partitioned (prefixed and non-prefixed) index as well as global partitioned (prefixed) indexes are supported for partitioned index-organized tables.

B-tree Indexes on UROWID Columns for Heap- and Index-Organized Tables

`UROWID` datatype columns can hold logical primary key-based rowids identifying rows of index-organized tables. Oracle supports indexes on `UROWID` datatypes of a heap- or index-organized table. The index supports equality predicates on `UROWID` columns. For predicates other than equality or for ordering on `UROWID` datatype columns, the index is not used.

Index-Organized Table Applications

The superior query performance for primary key based access, high availability aspects, and reduced storage requirements make index-organized tables ideal for the following kinds of applications:

- Online transaction processing (OLTP)
- Internet (for example, search engines and portals)
- E-commerce (for example, electronic stores and catalogs)
- Data warehousing
- Analytic functions

Overview of Application Domain Indexes

Oracle provides **extensible indexing** to accommodate indexes on customized complex datatypes such as documents, spatial data, images, and video clips and to make use of specialized indexing techniques. With extensible indexing, you can encapsulate application-specific index management routines as an **indextype** schema object and define a **domain index** (an application-specific index) on table columns or attributes of an object type. Extensible indexing also provides efficient processing of application-specific **operators**.

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

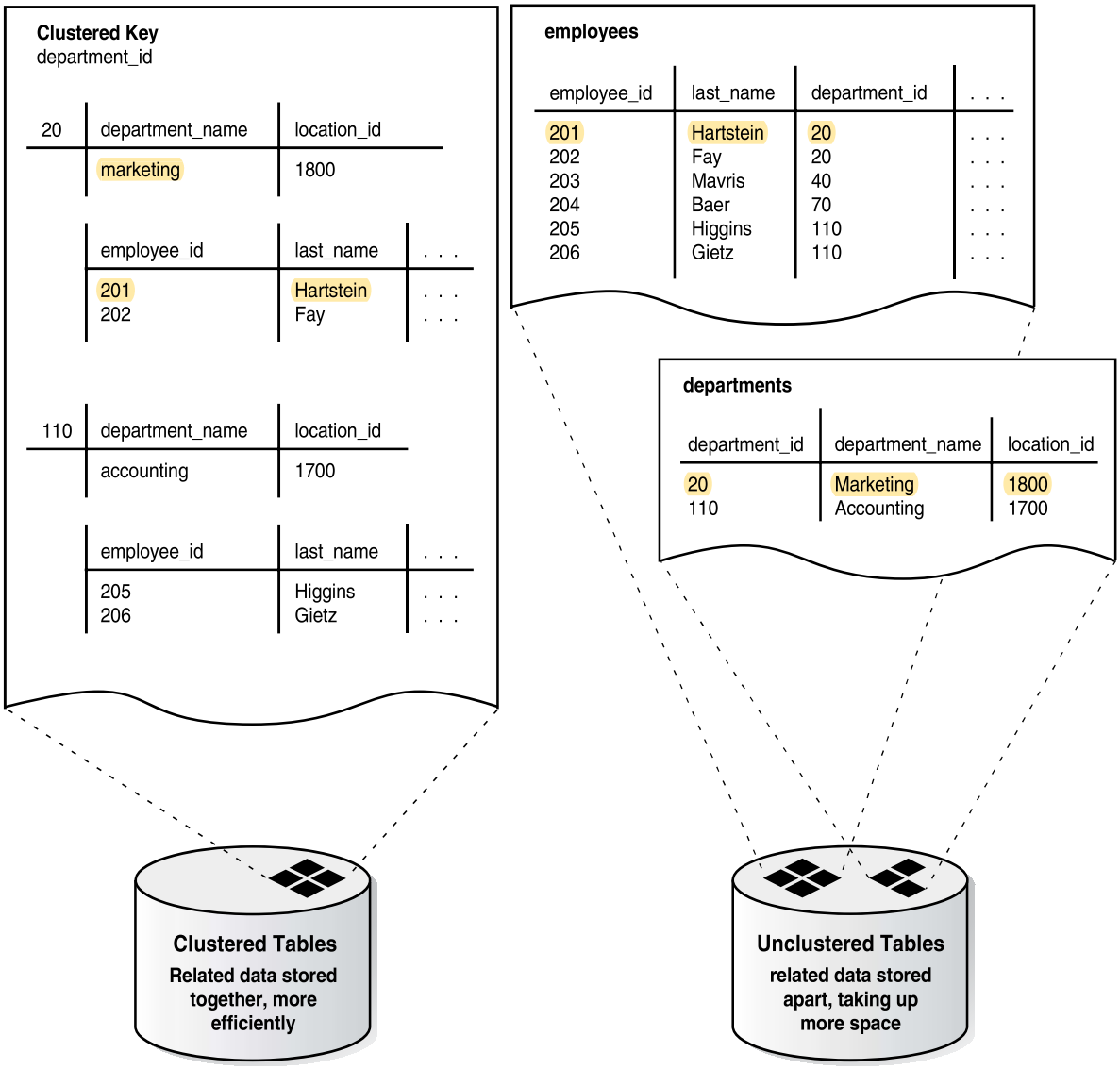
See Also: *Oracle Database Data Cartridge Developer's Guide* for information about using data cartridges within Oracle's extensibility architecture

Overview of Clusters

Clusters are an optional method of storing table data. A cluster is a group of tables that share the same data blocks because they share common columns and are often used together. For example, the `employees` and `departments` table share the `department_id` column. When you cluster the `employees` and `departments` tables, Oracle physically stores all rows for each department from both the `employees` and `departments` tables in the same data blocks.

Figure 5–10 shows what happens when you cluster the `employees` and `departments` tables:

Figure 5–10 Clustered Table Data



Because clusters store related rows of different tables together in the same data blocks, properly used clusters offers these benefits:

- Disk I/O is reduced for joins of clustered tables.
- Access time improves for joins of clustered tables.
- In a cluster, a **cluster key value** is the value of the cluster key columns for a particular row. Each cluster key value is stored only once each in the cluster and the cluster index, no matter how many rows of different tables contain the value. Therefore, less storage is required to store related table and index data in a cluster than is necessary in nonclustered table format. For example, in Figure 5–10, notice how each cluster key (each `department_id`) is stored just once for many rows that contain the same value in both the `employees` and `departments` tables.

> **See Also:** *Oracle Database Administrator's Guide* for information about creating and managing clusters

Overview of Hash Clusters

Hash clusters group table data in a manner similar to regular index clusters (clusters keyed with an index rather than a hash function). However, a row is stored in a hash cluster based on the result of applying a **hash function** to the row's cluster key value. All rows with the same key value are stored together on disk.

Hash clusters are a better choice than using an indexed table or index cluster when a table is queried frequently with equality queries (for example, return all rows for department 10). For such queries, the specified cluster key value is hashed. The resulting hash key value points directly to the area on disk that stores the rows.

Hashing is an optional way of storing table data to improve the performance of data retrieval. To use hashing, create a hash cluster and load tables into the cluster. Oracle physically stores the rows of a table in a hash cluster and retrieves them according to the results of a hash function.

Sorted hash clusters allow faster retrieval of data for applications where data is consumed in the order in which it was inserted.

Oracle uses a hash function to generate a distribution of numeric values, called **hash values**, which are based on specific cluster key values. The key of a hash cluster, like the key of an index cluster, can be a single column or composite key (multiple column key). To find or store a row in a hash cluster, Oracle applies the hash function to the row's cluster key value. The resulting hash value corresponds to a data block in the cluster, which Oracle then reads or writes on behalf of the issued statement.

A hash cluster is an alternative to a nonclustered table with an index or an index cluster. With an indexed table or index cluster, Oracle locates the rows in a table using key values that Oracle stores in a separate index. To find or store a row in an indexed table or cluster, at least two I/Os must be performed:

- One or more I/Os to find or store the key value in the index
- Another I/O to read or write the row in the table or cluster

> **See Also:** *Oracle Database Administrator's Guide* for information about creating and managing hash clusters