Part IV

Data

Part IV describes the data involved in database management.

Part IV contains the following chapters:

10

Schema Objects

This chapter discusses the different types of database objects contained in a user's schema. It includes:

- Introduction to Schema Objects
- Tables
- Views
- Materialized Views
- Dimensions
- The Sequence Generator
- Synonyms
- Indexes
- Index-Organized Tables
- Application Domain Indexes
- Clusters
- Hash Clusters

Introduction to Schema Objects

A **schema** is a collection of logical structures of data, or schema objects. A schema is owned by a database user and has the same name as that user. Each user owns a single schema. Schema objects can be created and manipulated with SQL and include the following types of objects:

- Clusters
- Database links
- Database triggers
- Dimensions
- External procedure libraries
- Indexes and index types
- Java classes, Java resources, and Java sources
- Materialized views and materialized view logs
- Object tables, object types, and object views
- Operators
- Sequences
- Stored functions, procedures, and packages
- Synonyms
- Tables and index-organized tables
- Views

Other types of objects are also stored in the database and can be created and manipulated with SQL but are not contained in a schema:

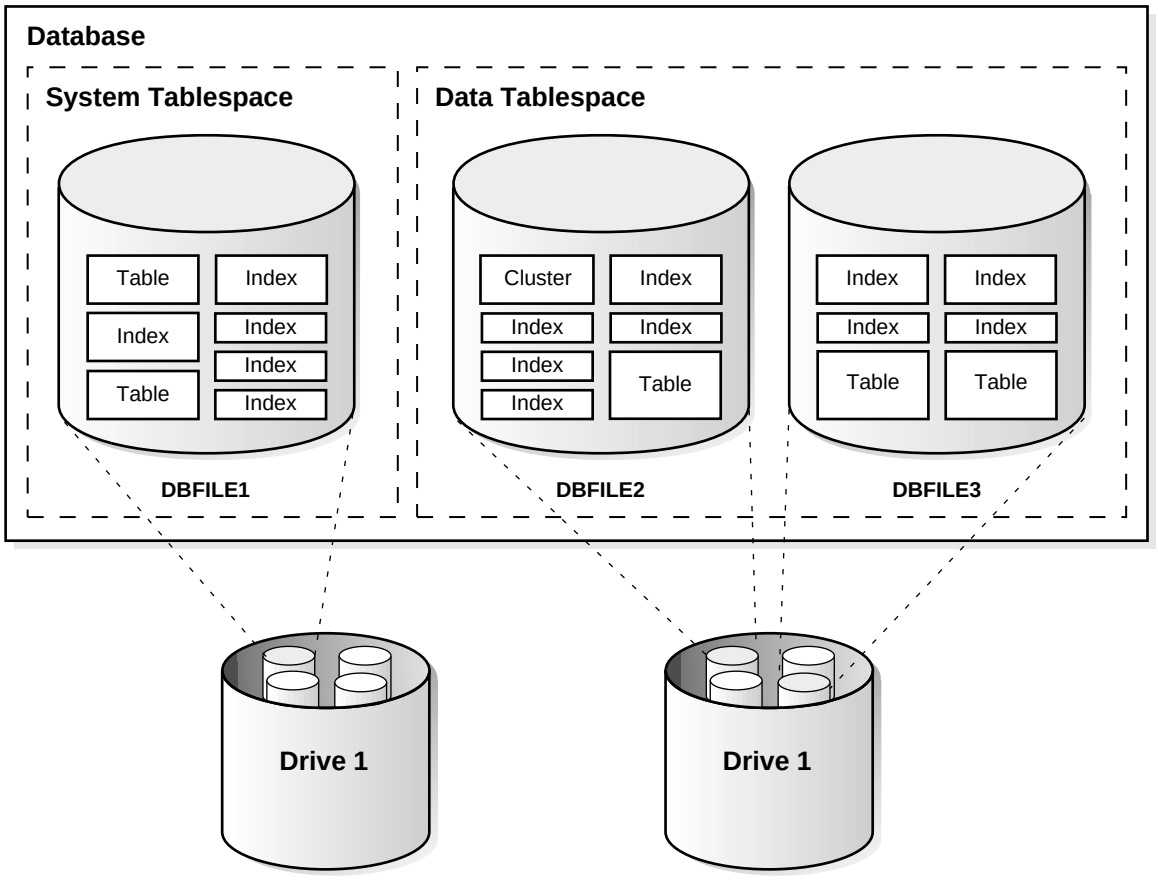
- Contexts
- Directories
- Profiles
- Roles
- Tablespaces
- Users
- Rollback segments

Schema objects are logical data storage structures. Schema objects do not have a one-to-one correspondence to physical files on disk that store their information. However, Oracle stores a schema object logically within a tablespace of the database. The data of each object is physically contained in one or more of the tablespace's datafiles. For some objects, such as tables, indexes, and clusters, you can specify how much disk space Oracle allocates for the object within the tablespace's datafiles.

There is no relationship between schemas and tablespaces: a tablespace can contain objects from different schemas, and the objects for a schema can be contained in different tablespaces.

Figure 10–1 illustrates the relationship among objects, tablespaces, and datafiles.

Figure 10–1 Schema Objects, Tablespaces, and Datafiles



See Also:

- *Oracle9i Database Administrator's Guide*
- "Stored Procedures and Functions" on page 14-21
- Chapter 14, "SQL, PL/SQL, and Java"
- Chapter 17, "Triggers"

Tables

Tables are the basic unit of data storage in an Oracle database. Data is stored in **rows** and **columns**. You define a table with a **table name** (such as `employees`) and set of columns. You give each column a **column name** (such as `employee_id`, `last_name`, and `job_id`), a **datatype** (such as VARCHAR2, DATE, or NUMBER), and a **width**. The width can be predetermined by the datatype, as in DATE. If columns are of the NUMBER datatype, define **precision** and **scale** instead of width. A row is a collection of column information corresponding to a single record.

You can specify rules for each column of a table. These rules are called **integrity constraints**. One example is a NOT NULL integrity constraint. This constraint forces the column to contain a value in every row.

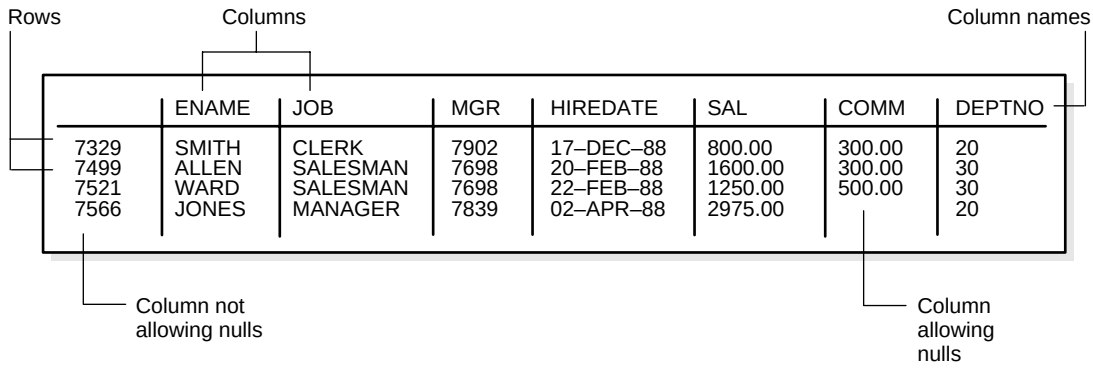
After you create a table, insert rows of data using SQL statements. Table data can then be queried, deleted, or updated using SQL.

Figure 10–2 shows a sample table named `emp`.

See Also:

- Chapter 12, "Native Datatypes" for a discussion of the Oracle datatypes
- Chapter 21, "Data Integrity" for more information about integrity constraints

Figure 10–2 The EMP Table



How Table Data Is Stored

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

- You can control the amount of space allocated to the data segment by setting the storage parameters for the data segment.
- You can control the use of the free space in the data blocks that constitute the data segment's extents by setting the PCTFREE and PCTUSED parameters for the data segment.

Oracle stores data for a clustered table in the data segment created for the cluster instead of in a data segment in a tablespace. Storage parameters cannot be specified when a clustered table is created or altered. The storage parameters set for the cluster always control the storage of all tables in the cluster.

The tablespace that contains a nonclustered table's data segment is either the table owner's default tablespace or a tablespace specifically named in the CREATE TABLE statement.

See Also: "User Tablespace Settings and Quotas" on page 22-14

Row Format and Size

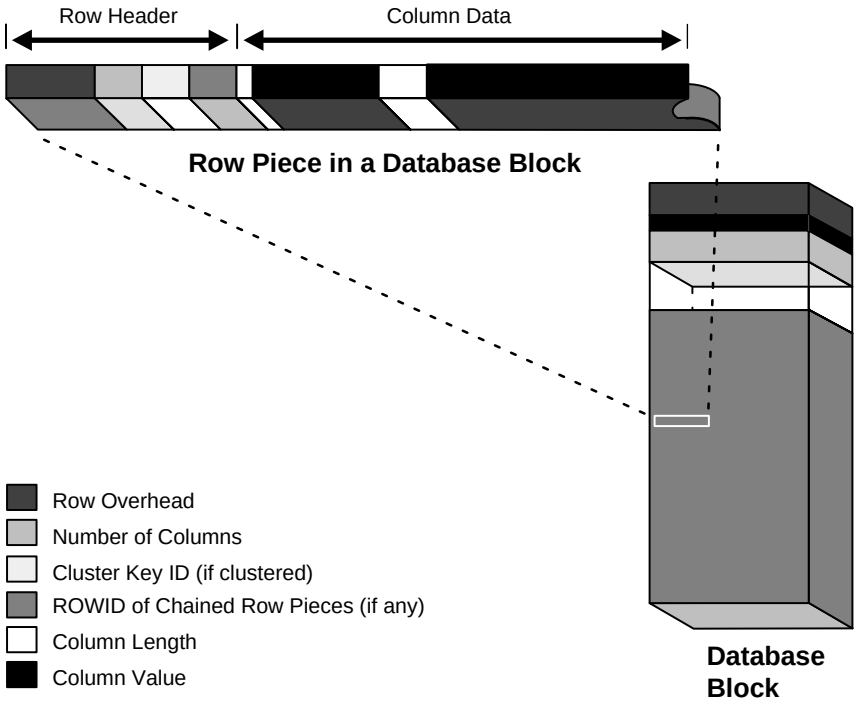
Oracle stores each row of a database table containing data for less than 256 columns as one or more row pieces. If an entire row can be inserted into a single data block, then Oracle stores the row as one row piece. However, if all of a row's data cannot

be inserted into a single data block or an update to an existing row causes the row to outgrow its data block, Oracle stores the row using multiple row pieces. A data block usually contains only one row piece for each row. When Oracle must store a row in more than one row piece, it is **chained** across multiple blocks.

When a table has more than 255 columns, rows that have data after the 255th column are likely to be chained within the same block. This is called **intra-block chaining**. A chained row's pieces are chained together using the rowids of the pieces. With intra-block chaining, users receive all the data in the same block. If the row fits in the block, users do not see an effect in I/O performance, because no extra I/O operation is required to retrieve the rest of the row.

Each row piece, chained or unchained, contains a **row header** and data for all or some of the row's columns. Individual columns can also span row pieces and, consequently, data blocks. Figure 10–3 shows the format of a row piece:

Figure 10–3 The Format of a Row Piece



The **row header** precedes the data and contains information about:

- Row pieces
- Chaining (for chained row pieces only)
- Columns in the row piece
- Cluster keys (for clustered data only)

A row fully contained in one block has at least 3 bytes of row header. After the row header information, each row contains column length and data. The column length requires 1 byte for columns that store 250 bytes or less, or 3 bytes for columns that store more than 250 bytes, and precedes the column data. Space required for column data depends on the datatype. If the datatype of a column is variable length, then the space required to hold a value can grow and shrink with updates to the data.

To conserve space, a null in a column only stores the column length (zero). Oracle does not store data for the null column. Also, for trailing null columns, Oracle does not even store the column length.

Note: Each row also uses 2 bytes in the data block header's row directory.

Clustered rows contain the same information as nonclustered rows. In addition, they contain information that references the cluster key to which they belong.

See Also:

- *Oracle9i Database Administrator's Guide* for more information about clustered rows and tables
- "Clusters" on page 10-63
- "Row Chaining and Migrating" on page 2-7
- "Nulls Indicate Absence of Value" on page 10-10
- "Row Directory" on page 2-5

Rowids of Row Pieces

The **rowid** identifies each row piece by its location or address. After they are assigned, a given row piece retains its rowid until the corresponding row is deleted or exported and imported using the Export and Import utilities. For clustered tables, if the cluster key values of a row change, then the row keeps the same rowid but also gets an additional pointer rowid for the new values.

Because rowids are constant for the lifetime of a row piece, it is useful to reference rowids in SQL statements such as SELECT, UPDATE, and DELETE.

See Also:

- "Clusters" on page 10-63
- "Physical Rowids" on page 12-17

Column Order

The column order is the same for all rows in a given table. Columns are usually stored in the order in which they were listed in the CREATE TABLE statement, but this is not guaranteed. For example, if you create a table with a column of datatype

LONG, then Oracle always stores this column last. Also, if a table is altered so that a new column is added, then the new column becomes the last column stored.

In general, try to place columns that frequently contain nulls last so that rows take less space. Note, though, that if the table you are creating includes a LONG column as well, then the benefits of placing frequently null columns last are lost.

Nulls Indicate Absence of Value

A **null** is the absence of a value in a column of a row. Nulls indicate missing, unknown, or inapplicable data. A null should not be used to imply any other value, such as zero. A column allows nulls unless a NOT NULL or PRIMARY KEY integrity constraint has been defined for the column, in which case no row can be inserted without a value for that column.

Nulls are stored in the database if they fall between columns with data values. In these cases they require 1 byte to store the length of the column (zero).

Trailing nulls in a row require no storage because a new row header signals that the remaining columns in the previous row are null. For example, if the last three columns of a table are null, no information is stored for those columns. In tables with many columns, the columns more likely to contain nulls should be defined last to conserve disk space.

Most comparisons between nulls and other values are by definition neither true nor false, but unknown. To identify nulls in SQL, use the IS NULL predicate. Use the SQL function NVL to convert nulls to non-null values.

Nulls are not indexed, except when the cluster key column value is null or the index is a bitmap index.

See Also:

- *Oracle9i SQL Reference* for more information about comparisons using IS NULL and the NVL function
- "Indexes and Nulls" on page 10-31
- "Bitmap Indexes and Nulls" on page 10-52

Default Values for Columns

You can assign a default value to a column of a table so that when a new row is inserted and a value for the column is omitted or keyword DEFAULT is supplied, a default value is supplied automatically. Default column values work as though an INSERT statement actually specifies the default value.

The datatype of the default literal or expression must match or be convertible to the column datatype.

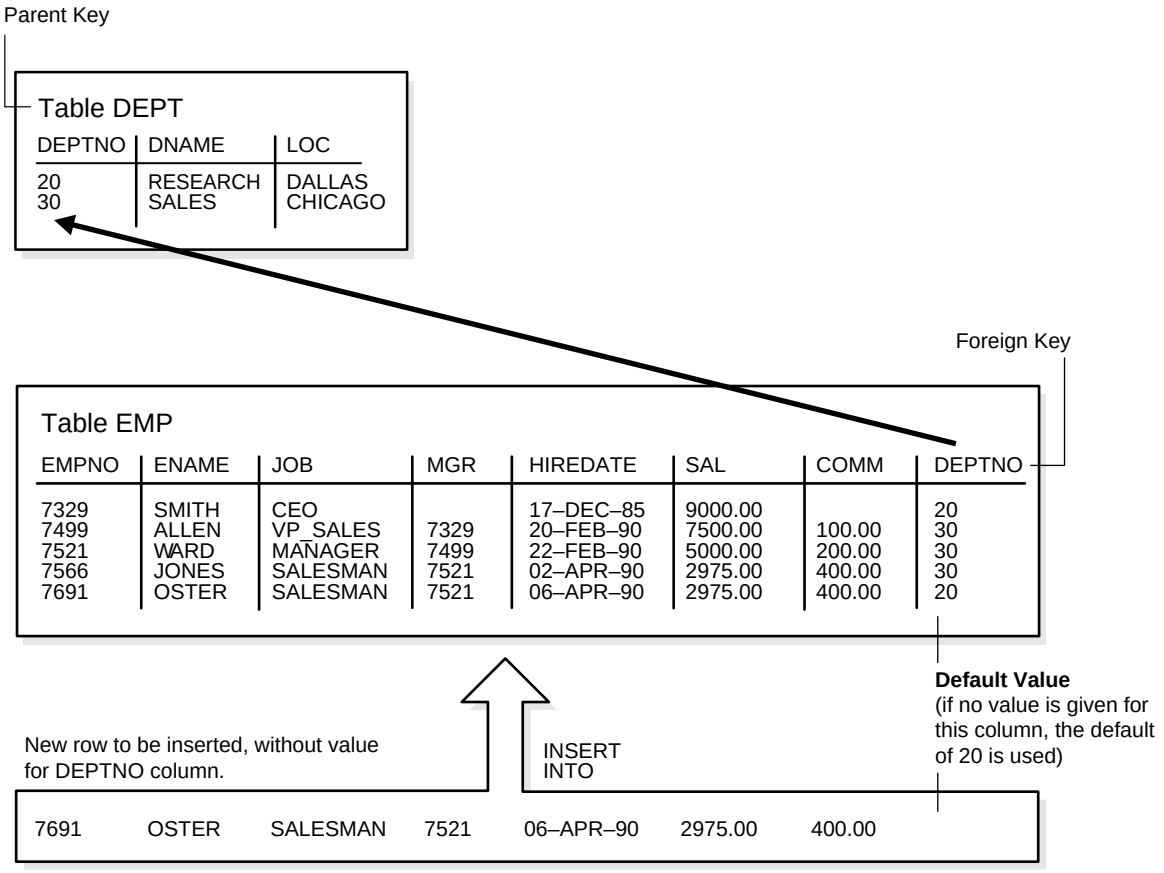
If a default value is not explicitly defined for a column, then the default for the column is implicitly set to NULL.

Default Value Insertion and Integrity Constraint Checking

Integrity constraint checking occurs after the row with a default value is inserted. For example, in Figure 10–4, a row is inserted into the emp table that does not include a value for the employee's department number. Because no value is supplied for the department number, Oracle inserts the deptno column's default value of 20. After inserting the default value, Oracle checks the FOREIGN KEY integrity constraint defined on the deptno column.

> **See Also:** Chapter 21, "Data Integrity" for more information about integrity constraints

Figure 10–4 DEFAULT Column Values



Partitioned Tables

Partitioned tables allow your data to be broken down into smaller, more manageable pieces called partitions, or even subpartitions. Indexes can be partitioned in similar fashion. Each partition can be managed individually, and can operate independently of the other partitions, thus providing a structure that can be better tuned for availability and performance.

See Also: Chapter 11, "Partitioned Tables and Indexes"

Nested Tables

You can create a table with a column whose datatype is another table. That is, tables can be **nested** within other tables as values in a column. The Oracle server stores nested table data out of line from the rows of the parent table, using a **store table** that is associated with the nested table column. The parent row contains a unique set identifier value associated with a nested table instance.

See Also:

- "Nested Tables Description" on page 13-12
- *Oracle9i Application Developer's Guide - Fundamentals*

Temporary Tables

In addition to permanent tables, Oracle can create **temporary tables** to hold session-private data that exists only for the duration of a transaction or session.

The `CREATE GLOBAL TEMPORARY TABLE` statement creates a temporary table that can be transaction-specific or session-specific. For transaction-specific temporary tables, data exists for the duration of the transaction. For session-specific temporary tables, data exists for the duration of the session. Data in a temporary table is private to the session. Each session can only see and modify its own data. DML locks are not acquired on the data of the temporary tables. The `LOCK` statement has no effect on a temporary table, because each session has its own private data.

A `TRUNCATE` statement issued on a session-specific temporary table truncates data in its own session. It does not truncate the data of other sessions that are using the same table.

DML statements on temporary tables do not generate redo logs for the data changes. However, undo logs for the data and redo logs for the undo logs are generated. Data from the temporary table is automatically dropped in the case of session termination, either when the user logs off or when the session terminates abnormally such as during a session or instance failure.

You can create indexes for temporary tables using the `CREATE INDEX` statement. Indexes created on temporary tables are also temporary, and the data in the index has the same session or transaction scope as the data in the temporary table.

You can create views that access both temporary and permanent tables. You can also create triggers on temporary tables.

The Export and Import utilities can export and import the definition of a temporary table. However, no data rows are exported even if you use the `ROWS` clause.

Similarly, you can replicate the definition of a temporary table, but you cannot replicate its data.

Segment Allocation

Temporary tables use temporary segments. Unlike permanent tables, temporary tables and their indexes do not automatically allocate a segment when they are created. Instead, segments are allocated when the first `INSERT` (or `CREATE TABLE AS SELECT`) is performed. This means that if a `SELECT`, `UPDATE`, or `DELETE` is performed before the first `INSERT`, then the table appears to be empty.

You can perform DDL statements (`ALTER TABLE`, `DROP TABLE`, `CREATE INDEX`, and so on) on a temporary table only when no session is currently bound to it. A session gets bound to a temporary table when an `INSERT` is performed on it. The session gets unbound by a `TRUNCATE`, at session termination, or by doing a `COMMIT` or `ABORT` for a transaction-specific temporary table.

Temporary segments are deallocated at the end of the transaction for transaction-specific temporary tables and at the end of the session for session-specific temporary tables.

See Also: "Extents in Temporary Segments" on page 2-11

Parent and Child Transactions

Transaction-specific temporary tables are accessible by user transactions and their child transactions. However, a given transaction-specific temporary table cannot be used concurrently by two transactions in the same session, although it can be used by transactions in different sessions.

If a user transaction does an `INSERT` into the temporary table, then none of its child transactions can use the temporary table afterward.

If a child transaction does an `INSERT` into the temporary table, then at the end of the child transaction, the data associated with the temporary table goes away. After that, either the user transaction or any other child transaction can access the temporary table.

External Tables

You can access data in external sources as if it were in a table in the database. You can connect to the database and create metadata for the external table, using DDL. The DDL for an external table consists of two parts: one part that describes the

Oracle column types, another part (the access parameters) which describes the mapping of the external data to the Oracle data columns.

An external table does not describe any data that is stored in the database. Nor does it describe how data is stored in the external source. Instead, it describes how the external table layer needs to present the data to the server. It is the responsibility of the access driver and the external table layer to do the necessary transformations required on the data in the data file so that it matches the external table definition.

External tables are read-only; therefore, no DML operations are possible, and no index can be created on them.

The Access Driver

When the database server needs to access data in an external source, it calls the appropriate access driver to get the data from an external source in a form that the database server expects. Oracle provides a default access driver that satisfies most requirements for accessing data in files.

It is important to remember that the description of the data in the data source is separate from the definition of the external table. The source file can contain more or fewer fields than the columns in the table. Also, the datatypes for fields in the data source can be different from the columns in the table. The access driver takes care of ensuring the data from the data source is processed so that it matches the definition of the external table.

Data Loading with External Tables

The main use for external tables is to use them as a row source for loading data into an actual table in the database. After you create an external table, you can then use a `CREATE TABLE AS SELECT` or `INSERT INTO ... AS SELECT` statement, using the external table as the source of the `SELECT` clause.

> **Note:** You cannot insert data into external tables or update records in them; external tables are read-only.

When you access the external table through a SQL statement, the fields of the external table can be used just like any other field in a regular table. In particular, you can use the fields as arguments for any SQL built-in function, PL/SQL function, or Java function. This lets you manipulate data from the external source. For data warehousing, you can do more sophisticated transformations in this way than you

can with simple datatype conversions. You can also use this mechanism in data warehousing to do data cleansing.

While external tables cannot contain a column object, constructor functions can be used to build a column object from attributes in the external table

Parallel Access to External Tables

After the metadata for an external table is created, you can query the external data directly and in parallel, using SQL. As a result, the external table acts as a view, which lets you run any SQL query against external data without loading the external data into the database.

The degree of parallel access to an external table is specified using standard parallel hints and with the PARALLEL clause. Using parallelism on an external table allows for concurrent access to the data files that comprise an external table. Whether a single file is accessed concurrently or not is dependent upon the access driver implementation, and attributes of the data file(s) being accessed (for example, record formats).

See Also:

- *Oracle9i Database Administrator's Guide* for information about managing external tables, external connections, and directories
- *Oracle9i Database Performance Tuning Guide and Reference* for information about tuning loads from external tables
- *Oracle9i Database Utilities* for information about import and export
- *Oracle9i SQL Reference* for information about creating and querying external tables

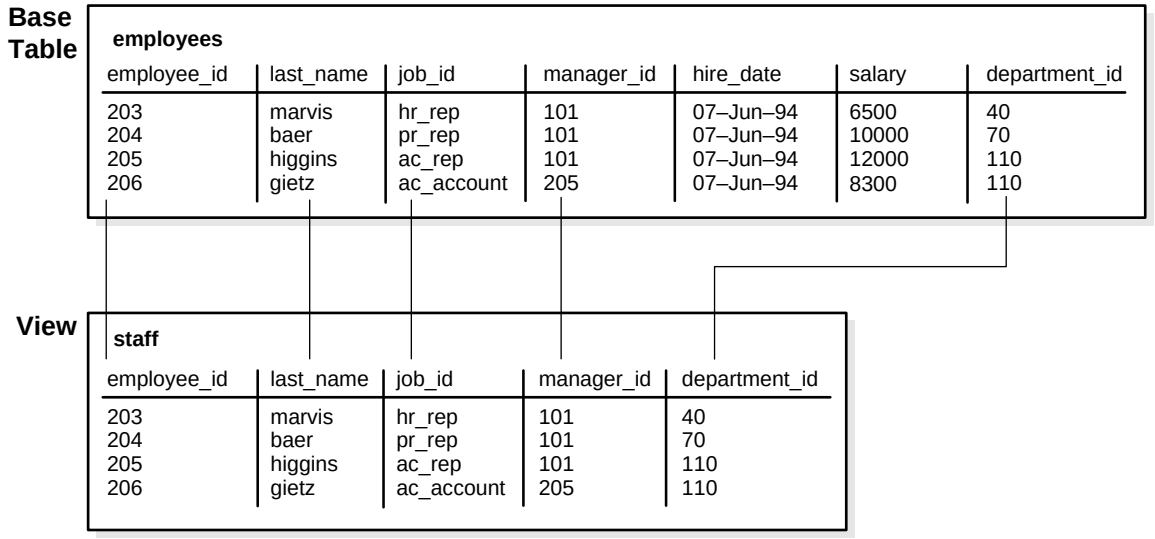
Views

A view is a tailored presentation of the data contained in one or more tables or other views. A view takes the output of a query and treats it as a table. Therefore, a view can be thought of as a stored query or a virtual table. You can use views in most places where a table can be used.

For example, the `employees` table has several columns and numerous rows of information. If you want users to see only five of these columns or only specific rows, then you can create a view of that table for other users to access.

Figure 10–5 shows an example of a view called STAFF derived from the base table `employees`. Notice that the view shows only five of the columns in the base table.

Figure 10–5 An Example of a View



Because views are derived from tables, they have many similarities. For example, you can define views with up to 1000 columns, just like a table. You can query views, and with some restrictions you can update, insert into, and delete from views. All operations performed on a view actually affect data in some base table of the view and are subject to the integrity constraints and triggers of the base tables.

Note: You cannot explicitly define triggers on views, but you can define them for the underlying base tables referenced by the view. Oracle does support definition of logical constraints on views.

See Also: *Oracle9i SQL Reference*

How Views are Stored

Unlike a table, a view is not allocated any storage space, nor does a view actually contain data. Rather, a view is defined by a query that extracts or derives data from the tables that the view references. These tables are called **base tables**. Base tables

can in turn be actual tables or can be views themselves (including materialized views). Because a view is based on other objects, a view requires no storage other than storage for the definition of the view (the stored query) in the data dictionary.

How Views Are Used

Views provide a means to present a different representation of the data that resides within the base tables. Views are very powerful because they let you tailor the presentation of data to different types of users. Views are often used to:

- Provide an additional level of table security by restricting access to a predetermined set of rows or columns of a table

  For example, Figure 10–5 shows how the STAFF view does not show the `salary` or `commission_pct` columns of the base table `employees`.

- Hide data complexity

  For example, a single view can be defined with a **join**, which is a collection of related columns or rows in multiple tables. However, the view hides the fact that this information actually originates from several tables.

- Simplify statements for the user

  For example, views allow users to select information from multiple tables without actually knowing how to perform a join.

- Present the data in a different perspective from that of the base table

  For example, the columns of a view can be renamed without affecting the tables on which the view is based.

- Isolate applications from changes in definitions of base tables

  For example, if a view's defining query references three columns of a four column table, and a fifth column is added to the table, then the view's definition is not affected, and all applications using the view are not affected.

- Express a query that cannot be expressed without using a view

  For example, a view can be defined that joins a GROUP BY view with a table, or a view can be defined that joins a `UNION` view with a table.

- Save complex queries

  For example, a query can perform extensive calculations with table information. By saving this query as a view, you can perform the calculations each time the view is queried.

See Also: *Oracle9i SQL Reference* for information about the GROUP BY or UNION views

Mechanics of Views

Oracle stores a view's definition in the data dictionary as the text of the query that defines the view. When you reference a view in a SQL statement, Oracle:

1. Merges the statement that references the view with the query that defines the view
2. Parses the merged statement in a shared SQL area
3. Executes the statement

Oracle parses a statement that references a view in a new shared SQL area *only* if no existing shared SQL area contains a similar statement. Therefore, you get the benefit of reduced memory use associated with shared SQL when you use views.

Globalization Support Parameters in Views

When Oracle evaluates views containing string literals or SQL functions that have globalization support parameters as arguments (such as TO_CHAR, TO_DATE, and TO_NUMBER), Oracle takes default values for these parameters from the globalization support parameters for the session. You can override these default values by specifying globalization support parameters explicitly in the view definition.

See Also: *Oracle9i Database Globalization Support Guide* for information about globalization support

Use of Indexes Against Views

Oracle determines whether to use indexes for a query against a view by transforming the original query when merging it with the view's defining query.

Consider the following view:

```
CREATE VIEW employees_view AS
  SELECT employee_id, last_name, salary, location_id
    FROM employees, departments
    WHERE employees.department_id = departments.department_id AND
                        departments.department_id = 10;
```

Now consider the following user-issued query:

```
SELECT last_name
  FROM employees_view
  WHERE employee_id = 9876;
```

The final query constructed by Oracle is:

```
SELECT last_name
  FROM employees, departments
  WHERE employees.department_id = departments.department_id AND
        departments.department_id = 10 AND
        employees.employee_id = 9876;
```

In all possible cases, Oracle merges a query against a view with the view's defining query and those of any underlying views. Oracle optimizes the merged query as if you issued the query without referencing the views. Therefore, Oracle can use indexes on any referenced base table columns, whether the columns are referenced in the view definition or in the user query against the view.

In some cases, Oracle cannot merge the view definition with the user-issued query. In such cases, Oracle may not use all indexes on referenced columns.

> **See Also:** *Oracle9i Database Performance Tuning Guide and Reference* for more information about query optimization

Dependencies and Views

Because a view is defined by a query that references other objects (tables, materialized views, or other views), a view depends on the referenced objects. Oracle automatically handles the dependencies for views. For example, if you drop a base table of a view and then create it again, Oracle determines whether the new base table is acceptable to the existing definition of the view.

> **See Also:** Chapter 15, "Dependencies Among Schema Objects" for a complete discussion of dependencies in a database

Updatable Join Views

A **join view** is defined as a view that has more than one table or view in its `FROM` clause (a **join**) and that does not use any of these clauses: `DISTINCT`, `AGGREGATION`, `GROUP BY`, `START WITH`, `CONNECT BY`, `ROWNUM`, and set operations (`UNION ALL`, `INTERSECT`, and so on).

An **updatable join view** is a join view that involves two or more base tables or views, where `UPDATE`, `INSERT`, and `DELETE` operations are permitted. The data

dictionary views ALL_UPDATABLE_COLUMNS, DBA_UPDATABLE_COLUMNS, and USER_UPDATABLE_COLUMNS contain information that indicates which of the view columns are updatable. In order to be inherently updatable, a view cannot contain any of the following constructs:

- A set operator
- A DISTINCT operator
- An aggregate or analytic function
- A GROUP BY, ORDER BY, CONNECT BY, or START WITH clause
- A collection expression in a SELECT list
- A subquery in a SELECT list
- Joins (with some exceptions). See *Oracle9i Database Administrator's Guide* for details.

Views that are not updatable can be modified using INSTEAD OF triggers.

See Also:

- *Oracle9i SQL Reference* for further information about updatable views
- "INSTEAD OF Triggers" on page 17-12

Object Views

In the Oracle object-relational database, an **object view** let you retrieve, update, insert, and delete relational data as if it was stored as an object type. You can also define views with columns that are object datatypes, such as objects, REFs, and collections (nested tables and VARRAYs).

See Also:

- Chapter 13, "Object Datatypes and Object Views"
- *Oracle9i Application Developer's Guide - Fundamentals*

Inline Views

An **inline view** is not a schema object. It is a subquery with an alias (correlation name) that you can use like a view within a SQL statement.

For example, this query joins the summary table SUMTAB to an inline view V defined on the TIME table to obtain T.YEAR, and then rolls up the aggregates in SUMTAB to the YEAR level:

```
SELECT v.year, s.prod_name, SUM(s.sum_sales)
    FROM sumtab s,
        (SELECT DISTINCT t.month, t.year FROM time t) v
    WHERE s.month = v.month
    GROUP BY v.year, s.prod_name;
```

See Also: *Oracle9i SQL Reference* for information about subqueries

Materialized Views

Materialized views are schema objects that can be used to summarize, compute, replicate, and distribute data. They are suitable in various computing environments such as data warehousing, decision support, and distributed or mobile computing:

- In data warehouses, materialized views are used to compute and store aggregated data such as sums and averages. Materialized views in these environments are typically referred to as **summaries** because they store summarized data. They can also be used to compute joins with or without aggregations. If compatibility is set to Oracle9*i* or higher, then materialized views can be used for queries that include filter selections.

  Cost-based optimization can use materialized views to improve query performance by automatically recognizing when a materialized view can and should be used to satisfy a request. The optimizer transparently rewrites the request to use the materialized view. Queries are then directed to the materialized view and not to the underlying detail tables or views.

- In distributed environments, materialized views are used to replicate data at distributed sites and synchronize updates done at several sites with conflict resolution methods. The materialized views as replicas provide local access to data that otherwise has to be accessed from remote sites.

- In mobile computing environments, materialized views are used to download a subset of data from central servers to mobile clients, with periodic refreshes from the central servers and propagation of updates by clients back to the central servers.

Materialized views are similar to indexes in several ways:

- They consume storage space.

- They must be refreshed when the data in their master tables changes.
- They improve the performance of SQL execution when they are used for query rewrites.
- Their existence is transparent to SQL applications and users.

Unlike indexes, materialized views can be accessed directly using a SELECT statement. Depending on the types of refresh that are required, they can also be accessed directly in an INSERT, UPDATE, or DELETE statement.

A materialized view can be partitioned. You can define a materialized view on a partitioned table and one or more indexes on the materialized view.

See Also:

- "Indexes" on page 10-28
- Chapter 11, "Partitioned Tables and Indexes"
- *Oracle9i Data Warehousing Guide* for information about materialized views in a data warehousing environment

Define Constraints on Views

Data warehousing applications recognize multidimensional data in the Oracle database by identifying Referential Integrity (RI) constraints in the relational schema. RI constraints represent primary and foreign key relationships among tables. By querying the Oracle data dictionary, applications can recognize RI constraints and therefore recognize the multidimensional data in the database. In some environments, database administrators, for schema complexity or security reasons, define views on fact and dimension tables. Oracle provides the ability to constrain views. By allowing constraint definitions between views, database administrators can propagate base table constraints to the views, thereby allowing applications to recognize multidimensional data even in a restricted environment.

Only logical constraints, that is, constraints that are declarative and not enforced by Oracle, can be defined on views. The purpose of these constraints is not to enforce any business rules but to identify multidimensional data. The following constraints can be defined on views:

- Primary key constraint
- Unique constraint
- Referential Integrity constraint

Given that view constraints are declarative, DISABLE, NOVALIDATE is the only valid state for a view constraint. However, the RELY or NORELY state is also allowed, because constraints on views may be used to enable more sophisticated query rewrites; a view constraint in the RELY state allows query rewrites to occur when the rewrite integrity level is set to trusted mode.

Note: Although view constraint definitions are declarative in nature, operations on views are subject to the integrity constraints defined on the underlying base tables, and constraints on views can be enforced through constraints on base tables.

Refresh Materialized Views

Oracle maintains the data in materialized views by refreshing them after changes are made to their master tables. The refresh method can be incremental *(***fast refresh***)* or complete. For materialized views that use the fast refresh method, a **materialized view log** or **direct loader log** keeps a record of changes to the master tables.

Materialized views can be refreshed either on demand or at regular time intervals. Alternatively, materialized views in the same database as their master tables can be refreshed whenever a transaction commits its changes to the master tables.

Materialized View Logs

A **materialized view log** is a schema object that records changes to a master table's data so that a materialized view defined on the master table can be refreshed incrementally.

Each materialized view log is associated with a single master table. The materialized view log resides in the same database and schema as its master table.

See Also:

- *Oracle9i Data Warehousing Guide* for information about materialized views and materialized view logs in a warehousing environment
- *Oracle9i Replication* for information about materialized views used for replication

Dimensions

A dimension defines hierarchical (parent/child) relationships between pairs of columns or column sets. Each value at the child level is associated with one and only one value at the parent level. A hierarchical relationship is a **functional dependency** from one level of a hierarchy to the next level in the hierarchy. A dimension is a container of logical relationships between columns, and it does not have any data storage assigned to it.

The `CREATE DIMENSION` statement specifies:

- Multiple `LEVEL` clauses, each of which identifies a column or column set in the dimension
- One or more `HIERARCHY` clauses that specify the parent/child relationships between adjacent levels
- Optional `ATTRIBUTE` clauses, each of which identifies an additional column or column set associated with an individual level

The columns in a dimension can come either from the same table (**denormalized**) or from multiple tables (**fully** or **partially normalized**). To define a dimension over columns from multiple tables, connect the tables using the `JOIN` clause of the `HIERARCHY` clause.

For example, a normalized time dimension can include a date table, a month table, and a year table, with join conditions that connect each date row to a month row, and each month row to a year row. In a fully denormalized time dimension, the date, month, and year columns are all in the same table. Whether normalized or denormalized, the hierarchical relationships among the columns need to be specified in the `CREATE DIMENSION` statement.

See Also:

- *Oracle9i Data Warehousing Guide* for information about how dimensions are used in a warehousing environment
- *Oracle9i SQL Reference* for information about creating dimensions

The Sequence Generator

The sequence generator provides a sequential series of numbers. The sequence generator is especially useful in multiuser environments for generating unique sequential numbers without the overhead of disk I/O or transaction locking. For

example, assume two users are simultaneously inserting new employee rows into the `employees` table. By using a sequence to generate unique employee numbers for the `employee_id` column, neither user has to wait for the other to enter the next available employee number. The sequence automatically generates the correct values for each user.

Therefore, the sequence generator reduces serialization where the statements of two transactions must generate sequential numbers at the same time. By avoiding the serialization that results when multiple users wait for each other to generate and use a sequence number, the sequence generator improves transaction throughput, and a user's wait is considerably shorter.

Sequence numbers are Oracle integers of up to 38 digits defined in the database. A sequence definition indicates general information, such as the following:

- The name of the sequence
- Whether the sequence ascends or descends
- The interval between numbers
- Whether Oracle should cache sets of generated sequence numbers in memory

Oracle stores the definitions of all sequences for a particular database as rows in a single data dictionary table in the `SYSTEM` tablespace. Therefore, all sequence definitions are always available, because the `SYSTEM` tablespace is always online.

Sequence numbers are used by SQL statements that reference the sequence. You can issue a statement to generate a new sequence number or use the current sequence number. After a statement in a user's session generates a sequence number, the particular sequence number is available only to that session. Each user that references a sequence has access to the current sequence number.

Sequence numbers are generated independently of tables. Therefore, the same sequence generator can be used for more than one table. Sequence number generation is useful to generate unique primary keys for your data automatically and to coordinate keys across multiple rows or tables. Individual sequence numbers can be skipped if they were generated and used in a transaction that was ultimately rolled back. Applications can make provisions to catch and reuse these sequence numbers, if desired.

Caution: If accountability for all sequence numbers is required, that is, if your application can never lose sequence numbers, then you cannot use Oracle sequences and you may choose to store sequence numbers in database tables.

Be careful when implementing sequence generators using database tables. Even in a single instance configuration, for a high rate of sequence values generation, a performance overhead is associated with the cost of locking the row that stores the sequence value.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for performance implications when using sequences
- *Oracle9i SQL Reference* for information about the CREATE SEQUENCE statement

Synonyms

A **synonym** is an alias for any table, view, materialized view, sequence, procedure, function, or package. Because a synonym is simply an alias, it requires no storage other than its definition in the data dictionary.

Synonyms are often used for security and convenience. For example, they can do the following:

- Mask the name and owner of an object
- Provide location transparency for remote objects of a distributed database
- Simplify SQL statements for database users
- Enable restricted access similar to specialized views when exercising fine-grained access control

You can create both public and private synonyms. A **public** synonym is owned by the special user group named PUBLIC and every user in a database can access it. A **private** synonym is in the schema of a specific user who has control over its availability to others.

Synonyms are very useful in both distributed and nondistributed database environments because they hide the identity of the underlying object, including its location in a distributed system. This is advantageous because if the underlying

object must be renamed or moved, then only the synonym needs to be redefined. Applications based on the synonym continue to function without modification.

Synonyms can also simplify SQL statements for users in a distributed database system. The following example shows how and why public synonyms are often created by a 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.

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);
```

You cannot create an index that references only one column in a table if another such index already exists.

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 speed of execution. Given a data value that has been indexed, the index points directly to the location of the rows containing that value.

Indexes are logically and physically independent of the data in the associated table. You can create or drop an index at any time without affecting the base tables or other indexes. If you drop an index, all applications continue to work. However, access of previously indexed data can be slower. Indexes, as independent structures, require storage space.

Oracle automatically maintains and uses indexes after they are created. Oracle automatically reflects changes to data, such as adding new rows, updating rows, or deleting rows, in all relevant indexes with no additional action by users.

Retrieval performance of indexed data remains almost constant, even as new rows are inserted. However, the presence of many indexes on a table decreases the performance of updates, deletes, and inserts, because Oracle must also update the indexes associated with the table.

The optimizer can use an existing index to build another index. This results in a much faster index build.

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). Nonunique indexes do not impose this restriction on the column values.

Oracle recommends that unique indexes be created explicitly, and not through enabling a unique constraint on a table.

Alternatively, you can define `UNIQUE` integrity constraints on the desired columns. Oracle enforces `UNIQUE` integrity constraints by automatically defining a unique index on the unique key. However, it is advisable that any index that exists for query performance, including unique indexes, be created explicitly.

> **See Also:** *Oracle9i Database Administrator's Guide* for information about creating unique indexes explicitly

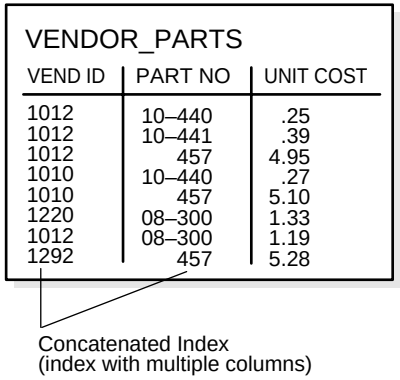
Composite Indexes

A **composite index** (also called a **concatenated index**) is an index that you create on multiple columns in a table. Columns in a composite index can appear in any order and 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. Generally, the most commonly accessed or most selective columns go first.

Figure 10–6 illustrates the `VENDOR_PARTS` table that has a composite index on the `VENDOR_ID` and `PART_NO` columns.

Figure 10–6 Composite Index Example



No more than 32 columns can form a regular composite index. For a bitmap index, the maximum number columns is 30. A key value cannot exceed roughly half (minus some overhead) the available data space in a data block.

> **See Also:** *Oracle9i Database Performance Tuning Guide and Reference* for more information about using composite indexes

Indexes and Keys

Although the terms are often used interchangeably, **indexes** and **keys** are different. **Indexes** are structures actually stored in the database, which users create, alter, and drop using SQL statements. You create an index to provide a fast access path to table data. **Keys** are strictly a logical concept. Keys correspond to another feature of Oracle called integrity constraints, which enforce the business rules of a database.

Because Oracle uses indexes to enforce some integrity constraints, the terms key and index are often are used interchangeably. However, do not confuse them with each other.

> **See Also:** Chapter 21, "Data Integrity"

Indexes and Nulls

`NULL` values in indexes are considered to be distinct except when all the non-`NULL` values in two or more rows of an index are identical, in which case the rows are considered to be identical. Therefore, `UNIQUE` indexes prevent rows containing

NULL values from being treated as identical. This does not apply if there are no non-NULL values—in other words, if the rows are entirely NULL.

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

See Also: "Bitmap Indexes and Nulls" on page 10-52

Function-Based Indexes

You can create indexes on functions and expressions that involve one or more columns in the table being indexed. A **function-based index** computes the value of the function or expression and stores it in the index. You can create a function-based index as either a B-tree or a bitmap index.

The function used for building the index can be an arithmetic expression or an expression that contains a PL/SQL function, package function, C callout, or SQL function. The expression cannot contain any aggregate functions, and it must be DETERMINISTIC. For building an index on a column containing an object type, the function can be a method of that object, such as a map method. However, you cannot build a function-based index on a LOB column, REF, or nested table column, nor can you build a function-based index if the object type contains a LOB, REF, or nested table.

See Also:

- "Bitmap Indexes"
- *Oracle9i Database Performance Tuning Guide and Reference* for more information about using function-based indexes

Uses of Function-Based Indexes

Function-based indexes provide an efficient mechanism for evaluating statements that contain functions in their WHERE clauses. The value of the expression is computed and stored in the index. When it processes INSERT and UPDATE statements, however, Oracle must still evaluate the function to process the statement.

For example, if you create the following index:

```
CREATE INDEX idx ON table_1 (a + b * (c - 1), a, b);
```

then Oracle can use it when processing queries such as this:

```
SELECT a FROM table_1 WHERE a + b * (c - 1) < 100;
```

Function-based indexes defined on UPPER(*column_name*) or LOWER(*column_name*) can facilitate case-insensitive searches. For example, the following index:

```
CREATE INDEX uppercase_idx ON employees (UPPER(first_name));
```

can facilitate processing queries such as this:

```
SELECT * FROM employees WHERE UPPER(first_name) = 'RICHARD';
```

A function-based index can also be used for a globalization support sort index that provides efficient linguistic collation in SQL statements.

> **See Also:** *Oracle9i Database Globalization Support Guide* for information about linguistic indexes

Optimization with Function-Based Indexes

You must gather statistics about function-based indexes for the optimizer. Otherwise, the indexes cannot be used to process SQL statements. Rule-based optimization never uses function-based indexes.

Cost-based optimization can use an index range scan on a function-based index for queries with expressions in WHERE clause. For example, in this query:

```
SELECT * FROM t WHERE a + b < 10;
```

the optimizer can use index range scan if an index is built on a+b. The range scan access path is especially beneficial when the predicate (WHERE clause) has low selectivity. In addition, the optimizer can estimate the selectivity of predicates involving expressions more accurately if the expressions are materialized in a function-based index.

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:** *Oracle9i Database Performance Tuning Guide and Reference* 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.

PL/SQL functions used in defining function-based indexes must be DETERMINISTIC. The index owner needs to have the EXECUTE privilege on the defining function. If the EXECUTE privilege is revoked, then the function-based index is marked DISABLED.

See Also:

- *Oracle9i Database Performance Tuning Guide and Reference* for information about DETERMINISTIC PL/SQL functions
- "Function-Based Index Dependencies" on page 15-8 for more information about dependencies and privileges for function-based indexes

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: "User Tablespace Settings and Quotas" on page 22-14

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. The number of bytes required for the overhead of an index block depends on the operating system.

See Also: Your Oracle operating system specific documentation for information about the overhead of an index block

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.

Although Oracle accepts the keywords `ASC`, `DESC`, `COMPRESS`, and `NOCOMPRESS` in the `CREATE INDEX` statement, they have no effect on index data, which is stored using rear compression in the branch nodes but not in the leaf nodes.

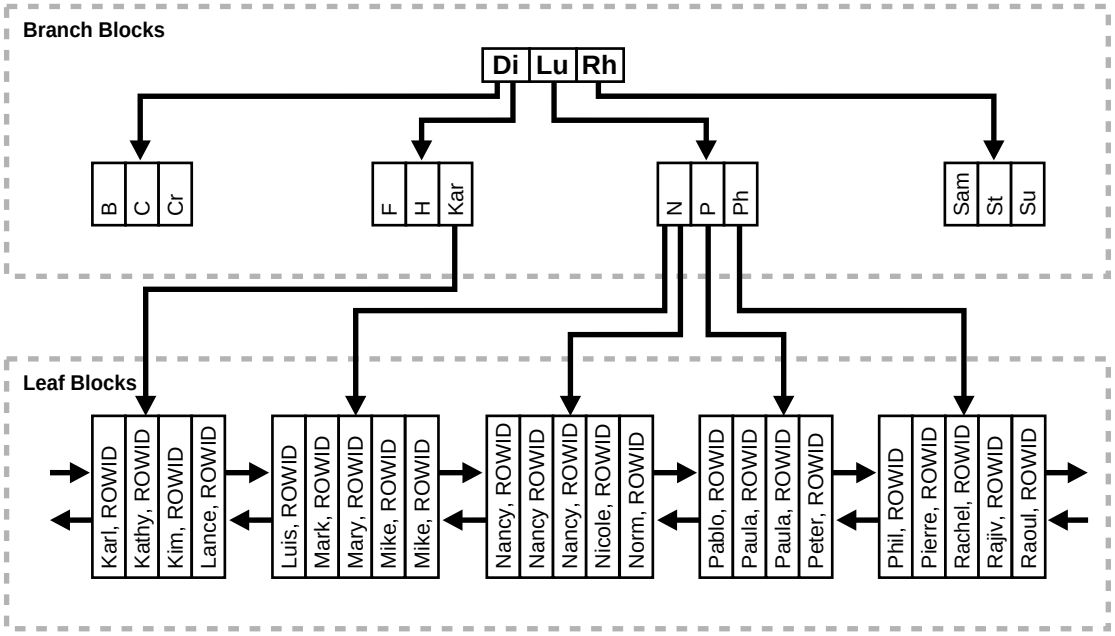
The Internal Structure of Indexes

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. Obviously 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 10–7 illustrates the structure of a B-tree index.

Figure 10–7 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

How Indexes Are Searched

Index Unique Scan

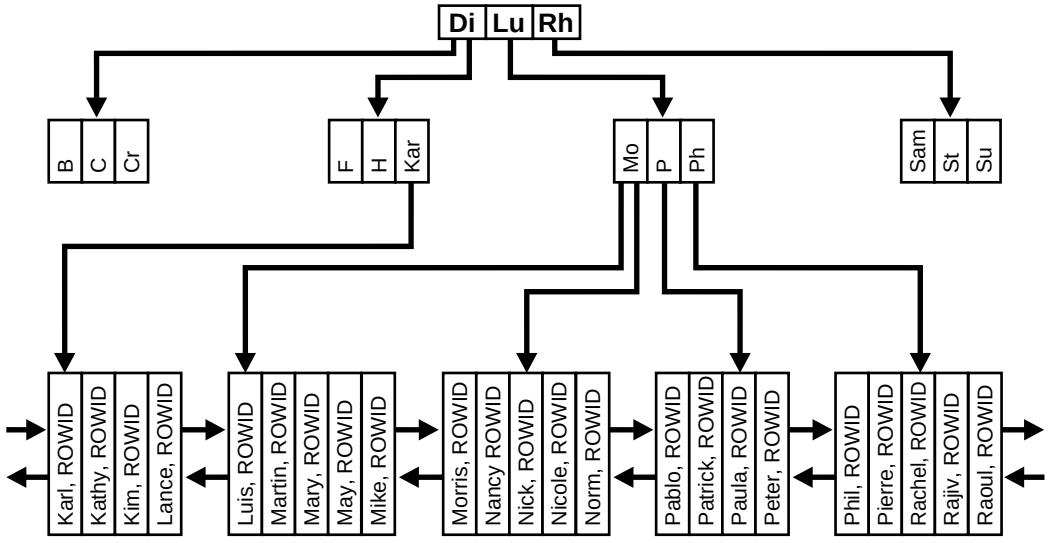
Index unique scan is one of the most efficient ways of accessing data. This access method is used for returning the data from B-tree indexes. The optimizer chooses a unique scan when all columns of a unique (B-tree) index are specified with equality conditions.

Steps in Index Unique Scans

1. Start with the root block.
2. Search the block keys for the smallest key greater than or equal to the value.
3. If key is greater than the value, then follow the link before this key to the child block.
4. If key is equal to the value, then follow this link to the child block.
5. If no key is greater than or equal to the value in Step 2, then follow the link after the highest key in the block.
6. Repeat steps 2 through 4 if the child block is a branch block.
7. Search the leaf block for key equal to the value.
8. If key is found, then return the `ROWID`.
9. If key is not found, then the row does not exist.

Figure 10–8 shows an example of an index unique scan and is described in the text that follows the figure.

Figure 10–8 Example of an Index Unique Scan



If searching for Patrick:

- In the root block, Rh is the smallest key >= Patrick.
- Follow the link before Rh to branch block (N, P, Ph).
- In this block, Ph is the smallest key >= Patrick.
- Follow the link before Ph to leaf block (Pablo, Patrick, Paula, Peter).
- In this block, search for key Patrick = Patrick.
- Found Patrick = Patrick, return (`KEY`, `ROWID`).

If searching for Meg:

- In the root block, Rh is the smallest key >= Meg.
- Follow the link before Rh to branch block (N, P, Ph).
- In this block, Mo is the smallest key >= Meg.
- Follow the link before Mo to leaf block (Luis,… , May, Mike).
- In this block, search for key = Meg.

- Did not find key = Meg, return 0 rows.

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 ROWIDs.

How Index Range Scans Work Index range scans can happen on both unique and non-unique indexes. B-tree non-unique indexes are identical to the unique B-tree indexes. However, they allow multiple values for the same key.

For a range scan, you can specify an equality condition. For example:

- `name = 'ALEX' - start key = 'ALEX', end key = 'ALEX'`

Alternatively, specify an interval bounded by start key and end key. For example:

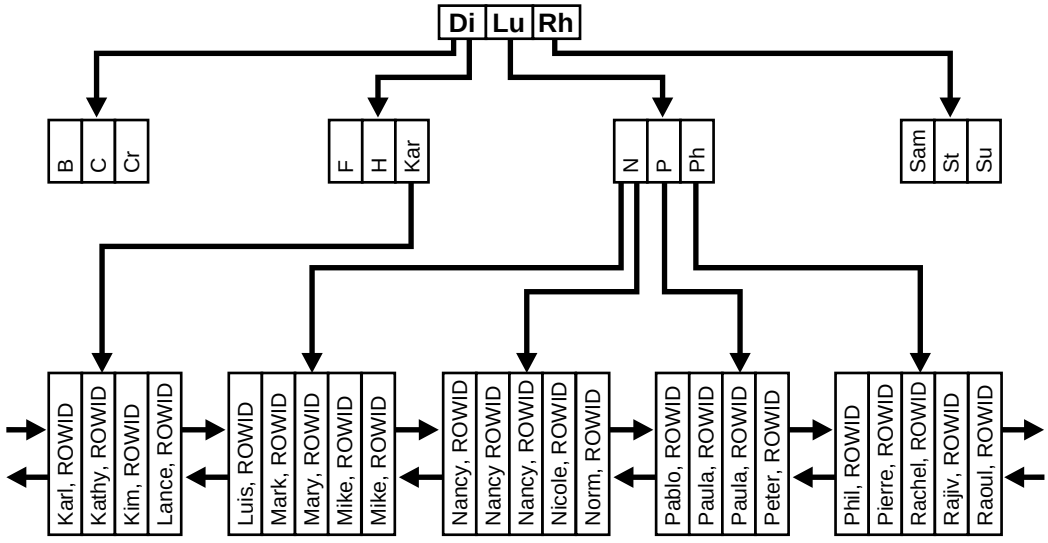
- `name LIKE 'AL%' - start key = 'AL, end key < 'AM'`
- `order_id BETWEEN 100 AND 120 - start key = 100, end key = 120`

Or, specify just a start key or an end key (unbounded range scan). For example:

- `order_book_date > SYSDATE - 30` (orders booked in last month)
- `employee_hire_date < SYSDATE - 3650` (employees with more than a decade of service)

Figure 10–9 shows an example of a bounded range scan and is described in the text that follows the figure.

Figure 10–9 Example of a Bounded Range Scan



Steps in a Bounded Range Scan

1. Start with the root block.
2. Search the block keys for the smallest key greater than or equal to the start key.
3. If key > start key, then follow the link before this key to the child block.
4. If key = start key, then follow this link to the child block.
5. If no key is greater than or equal to the start key in Step 2, then follow the link after the highest key in the block.
6. Repeat steps 2 through 4 if the child block is a branch block.
7. Search the leaf block keys for the smallest key greater than or equal to the start key.
8. While key <= end key:
   - If the key columns meet all `WHERE` clause conditions, then return the (value, `ROWID`).
   - Follow the link to the right.

Here, the range scans make use of the fact that all the leaf nodes are linked from left to right. In Step 7, extra filtering conditions on the index columns can be applied before accessing the table by ROWID.

Range scans bounded on the left (unbounded on the right) start the same. However, they do not check for the end point. They continue until they reach the right-most leaf key.

Range scans bounded on the right traverse the index tree to the left-most leaf key and then follow step #6 and # 7 until they reach a key greater then the specified condition.

With range scans using the non-unique B-tree index, if searching for Nancy:

- Start key = 'Nancy', end key < 'Nancy'.
- In the root block, Rh is the smallest key >= start key.
- Follow the link before Rh to branch block (N, P, Ph).
- In this block, P is the smallest key >= start key.
- Follow the link before P to leaf block (Nancy, …, Nicole, Norm).
- In this block, Nancy is the smallest key >= start key.
- Because Nancy <= end key, return the (KEY, ROWID).
- Next key Nancy <= end key, return the (KEY, ROWID).
- Next key Nancy <= end key, return the (KEY, ROWID).
- Next key Nicole > end key, terminate the range scan.

If searching for 'P%':

- Start key = 'P', end key < 'Q'.
- In the root block, Rh is the smallest key >= start key.
- Follow the link before Rh to branch block (N, P, Ph).
- In this block, P is the smallest key = start key.
- Follow this link to leaf block (Pablo,…, Peter).
- In this block, Pablo is the smallest key >= start key.
- Because Pablo <= end key, return the (KEY, ROWID).
- Next key Paula <= end key, return the (KEY, ROWID).
- Next key Paula <= end key, return the (KEY, ROWID).

- Next key Phil <= end key, return the (KEY, ROWID).
- Next key Pierre <= end key, return the (KEY, ROWID).
- Next key Rachel > end key, terminate the range scan.

Index Range Scan Descending

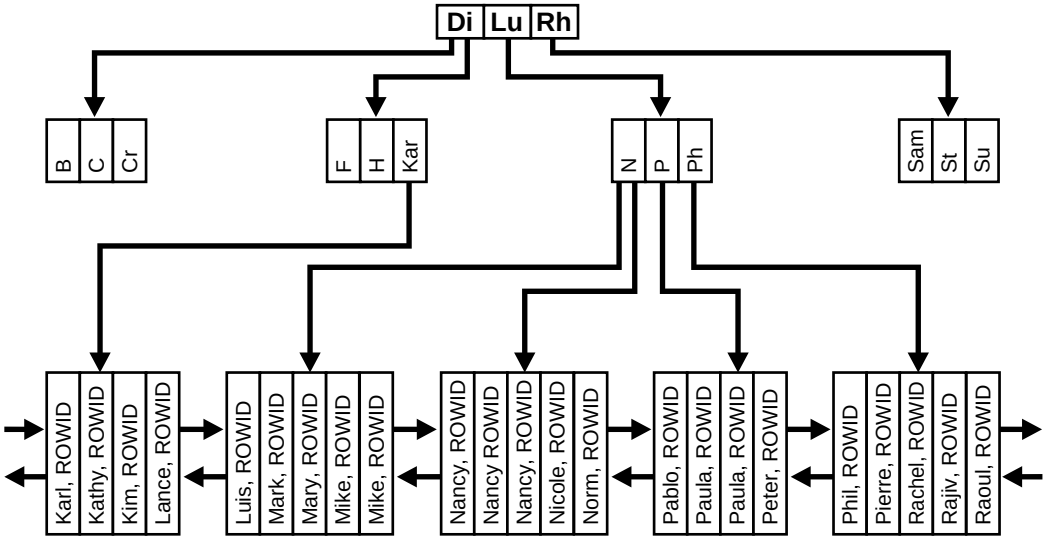
Steps in a Bounded Descending Range Scan For a descending range scan (like with the normal range scan), specify an equality condition or an interval.

1. Start with the root block.
2. Search the block keys for the biggest key less than or equal to the end key.
3. Follow the link to the child block.
4. If no key is less than or equal to the end key in step 2, then follow the link before the lowest key in the block.
5. Repeat steps 2 through 4 if the child block is a branch block.
6. Search the leaf block keys for the biggest key less than or equal to the end key.
7. While key >= start key:
   - If the key columns meet all WHERE clause conditions, then return the (value, ROWID).
   - Follow the link to the left.

Here, the range scans make use of the fact that all the leaf nodes are linked from right to left.

Figure 10–10 shows examples of a bounded range scan and is described in the text that follows the figure.

Figure 10–10 Examples of Range Scans Using the Non-Unique B-tree Index



If searching for Nancy:

- Start key = ‘Nancy’, end key < ‘Nancy’.
- In the root block, Lu is the biggest key <= end key.
- Follow the link to branch block (N, P, Ph).
- In this branch block, N is the biggest key <= end key.
- Follow the link after N to leaf block (Nancy, …, Nicole, Norm).
- In this leaf block, Nancy is the biggest key <= end key.
- Nancy >= start key, return the (`KEY`, `ROWID`).
- Prev key Nancy >= start key, return the (`KEY`, `ROWID`).
- Prev key Nancy >= start key, return the (`KEY`, `ROWID`).
- Prev key Mike < start key, terminate the range scan.

If searching for ‘P%’:

- Start key = ‘P’, end key < ‘Q’.

- In the root block key, Lu is the biggest key <= end key.
- Follow the link to branch block (N, P, Ph).
- In this branch block, Ph is the biggest key <= end key.
- Follow the link to leaf block (Phil,…,Raoul).
- In the leaf block, Pierre is the biggest key <= end key.
- Pierre >= start key, return the (KEY, ROWID).
- Prev key Phil >= start key, return the (KEY, ROWID).
- Prev key Peter >= start key, return the (KEY, ROWID).
- Prev key Paula >= start key, return the (KEY, ROWID).
- Prev key Pablo >= start key, return the (KEY, ROWID).
- Prev key Norm < start key, terminate the range scan.

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 ID (OID) is repeated for each element of the collection datatype. Key compression lets you compress the repeating OID 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: "Index-Organized Tables" on page 10-57

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 Oracle9*i* 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 Oracle9*i* 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

> **Note:** Bitmap indexes are available only if you have purchased the Oracle9*i* Enterprise Edition.
>
> See *Oracle9i Database New Features* for more information about the features available in Oracle9*i* and the Oracle9*i* Enterprise Edition.

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 for AND, OR, NOT, or equality queries.

Bitmap indexes are integrated with the Oracle cost-based optimization approach 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 10–1 shows a portion of a company's customer data.

Table 10–1 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 10–2 illustrates the bitmap index for the REGION column in this example. It consists of three separate bitmaps, one for each region.

Table 10–2 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 10–11. To identify the specific customers who satisfy the criteria, the resulting bitmap can be used to access the table.

Figure 10–11 Executing a Query Using Bitmap Indexes

Bitmap Indexes and Nulls

Bitmap indexes 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 and Nulls Example 1

```
SELECT COUNT(*) FROM employees;
```

Any bitmap index can be used for this query, because all table rows are indexed, including those that have NULL data. If NULLs were not indexed, then the optimizer could only use indexes on columns with NOT NULL constraints.

Bitmap Indexes and Nulls Example 2

```
SELECT COUNT(*) FROM employees WHERE commission_pct IS NULL;
```

This query can be optimized with a bitmap index on commission_pct.

Bitmap Indexes and Nulls Example 3

```
SELECT COUNT(*)
FROM customers
WHERE cust_gender = 'M' AND cust_state_province != 'CA';
```

This query can be answered by finding the bitmap for cust_gender = 'M' and subtracting the bitmap for cust_state_province = 'CA'. If cust_state_province can contain null values (that is, if it does not have a NOT NULL constraint), then the bitmaps for cust_state_province = 'NULL' must also be subtracted from the result.

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 11, "Partitioned Tables and Indexes" for information about partitioned tables and descriptions of local and global indexes
- *Oracle9i Database Performance Tuning Guide and Reference* for more information about using bitmap indexes

Bitmap Join Indexes

A join index is an index on one table that involves columns of one or more different tables through a join.

The bitmap join index, in its simplest form, is a bitmap index on a table `F` based on columns from table `D1,...,Dn`, where `Di` joins with `F` in a star or snowflake schema as described in "Creation of a Bitmap Join Index" on page 10-56. In the data warehousing environment, table `F` is usually a fact table, table `Di` is usually a dimension table, and the join condition is an equi-inner join between the primary key column(s) of the dimension tables and the foreign key column(s) in the fact table. For simplicity, from now on we call the table whose rowids are bitmapped the ***fact table***, and the other tables participating in the join of bitmap join index the ***dimension tables***.

The volume of data that must be joined can be reduced if join indexes are used as joins have already been precalculated. In addition, join indexes which contain multiple dimension tables can eliminate bitwise operations which are necessary in the star transformation with existing bitmap indexes. Finally, bitmap join indexes are much more efficient in storage than materialized join views which do not compress rowids of the fact tables.

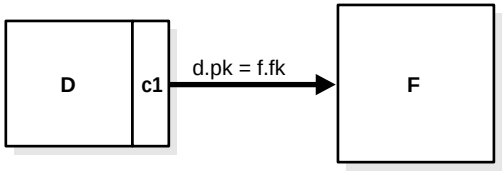
Four Join Models

The following section describes four join models in the star query framework and explains how they are addressed by bitmap join indexes. The accompanying figures are described by SQL statements in the text that follows each figure.

Notation

`F`*`i`* -- Fact table *i*
`D`*`i`* -- Dimension table *i*
`pk` -- The primary key column on the dimension table
`fk` -- The fact table column participating in the join with the dimension tables
`sales` -- The measurement column on the fact table

Figure 10–12 One Dimension Table Column Joins One Fact Table



In Figure 10–12, a bitmap join index on `F(D.c1)` can be represented by the following SQL statement:

```
CREATE BITMAP INDEX bji ON f (d.c1) FROM f, d WHERE d.pk = f.fk
```

Then the following query can be run by accessing the bitmap join index to avoid the join operation:

```
SELECT SUM(f.sales)
FROM d, f
WHERE d.pk = f.fk and d.c1 = 2
```

Similar to the materialized join view, a bitmap join index computes the join and stores it as a database object. The difference is that a materialized join view materializes the join into a table while a bitmap join index materializes the join into a bitmap index.

Figure 10–13 Two or More Dimension Table Columns Join One Fact Table

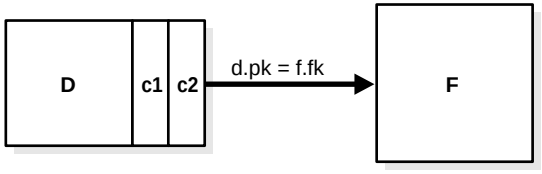


Figure 10–13 shows a simple extension of Figure 10–12, requiring a concatenated bitmap join index to represent it, as follows:

```
CREATE BITMAP INDEX bji ON f (d.c1, d.c2)
FROM F, d
WHERE d.pk = f.fk;
```

The result of the following query can be retrieved by accessing the bitmap join index `bji`.:

```
SELECT SUM(f.sales)
FROM d, f
WHERE d.pk = f.fk AND d.c1 = 1 AND d.c2 = 3;
```

Another query which references only the leading portion of the index key can also use bitmap join index `bji`:

```
SELECT SUM(f.sales)
FROM d, f
WHERE d.pk = f.fk AND d.c1 = 1
```

Figure 10–14 Multiple Dimension Tables Join One Fact Table

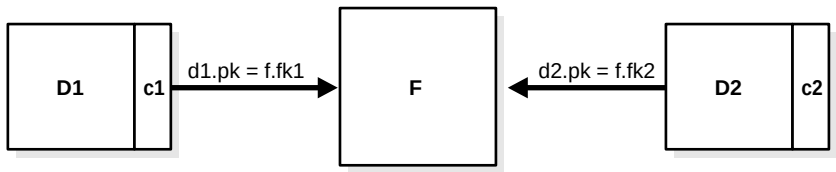


Figure 10–14 shows the third model, which requires a concatenated bitmap join index:

```
CREATE BITMAP INDEX bji ON f (d1.c1, d2.c2)
FROM f, d1, d2
WHERE d1.pk = f.fk1 AND d2.pk = f.fk2
```

Figure 10–15 Snow Flake Schema

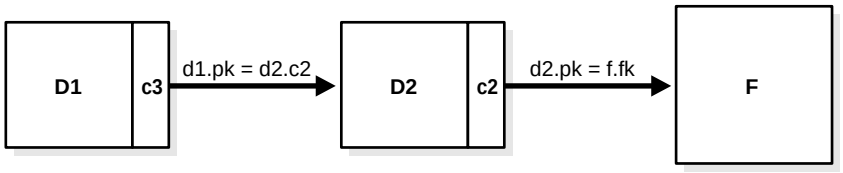


Figure 10–15 involves joins between two or more dimension tables. It can be expressed by a bitmap join index. The bitmap join index can be either single or concatenated depending on the number of columns in the dimension tables to be indexed. A bitmap join index on `d1.c3` with a join between `d1` and `d2` and a join between `d2` and `f` can be created as follows:

```
CREATE BITMAP INDEX bji ON f (d1.c3)
FROM f, d1, d2
WHERE d1.pk = d2.c2 AND d2.pk = f.fk;
```

A bitmap join index should be able to represent joins of the combination of the preceding models.

Creation of a Bitmap Join Index

Consider a star or snowflake schema with a single fact table `F` and multiple dimension tables `D1`,.., `Dn` as defined in "Bitmap Join Indexes" on page 10-53. These are the restrictions on the bitmap join index on `F` joined with `D1`,.., `Dn`.

- The bitmap join index is on a single table `F`.
- No table can appear twice in the `FROM` clause.
- Joins form either star or snowflake schema and all joins are through primary keys or keys with unique constraints as follows:
  - The dimension table column(s) participating the join with the fact table must be either the primary key column(s) or with the unique constraint
  - In the snow flake schema where a join is `D1><D2><F`, the column(s) on `D1` participating in the join `D1><D2` must be either the primary key column(s) or with the unique constraint.
  - For a composite primary key on the dimension table, each column of the key needs to be in the join.
- All joins are equi-inner joins and they are connected by `AND`s only.

- The current restrictions for creating a regular bitmap index also apply to a bitmap join index. For example, we cannot create a bitmap index with the `UNIQUE` attribute. See the *Oracle9i SQL Reference* for other restrictions.
- A bitmap join index must not be partitioned if the fact table is not partitioned. If the fact table is partitioned, the corresponding bitmap join index must be local partitioned with the fact table. Global partitioned bitmap join indexes are not supported.

Bitmap join index on IOT, functional bitmap join index and temporary bitmap join index are not yet allowed.

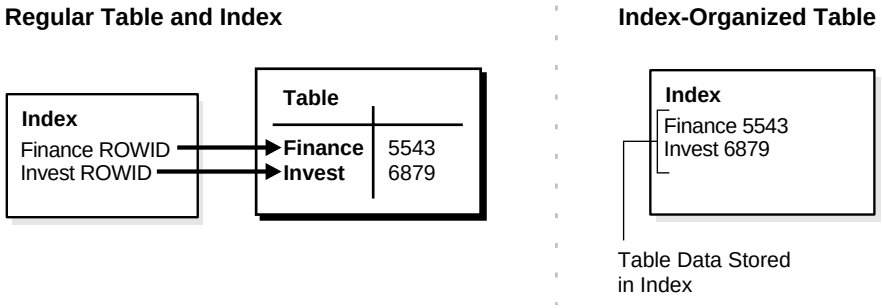
The primary key or unique constraint requirement is a correctness issue of a bitmap join index. For a regular bitmap index, there is a one-to-one mapping relation between a bit set in a bitmap and a rowid in the base table. For a bitmap join index, there should also be a one to one mapping between each row in the result set of the join and the rowids in the fact table. The primary key or unique constraint is used to enforce this one-to-one mapping.

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 10–16, 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 10–16 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 10–3 summarizes the differences between index-organized tables and ordinary tables.

Table 10–3 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 storage option (as described later) to push out infrequently accessed nonkey columns from the B-tree leaf block to an optional (heap-organized) overflow storage area. 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 10-45
- "Secondary Indexes on Index-Organized Tables" on page 10-60
- *Oracle9i 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, 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, and
- 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 storage area 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 storage area. Note that additional nonkey columns may sometimes need to be stored in the overflow due to PCTTHRESHOLD-based limits.

> **See Also:** *Oracle9i 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 optionally 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 12-21

Bitmap Indexes on Index-Organized Tables

Oracle supports bitmap indexes on 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 table, this physical rowid is then used to access the base table. However, in the case of index-organized table, 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 or HASH 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. Oracle9*i* 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
- Time-series applications

Application Domain Indexes

Oracle provides **extensible indexing** to accommodate indexes on customized complex data types 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**.

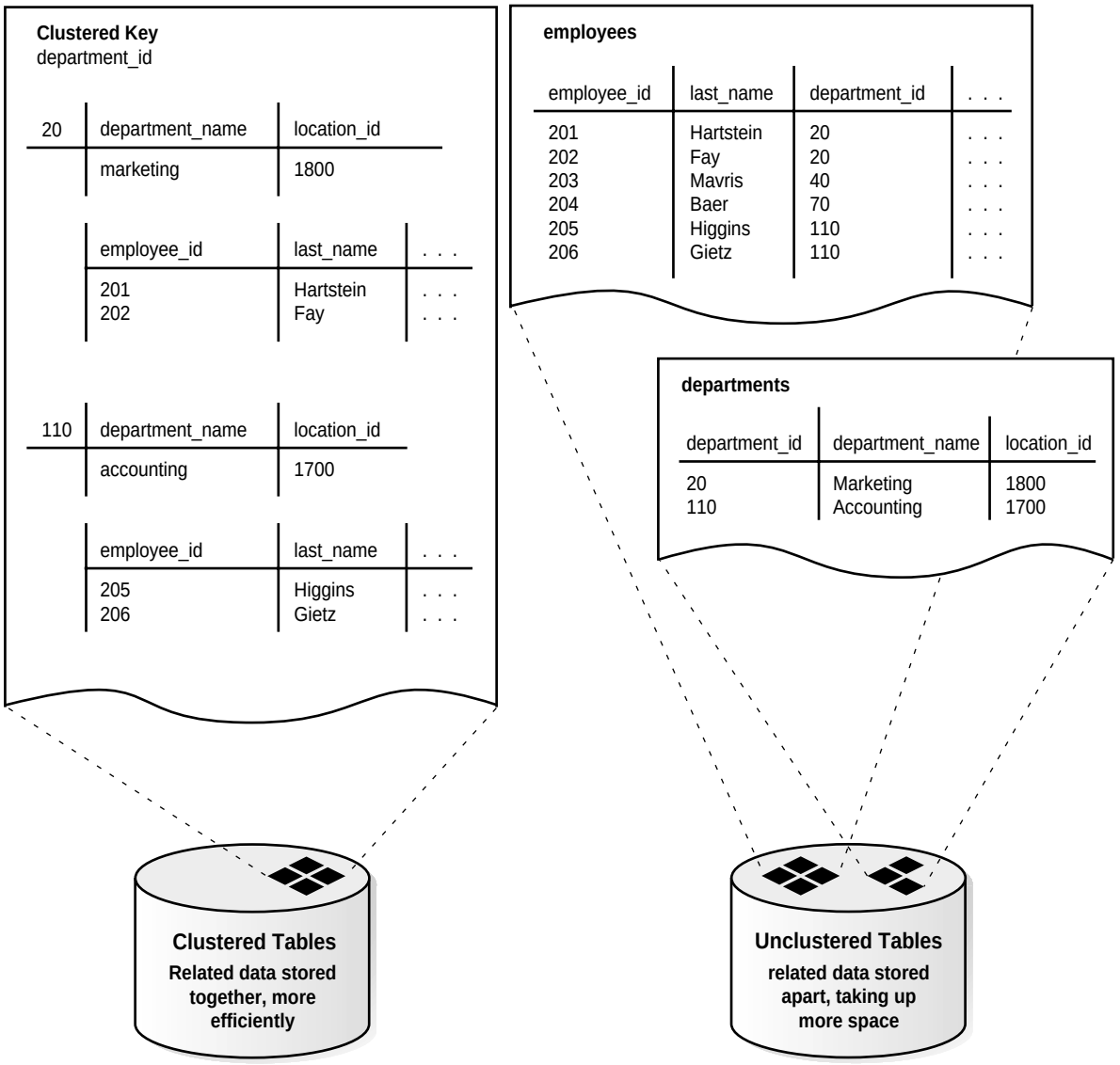
The application software, called the **cartridge**, controls the structure and content of a domain index. The Oracle server interacts with the application to build, maintain, and search the domain index. The index structure itself can be stored in the Oracle database as an index-organized table or externally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

> **See Also:** *Oracle9i Data Cartridge Developer's Guide* for information about using data cartridges within Oracle's extensibility architecture

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 10–17 shows what happens when you cluster the `employees` and `departments` tables:

Figure 10–17 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 10–17, 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: *Oracle9i Database Administrator's Guide* for information about creating and managing clusters

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.

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:** *Oracle9i Database Administrator's Guide* for information about creating and managing hash clusters

11

Partitioned Tables and Indexes

This chapter describes partitioned tables and indexes. It covers the following topics:

- Introduction to Partitioning
- Partitioning Methods
- Partitioned Indexes
- Partitioning to Improve Performance

> **Note:** Oracle supports partitioning only for tables, indexes on tables, materialized views, and indexes on materialized views. Oracle does not support partitioning of clustered tables or indexes on clustered tables.

Introduction to Partitioning

Partitioning addresses key issues in supporting very large tables and indexes by letting you decompose them into smaller and more manageable pieces called **partitions**. SQL queries and DML statements do not need to be modified in order to access partitioned tables. However, after partitions are defined, DDL statements can access and manipulate individuals partitions rather than entire tables or indexes. This is how partitioning can simplify the manageability of large database objects. Also, partitioning is entirely transparent to applications.

Each partition of a table or index must have the same logical attributes, such as column names, datatypes, and constraints, but each partition can have separate physical attributes such as pctfree, pctused, and tablespaces.

Partitioning is useful for many different types of applications, particularly applications that manage large volumes of data. OLTP systems often benefit from improvements in manageability and availability, while data warehousing systems benefit from performance and manageability.

Note: All partitions of a partitioned object must reside in tablespaces of a single block size.

See Also:

- "Multiple Block Sizes" on page 3-13
- *Oracle9i Data Warehousing Guide* for more information about partitioning

Partitioning offers these advantages:

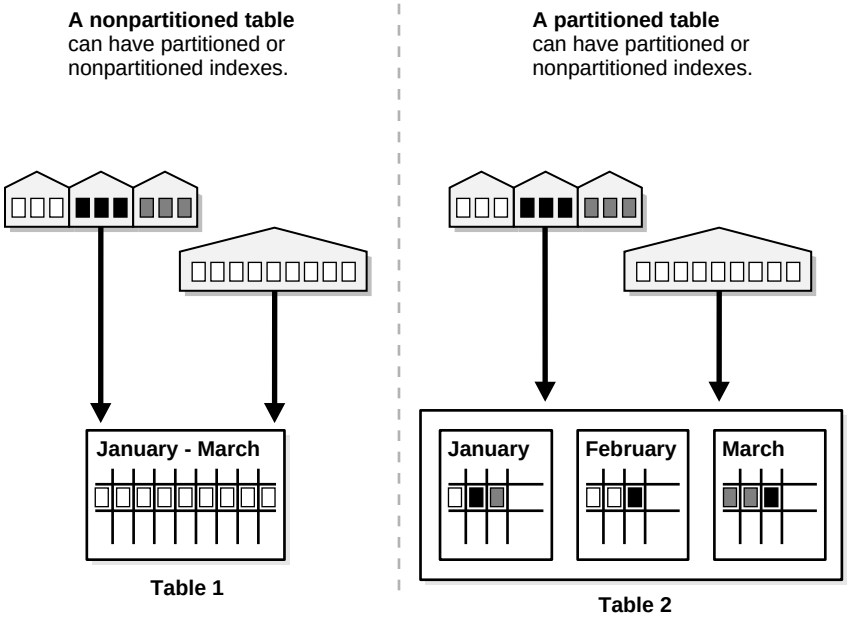
- Partitioning enables data management operations such data loads, index creation and rebuilding, and backup / recovery at the partition level, rather than on the entire table. This results in significantly reduced times for these operations.
- Partitioning improves query performance. In many cases, the results of a query can be achieved by accessing a subset of partitions, rather than the entire table. For some queries, this technique (called **partition pruning**) can provide order-of-magnitude gains in performance.
- Partitioning can significantly reduce the impact of scheduled downtime for maintenance operations.

Partition independence for partition maintenance operations lets you perform concurrent maintenance operations on different partitions of the same table or index. You can also run concurrent SELECT and DML operations against partitions that are unaffected by maintenance operations.

- Partitioning increases the availability of mission-critical databases if critical tables and indexes are divided into partitions to reduce the maintenance windows, recovery times, and impact of failures.
- Partitioning can be implemented without requiring any modifications to your applications. For example, you could convert a nonpartitioned table to a partitioned table without needing to modify any of the SELECT statements or DML statements which access that table. You do not need to rewrite your application code to take advantage of partitioning.

Figure 11–1 offers a graphical view of how partitioned tables differ from nonpartitioned tables.

Figure 11–1 A View of Partitioned Tables



Partition Key

Each row in a partitioned table is unambiguously assigned to a single partition. The partition key is a set of one or more columns that determines the partition for each row. Oracle9*i* automatically directs insert, update, and delete operations to the appropriate partition through the use of the partition key. A partition key:

- Consists of an ordered list of 1 to 16 columns
- Cannot contain a `LEVEL`, `ROWID`, or `MLSLABEL` pseudocolumn or a column of type `ROWID`
- Can contain columns that are `NULL`able

Partitioned Tables

Tables can be partitioned into up to 64,000 separate partitions. Any table can be partitioned except those tables containing columns with `LONG` or `LONG RAW` datatypes. You can, however, use tables containing columns with `CLOB` or `BLOB` datatypes.

Partitioned Index-Organized Tables

You can range partition index-organized tables. This feature is very useful for providing improved manageability, availability and performance for index-organized tables. In addition, data cartridges that use index-organized tables can take advantage of the ability to partition their stored data. Common examples of this are the Image and *inter*Media cartridges.

For partitioning an index-organized table:

- Only range and hash partitioning are supported
- Partition columns must be a subset of primary key columns
- Secondary indexes can be partitioned — locally and globally
- `OVERFLOW` data segments are always equipartitioned with the table partitions

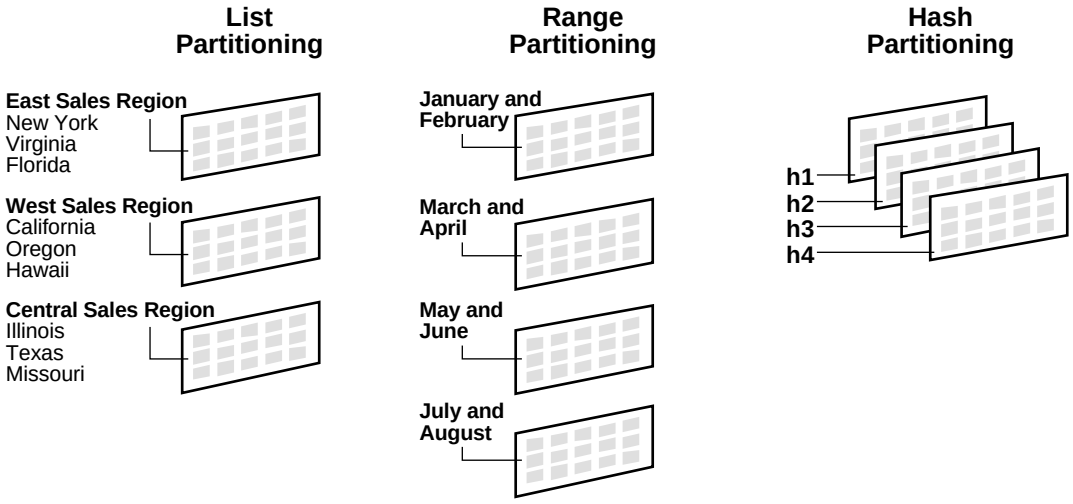
Partitioning Methods

Oracle provides the following partitioning methods:

- Range Partitioning
- List Partitioning
- Hash Partitioning
- Composite Partitioning

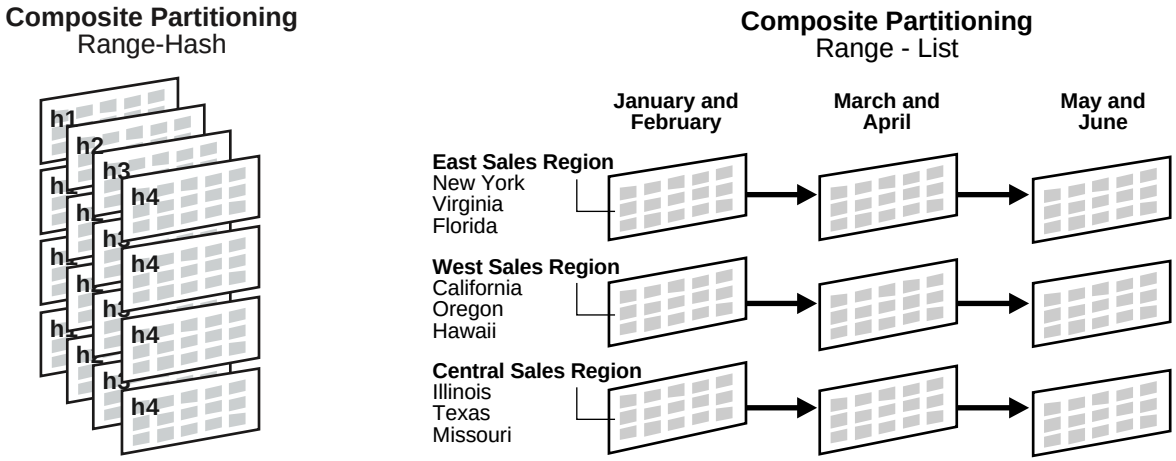
Figure 11–2 offers a graphical view of the methods of partitioning.

Figure 11–2 List, Range, and Hash Partitioning



Composite partitioning is a combination of other partitioning methods. Oracle currently supports range-hash and range-list composite partitioning. Figure 11–3 offers a graphical view of range-hash and range-list composite partitioning.

Figure 11–3 Composite Partitioning



Range Partitioning

Range partitioning maps data to partitions based on ranges of partition key values that you establish for each partition. It is the most common type of partitioning and is often used with dates. For example, you might want to partition sales data into monthly partitions.

When using range partitioning, consider the following rules:

- Each partition has a VALUES LESS THAN clause, which specifies a noninclusive upper bound for the partitions. Any binary values of the partition key equal to or higher than this literal are added to the next higher partition.
- All partitions, except the first, have an implicit lower bound specified by the VALUES LESS THAN clause on the previous partition.
- A MAXVALUE literal can be defined for the highest partition. MAXVALUE represents a virtual infinite value that sorts higher than any other possible value for the partition key, including the null value.

A typical example is given in the following section. The statement creates a table (`sales_range`) that is range partitioned on the `sales_date` field.

Range Partitioning Example

```
CREATE TABLE sales_range
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount  NUMBER(10),
sales_date    DATE)
PARTITION BY RANGE(sales_date)
(
PARTITION sales_jan2000 VALUES LESS THAN(TO_DATE('02/01/2000','DD/MM/YYYY')),
PARTITION sales_feb2000 VALUES LESS THAN(TO_DATE('03/01/2000','DD/MM/YYYY')),
PARTITION sales_mar2000 VALUES LESS THAN(TO_DATE('04/01/2000','DD/MM/YYYY')),
PARTITION sales_apr2000 VALUES LESS THAN(TO_DATE('05/01/2000','DD/MM/YYYY'))
);
```

List Partitioning

List partitioning enables you to explicitly control how rows map to partitions. You do this by specifying a list of discrete values for the partitioning key in the description for each partition. This is different from range partitioning, where a range of values is associated with a partition and from hash partitioning, where a hash function controls the row-to-partition mapping. The advantage of list partitioning is that you can group and organize unordered and unrelated sets of data in a natural way.

The details of list partitioning can best be described with an example. In this case, let's say you want to partition a sales table by region. That means grouping states together according to their geographical location as in the following example.

List Partitioning Example

```
CREATE TABLE sales_list
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_state   VARCHAR2(20),
sales_amount  NUMBER(10),
sales_date    DATE)
PARTITION BY LIST(sales_state)
(
PARTITION sales_west VALUES('California', 'Hawaii'),
PARTITION sales_east VALUES ('New York', 'Virginia', 'Florida'),
PARTITION sales_central VALUES('Texas', 'Illinois')
PARTITION sales_other VALUES(DEFAULT)
);
```

A row is mapped to a partition by checking whether the value of the partitioning column for a row falls within the set of values that describes the partition. For example, the rows are inserted as follows:

- (10, 'Jones', 'Hawaii', 100, '05-JAN-2000') maps to partition sales_west
- (21, 'Smith', 'Florida', 150, '15-JAN-2000') maps to partition sales_east
- (32, 'Lee', 'Colorado', 130, '21-JAN-2000') does not map to any partition in the table

Unlike range and hash partitioning, multicolumn partition keys are not supported for list partitioning. If a table is partitioned by list, the partitioning key can only consist of a single column of the table.

The DEFAULT partition enables you to avoid specifying all possible values for a list-partitioned table by using a default partition, so that all rows that do not map to any other partition do not generate an error.

Hash Partitioning

Hash partitioning enables easy partitioning of data that does not lend itself to range or list partitioning. It does this with a simple syntax and is easy to implement. It is a better choice than range partitioning when:

- You do not know beforehand how much data maps into a given range
- The sizes of range partitions would differ quite substantially or would be difficult to balance manually
- Range partitioning would cause the data to be undesirably clustered
- Performance features such as parallel DML, partition pruning, and partition-wise joins are important

The concepts of splitting, dropping or merging partitions do not apply to hash partitions. Instead, hash partitions can be added and coalesced.

Hash Partitioning Example

```
CREATE TABLE sales_hash
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount  NUMBER(10),
week_no       NUMBER(2))
PARTITION BY HASH(salesman_id)
PARTITIONS 4
STORE IN (data1, data2, data3, data4);
```

The preceding statement creates a table `sales_hash`, which is hash partitioned on `salesman_id` field. The tablespace names are `data1`, `data2`, `data3`, and `data4`.

Composite Partitioning

Composite partitioning partitions data using the range method, and within each partition, subpartitions it using the hash or list method. Composite range-hash partitioning provides the improved manageability of range partitioning and the data placement, striping, and parallelism advantages of hash partitioning. Composite range-list partitioning provides the manageability of range partitioning and the explicit control of list partitioning for the subpartitions.

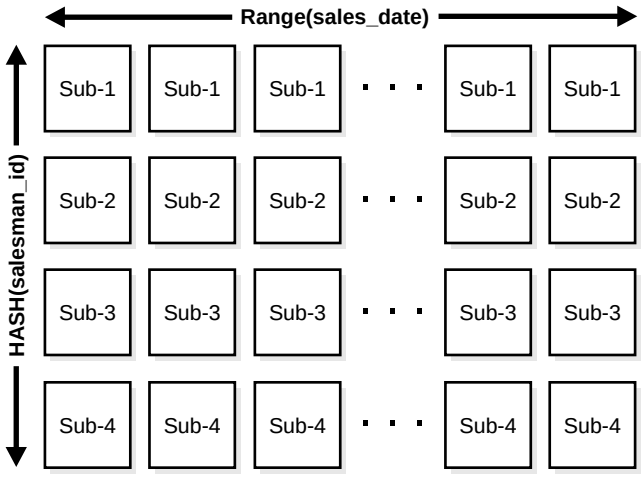
Composite partitioning supports historical operations, such as adding new range partitions, but also provides higher degrees of parallelism for DML operations and finer granularity of data placement through subpartitioning.

Composite Partitioning Range-Hash Example

```
CREATE TABLE sales_composite
(salesman_id  NUMBER(5),
 salesman_name VARCHAR2(30),
 sales_amount  NUMBER(10),
 sales_date    DATE)
PARTITION BY RANGE(sales_date)
SUBPARTITION BY HASH(salesman_id)
SUBPARTITION TEMPLATE(
SUBPARTITION sp1 TABLESPACE data1,
SUBPARTITION sp2 TABLESPACE data2,
SUBPARTITION sp3 TABLESPACE data3,
SUBPARTITION sp4 TABLESPACE data4)
(PARTITION sales_jan2000 VALUES LESS THAN(TO_DATE('02/01/2000','DD/MM/YYYY'))
 PARTITION sales_feb2000 VALUES LESS THAN(TO_DATE('03/01/2000','DD/MM/YYYY'))
 PARTITION sales_mar2000 VALUES LESS THAN(TO_DATE('04/01/2000','DD/MM/YYYY'))
 PARTITION sales_apr2000 VALUES LESS THAN(TO_DATE('05/01/2000','DD/MM/YYYY'))
 PARTITION sales_may2000 VALUES LESS THAN(TO_DATE('06/01/2000','DD/MM/YYYY')));
```

This statement creates a table `sales_composite` that is range partitioned on the `sales_date` field and hash subpartitioned on `salesman_id`. When you use a template, Oracle names the subpartitions by concatenating the partition name, an underscore, and the subpartition name from the template. Oracle places this subpartition in the tablespace specified in the template. In the previous statement, `sales_jan2000_sp1` is created and placed in tablespace `data1` while `sales_jan2000_sp4` is created and placed in tablespace `data4`. In the same manner, `sales_apr2000_sp1` is created and placed in tablespace `data1` while `sales_apr2000_sp4` is created and placed in tablespace `data4`. Figure 11–4 offers a graphical view of the previous example.

Figure 11–4 Composite Range-Hash Partitioning



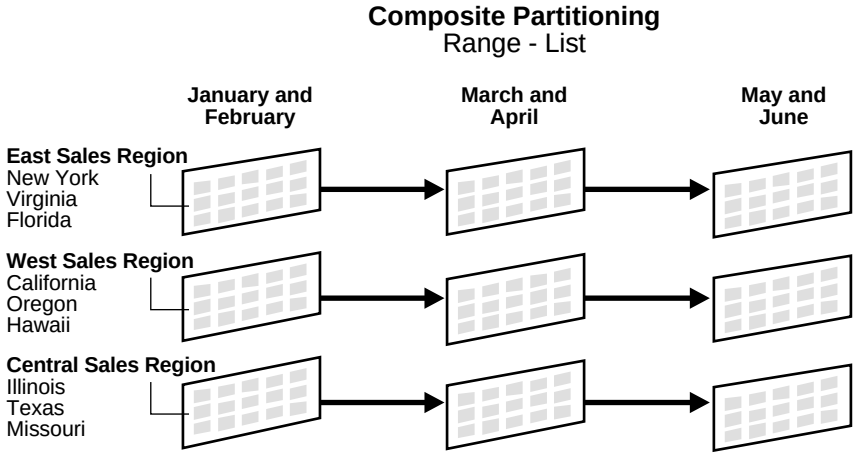
Composite Partitioning Range-List Example

```
CREATE TABLE bimonthly_regional_sales
(deptno NUMBER,
 item_no VARCHAR2(20),
 txn_date DATE,
 txn_amount NUMBER,
 state VARCHAR2(2))
PARTITION BY RANGE (txn_date)
SUBPARTITION BY LIST (state)
SUBPARTITION TEMPLATE(
    SUBPARTITION east VALUES('NY', 'VA', 'FL') TABLESPACE ts1,
    SUBPARTITION west VALUES('CA', 'OR', 'HI') TABLESPACE ts2,
    SUBPARTITION central VALUES('IL', 'TX', 'MO') TABLESPACE ts3)
(
PARTITION janfeb_2000 VALUES LESS THAN (TO_DATE('1-MAR-2000','DD-MON-YYYY')),
PARTITION marapr_2000 VALUES LESS THAN (TO_DATE('1-MAY-2000','DD-MON-YYYY')),
PARTITION mayjun_2000 VALUES LESS THAN (TO_DATE('1-JUL-2000','DD-MON-YYYY'))
);
```

This statement creates a table `bimonthly_regional_sales` that is range partitioned on the `txn_date` field and list subpartitioned on `state`. When you use a template, Oracle names the subpartitions by concatenating the partition name, an underscore, and the subpartition name from the template. Oracle places this subpartition in the tablespace specified in the template. In the previous statement,

janfeb_2000_east is created and placed in tablespace ts1 while janfeb_2000_central is created and placed in tablespace ts3. In the same manner, mayjun_2000_east is placed in tablespace ts1 while mayjun_2000_central is placed in tablespace ts3. Figure 11–5 offers a graphical view of the table bimonthly_regional_sales and its 9 individual subpartitions.

Figure 11–5 Composite Range-List Partitioning



When to Partition a Table

Here are some suggestions for when to partition a table:

- Tables greater than 2GB should always be considered for partitioning.
- Tables containing historical data, in which new data is added into the newest partition. A typical example is a historical table where only the current month's data is updatable and the other 11 months are read-only.

Partitioned Indexes

Just like partitioned tables, partitioned indexes improve manageability, availability, performance, and scalability. They can either be partitioned independently (global indexes) or automatically linked to a table's partitioning method (local indexes).

See Also: *Oracle9i Data Warehousing Guide* for more information about partitioned indexes

Local Partitioned Indexes

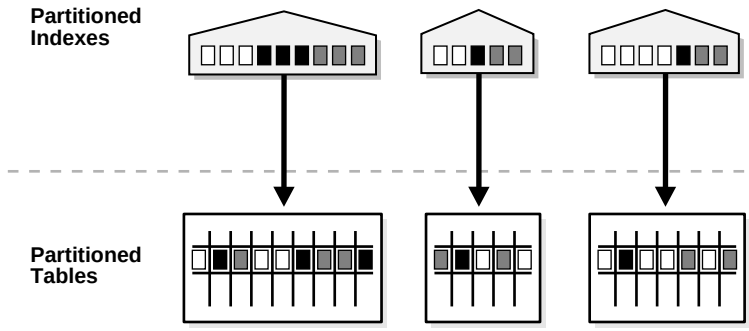
Local partitioned indexes are easier to manage than other types of partitioned indexes. They also offer greater availability and are common in DSS environments. The reason for this is equipartitioning: each partition of a local index is associated with exactly one partition of the table. This enables Oracle to automatically keep the index partitions in sync with the table partitions, and makes each table-index pair independent. Any actions that make one partition's data invalid or unavailable only affect a single partition.

You cannot explicitly add a partition to a local index. Instead, new partitions are added to local indexes only when you add a partition to the underlying table. Likewise, you cannot explicitly drop a partition from a local index. Instead, local index partitions are dropped only when you drop a partition from the underlying table.

A local index can be unique. However, in order for a local index to be unique, the partitioning key of the table must be part of the index's key columns. Unique local indexes are useful for OLTP environments.

Figure 11–6 offers a graphical view of local partitioned indexes.

Figure 11–6 Local Partitioned Index



Global Partitioned Indexes

Global partitioned indexes are flexible in that the degree of partitioning and the partitioning key are independent from the table's partitioning method. They are commonly used for OLTP environments and offer efficient access to any individual record.

The highest partition of a global index must have a partition bound, all of whose values are MAXVALUE. This ensures that all rows in the underlying table can be represented in the index. Global prefixed indexes can be unique or nonunique.

You cannot add a partition to a global index because the highest partition always has a partition bound of MAXVALUE. If you wish to add a new highest partition, use the ALTER INDEX SPLIT PARTITION statement. If a global index partition is empty, you can explicitly drop it by issuing the ALTER INDEX DROP PARTITION statement. If a global index partition contains data, dropping the partition causes the next highest partition to be marked unusable. You cannot drop the highest partition in a global index.

Maintenance of Global Partitioned Indexes

By default, the following operations on partitions on a heap-organized table mark all global indexes as unusable:

```
ADD (HASH)
COALESCE (HASH)
DROP
EXCHANGE
MERGE
MOVE
SPLIT
TRUNCATE
```

These indexes can be maintained by appending the clause `UPDATE GLOBAL INDEXES` to the SQL statements for the operation. The two advantages to maintaining global indexes:

- The index remains available and online throughout the operation. Hence no other applications are affected by this operation.
- The index doesn't have to be rebuilt after the operation.

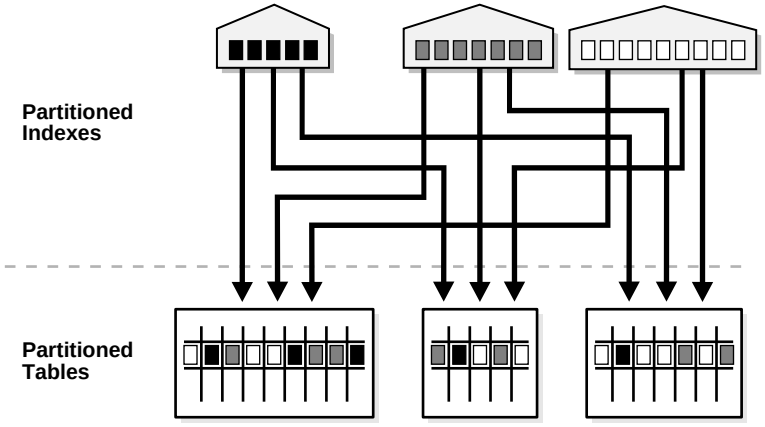
Example: `ALTER TABLE DROP PARTITION P1 UPDATE GLOBAL INDEXES`

Note: This feature is supported only for heap organized tables.

See Also: *Oracle9i SQL Reference* for more information about the `UPDATE GLOBAL INDEX` clause

Figure 11–7 offers a graphical view of global partitioned indexes.

Figure 11–7 Global Partitioned Index

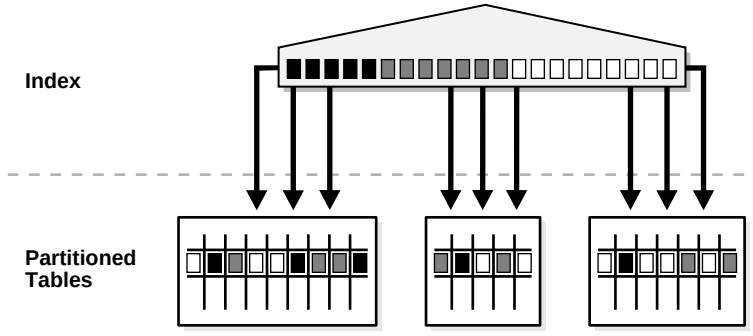


Global Nonpartitioned Indexes

Global nonpartitioned indexes behave just like a nonpartitioned index. They are commonly used in OLTP environments and offer efficient access to any individual record.

Figure 11–8 offers a graphical view of global nonpartitioned indexes.

Figure 11–8 Global Nonpartitioned Index



Partitioned Index Examples

Example of Index Creation: Starting Table Used for Examples

```
CREATE TABLE employees
(employee_id NUMBER(4) NOT NULL,
 last_name VARCHAR2(10),
 department_id NUMBER(2))
PARTITION BY RANGE (department_id)
(PARTITION employees_part1 VALUES LESS THAN (11) TABLESPACE part1,
 PARTITION employees_part2 VALUES LESS THAN (21) TABLESPACE part2,
 PARTITION employees_part3 VALUES LESS THAN (31) TABLESPACE part3);
```

Example of a Local Index Creation

```
CREATE INDEX employees_local_idx ON employees (employee_id) LOCAL;
```

Example of a Global Index Creation

```
CREATE INDEX employees_global_idx ON employees(employee_id);
```

Example of a Global Partitioned Index Creation

```
CREATE INDEX employees_global_part_idx ON employees(employee_id)
GLOBAL PARTITION BY RANGE(employee_id)
(PARTITION p1 VALUES LESS THAN(5000),
 PARTITION p2 VALUES LESS THAN(MAXVALUE));
```

Example of a Partitioned Index-Organized Table Creation

```
CREATE TABLE sales_range
(
salesman_id   NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount  NUMBER(10),
sales_date    DATE,
PRIMARY KEY(sales_date, salesman_id))
ORGANIZATION INDEX INCLUDING salesman_id
OVERFLOW TABLESPACE tabsp_overflow
PARTITION BY RANGE(sales_date)
(PARTITION sales_jan2000 VALUES LESS THAN(TO_DATE('02/01/2000','DD/MM/YYYY'))
 OVERFLOW TABLESPACE p1_overflow,
 PARTITION sales_feb2000 VALUES LESS THAN(TO_DATE('03/01/2000','DD/MM/YYYY'))
 OVERFLOW TABLESPACE p2_overflow,
 PARTITION sales_mar2000 VALUES LESS THAN(TO_DATE('04/01/2000','DD/MM/YYYY'))
 OVERFLOW TABLESPACE p3_overflow,
 PARTITION sales_apr2000 VALUES LESS THAN(TO_DATE('05/01/2000','DD/MM/YYYY'))
 OVERFLOW TABLESPACE p4_overflow);
```

Miscellaneous Information about Creating Indexes on Partitioned Tables

You can create bitmap indexes on partitioned tables, with the restriction that the bitmap indexes must be local to the partitioned table. They cannot be global indexes.

Global indexes can be unique. Local indexes can only be unique if the partitioning key is a part of the index key.

Using Partitioned Indexes in OLTP Applications

Here are a few guidelines for OLTP applications:

- Global indexes and unique, local indexes provide better performance than nonunique local indexes because they minimize the number of index partition probes.
- Local indexes offer better availability when there are partition or subpartition maintenance operations on the table.

Using Partitioned Indexes in Data Warehousing and DSS Applications

Here are a few guidelines for data warehousing and DSS applications:

- Local indexes are preferable because they are easier to manage during data loads and during partition-maintenance operations.
- Local indexes can improve performance because many index partitions can be scanned in parallel by range queries on the index key.

Partitioned Indexes on Composite Partitions

Here are a few points to remember when using partitioned indexes on composite partitions:

- Only range partitioned global indexes are supported.
- Subpartitioned indexes are always local and stored with the table subpartition by default.
- Tablespaces can be specified at either index or index subpartition levels.

Partitioning to Improve Performance

Partitioning can help you improve performance and manageability. Some topics to keep in mind when using partitioning for these reasons are:

- Partition Pruning
- Partition-wise Joins
- Parallel DML

Partition Pruning

The Oracle server explicitly recognizes partitions and subpartitions. It then optimizes SQL statements to mark the partitions or subpartitions that need to be accessed and eliminates (prunes) unnecessary partitions or subpartitions from access by those SQL statements. In other words, partition pruning is the skipping of unnecessary index and data partitions or subpartitions in a query.

For each SQL statement, depending on the selection criteria specified, unneeded partitions or subpartitions can be eliminated. For example, if a query only involves March sales data, then there is no need to retrieve data for the remaining eleven months. Such intelligent pruning can dramatically reduce the data volume, resulting in substantial improvements in query performance.

If the optimizer determines that the selection criteria used for pruning are satisfied by all the rows in the accessed partition or subpartition, it removes those criteria

from the predicate list (WHERE clause) during evaluation in order to improve performance. However, the optimizer cannot prune partitions if the SQL statement applies a function to the partitioning column (with the exception of the TO_DATE function). Similarly, the optimizer cannot use an index if the SQL statement applies a function to the indexed column, unless it is a function-based index.

Pruning can eliminate index partitions even when the underlying table's partitions cannot be eliminated, but only when the index and table are partitioned on different columns. You can often improve the performance of operations on large tables by creating partitioned indexes that reduce the amount of data that your SQL statements need to access or modify.

Equality, range, LIKE, and IN-list predicates are considered for partition pruning with range or list partitioning, and equality and IN-list predicates are considered for partition pruning with hash partitioning.

Partition Pruning Example

We have a partitioned table called orders. The partition key for orders is order_date. Let's assume that orders has six months of data, January to June, with a partition for each month of data. If the following query is run:

```
SELECT SUM(value)
FROM orders
WHERE order_date BETWEEN '28-MAR-98' AND '23-APR-98'
```

Partition pruning is achieved by:

- First, partition elimination of January, February, May, and June data partitions. Then either:
  - An index scan of the March and April data partition due to high index selectivity

    or
  - A full scan of the March and April data partition due to low index selectivity

Partition-wise Joins

A partition-wise join is a join optimization that you can use when joining two tables that are both partitioned along the join column(s). With partition-wise joins, the join operation is broken into smaller joins that are performed sequentially or in parallel. Another way of looking at partition-wise joins is that they minimize the amount of

data exchanged among parallel slaves during the execution of parallel joins by taking into account data distribution.

> **See Also:** *Oracle9i Data Warehousing Guide* for more information about partitioning methods and partition-wise joins

Parallel DML

Parallel execution dramatically reduces response time for data-intensive operations on large databases typically associated with decision support systems and data warehouses. In addition to conventional tables, you can use parallel query and parallel DML with range- and hash-partitioned tables. By doing so, you can enhance scalability and performance for batch operations.

The semantics and restrictions for parallel DML sessions are the same whether you are using index-organized tables or not.

> **See Also:** *Oracle9i Data Warehousing Guide* for more information about parallel DML and its use with partitioned tables

12

Native Datatypes

This chapter discusses the Oracle built-in datatypes, their properties, and how they map to non-Oracle datatypes. Topics include:

Introduction to Oracle Datatypes

Each column value and constant in a SQL statement has a **datatype**, which is associated with a specific storage format, constraints, and a valid range of values. When you create a table, you must specify a datatype for each of its columns.

Oracle provides the following built-in datatypes:

- Character Datatypes
  - CHAR Datatype
  - VARCHAR2 and VARCHAR Datatypes
  - NCHAR and NVARCHAR2 Datatypes
  - LONG Datatype
- NUMBER Datatype
- DATE Datatype
- LOB Datatypes
  - BLOB Datatype
  - CLOB and NCLOB Datatypes
  - BFILE Datatype
- RAW and LONG RAW Datatypes
- ROWID and UROWID Datatypes
  - Physical Rowids
  - Logical Rowids
  - Rowids in Non-Oracle Databases

Note: PL/SQL has additional datatypes for constants and variables, which include BOOLEAN, reference types, composite types (collections and records), and user-defined subtypes.

See Also:

- *PL/SQL User's Guide and Reference* for information about PL/SQL datatypes and a summary of the characteristics of each Oracle datatype
- *Oracle9i Application Developer's Guide - Fundamentals* for information about how to use the built-in datatypes

The following sections that describe each of the built-in datatypes in more detail.

Character Datatypes

The character datatypes store character (alphanumeric) data in strings, with byte values corresponding to the character encoding scheme, generally called a character set or code page.

The database's character set is established when you create the database. Examples of character sets are 7-bit ASCII (American Standard Code for Information Interchange), EBCDIC (Extended Binary Coded Decimal Interchange Code), Code Page 500, Japan Extended UNIX, and Unicode UTF-8. Oracle supports both single-byte and multibyte encoding schemes.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for information about how to select a character datatype
- *Oracle9i Database Globalization Support Guide* for more information about converting character data

CHAR Datatype

The CHAR datatype stores fixed-length character strings. When you create a table with a CHAR column, you must specify a string length (in bytes or characters) between 1 and 2000 bytes for the CHAR column width. The default is 1 byte. Oracle then guarantees that:

- When you insert or update a row in the table, the value for the CHAR column has the fixed length.
- If you give a shorter value, then the value is blank-padded to the fixed length.
- If you give a longer value with trailing blanks, then blanks are trimmed from the value to the fixed length.

- If a value is too large, Oracle returns an error.

Oracle compares CHAR values using blank-padded comparison semantics.

> **See Also:** *Oracle9i SQL Reference* for details about blank-padded comparison semantics

VARCHAR2 and VARCHAR Datatypes

The VARCHAR2 datatype stores variable-length character strings. When you create a table with a VARCHAR2 column, you specify a maximum string length (in bytes or characters) between 1 and 4000 bytes for the VARCHAR2 column. For each row, Oracle stores each value in the column as a variable-length field unless a value exceeds the column's maximum length, in which case Oracle returns an error. Using VARCHAR2 and VARCHAR saves on space used by the table.

For example, assume you declare a column VARCHAR2 with a maximum size of 50 characters. In a single-byte character set, if only 10 characters are given for the VARCHAR2 column value in a particular row, the column in the row's row piece stores only the 10 characters (10 bytes), not 50.

Oracle compares VARCHAR2 values using nonpadded comparison semantics.

> **See Also:** *Oracle9i SQL Reference* for details about nonpadded comparison semantics

VARCHAR Datatype

The VARCHAR datatype is synonymous with the VARCHAR2 datatype. To avoid possible changes in behavior, always use the VARCHAR2 datatype to store variable-length character strings.

Length Semantics for Character Datatypes

Globalization support allows the use of various character sets for the character datatypes. Globalization support lets you process single-byte and multibyte character data and convert between character sets. Client sessions can use client character sets that are different from the database character set.

Consider the size of characters when you specify the column length for character datatypes. You must consider this issue when estimating space for tables with columns that contain character data.

The length semantics of character datatypes can be measured in bytes or characters.

- **Byte semantics** treat strings as a sequence of bytes. This is the default for character datatypes.
- **Character semantics** treat strings as a sequence of characters. A character is technically a codepoint of the database character set.

For single byte character sets, columns defined in character semantics are basically the same as those defined in byte semantics. Character semantics are useful for defining varying-width multibyte strings; it reduces the complexity when defining the actual length requirements for data storage. For example, in a Unicode database (UTF8), you need to define a VARCHAR2 column that can store up to five Chinese characters together with five English characters. In byte semantics, this would require (5*3 bytes) + (1*5 bytes) = 20 bytes; in character semantics, the column would require 10 characters.

VARCHAR2(20 BYTE) and SUBSTRB(<*string*>, 1, 20) use byte semantics.
VARCHAR2(10 CHAR) and SUBSTR(<*string*>, 1, 10) use character semantics.

The parameter NLS_LENGTH_SEMANTICS decides whether a new column of character datatype uses byte or character semantics. The default length semantic is byte. If all character datatype columns in a database use byte semantics (or all use character semantics) then users do not have to worry about which columns use which semantics. The BYTE and CHAR qualifiers shown earlier should be avoided when possible, because they lead to mixed-semantics databases. Instead, the NLS_LENGTH_SEMANTICS initialization parameter should be set appropriately in INIT.ORA, and columns should use the default semantics.

See Also:

- "Use of Unicode Data in an Oracle Database" on page 12-6
- *Oracle9i Database Globalization Support Guide* for more information about Oracle's globalization support feature
- *Oracle9i Application Developer's Guide - Fundamentals* for information about setting length semantics and choosing the appropriate Unicode character set.
- *Oracle9i Database Migration* for information about migrating existing columns to character semantics

NCHAR and NVARCHAR2 Datatypes

NCHAR and NVARCHAR2 are Unicode data types that store Unicode character data. The character set of NCHAR and NVARCHAR2 datatypes can only be either

AL16UTF16 or UTF8 and is specified at database creation time as the national character set. AL16UTF16 and UTF8 are both Unicode encoding.

- The NCHAR datatype stores fixed-length character strings that correspond to the national character set.
- The NVARCHAR2 datatype stores variable length character strings.

When you create a table with an NCHAR or NVARCHAR2 column, the maximum size specified is always in character length semantics. Character length semantics is the default and only length semantics for NCHAR or NVARCHAR2.

Example 12–1 Defining Maximum Byte Length of a Column

If national character set is UTF8, the following statement defines the maximum byte length of 90 bytes:

```
CREATE TABLE tab1 (col1 NCHAR(30));
```

This statement creates a column with maximum character length of 30. The maximum byte length is the multiple of the maximum character length and the maximum number of bytes in each character.

NCHAR

The maximum length of an NCHAR column is 2000 bytes. It can hold up to 2000 characters. The actual data is subject to the maximum byte limit of 2000. The two size constraints must be satisfied simultaneously at run time.

NVARCHAR2

The maximum length of an NVARCHAR2 column is 4000 bytes. It can hold up to 4000 characters. The actual data is subject to the maximum byte limit of 4000. The two size constraints must be satisfied simultaneously at run time.

> **See Also:** *Oracle9i Database Globalization Support Guide* for more information about the NCHAR and NVARCHAR2 datatypes

Use of Unicode Data in an Oracle Database

Unicode is an effort to have a unified encoding of every character in every language known to man. It also provides a way to represent privately-defined characters. A database column that stores Unicode can store text written in any language.

Oracle users deploying globalized applications have a strong need to store Unicode data in Oracle databases. They need a datatype which is guaranteed to be Unicode regardless of the database character set.

Oracle supports a reliable Unicode data type through NCHAR, NVARCHAR2, and NCLOB. These data types are guaranteed to be Unicode encoding and always use character length semantics. The character sets used by NCHAR/NVARCHAR2 can be either UTF8 or AL16UTF16, depending on the setting of the national character set when the database is created. These data types allow character data in Unicode to be stored in a database that may or may not use Unicode as database character set.

Implicit Type Conversion

In addition to all the implicit conversions for CHAR/VARCHAR2, Oracle also supports implicit conversion for NCHAR/NVARCHAR2. Implicit conversion between CHAR/VARCHAR2 and NCHAR/NVARCHAR2 is also supported.

LOB Character Datatypes

The LOB datatypes for character data are CLOB and NCLOB. They can store up to 4 gigabytes of character data (CLOB) or national character set data (NCLOB). LOB datatypes are intended to replace the LONG datatype functionality.

See Also: "LOB Datatypes" on page 12-13

LONG Datatype

Note: The LONG datatype is provided for backward compatibility with existing applications. In new applications, use CLOB and NCLOB datatypes for large amounts of character data.

Columns defined as LONG can store variable-length character data containing up to 2 gigabytes of information. LONG data is text data that is to be appropriately converted when moving among different systems.

LONG datatype columns are used in the data dictionary to store the text of view definitions. You can use LONG columns in SELECT lists, SET clauses of UPDATE statements, and VALUES clauses of INSERT statements.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for information about the restrictions on the LONG datatype
- "RAW and LONG RAW Datatypes" on page 12-15 for information about the LONG RAW datatype

NUMBER Datatype

The NUMBER datatype stores fixed and floating-point numbers. Numbers of virtually any magnitude can be stored and are guaranteed portable among different systems operating Oracle, up to 38 digits of precision.

The following numbers can be stored in a NUMBER column:

- Positive numbers in the range 1×10^{-130} to $9.99...9 \times 10^{125}$ with up to 38 significant digits
- Negative numbers from -1×10^{-130} to $9.99...99 \times 10^{125}$ with up to 38 significant digits
- Zero
- Positive and negative infinity (generated only by importing from an Oracle Version 5 database)

For numeric columns, you can specify the column as:

```
column_name NUMBER
```

Optionally, you can also specify a **precision** (total number of digits) and **scale** (number of digits to the right of the decimal point):

```
column_name NUMBER (precision, scale)
```

If a precision is not specified, the column stores values as given. If no scale is specified, the scale is zero.

Oracle guarantees portability of numbers with a precision equal to or less than 38 digits. You can specify a scale and no precision:

```
column_name NUMBER (*, scale)
```

In this case, the precision is 38, and the specified scale is maintained.

When you specify numeric fields, it is a good idea to specify the precision and scale. This provides extra integrity checking on input.

Table 12–1 shows examples of how data would be stored using different scale factors.

Table 12–1 How Scale Factors Affect Numeric Data Storage

Input Data	Specified As	Stored As
7,456,123.89	`NUMBER`	7456123.89
7,456,123.89	`NUMBER(*,1)`	7456123.9
7,456,123.89	`NUMBER(9)`	7456124
7,456,123.89	`NUMBER(9,2)`	7456123.89
7,456,123.89	`NUMBER(9,1)`	7456123.9
7,456,123.89	`NUMBER(6)`	(not accepted, exceeds precision)
7,456,123.89	`NUMBER(7,-2)`	7456100

If you specify a negative scale, Oracle rounds the actual data to the specified number of places to the left of the decimal point. For example, specifying `(7,-2)` means Oracle rounds to the nearest hundredths, as shown in Table 12–1.

For input and output of numbers, the standard Oracle default decimal character is a period, as in the number 1234.56. The decimal is the character that separates the integer and decimal parts of a number. You can change the default decimal character with the initialization parameter `NLS_NUMERIC_CHARACTERS`. You can also change it for the duration of a session with the `ALTER SESSION` statement. To enter numbers that do not use the current default decimal character, use the `TO_NUMBER` function.

Internal Numeric Format

Oracle stores numeric data in variable-length format. Each value is stored in scientific notation, with 1 byte used to store the exponent and up to 20 bytes to store the mantissa. The resulting value is limited to 38 digits of precision. Oracle does not store leading and trailing zeros. For example, the number 412 is stored in a format similar to 4.12×10^2, with 1 byte used to store the exponent(2) and 2 bytes used to store the three significant digits of the mantissa(`4,1,2`). Negative numbers include the sign in their length.

Taking this into account, the column size in bytes for a particular numeric data value NUMBER(*p*), where *p* is the precision of a given value, can be calculated using the following formula:

```
ROUND((length(p)+s)/2))+1
```

where *s* equals zero if the number is positive, and *s* equals 1 if the number is negative.

Zero and positive and negative infinity (only generated on import from Version 5 Oracle databases) are stored using unique representations. Zero and negative infinity each require 1 byte; positive infinity requires 2 bytes.

DATE Datatype

The DATE datatype stores point-in-time values (dates and times) in a table. The DATE datatype stores the year (including the century), the month, the day, the hours, the minutes, and the seconds (after midnight).

Oracle can store dates in the Julian era, ranging from January 1, 4712 BCE through December 31, 4712 CE (Common Era). Unless BCE ('BC' in the format mask) is specifically used, CE date entries are the default.

Oracle uses its own internal format to store dates. Date data is stored in fixed-length fields of seven bytes each, corresponding to century, year, month, day, hour, minute, and second.

For input and output of dates, the standard Oracle date format is DD-MON-YY, as follows:

```
'13-NOV-92'
```

You can change this default date format for an instance with the parameter NLS_DATE_FORMAT. You can also change it during a user session with the ALTER SESSION statement. To enter dates that are not in standard Oracle date format, use the TO_DATE function with a format mask:

```
TO_DATE ('November 13, 1992', 'MONTH DD, YYYY')
```

Oracle stores time in 24-hour format—HH:MI:SS. By default, the time in a date field is 00:00:00 A.M. (midnight) if no time portion is entered. In a time-only entry, the date portion defaults to the first day of the current month. To enter the time portion of a date, use the TO_DATE function with a format mask indicating the time portion, as in:

```
INSERT INTO birthdays (bname, bday) VALUES
    ('ANDY',TO_DATE('13-AUG-66 12:56 A.M.','DD-MON-YY HH:MI A.M.'));
```

Use of Julian Dates

Julian dates allow continuous dating by the number of days from a common reference. (The reference is 01-01-4712 years BCE, so current dates are somewhere in the 2.4 million range.) A Julian date is nominally a noninteger, the fractional part being a portion of a day. Oracle uses a simplified approach that results in integer values. Julian dates can be calculated and interpreted differently. The calculation method used by Oracle results in a seven-digit number (for dates most often used), such as 2449086 for 08-APR-93.

> **Note:** Oracle Julian dates might not be compatible with Julian dates generated by other date algorithms.

The format mask `'J'` can be used with date functions (`TO_DATE` or `TO_CHAR`) to convert date data into Julian dates. For example, the following query returns all dates in Julian date format:

```
SELECT TO_CHAR (hire_date, 'J') FROM employees;
```

You must use the `TO_NUMBER` function if you want to use Julian dates in calculations. You can use the `TO_DATE` function to enter Julian dates:

```
INSERT INTO employees (hire_date) VALUES (TO_DATE(2448921, 'J'));
```

Date Arithmetic

Oracle date arithmetic takes into account the anomalies of the calendars used throughout history. For example, the switch from the Julian to the Gregorian calendar, 15-10-1582, eliminated the previous 10 days (05-10-1582 through 14-10-1582). The year 0 does not exist.

You can enter missing dates into the database, but they are ignored in date arithmetic and treated as the next "real" date. For example, the next day after 04-10-1582 is 15-10-1582, and the day following 05-10-1582 is also 15-10-1582.

> **Note:** This discussion of date arithmetic might not apply to all countries' date standards (such as those in Asia).

Centuries and the Year 2000

Oracle stores year data with the century information. For example, the Oracle database stores 1996 or 2001, and not simply 96 or 01. The DATE datatype always stores a four-digit year internally, and all other dates stored internally in the database have four digit years. Oracle utilities such as import, export, and recovery also deal with four-digit years.

Daylight Savings Support

Oracle9*i* provides daylight savings support for DATETIME datatypes in the server. You can insert and query DATETIME values based on local time in a specific region. The DATETIME datatypes TIMESTAMP WITH TIME ZONE and TIMESTAMP WITH LOCAL TIME ZONE are time-zone aware.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for more information about centuries and date format masks
- *Oracle9i SQL Reference* for information about date format codes

Time Zones

You can include the time zone in your date/time data and provides support for fractional seconds. Three new datatypes are added to DATE, with the following differences:

Datatype	Time Zone	Fractional Seconds
DATE	No	No
TIMESTAMP	No	Yes
TIMESTAMP WITH TIME ZONE	Explicit	Yes
TIMESTAMP WITH LOCAL TIME ZONE	Relative	Yes

TIMESTAMP WITH LOCAL TIME ZONE is stored in the database time zone. When a user selects the data, the value is adjusted to the user's session time zone.

Example:

A San Francisco database has system time zone = -8:00. When a New York client (session time zone = -5:00) inserts into or selects from the San Francisco database, TIMESTAMP WITH LOCAL TIME ZONE data is adjusted as follows:

- The New York client inserts TIMESTAMP'1998-1-23 6:00:00-5:00' into a TIMESTAMP WITH LOCAL TIME ZONE column in the San Francisco database. The inserted data is stored in San Francisco as binary value 1998-1-23 3:00:00.
- When the New York client selects that inserted data from the San Francisco database, the value displayed in New York is '1998-1-23 6:00:00'.
- A San Francisco client, selecting the same data, see the value '1998-1-23 3:00:00'.

Note: To avoid unexpected results in your DML operations on datatime data, you can verify the database and session time zones by querying the built-in SQL functions DBTIMEZONE and SESSIONTIMEZONE. If the database time zone or the session time zone has not been set manually, Oracle uses the operating system time zone by default. If the operating system time zone is not a valid Oracle time zone, Oracle uses UTC as the default value.

See Also: *Oracle9i SQL Reference* for details about the syntax of creating and entering data in time stamp columns

LOB Datatypes

The LOB datatypes BLOB, CLOB, NCLOB, and BFILE enable you to store large blocks of unstructured data (such as text, graphic images, video clips, and sound waveforms) up to 4 gigabytes in size. They provide efficient, random, piece-wise access to the data. Oracle Corporation recommends that you always use LOB datatypes over LONG datatypes.

You can perform parallel queries (but not parallel DML or DDL) on LOB columns.

LOB datatypes differ from LONG and LONG RAW datatypes in several ways. For example:

- A table can contain multiple LOB columns but only one LONG column.

- A table containing one or more LOB columns can be partitioned, but a table containing a LONG column cannot be partitioned.
- The maximum size of a LOB is 4 gigabytes, but the maximum size of a LONG is 2 gigabytes.
- LOBs support random access to data, but LONGs support only sequential access.
- LOB datatypes (except NCLOB) can be attributes of a user-defined object type but LONG datatypes cannot.
- Temporary LOBs that act like local variables can be used to perform transformations on LOB data. Temporary internal LOBs (BLOBs, CLOBs, and NCLOBs) are created in the user's temporary tablespace and are independent of tables. For LONG datatypes, however, no temporary structures are available.
- Tables with LOB columns can be replicated, but tables with LONG columns cannot.

SQL statements define LOB columns in a table and LOB attributes in a user-defined object type. When defining LOBs in a table, you can explicitly specify the tablespace and storage characteristics for each LOB.

LOB datatypes can be stored inline (within a table), out-of-line (within a tablespace, using a LOB locator), or in an external file (BFILE datatypes).

With compatibility set to Oracle9*i* or higher, you can use LOBs with SQL VARCHAR operators and functions.

See Also:

- *Oracle9i SQL Reference* for a complete list of differences between the LOB datatypes and the LONG and LONG RAW datatypes
- *Oracle9i Application Developer's Guide - Large Objects (LOBs)*for more information about LOB storage and LOB locators

BLOB Datatype

The BLOB datatype stores unstructured binary data in the database. BLOBs can store up to 4 gigabytes of binary data.

BLOBs participate fully in transactions. Changes made to a BLOB value by the DBMS_LOB package, PL/SQL, or the OCI can be committed or rolled back. However, BLOB locators cannot span transactions or sessions.

CLOB and NCLOB Datatypes

The CLOB and NCLOB datatypes store up to 4 gigabytes of character data in the database. CLOBs store database character set data and NCLOBs store Unicode national character set data. For varying-width database character sets, the CLOB value is stored in the database using the two-byte Unicode character set, which has a fixed width. Oracle translates the stored Unicode value to the character set requested on the client or on the server, which can be fixed-width or varying width. When you insert data into a CLOB column using a varying-width character set, Oracle converts the data into Unicode before storing it in the database.

CLOBs and NCLOBs participate fully in transactions. Changes made to a CLOB or NCLOB value by the DBMS_LOB package, PL/SQL, or the OCI can be committed or rolled back. However, CLOB and NCLOB locators cannot span transactions or sessions.

You cannot create an object type with NCLOB attributes, but you can specify NCLOB parameters in a method for an object type.

> **See Also:** *Oracle9i Database Globalization Support Guide* for more information about national character set data and the Unicode character set

BFILE Datatype

The BFILE datatype stores unstructured binary data in operating-system files outside the database. A BFILE column or attribute stores a file locator that points to an external file containing the data. BFILEs can store up to 4 gigabytes of data.

BFILEs are read-only; you cannot modify them. They support only random (not sequential) reads, and they do not participate in transactions. The underlying operating system must maintain the file integrity, security, and durability for BFILEs. The database administrator must ensure that the file exists and that Oracle processes have operating-system read permissions on the file.

RAW and LONG RAW Datatypes

Note: The LONG RAW datatype is provided for backward compatibility with existing applications. For new applications, use the BLOB and BFILE datatypes for large amounts of binary data.

The RAW and LONG RAW datatypes are used for data that is not to be interpreted (not converted when moving data between different systems) by Oracle. These datatypes are intended for binary data or byte strings. For example, LONG RAW can be used to store graphics, sound, documents, or arrays of binary data. The interpretation depends on the use.

RAW is a variable-length datatype like the VARCHAR2 character datatype, except Oracle Net Services (which connects user sessions to the instance) and the Import and Export utilities do not perform character conversion when transmitting RAW or LONG RAW data. In contrast, Oracle Net Services and Import/Export automatically convert CHAR, VARCHAR2, and LONG data between the database character set and the user session character set (set by the NLS_LANGUAGE parameter of the ALTER SESSION statement), if the two character sets are different.

When Oracle automatically converts RAW or LONG RAW data to and from CHAR data, the binary data is represented in hexadecimal form with one hexadecimal character representing every four bits of RAW data. For example, one byte of RAW data with bits 11001011 is displayed and entered as 'CB.'

LONG RAW data cannot be indexed, but RAW data can be indexed.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for information about other restrictions on the LONG RAW datatype

ROWID and UROWID Datatypes

Oracle uses a ROWID datatype to store the address *(***rowid***)* of every row in the database.

- **Physical rowids** store the addresses of rows in ordinary tables (excluding index-organized tables), clustered tables, table partitions and subpartitions, indexes, and index partitions and subpartitions.
- **Logical rowids** store the addresses of rows in index-organized tables.

A single datatype called the **universal rowid**, or UROWID, supports both logical and physical rowids, as well as rowids of foreign tables such as non-Oracle tables accessed through a gateway.

A column of the UROWID datatype can store all kinds of rowids. The value of the COMPATIBLE initialization parameter must be set to 8.1 or higher to use UROWID columns.

> **See Also:** "Rowids in Non-Oracle Databases" on page 12-23

The ROWID Pseudocolumn

Each table in an Oracle database internally has a **pseudocolumn** named ROWID. This pseudocolumn is not evident when listing the structure of a table by executing a SELECT * FROM ... statement, or a DESCRIBE ... statement using SQL*Plus, nor does the pseudocolumn take up space in the table. However, each row's address can be retrieved with a SQL query using the reserved word ROWID as a column name, for example:

```
SELECT ROWID, last_name FROM employees;
```

You cannot set the value of the pseudocolumn ROWID in INSERT or UPDATE statements, and you cannot delete a ROWID value. Oracle uses the ROWID values in the pseudocolumn ROWID internally for the construction of indexes.

You can reference rowids in the pseudocolumn ROWID like other table columns (used in SELECT lists and WHERE clauses), but rowids are not stored in the database, nor are they database data. However, you can create tables that contain columns having the ROWID datatype, although Oracle does not guarantee that the values of such columns are valid rowids. The user must ensure that the data stored in the ROWID column truly is a valid ROWID.

See Also: "How Rowids Are Used" on page 12-21

Physical Rowids

Physical rowids provide the fastest possible access to a row of a given table. They contain the physical address of a row (down to the specific block) and allow you to retrieve the row in a single block access. Oracle guarantees that as long as the row exists, its rowid does not change. These performance and stability qualities make rowids useful for applications that select a set of rows, perform some operations on them, and then access some of the selected rows again, perhaps with the purpose of updating them.

Every row in a nonclustered table is assigned a unique rowid that corresponds to the physical address of a row's row piece (or the initial row piece if the row is chained among multiple row pieces). In the case of clustered tables, rows in different tables that are in the same data block can have the same rowid.

A row's assigned rowid remains unchanged unless the row is exported and imported using the Import and Export utilities. When you delete a row from a table and then commit the encompassing transaction, the deleted row's associated rowid can be assigned to a row inserted in a subsequent transaction.

A physical rowid datatype has one of two formats:

- The **extended rowid** format supports tablespace-relative data block addresses and efficiently identifies rows in partitioned tables and indexes as well as nonpartitioned tables and indexes. Tables and indexes created by an Oracle8*i* (or higher) server always have extended rowids.
- A **restricted rowid** format is also available for backward compatibility with applications developed with Oracle7 or earlier releases.

Extended Rowids

Extended rowids use a base 64 encoding of the physical address for each row selected. The encoding characters are `A-Z`, `a-z`, `0-9`, `+`, and `/`. For example, the following query:

```
SELECT ROWID, last_name FROM employees WHERE department_id = 20;
```

can return the following row information:

```
ROWID              LAST_NAME
------------------ ----------
AAAAaoAATAAABrXAAA BORTINS
AAAAaoAATAAABrXAAE RUGGLES
AAAAaoAATAAABrXAAG CHEN
AAAAaoAATAAABrXAAN BLUMBERG
```

An extended rowid has a four-piece format, `OOOOOOFFFBBBBBBRRR`:

- `OOOOOO`: The **data object number** that identifies the database segment (`AAAAao` in the example). Schema objects in the same segment, such as a cluster of tables, have the same data object number.
- `FFF`: The tablespace-relative **datafile number** of the datafile that contains the row (file `AAT` in the example).
- `BBBBBB`: The **data block** that contains the row (block `AAABrX` in the example). Block numbers are relative to their datafile, **not** tablespace. Therefore, two rows with identical block numbers could reside in two different datafiles of the same tablespace.
- `RRR`: The **row** in the block.

You can retrieve the data object number from data dictionary views `USER_OBJECTS`, `DBA_OBJECTS`, and `ALL_OBJECTS`. For example, the following query returns the data object number for the `employees` table in the `SCOTT` schema:

```
SELECT DATA_OBJECT_ID FROM DBA_OBJECTS
    WHERE OWNER = 'SCOTT' AND OBJECT_NAME = 'EMPLOYEES';
```

You can also use the DBMS_ROWID package to extract information from an extended rowid or to convert a rowid from extended format to restricted format (or vice versa).

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for information about the DBMS_ROWID package

Restricted Rowids

Restricted rowids use a binary representation of the physical address for each row selected. When queried using SQL*Plus, the binary representation is converted to a VARCHAR2/hexadecimal representation. The following query:

```
SELECT ROWID, last_name FROM employees
    WHERE department_id = 30;
```

can return the following row information:

```
ROWID              ENAME
------------------ ----------
00000DD5.0000.0001 KRISHNAN
00000DD5.0001.0001 ARBUCKLE
00000DD5.0002.0001 NGUYEN
```

As shown, a restricted rowid's VARCHAR2/hexadecimal representation is in a three-piece format, **block.row.file**:

- The **data block** that contains the row (block DD5 in the example). Block numbers are relative to their datafile, **not** tablespace. Therefore, two rows with identical block numbers could reside in two different datafiles of the same tablespace.
- The **row** in the block that contains the row (rows 0, 1, 2 in the example). Row numbers of a given block always start with 0.
- The **datafile** that contains the row (file 1 in the example). The first datafile of every database is always 1, and file numbers are unique within a database.

Examples of Rowid Use

You can use the function SUBSTR to break the data in a rowid into its components. For example, you can use SUBSTR to break an extended rowid into its four components (database object, file, block, and row):

```
SELECT ROWID,
```

```
       SUBSTR(ROWID,1,6) "OBJECT",
       SUBSTR(ROWID,7,3) "FIL",
       SUBSTR(ROWID,10,6) "BLOCK",
       SUBSTR(ROWID,16,3) "ROW"
       FROM products;

ROWID              OBJECT  FIL  BLOCK   ROW
-----------------  ------  ---  ------  ----
AAAA8mAALAAAAQkAAA  AAAA8m  AAL  AAAAQk  AAA
AAAA8mAALAAAAQkAAF  AAAA8m  AAL  AAAAQk  AAF
AAAA8mAALAAAAQkAAI  AAAA8m  AAL  AAAAQk  AAI
```

Or you can use SUBSTR to break a restricted rowid into its three components (block, row, and file):

```
SELECT ROWID, SUBSTR(ROWID,15,4) "FILE",
       SUBSTR(ROWID,1,8) "BLOCK",
       SUBSTR(ROWID,10,4) "ROW"
       FROM products;

ROWID              FILE  BLOCK     ROW
-----------------  ----  --------  ----
00000DD5.0000.0001  0001  00000DD5  0000
00000DD5.0001.0001  0001  00000DD5  0001
00000DD5.0002.0001  0001  00000DD5  0002
```

Rowids can be useful for revealing information about the physical storage of a table's data. For example, if you are interested in the physical location of a table's rows (such as for table striping), the following query of an extended rowid tells how many datafiles contain rows of a given table:

```
SELECT COUNT(DISTINCT(SUBSTR(ROWID,7,3))) "FILES" FROM tablename;

   FILES
--------
       2
```

See Also:

- *Oracle9i SQL Reference*
- *PL/SQL User's Guide and Reference*
- *Oracle9i Database Performance Tuning Guide and Reference*

for more examples using rowids

How Rowids Are Used

Oracle uses rowids internally for the construction of indexes. Each key in an index is associated with a rowid that points to the associated row's address for fast access. End users and application developers can also use rowids for several important functions:

- Rowids are the fastest means of accessing particular rows.
- Rowids can be used to see how a table is organized.
- Rowids are unique identifiers for rows in a given table.

Before you use rowids in DML statements, they should be verified and guaranteed not to change. The intended rows should be locked so they cannot be deleted. Under some circumstances, requesting data with an invalid rowid could cause a statement to fail.

You can also create tables with columns defined using the `ROWID` datatype. For example, you can define an exception table with a column of datatype `ROWID` to store the rowids of rows in the database that violate integrity constraints. Columns defined using the `ROWID` datatype behave like other table columns: values can be updated, and so on. Each value in a column defined as datatype `ROWID` requires six bytes to store pertinent column data.

Logical Rowids

Rows in index-organized tables do not have permanent physical addresses—they are stored in the index leaves and can move within the block or to a different block as a result of insertions. Therefore their row identifiers cannot be based on physical addresses. Instead, Oracle provides index-organized tables with logical row identifiers, called **logical rowids**, that are based on the table's primary key. Oracle uses these logical rowids for the construction of secondary indexes on index-organized tables.

Each logical rowid used in a secondary index can include a **physical guess**, which identifies the block location of the row in the index-organized table at the time the guess was made; that is, when the secondary index was created or rebuilt.

Oracle can use guesses to probe into the leaf block directly, bypassing the full key search. This ensures that rowid access of nonvolatile index-organized tables gives comparable performance to the physical rowid access of ordinary tables. In a volatile table, however, if the guess becomes stale the probe can fail, in which case a primary key search must be performed.

The values of two logical rowids are considered equal if they have the same primary key values but different guesses.

Comparison of Logical Rowids with Physical Rowids

Logical rowids are similar to the physical rowids in the following ways:

- Logical rowids are accessible through the `ROWID` pseudocolumn.

  You can use the `ROWID` pseudocolumn to select logical rowids from an index-organized table. The `SELECT ROWID` statement returns an opaque structure, which internally consists of the table's primary key and the physical guess (if any) for the row, along with some control information.

  You can access a row using predicates of the form `WHERE ROWID` = *`value`*, where *`value`* is the opaque structure returned by `SELECT ROWID`.

- Access through the logical rowid is the fastest way to get to a specific row, although it can require more than one block access.
- A row's logical rowid does not change as long as the primary key value does not change. This is less stable than the physical rowid, which stays immutable through all updates to the row.
- Logical rowids can be stored in a column of the `UROWID` datatype

One difference between physical and logical rowids is that logical rowids cannot be used to see how a table is organized.

Note: An opaque type is one whose internal structure is not known to the database. The database provides storage for the type. The type designer can provide access to the contents of the type by implementing functions, typically 3GL routines.

See Also: "ROWID and UROWID Datatypes" on page 12-16

Guesses in Logical Rowids

When a row's physical location changes, the logical rowid remains valid even if it contains a guess, although the guess could become stale and slow down access to the row. Guess information cannot be updated dynamically. For secondary indexes on index-organized tables, however, you can rebuild the index to obtain fresh guesses. Note that rebuilding a secondary index on an index-organized table involves reading the base table, unlike rebuilding an index on an ordinary table.

Collect index statistics with the DBMS_STATS package or ANALYZE statement to keep track of the staleness of guesses, so Oracle does not use them unnecessarily. This is particularly important for applications that store rowids with guesses persistently in a UROWID column, then retrieve the rowids later and use them to fetch rows.

When you collect index statistics with the DBMS_STATS package or ANALYZE statement, Oracle checks whether the existing guesses are still valid and records the percentage of stale / valid guesses in the data dictionary. After you rebuild a secondary index (recomputing the guesses), collect index statistics again.

In general, logical rowids without guesses provide the fastest possible access for a highly volatile table. If a table is static or if the time between getting a rowid and using it is sufficiently short to make row movement unlikely, logical rowids with guesses provide the fastest access.

> **See Also:** *Oracle9i Database Performance Tuning Guide and Reference* for more information about collecting statistics

Rowids in Non-Oracle Databases

Oracle database applications can be run against non-Oracle database servers using SQL*Connect or the Oracle Transparent Gateway. In such cases, the format of rowids varies according to the characteristics of the non-Oracle system. Furthermore, no standard translation to VARCHAR2/hexadecimal format is available. Programs can still use the ROWID datatype. However, they must use a nonstandard translation to hexadecimal format of length up to 256 bytes.

Rowids of a non-Oracle database can be stored in a column of the UROWID datatype.

> **See Also:**
>
> - *Oracle Call Interface Programmer's Guide* for further details on handling rowids with non-Oracle systems
> - "ROWID and UROWID Datatypes" on page 12-16

ANSI, DB2, and SQL/DS Datatypes

SQL statements that create tables and clusters can also use ANSI datatypes and datatypes from IBM's products SQL / DS and DB2. Oracle recognizes the ANSI or IBM datatype name that differs from the Oracle datatype name, records it as the name of the datatype of the column, and then stores the column's data in an Oracle datatype based on the conversions shown in Table 12–2 and Table 12–3.

Table 12–2 ANSI Datatypes Converted to Oracle Datatypes

ANSI SQL Datatype	Oracle Datatype
CHARACTER(n)	CHAR(n)
CHAR(n)	
CHARACTER VARYING(n)	VARCHAR(n)
CHAR VARYING(n)	
NATIONAL CHARACTER(n)	NCHAR(n)
NATIONAL CHAR(n)	
NCHAR(n)	
NATIONAL CHARACTER VARYING(n)	NVARCHAR2(n)
NATIONAL CHAR VARYING(n)	
NCHAR VARYING(n)	
NUMERIC(p,s)	NUMBER(p,s)
DECIMAL(p,s)[a]	
INTEGER	NUMBER(38)
INT	
SMALLINT	
FLOAT(b)[b]	NUMBER
DOUBLE PRECISION[c]	
REAL[d]	

[a]The NUMERIC and DECIMAL datatypes can specify only fixed-point numbers. For these datatypes, s defaults to 0.

[b]The FLOAT datatype is a floating-point number with a binary precision b. The default precision for this datatype is 126 binary, or 38 decimal.

[c]The DOUBLE PRECISION datatype is a floating-point number with binary precision 126.

[d]The REAL datatype is a floating-point number with a binary precision of 63, or 18 decimal.

Table 12–3 SQL/DS and DB2 Datatypes Converted to Oracle Datatypes

SQL/DS or DB2 Datatype	Oracle Datatype
CHARACTER(n)	CHAR(n)
VARCHAR(n)	VARCHAR(n)
LONG VARCHAR(n)	LONG
DECIMAL(p,s)[a]	NUMBER(p,s)
INTEGER	NUMBER(38)
SMALLINT	
FLOAT(b)[b]	NUMBER

[a]The DECIMAL datatype can specify only fixed-point numbers. For this datatype, *s* defaults to 0.

[b]The FLOAT datatype is a floating-point number with a binary precision *b*. The default precision for this datatype is 126 binary, or 38 decimal.

Do not define columns with the following SQL/DS and DB2 datatypes, because they have no corresponding Oracle datatype:

- GRAPHIC
- LONG VARGRAPHIC
- VARGRAPHIC
- TIME

Note that data of type TIME can also be expressed as Oracle DATE data.

XML Datatypes

Oracle provides the XMLType datatype to handle XML data.

XMLType Datatype

XMLType can be used like any other user-defined type. XMLType can be used as the datatype of columns in tables and views. Variables of XMLType can be used in PL/SQL stored procedures as parameters, return values, and so on. You can also use XMLType in PL/SQL, SQL and Java, and through JDBC and OCI.

A number of useful functions that operate on XML content have been provided. Many of these are provided both as SQL functions and as member functions of `XMLType`. For example, function extract() extracts a specific node(s) from an `XMLType` instance.

You can use `XMLType` in SQL queries in the same way as any other user-defined datatypes in the system.

See Also:

- *Oracle9i XML Developer's Kits Guide - XDK*
- *Oracle9i XML Database Developer's Guide - Oracle XML DB*
- *Oracle9i Application Developer's Guide - Advanced Queuing* for information about using `XMLType` with Oracle Advanced Queuing
- Chapter 1, "Introduction to the Oracle Server"

URI Datatypes

A URI, or uniform resource identifier, is a generalized kind of URL. Like a URL, it can reference any document, and can reference a specific part of a document. It is more general than a URL because it has a powerful mechanism for specifying the relevant part of the document.

By using `UriType`, you can do the following:

- Create table columns that point to data inside or outside the database.
- Query the database columns using functions provided by `UriType`.

See Also: *Oracle9i XML Database Developer's Guide - Oracle XML DB*

Data Conversion

In some cases, Oracle supplies data of one datatype where it expects data of a different datatype. This is allowed when Oracle can automatically convert the data to the expected datatype. These are some of the functions used:

```
TO_NUMBER()
TO_CHAR()
TO_NCHAR()
TO_DATE()
```

```
TO_CLOB()
TO_NCLOB()
CHARTOROWID()
ROWIDTOCHAR()
ROWIDTONCHAR()
HEXTORAW()
RAWTOHEX()
RAWTONHEX()
REFTOHEX()
```

See Also: *Oracle9i SQL Reference* for the rules for implicit datatype conversions

13

Object Datatypes and Object Views

Object types and other user-defined datatypes let you define datatypes that model the structure and behavior of the data in their applications. An object view is a virtual object table.

This chapter contains the following major sections:

Introduction to Object Datatypes

Relational database management systems (RDBMSs) are the standard tool for managing business data. They provide reliable access to huge amounts of data for millions of businesses around the world every day.

Oracle is an **object-relational** database management system (ORDBMS), which means that users can define additional kinds of data—specifying both the structure of the data and the ways of operating on it—and use these types within the relational model. This approach adds value to the data stored in a database. Object datatypes make it easier for application developers to work with complex data such as images, audio, and video. Object types store structured business data in its natural form and allow applications to retrieve it that way. For that reason, they work efficiently with applications developed using object-oriented programming techniques.

Complex Data Models

The Oracle server lets you define complex business models in SQL and make them part of your database schema. Applications that manage and share your data need only contain the application logic, not the data logic.

Complex Data Model Example

For example, your firm might use purchase orders to organize its purchasing, accounts payable, shipping, and accounts receivable functions.

A purchase order contains an associated supplier or customer and an indefinite number of line items. In addition, applications often need dynamically computed status information about purchase orders. For example, you may need the current value of the shipped or unshipped line items.

Later sections of this chapter show how you can define a schema object, called an **object type**, that serves as a template for all purchase order data in your applications. An object type specifies the elements, called **attributes**, that make up a structured data unit, such as a purchase order. Some attributes, such as the list of line items, can be other structured data units. The object type also specifies the operations, called **methods**, you can perform on the data unit, such as determining the total value of a purchase order.

You can create purchase orders that match the template and store them in table columns, just as you would numbers or dates.

You can also store purchase orders in **object tables**, where each row of the table corresponds to a single purchase order and the table columns are the purchase order's attributes.

Because the logic of the purchase order's structure and behavior is in your schema, your applications do not need to know the details and do not have to keep up with most changes.

Oracle uses schema information about object types to achieve substantial transmission efficiencies. A client-side application can request a purchase order from the server and receive all the relevant data in a single transmission. The application can then, without knowing storage locations or implementation details, navigate among related data items without further transmissions from the server.

Multimedia Datatypes

Many efficiencies of database systems arise from their optimized management of basic datatypes like numbers, dates, and characters. Facilities exist for comparing values, determining their distributions, building efficient indexes, and performing other optimizations.

Text, video, sound, graphics, and spatial data are examples of important business entities that do not fit neatly into those basic types. Oracle Enterprise Edition supports modeling and implementation of these complex datatypes.

Object Datatype Categories

There are two categories of object datatypes:

- Object types
- Collection types

Object datatypes use the built-in datatypes and other user-defined datatypes as the building blocks for datatypes that model the structure and behavior of data in applications.

Object types are schema objects. Their use is subject to the same kinds of administrative control as other schema objects.

See Also:

- Chapter 12, "Native Datatypes"
- *Oracle9i Application Developer's Guide - Object-Relational Features*

Object Types

Object types are abstractions of the real-world entities—for example, purchase orders—that application programs deal with. An object type is a schema object with three kinds of components:

- A **name**, which serves to identify the object type uniquely within that schema
- **Attributes**, which model the structure and state of the real-world entity. Attributes are built-in types or other user-defined types.
- **Methods**, which are functions or procedures written in PL/SQL or Java and stored in the database, or written in a language such as C and stored externally. Methods implement operations the application can perform on the real-world entity.

An object type is a template. A structured data unit that matches the template is called an **object**.

Purchase Order Example

Here is an example of how you can define object types called `external_person`, `lineitem`, and `purchase_order`.

The object types `external_person` and `lineitem` have attributes of built-in types. The object type `purchase_order` has a more complex structure, which closely matches the structure of real purchase orders.

The attributes of `purchase_order` are `id`, `contact`, and `lineitems`. The attribute `contact` is an object, and the attribute `lineitems` is a nested table.

```
CREATE TYPE external_person AS OBJECT (
  name        VARCHAR2(30),
  phone       VARCHAR2(20) );

CREATE TYPE lineitem AS OBJECT (
  item_name   VARCHAR2(30),
  quantity    NUMBER,
  unit_price  NUMBER(12,2) );

CREATE TYPE lineitem_table AS TABLE OF lineitem;
```

```
CREATE TYPE purchase_order AS OBJECT (
  id          NUMBER,
  contact     external_person,
  lineitems   lineitem_table,

  MEMBER FUNCTION
  get_value   RETURN NUMBER );
```

This is a simplified example. It does not show how to specify the body of the method `get_value`, nor does it show the full complexity of a real purchase order.

An object type is a template. Defining it does not result in storage allocation. You can use `lineitem`, `external_person`, or `purchase_order` in SQL statements in most of the same places you can use types like `NUMBER` or `VARCHAR2`.

For example, you can define a relational table to keep track of your contacts:

```
CREATE TABLE contacts (
  contact     external_person
  date        DATE );
```

The `contacts` table is a relational table with an object type defining one of its columns. Objects that occupy columns of relational tables are called **column objects**.

See Also:

- "Nested Tables Description" on page 13-12
- "Row Objects and Column Objects" on page 13-8
- *Oracle9i Application Developer's Guide - Object-Relational Features* for a complete purchase order example

Types of Methods

Methods of an object type model the behavior of objects. The methods of an object type broadly fall into these categories:

- A **Member** method is a function or a procedure that always has an implicit `SELF` parameter as its first parameter, whose type is the containing object type.
- A **Static** method is a function or a procedure that does not have an implicit `SELF` parameter. Such methods can be invoked by qualifying the method with the type name, as in `TYPE_NAME.METHOD()`. Static methods are useful for specifying user-defined constructors or cast methods.

- **Comparison** methods are used for comparing instances of objects.

Oracle supports the choice of implementing type methods in PL/SQL, Java, and C.

In the example, `purchase_order` has a method named `get_value`. Each purchase order object has its own `get_value` method. For example, if *x* and *y* are PL/SQL variables that hold purchase order objects and *w* and *z* are variables that hold numbers, the following two statements can leave *w* and *z* with different values:

```
w = x.get_value();
z = y.get_value();
```

After those statements, *w* has the value of the purchase order referred to by variable *x*; *z* has the value of the purchase order referred to by variable *y*.

The term *x*`.get_value` () is an invocation of the method `get_value`. Method definitions can include parameters, but `get_value` does not need them, because it finds all of its arguments among the attributes of the object to which its invocation is tied. That is, in the first of the sample statements, it computes its value using the attributes of purchase order *x*. In the second it computes its value using the attributes of purchase order *y*. This is called the **selfish style** of method invocation.

Every object type also has one implicitly defined method that is not tied to specific objects, the object type's constructor method.

Object Type Constructor Methods Every object type has a system-defined **constructor method**; that is, a method that makes a new object according to the object type's specification. The name of the constructor method is the name of the object type. Its parameters have the names and types of the object type's attributes. The constructor method is a function. It returns the new object as its value.

For example, the expression:

```
purchase_order(
  1000376,
  external_person ("John Smith","1-800-555-1212"),
  NULL )
```

represents a purchase order object with the following attributes:

```
id         1000376
contact    external_person("John Smith","1-800-555-1212")
lineitems  NULL
```

The expression `external_person ("John Smith", "1-800-555-1212")` is an invocation of the constructor function for the object type `external_person`. The object that it returns becomes the contact attribute of the purchase order.

You can also define your own constructor functions to use in place of the constructor functions that the system implicitly defines for every object type.

> **See Also:** *Oracle9i Application Developer's Guide - Object-Relational Features*

Comparison Methods Methods play a role in comparing objects. Oracle has facilities for comparing two data items of a given built-in type (for example, two numbers), and determining whether one is greater than, equal to, or less than the other. Oracle cannot, however, compare two items of an arbitrary user-defined type without further guidance from the definer. Oracle provides two ways to define an order relationship among objects of a given object type: map methods and order methods.

Map methods use Oracle's ability to compare built-in types. Suppose, for example, that you have defined an object type called `rectangle`, with attributes `height` and `width`. You can define a map method area that returns a number, namely the product of the rectangle's `height` and `width` attributes. Oracle can then compare two rectangles by comparing their areas.

Order methods are more general. An order method uses its own internal logic to compare two objects of a given object type. It returns a value that encodes the order relationship. For example, it could return -1 if the first is smaller, 0 if they are equal, and 1 if the first is larger.

Suppose, for example, that you have defined an object type called `address`, with attributes `street`, `city`, `state`, and `zip`. **Greater than** and **less than** may have no meaning for addresses in your application, but you may need to perform complex computations to determine when two addresses are equal.

In defining an object type, you can specify either a map method or an order method for it, but not both. If an object type has no comparison method, Oracle cannot determine a greater than or less than relationship between two objects of that type. It can, however, attempt to determine whether two objects of the type are equal.

Oracle compares two objects of a type that lacks a comparison method by comparing corresponding attributes:

- If all the attributes are non-null and equal, Oracle reports that the objects are equal.
- If there is an attribute for which the two objects have unequal non-null values, Oracle reports them unequal.
- Otherwise, Oracle reports that the comparison is not available (null).

> **See Also:** *Oracle9i Application Developer's Guide - Object-Relational Features* for examples of how to specify and use comparison methods

Object Tables

An **object table** is a special kind of table that holds objects and provides a relational view of the attributes of those objects.

For example, the following statement defines an object table for objects of the `external_person` type defined earlier:

```
CREATE TABLE external_person_table OF external_person;
```

Oracle lets you view this table in two ways:

- A single column table in which each entry is an `external_person` object.
- A multicolumn table in which each of the attributes of the object type `external_person`, namely `name` and `phone`, occupies a column

For example, you can run the following instructions:

```
INSERT INTO external_person_table VALUES (
       "John Smith",
       "1-800-555-1212" );

SELECT VALUE(p) FROM external_person_table p
        WHERE p.name = "John Smith";
```

The first instruction inserts an `external_person` object into `external_person_table` as a multicolumn table. the second selects from `external_person_table` as a single column table.

Row Objects and Column Objects Objects that appear in object tables are called **row objects**. Objects that appear in table columns or as attributes of other objects are called **column objects**.

Object Identifiers

Every row object in an object table has an associated logical object identifier (OID). Oracle assigns a unique system-generated identifier of length 16 bytes as the OID for each row object by default.

The OID column of an object table is a hidden column. Although the OID value in itself is not very meaningful to an object-relational application, Oracle uses this value to construct object references to the row objects. Applications need to be concerned with only object references that are used for fetching and navigating objects.

The purpose of the OID for a row object is to uniquely identify it in an object table. To do this Oracle implicitly creates and maintains an index on the OID column of an object table. The system-generated unique identifier has many advantages, among which are the unambiguous identification of objects in a distributed and replicated environment.

Primary-Key Based Object Identifiers For applications that do not require the functionality provided by globally unique system-generated identifiers, storing 16 extra bytes with each object and maintaining an index on it may not be efficient. Oracle allows the option of specifying the primary key value of a row object as the object identifier for the row object.

Primary-key based identifiers also have the advantage of enabling a more efficient and easier loading of the object table. By contrast, system-generated object identifiers need to be remapped using some user-specified keys, especially when references to them are also stored persistently.

Object Views Description

An object view is a virtual object table. Its rows are row objects. Oracle materializes object identifiers, which it does not store persistently, from primary keys in the underlying table or view.

See Also: "Introduction to Object Views" on page 13-23

REFs

In the relational model, foreign keys express many-to-one relationships. Oracle object types provide a more efficient means of expressing many-to-one relationships when the "one" side of the relationship is a row object.

Oracle provides a built-in datatype called REF to encapsulate references to row objects of a specified object type. From a modeling perspective, REFs provide the

ability to capture an association between two row objects. Oracle uses object identifiers to construct such REFs.

You can use a REF to examine or update the object it refers to. You can also use a REF to obtain a copy of the object it refers to. The only changes you can make to a REF are to replace its contents with a reference to a different object of the same object type or to assign it a null value.

Scoped REFs In declaring a column type, collection element, or object type attribute to be a REF, you can constrain it to contain only references to a specified object table. Such a REF is called a **scoped** REF. Scoped REFs require less storage space and allow more efficient access than unscoped REFs.

Dangling REFs It is possible for the object identified by a REF to become unavailable through either deletion of the object or a change in privileges. Such a REF is called **dangling**. Oracle SQL provides a predicate (called IS DANGLING) to allow testing REFs for this condition.

Dereference REFs Accessing the object referred to by a REF is called **dereferencing** the REF. Oracle provides the DEREF operator to do this. Dereferencing a dangling REF results in a null object.

Oracle provides **implicit dereferencing** of REFs. For example, consider the following:

```
CREATE TYPE person AS OBJECT (
  name    VARCHAR2(30),
  manager REF person );
```

If *x* represents an object of type PERSON, then the expression:

```
x.manager.name
```

represents a string containing the name attribute of the person object referred to by the manager attribute of *x*. The previous expression is a shortened form of:

```
y.name, where y = DEREF(x.manager)
```

Obtain REFs You can obtain a REF to a row object by selecting the object from its object table and applying the REF operator. For example, you can obtain a REF to the purchase order with identification number 1000376 as follows:

```
DECLARE OrderRef REF to purchase_order;

SELECT REF(po) INTO OrderRef
               FROM purchase_order_table po
               WHERE po.id = 1000376;
```

See Also: *Oracle9i Application Developer's Guide - Object-Relational Features* for examples of how to use REFs

Collection Types

Each collection type describes a data unit made up of an indefinite number of elements, all of the same datatype. The collection types are **array types** and **table types**.

Array types and table types are schema objects. The corresponding data units are called **VARRAYs** and **nested tables**. When there is no danger of confusion, we often refer to the collection types as VARRAYs and nested tables.

Collection types have constructor methods. The name of the constructor method is the name of the type, and its argument is a comma separated list of the new collection's elements. The constructor method is a function. It returns the new collection as its value.

An expression consisting of the type name followed by empty parentheses represents a call to the constructor method to create an empty collection of that type. An empty collection is different from a null collection.

VARRAYs

An **array** is an ordered set of data **elements**. All elements of a given array are of the same datatype. Each element has an **index**, which is a number corresponding to the element's position in the array.

The number of elements in an array is the **size** of the array. Oracle allows arrays to be of variable size, which is why they are called VARRAYs. You must specify a maximum size when you declare the array type.

For example, the following statement declares an array type:

```
CREATE TYPE prices AS VARRAY(10) OF NUMBER(12,2);
```

The VARRAYs of type `prices` have no more than 10 elements, each of datatype `NUMBER(12,2)`.

Creating an array type does not allocate space. It defines a datatype, which you can use as:

- The datatype of a column of a relational table
- An object type attribute
- A PL/SQL variable, parameter, or function return type.

A VARRAY is normally stored in line; that is, in the same tablespace as the other data in its row. If it is sufficiently large, however, Oracle stores it as a BLOB.

> **See Also:** *Oracle9i Application Developer's Guide - Object-Relational Features* for more information about using VARRAYs

Nested Tables Description

A **nested table** is an unordered set of data **elements**, all of the same datatype. It has a single column, and the type of that column is a built-in type or an object type. If an object type, the table can also be viewed as a multicolumn table, with a column for each attribute of the object type. If compatibility is set to Oracle9*i* or higher, nested tables can contain other nested tables.

For example, in the purchase order example, the following statement declares the table type used for the nested tables of line items:

```
CREATE TYPE lineitem_table AS TABLE OF lineitem;
```

A table type definition does not allocate space. It defines a type, which you can use as:

- The datatype of a column of a relational table
- An object type attribute
- A PL/SQL variable, parameter, or function return type

When a table type appears as the type of a column in a relational table or as an attribute of the underlying object type of an object table, Oracle stores all of the nested table data in a single table, which it associates with the enclosing relational or object table. For example, the following statement defines an object table for the object type `purchase_order`:

```
CREATE TABLE purchase_order_table OF purchase_order
   NESTED TABLE lineitems STORE AS lineitems_table;
```

The second line specifies `lineitems_table` as the storage table for the `lineitems` attributes of all of the `purchase_order` objects in `purchase_order_table`.

A convenient way to access the elements of a nested table individually is to use a nested cursor.

See Also:

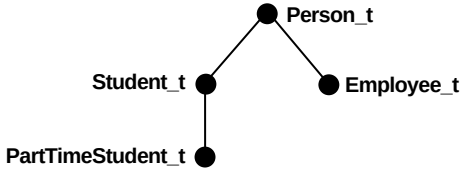
- *Oracle9i Database Reference* for information about nested cursors
- *Oracle9i Application Developer's Guide - Object-Relational Features* for more information about using nested tables

Type Inheritance

An object type can be created as a subtype of an existing object type. A single inheritance model is supported: the subtype can be derived from only one parent type. A type inherits all the attributes and methods of its direct supertype. It can add new attributes and methods, and it can override any of the inherited methods.

Figure 13–1 illustrates two subtypes, `Student_t` and `Employee_t`, created under `Person_t`.

Figure 13–1 A Type Hierarchy



Furthermore, a subtype can itself be refined by defining another subtype under it, thus building up type hierarchies. In the preceding diagram, `PartTimeStudent_t` is derived from subtype `Student_t`.

FINAL and NOT FINAL Types

A type declaration must have the `NOT FINAL` keyword, if you want it to have subtypes. The default is that the type is `FINAL`; that is, no subtypes can be created for the type. This allows for backward compatibility.

Example of Creating a NOT FINAL Object Type

```
CREATE TYPE Person_t AS OBJECT
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100)) NOT FINAL;
```

`Person_t` is declared to be a `NOT FINAL` type. This enables definition of subtypes of `Person_t`.

`FINAL` types can be altered to be `NOT FINAL`. In addition, `NOT FINAL` types with no subtypes can be altered to be `FINAL`.

NOT INSTANTIABLE Types and Methods

A type can be declared to be `NOT INSTANTIABLE`. This implies that there is no constructor (default or user-defined) for the type. Thus, it is not possible to construct instances of this type. The typical use would be define instantiable subtypes for such a type, as follows:

```
CREATE TYPE Address_t AS OBJECT(...) NOT INSTANTIABLE NOT FINAL;
CREATE TYPE USAddress_t UNDER Address_t(...);
CREATE TYPE IntlAddress_t UNDER Address_t(...);
```

A method of a type can be declared to be `NOT INSTANTIABLE`. Declaring a method as `NOT INSTANTIABLE` means that the type is not providing an implementation for that method. Furthermore, a type that contains any non-instantiable methods must necessarily be declared `NOT INSTANTIABLE`.

For example:

```
CREATE TYPE T AS OBJECT
(
  x NUMBER,
  NOT INSTANTIABLE MEMBER FUNCTION func1() RETURN NUMBER
) NOT INSTANTIABLE;
```

A subtype of a `NOT INSTANTIABLE` type can override any of the non-instantiable methods of the supertype and provide concrete implementations. If there are any non-instantiable methods remaining, the subtype must also necessarily be declared `NOT INSTANTIABLE`.

A non-instantiable subtype can be defined under an instantiable supertype. Declaring a non-instantiable type to be `FINAL` is not allowed.

See Also: *PL/SQL User's Guide and Reference*

User-Defined Aggregate Functions

Oracle supports a fixed set of aggregate functions, such as `MAX`, `MIN`, and `SUM`. These is also a mechanism to implement new aggregate functions with user-defined aggregation logic.

Why Have User-Defined Aggregate Functions?

User-defined aggregate functions (UDAGs) refer to aggregate functions with user-specified aggregation semantics. Users can create a new aggregate function and provide the aggregation logic through a set of routines. After it is created, the user-defined aggregate function can be used in SQL DML statements in a manner similar to built-in aggregates. The Oracle server evaluates the UDAG by invoking the user-provided aggregation routines appropriately.

Databases are increasingly being used to store complex data such as image, spatial, audio, video, and so on. The complex data is typically stored in the database using object types, opaque types, or LOBs. User-defined aggregates are primarily useful in specifying aggregation over such new domains of data.

Furthermore, UDAGs can be used to create new aggregate functions over traditional scalar data types for financial or scientific applications. Because it is not possible to provide native support for all forms of aggregates, it is desirable to provide application developers with a flexible mechanism to add new aggregate functions.

See Also:

- *Oracle9i Data Cartridge Developer's Guide* for information about implementing user-defined aggregates
- *Oracle9i Data Warehousing Guide* for more information about using UDAGs in data warehousing
- Chapter 12, "Native Datatypes" for more information on opaque types

Creation and Use of UDAGs

The following is the procedure for implementing user-defined aggregates:

1. Implement the `ODCIAggregate` interface routines as methods of an object type.

2. Create a UDAG, using the CREATE FUNCTION statement and specify the implementation type created in Step 1:

   ```
   CREATE FUNCTION MyUDAG ... AGGREGATE USING MyUDAGRoutines;
   ```

3. Use the UDAG in SQL DML statements the same way you use built-in aggregates:

   ```
   SELECT col1, MyUDAG(col2) FROM tab GROUP BY col1;
   ```

How Do Aggregate Functions Work?

An aggregate function conceptually takes a set of values as input and returns a single value. The sets of values for aggregation are typically identified using a GROUP BY clause. For example:

```
SELECT AVG(T.Sales)
FROM AnnualSales T
GROUP BY T.State
```

The evaluation of an aggregate function can be decomposed into three primitive operations. Considering the preceding example of AVG(), they are:

1. Initialize : initialize the computation

   ```
   runningSum = 0; runningCount = 0;
   ```

2. Iterate : process new input value

   ```
   runningSum += inputval; runningCount++;
   ```

3. Terminate : compute the result

   ```
   return (runningSum/runningCount);
   ```

The variables runningSum and runningCount, in the preceding example, determine the **state** of the aggregation. Thus, the **aggregation context** can be viewed as an object that contains runningSum and runningCount attributes. The Initialize method initializes the aggregation context, Iterate updates it and Terminate method uses the context to return the resultant aggregate value.

In addition, we require one more primitive operation to merge two aggregation contexts and create a new context. This operation is needed to combine the results of aggregation over subsets and obtain the aggregate over the entire set. This situation can arise during both serial and parallel evaluations of the aggregate.

4. Merge: combine the two aggregation contexts and return a single context

```
runningSum = runningSum1 + runningSum2;
runningCount = runningCount1 + runningCount2;
```

Oracle lets you register new aggregate functions by providing specific implementations for these primitive operations.

Application Interfaces

Oracle provides several facilities for using object datatypes in application programs:

- SQL
- PL/SQL
- Pro*C/C++
- OCI
- OTT
- JPublisher
- JDBC
- SQLJ

SQL

Oracle SQL data definition language provides the following support for object datatypes:

- Defining object types, nested tables, and arrays
- Specifying privileges
- Specifying table columns of object types
- Creating object tables

Oracle SQL data manipulation language provides the following support for object datatypes:

- Querying and updating objects and collections
- Manipulating REFs

> **See Also:** *Oracle9i SQL Reference* for a complete description of SQL syntax

PL/SQL

PL/SQL is a procedural language that extends SQL. It offers features such as packages, data encapsulation, information hiding, overloading, and exception handling. Most stored procedures are written in PL/SQL.

PL/SQL allows use from within functions and procedures of the SQL features that support object types. The parameters and variables of PL/SQL functions and procedures can be of user-defined types.

PL/SQL provides all the capabilities necessary to implement the methods associated with object types. These methods (functions and procedures) reside on the server as part of a user's schema.

> **See Also:** *PL/SQL User's Guide and Reference* for a complete description of PL/SQL

Pro*C/C++

The Oracle Pro*C/C++ precompiler allows programmers to use object datatypes in C and C++ programs. Pro*C developers can use the Object Type Translator to map Oracle object types and collections into C datatypes to be used in the Pro*C application.

Pro*C provides compile time type checking of object types and collections and automatic type conversion from database types to C datatypes. Pro*C includes an EXEC SQL syntax to create and destroy objects and offers two ways to access objects in the server:

- SQL statements and PL/SQL functions or procedures embedded in Pro*C programs
- A simple interface to the object cache, where objects can be accessed by traversing pointers, then modified and updated on the server

> **See Also:**
>
> - "OCI" on page 13-20
> - *Pro*C/C++ Precompiler Programmer's Guide* for a complete description of the Pro*C precompiler

Dynamic Creation and Access of Type Descriptions

Oracle provides a C API to enable dynamic creation and access of type descriptions. Additionally, you can create transient type descriptions, type descriptions that are not stored persistently in the DBMS.

The C API enables creation and access of `LNOCIAnyData` and `LNOCIAnyDataSet`.

- The `LNOCIAnyData` type models a self descriptive (with regard to type) data instance of a given type.
- The `LNOCIAnyDataSet` type models a set of data instances of a given type.

Oracle also provides SQL data types (in Oracle's Open Type System) that correspond to these data types.

- `SYS.ANYTYPE` corresponds to `LNOCIType`
- `SYS.ANYDATA` corresponds to `LNOCIAnyData`
- `SYS.ANYDATASET` corresponds to `LNOCIAnyDataSet`

You can create database table columns and SQL queries on such data.

The new C API uses the following terms:

- **Transient types** - Type descriptions (type metadata) that are not stored persistently in the database.
- **Persistent types** - SQL types created using the `CREATE TYPE` SQL statement. Their type descriptions are stored persistently in the database.
- **Self-descriptive data** - Data encapsulating type information along with the actual contents. The `ANYDATA` type (`LNOCIAnyData`) models such data. A data value of any SQL type can be converted to an `ANYDATA`, which can be converted back to the old data value. An incorrect conversion attempt results in an exception.
- **Self-descriptive MultiSet** - Encapsulation of a set of data instances (all of the same type), along with their type description.

See Also:

- *Oracle9i Application Developer's Guide - Object-Relational Features*
- *Oracle Call Interface Programmer's Guide*

OCI

The Oracle call interface (OCI) is a set of C language interfaces to the Oracle server. It provides programmers great flexibility in using the server's capabilities.

An important component of OCI is a set of calls to allow application programs to use a workspace called the object cache. The **object cache** is a memory block on the client side that allows programs to store entire objects and to navigate among them without round trips to the server.

The object cache is completely under the control and management of the application programs using it. The Oracle server has no access to it. The application programs using it must maintain data coherency with the server and protect the workspace against simultaneous conflicting access.

LNOCI provides functions to:

- Access objects on the server using SQL
- Access, manipulate and manage objects in the object cache by traversing pointers or `REFs`
- Convert Oracle dates, strings and numbers to C data types
- Manage the size of the object cache's memory
- Create transient type descriptions. Transient type descriptions are not stored persistently in the DBMS. Compatibility must be set to Oracle9*i* or higher.

LNOCI improves concurrency by allowing individual objects to be locked. It improves performance by supporting complex object retrieval.

LNOCI developers can use the object type translator to generate the C datatypes corresponding to a Oracle object types.

See Also: *Oracle Call Interface Programmer's Guide*

OTT

The Oracle type translator (OTT) is a program that automatically generates C language structure declarations corresponding to object types. OTT facilitates using the Pro*C precompiler and the OCI server access package.

See Also:

- *Oracle Call Interface Programmer's Guide*
- *Pro*C/C++ Precompiler Programmer's Guide*

JPublisher

Java Publisher (JPublisher) is a program that automatically generates Java class definitions corresponding to object types in the database. Java Publisher facilitates using SQLJ and the JDBC server access package.

See Also: *Oracle9i JPublisher User's Guide*

JDBC

Java Database Connectivity (JDBC) is a set of Java interfaces to the Oracle server. Oracle's JDBC:

- Allows access to objects and collection types defined in the database from Java programs through dynamic SQL
- Provides for translation of types defined in the database into Java classes through default or customizable mappings

See Also: *Oracle9i JDBC Developer's Guide and Reference*

SQLJ

SQLJ allows developers to use object datatypes in Java programs. Developers can use JPublisher to map Oracle object and collection types into Java classes to be used in the application.

SQLJ provides access to server objects using SQL statements embedded in the Java code. SQLJ provides compile-time type checking of object types and collections in the SQL statements.

The syntax is based on an ANSI standard (SQLJ Consortium).

SQLJ Object Types

You can specify Java classes as SQL user-defined object types. You can define columns or rows of this SQLJ type. You can also query and manipulate the objects of this type as if they were SQL primitive types.

Additionally, you can do the following:

- Make the static fields of a class visible in SQL
- Allow the user to call a Java constructor
- Maintain the dependency between the Java class and its corresponding type

See Also:

- *Oracle9i SQL Reference*
- *Oracle9i Application Developer's Guide - Object-Relational Features*
- *Oracle9i SQLJ Developer's Guide and Reference*

Datatype Evolution

An object datatype can be referenced by any of the following schema objects:

- Table or subtable
- Type or subtype
- Program unit (PL/SQL block): procedure, function, package, trigger
- Indextype
- View (including object view)
- Functional index
- Operator

When any of these objects references a type, either directly or indirectly through another type or subtype, it becomes a dependent object on that type. Whenever a type is modified, all dependent program units, views, operators and indextypes are marked *invalid*. The next time each of these invalid objects is referenced, it is revalidated, using the new type definition. If it is recompiled successfully, then it becomes valid and can be used again.

When a type has either type or table dependents, altering a type definition becomes more complicated because existing persistent data relies on the current type definition.

You can change an object type and propagate the type change to its dependent types and tables. ALTER TYPE lets you add or drop methods and attributes from existing types and optionally propagate the changes to dependent types, tables, and even the table data. You can also modify certain attributes of a type.

See Also:

- *Oracle9i SQL Reference* for details about syntax
- *PL/SQL User's Guide and Reference* for details about type specification and body compilation
- *Oracle9i Application Developer's Guide - Object-Relational Features* for details about managing type versions

Introduction to Object Views

Just as a view is a virtual table, an **object view** is a virtual object table.

Oracle provides object views as an extension of the basic relational view mechanism. By using object views, you can create virtual object tables from data—of either built-in or user-defined types—stored in the columns of relational or object tables in the database.

Object views provide the ability to offer specialized or restricted access to the data and objects in a database. For example, you can use an object view to provide a version of an employee object table that does not have attributes containing sensitive data and does not have a deletion method.

Object views allow the use of relational data in object-oriented applications. They let users:

- Try object-oriented programming techniques without converting existing tables
- Convert data gradually and transparently from relational tables to object-relational tables
- Use legacy RDBMS data with existing object-oriented applications

Advantages of Object Views

Using object views can lead to better performance. Relational data that make up a row of an object view traverse the network as a unit, potentially saving many round trips.

You can fetch relational data into the client-side object cache and map it into C or *C++* structures so 3GL applications can manipulate it just like native structures.

Object views provide a gradual upgrade path for legacy data. They provide for co-existence of relational and object-oriented applications, and they make it easier

to introduce object-oriented applications to existing relational data without having to make a drastic change from one paradigm to another.

Object views provide the flexibility of looking at the same relational or object data in more than one way. Thus you can use different in-memory object representations for different applications without changing the way you store the data in the database.

How Object Views Are Defined

Conceptually, the process of defining an object view is simple. It consists of the following actions:

- Defining an object type to be represented by rows of the object view.
- Writing a query that specifies which data in which relational tables contain the attributes for objects of that type.
- Specifying an object identifier, based on attributes of the underlying data, to allow REFs to the objects (rows) of the object view.

The object identifier corresponds to the unique object identifier that Oracle generates automatically for rows of object tables. In the case of object views, however, the declaration must specify something that is unique in the underlying data (for example, a primary key).

If the object view is based on a table or another object view and you do not specify an object identifier, Oracle uses the object identifier from the original table or object view.

If you want to be able to update a complex object view, you might need to take another action:

- Write an INSTEAD OF trigger procedure for Oracle to run whenever an application program tries to update data in the object view.

After doing these four things, you can use an object view just like an object table.

For example, the following SQL statements define an object view:

```
CREATE TABLE emp_table  (
    empnum   NUMBER (5),
    ename    VARCHAR2 (20),
    salary   NUMBER (9, 2),
    job      VARCHAR2 (20) );
```

```
CREATE TYPE employee_t  AS OBJECT(
    empno    NUMBER (5),
    ename    VARCHAR2 (20),
    salary   NUMBER (9, 2),
    job      VARCHAR2 (20) );

CREATE VIEW emp_view1 OF employee_t
    WITH OBJECT OID (empno) AS
        SELECT   e.empnum, e.ename, e.salary, e.job
            FROM     emp_table e
            WHERE    job = 'Developer';
```

The object view looks to the user like an object table whose underlying type is `employee_t`. Each row contains an object of type `employee_t`. Each row has a unique object identifier.

Oracle constructs the object identifier based on the specified key. In most cases, it is the primary key of the base table. If the query that defines the object view involves joins, however, you must provide a key across all tables involved in the joins, so that the key still uniquely identifies rows of the object view.

Note: Columns in the `WITH OBJECT OID` clause (`empno` in the example) must also be attributes of the underlying object type (`employee_t` in the example). This makes it easy for trigger programs to identify the corresponding row in the base table uniquely.

See Also:

- *Oracle9i Database Administrator's Guide* for specific directions for defining object views
- "Updates of Object Views" on page 13-26 for more information about writing an `INSTEAD OF` trigger

Use of Object Views

Data in the rows of an object view can come from more than one table, but the object still traverses the network in one operation. When the instance is in the client side object cache, it appears to the programmer as a C or C++ structure or as a PL/SQL object variable. You can manipulate it like any other native structure.

You can refer to object views in SQL statements the same way you refer to an object table. For example, object views can appear in a SELECT list, in an UPDATE SET clause, or in a WHERE clause. You can also define object views on object views.

You can access object view data on the client side using the same OCI calls you use for objects from object tables. For example, you can use LNOCIObjectPin() for pinning a REF and LNOCIObjectFlush() for flushing an object to the server. When you update or flush to the server an object in an object view, Oracle updates the object view.

> **See Also:** *Oracle Call Interface Programmer's Guide* for more information about OCI calls

Updates of Object Views

You can update, insert, and delete the data in an object view using the same SQL DML you use for object tables. Oracle updates the base tables of the object view if there is no ambiguity.

A view is not updatable if its view query contains joins, set operators, aggregate functions, GROUP BY, or DISTINCT. If a view query contains pseudocolumns or expressions, the corresponding view columns are not updatable. Object views often involve joins.

To overcome these obstacles Oracle provides **INSTEAD OF triggers**. They are called INSTEAD OF triggers because Oracle runs the trigger body instead of the actual DML statement.

INSTEAD OF triggers provide a transparent way to update object views or relational views. You write the same SQL DML (INSERT, DELETE, and UPDATE) statements as for an object table. Oracle invokes the appropriate trigger instead of the SQL statement, and the actions specified in the trigger body take place.

> **See Also:**
>
> - *Oracle9i Application Developer's Guide - Object-Relational Features* for a purchase order/line item example that uses an INSTEAD OF trigger
> - Chapter 17, "Triggers"

Updates of Nested Table Columns in Views

A nested table can be modified by inserting new elements and updating or deleting existing elements. Nested table columns that are virtual or synthesized, as in a view,

are not usually updatable. To overcome this, Oracle allows INSTEAD OF triggers to be created on these columns.

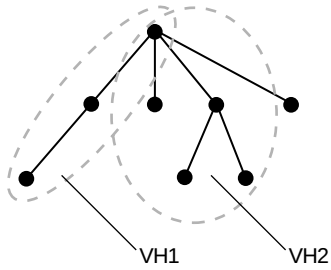
The INSTEAD OF trigger defined on a nested table column of a view is fired when the column is modified. If the entire collection is replaced by an update of the parent row, then the INSTEAD OF trigger on the nested table column is not fired.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for a purchase order / line item example that uses an INSTEAD OF trigger on a nested table column

View Hierarchies

An object view can be created as a subview of another object view. The type of the superview must be the immediate supertype of the type of the object view being created. Thus, you can build an object view hierarchy which has a one-to-one correspondence to the type hierarchy. This does not imply that every view hierarchy must span the entire corresponding type hierarchy. The view hierarchy can be rooted at any subtype of the type hierarchy. Furthermore, it does not have to encompass the entire subhierarchy.

Figure 13–2 Multiple View Hierarchies



By default, the rows of an object view in a view hierarchy include all the rows of all its subviews (direct and indirect) projected over the columns of the given view.

Only one object view can be created as a subview of a given view corresponding to the given subtype; that is, the same view cannot participate in many different view hierarchies. An object view can be created as a subview of only one superview; multiple inheritance is not supported.

The subview inherits the object identifier (OID) from its superview and cannot be explicitly specified in any subview.