ORACLE UNIVERSITY

Oracle Database 12*c*: SQL and PL/SQL New Features

Student Guide

D90165GC10
Edition 1.0 | February 2015 | D90450

Learn more from Oracle University at **oracle.com/education/**

ORACLE

Authors

Anupama Mandya
Lauran Serhal
Swarnapriya Shridhar
Dimpi Sarmah

Technical Contributors

Steven Feuerstein
John Haydu
Hermann Baer
Srinivasan Ramakrishnan
Charles Sperry
A.A. Hopeman
Suresh Sridharan
Paul Lane
S. Matt Taylor Jr.

Reviewers

Steven Feuerstein
Iloon Ellen
Gerlinde Frenzen

Editors

Vijayalakshmi Narasimhan
Aju Kumar

Graphic Designer

Rajiv Chandrabhanu

Publishers

Joseph Fernandez
Srividya Rameshkumar
Veena Narasimhan

Contents

1

Course Overview

Course Objectives

After completing this course, you should be able to:

- Identify the new features available in Oracle Database 12*c* for SQL Language and data types
  - Use a SQL row limiting clause in a query
  - Use invisible or hidden columns
  - Understand enhanced online DDL capabilities
  - Understand increase in length limits of some Oracle data types
  - Use an `IDENTITY` column in SQL
  - Use SQL column defaulting on explicit `NULL` and sequence
- Identify the new features available in Oracle Database 12*c* for PL/SQL
  - Understand fine-grained access control for packaged procedures

ORACLE

This course covers the new features that are available in Oracle Database 12*c* for the SQL Language, data types, PL/SQL, and data warehousing. In this course, you identify the new features in the SQL language and data types such as using the SQL row limiting `FETCH` clause in SQL queries, creating invisible columns in a table, using enhanced DDL with the `ONLINE` keyword, using increased length limits of SQL data types, using the `SQL IDENTITY` column, and using SQL column enhancements that default on explicit `NULL` and sequence.

You identify some of the PL/SQL new features in Oracle Database 12*c*, such as providing fine-grained access control for packaged procedures, understanding the Invoker's Rights function that can be result cached, implementing the `ACCESSIBLE BY` clause, granting roles to PL/SQL packages and stand-alone stored subprograms, and identifying PL/SQL functions that can run faster in SQL.

Course Objectives

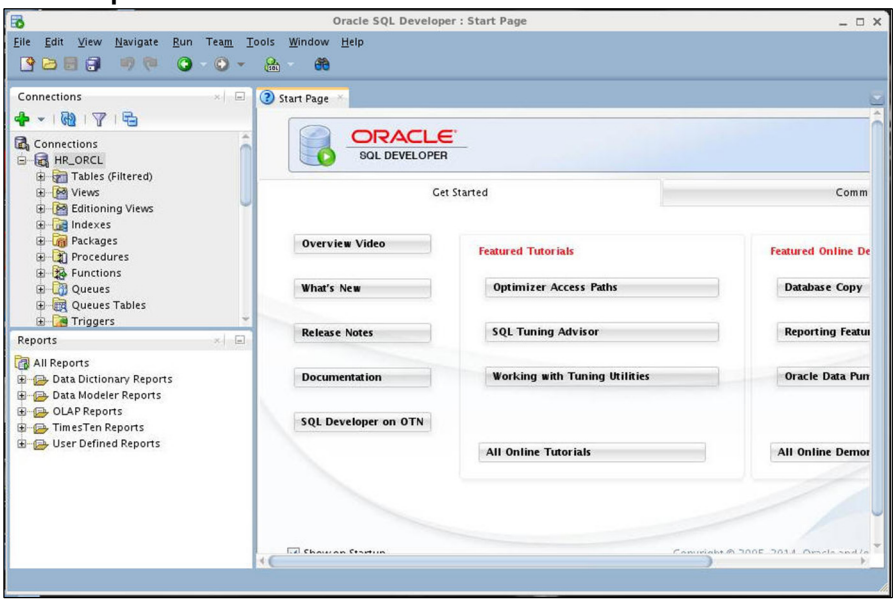
  - Define the Invoker's Rights function that can be result cached
  - Grant roles to PL/SQL packages and stand-alone stored subprograms
  - Identify PL/SQL functions that can run faster in SQL
- Identify data warehousing enhancements
  - Understand multi partition maintenance operations
  - Analyze partial global indexes for partitioned tables
  - Maintain an asynchronous global index
  - Determine SQL for pattern matching
  - Understand synchronous materialized view refresh
  - Identify ways to improve query performance against OLAP Cubes

ORACLE

You also identify some of the data warehousing enhancements in Oracle Database 12*c* such as understanding multi partition maintenance operations, analyzing partial global indexes for partitioned tables, maintaining an asynchronous global index, determining SQL for pattern matching, understanding synchronous materialized view refresh, and identifying ways to improve query performance against OLAP Cubes.

Oracle SQL Developer

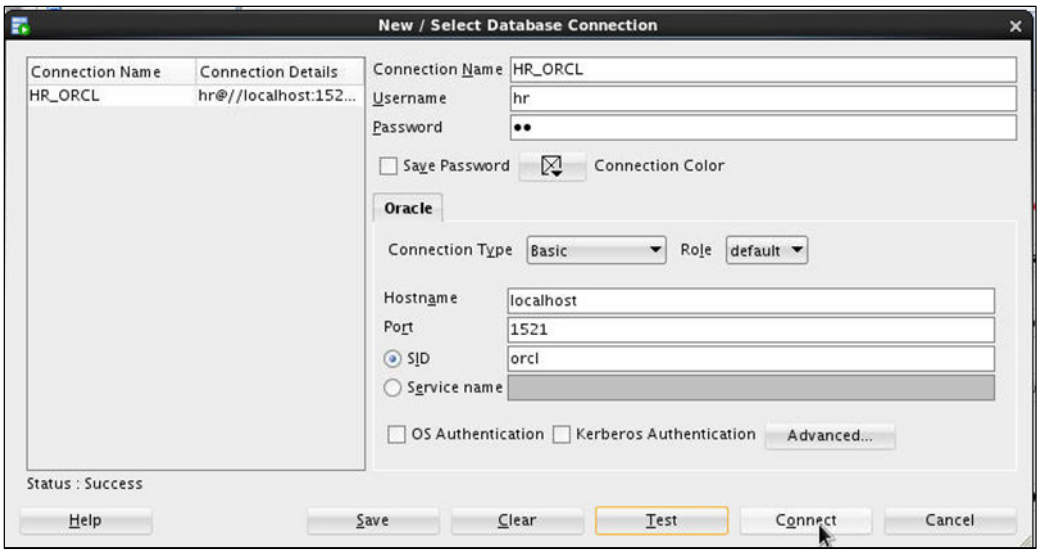
This tool is used throughout the course to run SQL statements and SQL scripts.



Oracle SQL Developer is a free graphical tool that enhances productivity and simplifies database development tasks. With SQL Developer, you can browse database objects, run SQL statements and SQL scripts, edit and debug PL/SQL statements, manipulate and export data, and view and create reports.

You can connect to any target Oracle database schema by using standard Oracle Database authentication. After you are connected, you can perform operations on the objects in the database.

Creating Database Connections by Using SQL Developer

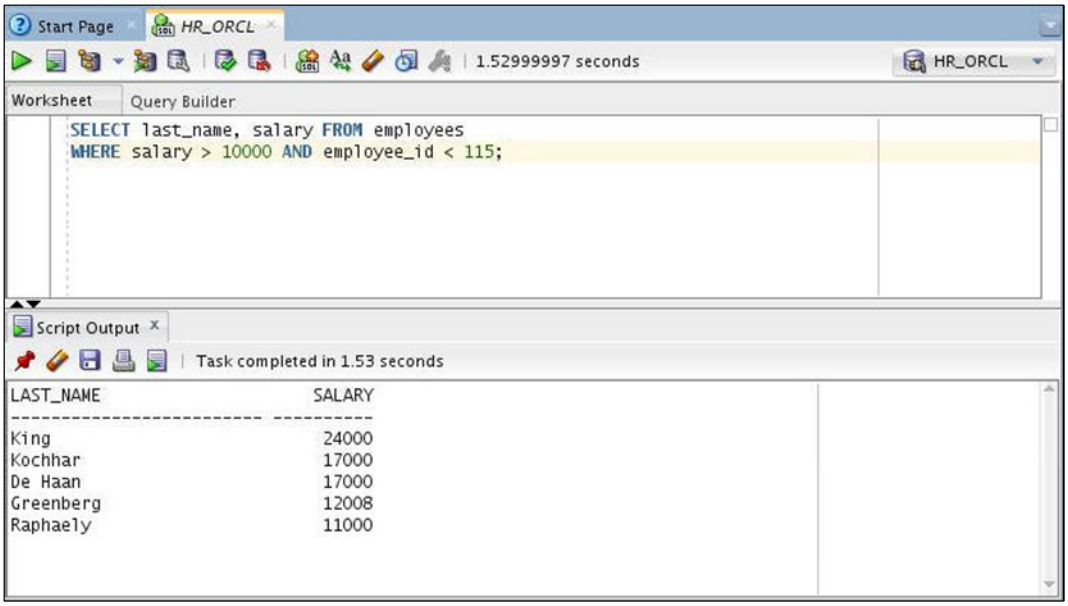


A connection is a SQL Developer object that specifies the necessary information for connecting to a specific database as a specific user of that database. You must have at least one database connection to use SQL Developer.

To create a new database connection, right-click the Connections node and select New Database Connection. Use the dialog box (shown in the screenshot in the slide) to specify information about the connection.

In this course, the `hr` schema is being used.

Using the SQL Worksheet



You can use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements. You can specify any actions that can be processed by the database connection associated with the worksheet, such as creating a table, inserting data, creating and editing a trigger, selecting data from a table, and saving that data to a file.

You can display a SQL Worksheet by right-clicking a connection in the Connections Navigator and selecting Open SQL Worksheet, by selecting Tools, and then SQL Worksheet, or by clicking the Use SQL Worksheet icon on the menu bar.

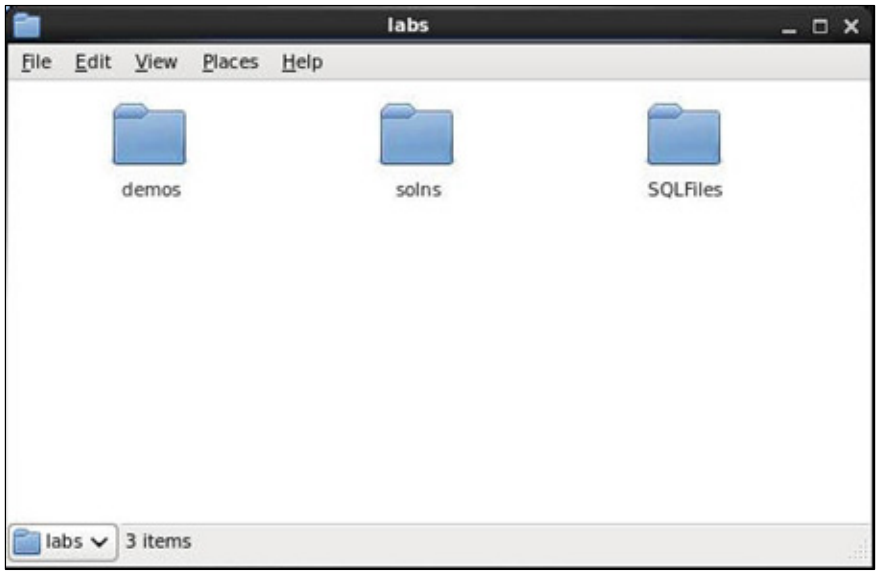
The SQL Worksheet has the Worksheet and Query Builder tabs.

Additional Resources

For additional information about SQL Developer and the new features in Oracle Database 12*c*, refer to the following:

- Oracle Technology Network:
  - http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html
- Oracle SQL Developer User's Guide Release 4.0
  - http://docs.oracle.com/cd/E39885_01/index.htm
- Oracle Database 12*c* SQL Language Reference
  - https://docs.oracle.com/database/121/SQLRF/toc.htm
- Oracle Database 12*c* PL/SQL Language Reference
  - http://docs.oracle.com/database/121/LNPLS/release_changes.htm#LNPLS105
- Oracle Database 12*c* Data Warehousing Reference
  - https://docs.oracle.com/database/121/DWHSG/release_changes.htm#DWHSG9284

Accessing the `labs` Directory



All the files that are required to complete the practices are available in the `labs` directory.

To access the `labs` directory, from the Applications menu, select System Tools > File Browser. From the `oracle` directory, open the `labs` directory. You will see three folders and their contents:

- **demos:** The videos of demonstrations in a lesson
- **solns:** The solution scripts given in the Activity Guide
- **SQLFiles:** All the files that you need to complete the practices

2

SQL Language Enhancements

Objectives

After completing this lesson, you should be able to:

- Use the `FETCH` clause to limit the rows returned by a query
- Create an invisible column in a table
- Utilize the enhanced DDL capabilities by using the `ONLINE` keyword
- Identify the additional new features that are available in the SQL language

In this lesson, you learn about using the SQL row limiting clause, creating an invisible column in a table, understanding enhanced DDL capabilities by using the `ONLINE` keyword, and identifying some additional SQL language new features that are available in Oracle Database 12*c*.

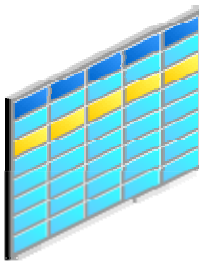
Lesson Agenda

- SQL Row Limiting Clause
- Invisible and Hidden Columns
- Enhanced DDL operations by using the `ONLINE` keyword
- Additional new features in the SQL Language

SQL Row Limiting Clause

`row_limiting_clause` can be used to:

- Limit the rows returned by a query
- Implement top-N reporting
- Specify the number or percentage of rows to be returned by using the `FETCH FIRST` keyword
- Specify that the returned rows should begin with a row after the first row of the full result set by using the `OFFSET` keyword
- Include additional rows with the same ordering keys as the last row of the row-limited result set by using the `WITH TIES` keyword



In Oracle Database 12*c* Release 1, the SQL `SELECT` syntax has been enhanced to allow a `row_limiting_clause`, which limits the number of rows that are returned in the result set. The `row_limiting_clause` provides both easy-to-understand syntax and expressive power. Limiting the number of rows returned can be valuable for reporting, analysis, data browsing, and other tasks. Queries that order data, and then limit row output, are widely used, and are often referred to as Top-N queries. Top-N queries sort their result set, and then return only the first *n* rows.

You can specify the number of rows or percentage of rows to return with the `FETCH FIRST` keywords. You can use the `OFFSET` keyword to specify that the returned rows should begin with a row after the first row of the full result set. The `WITH TIES` keyword includes rows with the same ordering keys as the last row of the row-limited result set. (You must specify the `ORDER BY` clause with the `WITH TIES` clause in the query.)

`FETCH FIRST` and `OFFSET` simplify syntax and comply with the ANSI SQL standard.

There are certain limitations of the SQL row limiting clause:

- You cannot specify this clause with the `for_update_clause`.
- You cannot specify this clause in the subquery of a `DELETE` or an `UPDATE` statement.
- If you specify this clause, the select list cannot contain the sequence pseudocolumns `CURRVAL` or `NEXTVAL`.

SQL Row Limiting Clause: Syntax

You specify the `row_limiting_clause` in the `SQL SELECT` statement by placing it after the `ORDER BY` clause.

```
SELECT …
   FROM …
 [ WHERE …  ]
 [ ORDER BY …  ]
 [OFFSET offset { ROW | ROWS }]
[FETCH { FIRST | NEXT } [{ row_count | percent PERCENT
   }] { ROW | ROWS }
  { ONLY | WITH TIES }]
```


You specify the `row_limiting_clause` in the SQL `SELECT` statement by placing it after the `ORDER BY` clause. Note that an `ORDER BY` clause is required if you want to sort the rows for consistency.

OFFSET: Use this clause to specify the number of rows to skip before row limiting begins. The value for offset must be a number. If you specify a negative number, offset is treated as 0. If you specify `NULL` or a number greater than or equal to the number of rows that are returned by the query, 0 rows are returned.

ROW | ROWS: Use these keywords interchangeably. They are provided for semantic clarity.

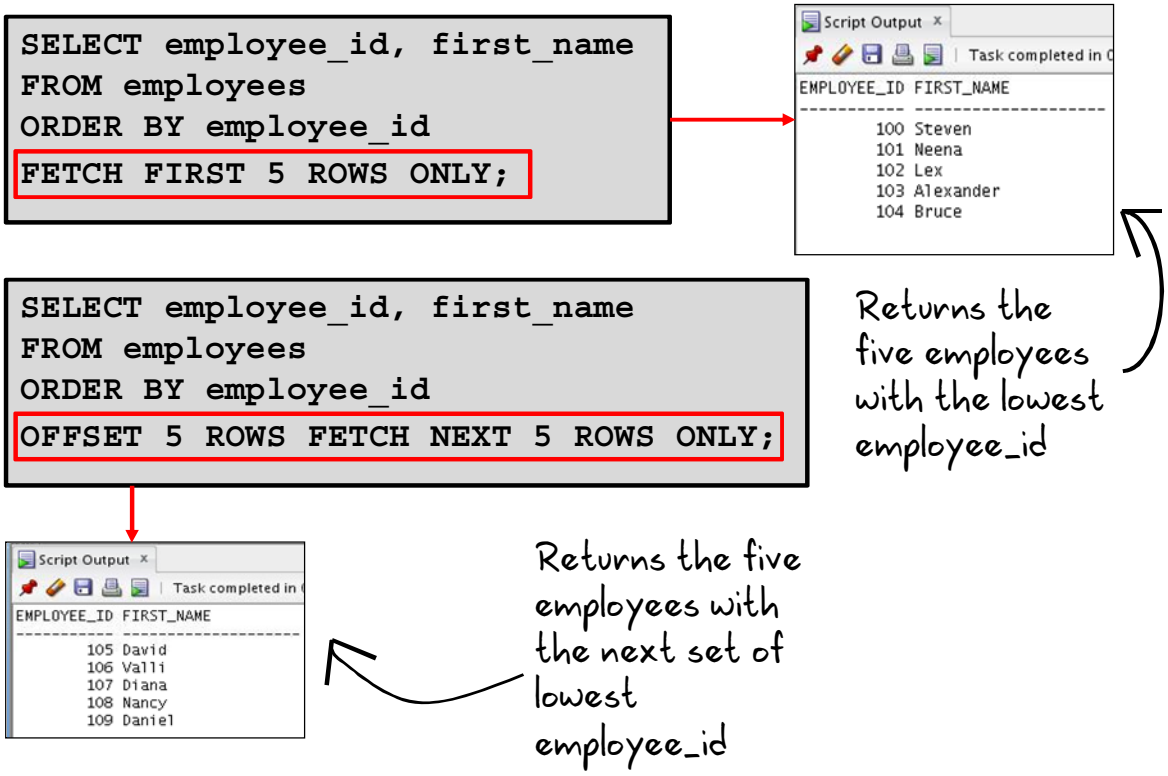
FETCH: Use this clause to specify the number of rows or percentage of rows to return.

FIRST | NEXT: Use these keywords interchangeably. They are provided for semantic clarity.

row_count | *percent* PERCENT: Use `row_count` to specify the number of rows to return. Use *percent* `PERCENT` to specify the percentage of the total number of selected rows to return. The value for percent must be a number.

ONLY | WITH TIES: Specify `ONLY` to return exactly the specified number of rows or percentage of rows. Specify `WITH TIES` to return additional rows with the same sort key as the last row fetched. If you specify `WITH TIES`, you must specify `order_by_clause`. If you do not specify `order_by_clause`, no additional rows will be returned.

SQL Row Limiting Clause: Examples



The first code example in the slide returns the five employees with the lowest `employee_id`.

The second code example returns the five employees with the next set of lowest `employee_id`.

Note: If `employee_id` is assigned sequentially by the date when the employee joined the organization, these examples give you the top 5 employees, and then employees 6–10 depending on the date of joining.

Quiz

You can use the `OFFSET` keyword to specify that the returned rows begin with a row after the first row of the full result set.

a. True
b. False

Answer: a

Practice 2-1 Overview: SQL Row Limiting Clause

This practice covers the following topics:

- Executing a `SQL SELECT` statement that returns employees with the lowest `employee_id` values
- Executing a `SQL SELECT` statement that returns the next five employees with the lowest `employee_id` values
- Executing a `SQL SELECT` statement that returns 5 percent of the employees with the lowest salaries
- Executing a `SQL SELECT` statement that returns 5 percent of the employees with the lowest salaries by using the `WITH TIES` keyword

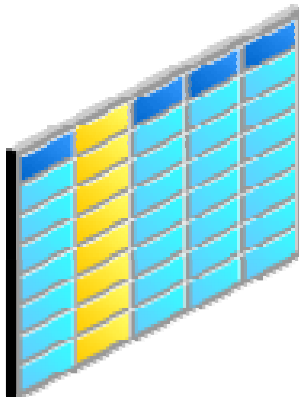
Lesson Agenda

- SQL Row Limiting Clause
- Invisible and Hidden Columns
- Enhanced DDL operations by using the `ONLINE` keyword
- Additional new features in SQL Language

Invisible and Hidden Columns

Invisible columns are:

- User-specified hidden columns
- Added to the table and can be made visible later
- Added to the table without affecting existing applications that access the table



Invisible columns were introduced in Oracle Database 12*c* so that developers could enhance applications to meet the constant changes in business rules and requirements. Invisible columns are user-specified hidden columns. With invisible columns, you can access the application while a developer is enhancing it to meet new business requirements. You expose the invisible column after the new business requirement is incorporated into the application. You can make a column invisible during table creation or when you add a column to a table, and you can later alter the table to make the same column visible. You can also alter a table to make a visible column invisible.

You can seamlessly add an invisible column to the table without affecting existing applications that access the table.

Specifying an Invisible Column

You can:

- Specify invisible columns in a `column_expression`
- Display or assign a value to an invisible column by specifying its name explicitly

You can specify invisible columns in a `column_expression`. You can display or assign a value to an invisible column by specifying its name explicitly.

Displaying Invisible Columns

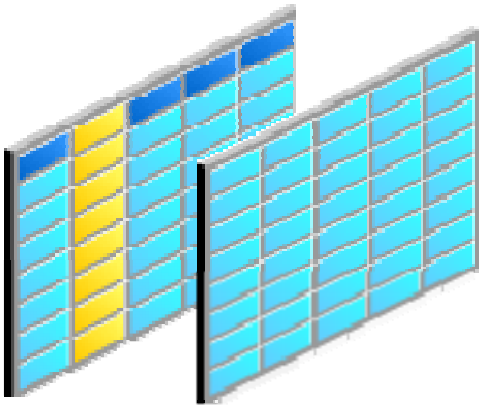
- To display an invisible column:
  - Include the invisible column in the select list of a `SELECT` statement
  - Configure SQL*Plus to allow `INVISIBLE` column information to be viewed with the `DESCRIBE` command by setting `COLINVISIBLE` to `ON`
- The following operations do not display an invisible column:
  - The `SELECT * FROM` statement in SQL (unless the invisible column is explicitly included)
  - `DESCRIBE` commands in SQL*Plus
  - `%ROWTYPE` attribute declarations in PL/SQL
  - Describe calls in Oracle Call Interface (OCI)

An invisible column in a table can be accessed only by referring to its name explicitly in queries and other operations. To display or assign a value to an `INVISIBLE` column, you must specify its name explicitly. For example:

- The `SELECT *` syntax will not display an `INVISIBLE` column. However, if you include an `INVISIBLE` column in the select list of a `SELECT` statement, the column will be displayed.
- You cannot implicitly specify a value for an `INVISIBLE` column in the `VALUES` clause of an `INSERT` statement. You must specify the `INVISIBLE` column in the column list.
- You must explicitly specify an `INVISIBLE` column in Oracle Call Interface (OCI) describe calls and PL/SQL `%ROWTYPE` attributes.
- You can configure SQL*Plus to allow `INVISIBLE` column information to be viewed with the `DESCRIBE` command by changing the default setting of `COLINVISIBLE` to `ON`.

Difference Between Invisible and Hidden Columns

- Invisible columns are not system-generated hidden columns.
- Invisible columns can be made visible, but hidden columns cannot be made visible.



Invisible columns are not the same as system-generated hidden columns. You can make invisible columns visible, but you cannot make hidden columns visible.

Facts About Invisible Columns

Invisible columns:

- Can be specified as a virtual column
- Cannot have values specified in the `VALUES` clause of an `INSERT` statement
- Must be specified in the column list
- Can be used as a partitioning key when specified as part of `CREATE TABLE`
- Are not supported in external tables, cluster tables, and temporary tables
- Cannot have user-defined type attributes

Column Ordering for Invisible Columns

- Invisible columns are not included in the column order for a table.

```
CREATE TABLE mytable (a INT, b INT INVISIBLE, c
INT);
```

Columns	Column Order
a	1
c	2

- Invisible columns, when made visible, will appear as the last column in the table's column order.

```
ALTER TABLE mytable MODIFY (b VISIBLE);
```

Columns	Column Order
a	1
c	2
b	3

ORACLE

The database stores columns in the order in which they are listed in the `CREATE TABLE` statement. If you add a new column to a table, the new column becomes the last column in the table's column order.

When a table contains one or more invisible columns, the invisible columns are not included in the column order for the table. Column ordering is important when all the columns in a table are accessed.

For example, a `SELECT * FROM` statement displays the columns in a table's column order. Because invisible columns are not included in this type of generic access of a table, they are not included in the column order.

When you make an invisible column visible, the column is included in the table's column order as the last column. When you make a visible column invisible, the invisible column is not included in the column order and the order of the visible columns in the table might be re-arranged.

The SET COLINVISIBLE Option and the DESCRIBE Command

- The SET COLINVISIBLE option enables users to display the invisible column formation in a DESCRIBE command.
- Syntax:

```
SET COLINVI[SIBLE] [ON|OFF]
```

  where:
  - ON displays the metadata for the invisible column in the DESCRIBE command
  - OFF does not display the metadata. The default value is OFF.
- The DESCRIBE command describes the invisible column when the SET COLINVISIBLE flag is set.

Currently, SQL*Plus (a command-line tool for accessing Oracle database) cannot display the metadata information for the invisible column with the DESCRIBE command. It needs to provide the SET option so that you can describe the invisible column in a table.

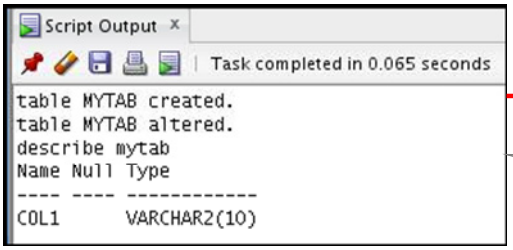
The SET COLINVISIBLE option enables users to display the invisible column formation in a DESCRIBE command. The syntax has an ON/OFF option. ON means that metadata for the invisible column is displayed in the DESCRIBE command and OFF means that no metadata is displayed.

The DESCRIBE command describes the invisible column when the SET COLINVISIBLE flag is set. When a table does not contain invisible columns, the COLINVISIBLE flag does not affect the DESCRIBE command. Queries and other operations that need to access the invisible columns can refer to them explicitly. There are no behavior changes in the query statements.

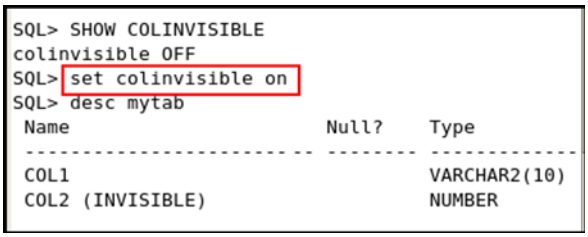
SET COLINVISIBLE Option: Example

```
create table mytab (col1 varchar2(10));
alter table mytab add (col2 number invisible);
describe mytab ;
```

OUTPUT



The invisible column is not displayed when the DESCRIBE command is executed.



The invisible column is displayed when the metadata for the invisible column is set to ON.

The code example in the slide shows the use of the SET COLINVISIBLE option in the CREATE TABLE statement. Take a moment to review the code in the slide. Note that the invisible column is displayed only when metadata is set to ON.

Quiz

The `SET COLINVISIBLE` option allows you to display the invisible column definition in a `DESCRIBE` command.

a. True
b. False

Answer: a

Quiz

The `COLINVISIBLE` flag does not affect the `DESCRIBE` command when the table has no invisible column.

a. True
b. False

Answer: a

Practice 2-2 Overview: Invisible and Hidden Columns

This practice covers the following topics:

- Creating a table with an invisible column
- Using the `DESCRIBE` command to list the table columns

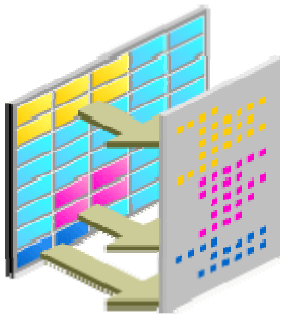
Lesson Agenda

- SQL Row Limiting Clause
- Invisible and Hidden Columns
- Enhanced DDL operations by using the `ONLINE` keyword
- Additional new features in SQL Language

Enhanced DDL Capabilities By Using the ONLINE Keyword

The ONLINE keyword allows execution of DML statements during the following DDL operations:

- DROP INDEX
- DROP CONSTRAINT
- ALTER INDEX UNUSABLE
- SET COLUMN UNUSED



Beginning with Oracle Database 12*c* Release 1 (12.1), you can use the new ONLINE keyword to allow the execution of DML statements during the following DDL operations:

- DROP INDEX
- DROP CONSTRAINT
- ALTER INDEX UNUSABLE
- SET COLUMN UNUSED

This enhancement enables simpler application development, especially for application migrations. There are no application disruptions for schema maintenance operations.

ONLINE Keyword in DROP INDEX/CONSTRAINT Statements

- ONLINE indicates that DML operations on a table or partition are allowed while dropping an index.
- DROP INDEX ONLINE is supported for partitioned and non-partitioned indexes.

```
SQL> DROP INDEX schema.index ONLINE FORCE;
```

- DROP CONSTRAINT ONLINE enables you to drop an integrity constraint from a database with some restrictions:
  - Cannot drop a constraint with CASCADE
  - Cannot drop a referencing constraint

```
SQL> ALTER TABLE hr.employees
  2  DROP CONSTRAINT emp_email_uk ONLINE;
```

ORACLE

The DROP INDEX statement is used to remove an index or a domain index from a database. Drop index online is supported for partitioned and non-partitioned indexes.

- ONLINE: Specify ONLINE to indicate that DML operations on a table or partition are allowed while dropping an index.
- FORCE: FORCE applies only to domain indexes. This clause drops a domain index even if the index type routine invocation returns an error or if the index is marked IN PROGRESS. Without FORCE, you cannot drop a domain index if its index type routine invocation returns an error or if the index is marked IN PROGRESS.

Restrictions on dropping indexes:

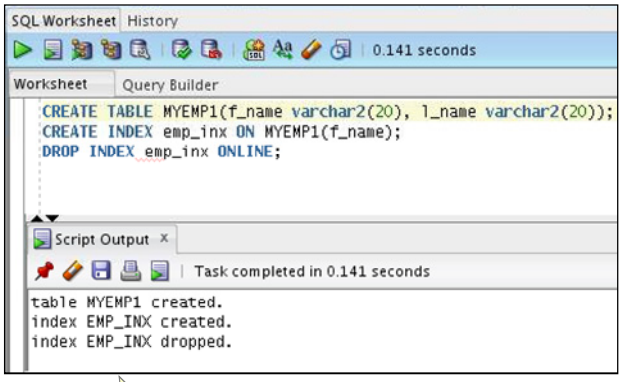
- You cannot drop a domain index if the index or any of its index partitions is marked IN_PROGRESS.
- You cannot specify the ONLINE clause when dropping a domain index, a cluster index, or an index on a queue table.

Restrictions on dropping constraints:

- You cannot drop a constraint with CASCADE.
- You cannot drop a constraint that holds references for other dependencies.

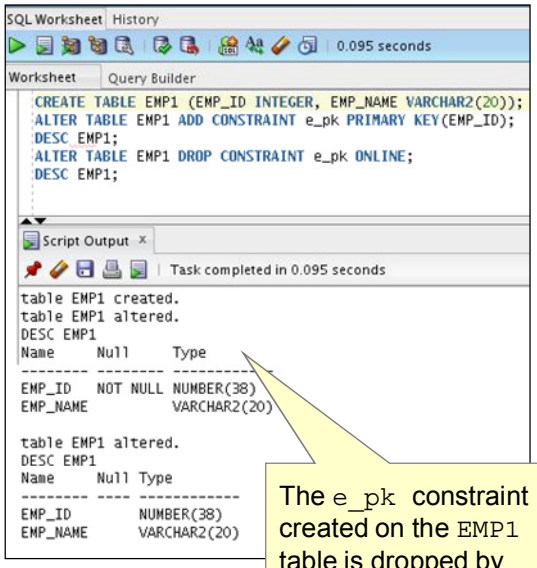
DROP INDEX and DROP CONSTRAINT: Examples

DROP INDEX ONLINE



The EMP_INX index created on the MYEMP1 table is dropped by using the ONLINE keyword.

DROP CONSTRAINT ONLINE



The e_pk constraint created on the EMP1 table is dropped by using the ONLINE keyword.

The example shows the use of the ONLINE keyword to indicate that DML operations on a table or partition are allowed while dropping an index and a constraint.

UNUSABLE Keyword in the ALTER INDEX Statement

- Specify the UNUSABLE clause to mark the index, index partitions, or index subpartitions as unusable.
- Specify the ONLINE clause to indicate that DML operations on a table or partition are allowed while marking an index as UNUSABLE.

```
SQL> ALTER INDEX hr.i_emp_ix UNUSABLE ONLINE;
```

```
SQL> SELECT status FROM user_indexes
  2  WHERE table_name='EMP';

INDEX_NAME                     STATUS
------------------------------ ------------
I_EMP_IX                       UNUSABLE
```


Specify the UNUSABLE keyword to mark the index, index partitions, or index subpartitions as unusable. The space allocated for an index, index partition, or subpartition is freed immediately when you mark an object as UNUSABLE. An unusable index must be rebuilt, or it must be dropped and re-created, before it can be used. While one partition is marked as UNUSABLE, the other partitions of the index are still valid. As of Oracle Database 12*c*, the ALTER INDEX UNUSABLE clause is made online by specifying an ONLINE keyword to indicate that DML operations on the table or partition are allowed while marking the index as UNUSABLE. If you specify this clause, the database does not drop the index segments.

Note: You cannot specify UNUSABLE for an index on a temporary table.

SET UNUSED Keyword to Mark Unused Columns

- The SET UNUSED clause can be the first step to free space in the database by dropping columns no longer needed.
- The **ONLINE** keyword indicates that DML operations on the table are allowed while marking columns as UNUSED.

```
SQL> CREATE TABLE emp (ename VARCHAR2(20), id
NUMBER);
SQL> INSERT INTO emp VALUES('Tim',4);
SQL> SELECT * FROM emp;
SQL> ALTER TABLE emp SET UNUSED (ename) ONLINE;
SQL> DESC emp;
Name                 Null?  Type
-------------------- ------ ----------------------------
ID                          NUMBER
```


The drop_column_clause can be the first step to free space in the database by dropping columns that you no longer need or by marking them to be dropped at a future time when the demand on system resources is less.

You specify the SET UNUSED keyword to mark one or more columns as unused. When you specify this clause for a column in an external table, the clause is transparently converted to an ALTER TABLE ... DROP COLUMN statement. Because any operation on an external table is a metadata-only operation, there is no difference in the performance of the two commands. Unused columns are treated as if they were dropped, even though their column data remains in the table rows. After marking a column UNUSED, you have no access to it. A SELECT * query does not retrieve data from unused columns. In addition, the names and types of columns that are marked as UNUSED are not displayed during a DESCRIBE, and you can add a new column to the table with the same name as the unused column.

You cannot specify the ONLINE clause when marking a column with a DEFERRABLE constraint as unused. A subsequent DROP UNUSED COLUMNS physically removes all unused columns from a table, similar to a DROP COLUMN.

Quiz

`SET UNUSED COLUMN` is used to:

a. Remove an index or a domain index from a database
b. Drop an integrity constraint from a database
c. Mark one or more columns as unused
d. Not drop the index segments in the database

Answer: c

Practice 2-3 Overview: Enhanced DDL Capabilities By Using the ONLINE Keyword

This practice covers the following topics:

- Creating an index on a table and executing the `DROP INDEX` statement with the `ONLINE` keyword to drop the index
- Creating a table, adding a primary key constraint to a column in the table, and executing the `DROP CONSTRAINT` statement with the `ONLINE` keyword
- Creating a table and an index for the table, and altering the index by using the `ALTER INDEX` statement with the `UNUSABLE ONLINE` keyword
- Creating a table, inserting values into the table, altering the table, and setting a column as unused by using the `SET UNUSED` keyword

ORACLE

Lesson Agenda

- SQL Row Limiting Clause
- Invisible and Hidden Columns
- Enhanced DDL operations by using the `ONLINE` keyword
- Additional new features in SQL Language

Database Migration Assistant for Unicode (DMU)

- Is a migration tool that provides an end-to-end solution for migrating databases from legacy character sets to Unicode
- Supports migration of almost all Oracle data types that directly or indirectly contain textual data

The Database Migration Assistant for Unicode (DMU) is a unique, next-generation migration tool that provides a streamlined, end-to-end solution for migrating databases from legacy character sets to Unicode.

The legacy utilities, CSSCAN and CSALTER, are removed from the database installation and de-supported.

The DMU supports migration of almost all Oracle data types that directly or indirectly contain textual data, and deals with database objects such as materialized views, indexes, constraints, and triggers that are affected by conversion of tables, so that they are properly synced after the migration.

This feature significantly reduces migration down time. It also helps to lower the cost and improve the efficiency of migrating to Unicode.

Database Migration Assistant for Unicode: Features

- Guided end-to-end migration workflow
- Intuitive graphical user interface
- Advanced data analysis and cleansing tools
- Validation mode to check data integrity for compliance with the Unicode Standard

Some of the features of the Database Migration Assistant for Unicode (DMU) are listed in the slide. DMU's intuitive user interface greatly simplifies the migration process and lessens the need for character set migration expertise by guiding the DBA through the entire migration process, as well as automating many of the migration tasks. It comes with a scalable, in-place migration architecture, which significantly reduces the effort and down time required for data conversion, comparing to conventional export and import migration methods.

For post-migration and existing databases that are already using the Unicode character set, DMU also has a validation mode, where it identifies data that is not correctly encoded in Unicode, thus providing a health check on potential issues with implementation of Unicode in database applications.

Database Migration Assistant for Unicode: Benefits

- Alleviates costly manual workload through automated migration tasks
- Greatly simplifies the data preparation process and prevents data loss
- Provides robust error-handling and failure recovery
- Provides health check on data integrity in databases encoded with the Unicode Standard
- Supports migration of 12*c* pluggable databases to Unicode
- Supports enhanced bulk cleansing features with pattern-based replacement

The pluggable databases in the multitenant architecture of Oracle 12*c* refer to many databases being managed together in one Oracle Database instance. Using Unicode for these databases enables the support of character data in any languages and assures maximum compatibility among the databases that are to be consolidated. It is recommended that you use DMU to migrate the databases to Unicode before consolidating them as pluggable databases. However, with DMU 2.0, the migration, especially the data conversion phase, can also be performed after the consolidation.

Other new features in DMU 2.0 include enhanced bulk cleansing features with pattern-based replacement to resolve certain classes of data convertibility issues, support for migrating PeopleSoft databases to Unicode, and improved conversion error-handling mechanisms.

You can learn more about the DMU feature in the *Oracle Database Migration Assistant for Unicode Guide.*

Unicode and UCA Conformance

Unicode Update

- It allows text and symbols from all languages to be consistently represented and manipulated by computers.
- It provides uniform representation of textual information, independent of the platform and programming language.
- Oracle Database 12*c* supports Unicode Standard 6.1.

Unicode Collation Algorithm (UCA) Conformance

- Oracle Database 12*c* introduces UCA support in addition to the existing database monolingual and multilingual linguistic collations.
- You can enable more fine-grained control on searching, sorting, and matching of multilingual data by using UCA collations.

ORACLE

Unicode: Unicode is an industry standard that allows text and symbols from all languages to be consistently represented and manipulated by computers. It provides uniform representation of textual information, independent of the platform or programming language. Oracle Database 11*g* R2 supports Unicode 5.0. Currently, Oracle Database 12*c* supports Unicode 6.1, which is the latest version. You must keep Oracle up-to-date with the latest Unicode Standard for compatibility and from a competitive perspective.

As of version 6.1, the Unicode Standard encodes more than 110,000 characters. Improvements have also been made in character properties and collation algorithms. Oracle's Unicode database character set AL32UTF8 and national character set AL16UTF16 definitions have been updated to conform to Unicode 6.1. Using these character sets in Oracle Database 12*c* will equip you with the most comprehensive character repertoire support in multilingual databases for all your character data processing needs.

UCA: Unicode Collation Algorithm (UCA) is a Unicode Standard for determining the linguistic order of Unicode strings. UCA defines a Default Unicode Collation Element Table (DUCET) that supplies a reasonable default collation for all Unicode characters.

Oracle Database 12*c* introduces UCA support, in addition to the existing database monolingual and multilingual linguistic collations. Oracle's implementation of UCA is compliant with Unicode Standard 6.1. In addition to the collation based on DUCET, it provides customized collations for a number of commonly used languages.

UCA is the recommended mechanism for sorting multilingual data. With the newly added UCA collations, customers can attain more fine-grained control on how searching, sorting, and matching of multilingual data is performed.

The linguistic operations involve transforming the character data into binary values called collation keys before evaluating the relative order. Because the collation keys are represented in Oracle with the RAW data type, you can now sort longer text with higher precision in Oracle Database 12*c*. The maximum length limits of the `VARCHAR2`, `NVARCHAR2`, and `RAW` data types have now been extended to 32767 bytes.

Locale Coverage and Secure Files

- New Locale Coverage:
  - Oracle Database 12*c* has introduced a set of 12 new languages and 32 new territories to support database locales, covering additional regions of Asia, Africa, Americas, and Europe.
- SecureFiles:
  - These are made default storage mechanisms for large objects (`LOBs`).
  - The default value for `db_securefile` is `PREFERRED` when the compatible `init.ora` parameter is set to 12.0.0.0.0 or higher. To change this behavior, set the value to `PERMITTED`.
  - This provides optimal performance for storing unstructured data in the database.

Locale Coverage: Oracle Database 12*c* has introduced a set of 12 new languages and 32 new territories to the supported database locales, covering additional regions of Asia, Africa, Americas, and Europe to expand the scope of globalization support and address fast evolving customer requirements.

The new languages are Amharic, Armenian, Dari, Divehi, Khmer, Lao, Latin Bosnian, Maltese, Nepali, Persian, Sinhala, and Swahili.

The new territories are Afghanistan, Armenia, Bahamas, Belize, Bermuda, Bolivia, Bosnia and Herzegovina, Cambodia, Cameroon, Congo Brazzaville, Congo Kinshasa, Ethiopia, Gabon, Honduras, Iran, Ivory Coast, Kenya, Laos, Maldives, Malta, Montenegro, Nepal, Nigeria, Pakistan, Paraguay, Senegal, Serbia, Sri Lanka, Tanzania, Uganda, Uruguay, and Zambia.

It also includes support for the Ethiopian calendar, which is a calendar system based on the Coptic calendar with a 13th month of either 5 or 6 days in length.

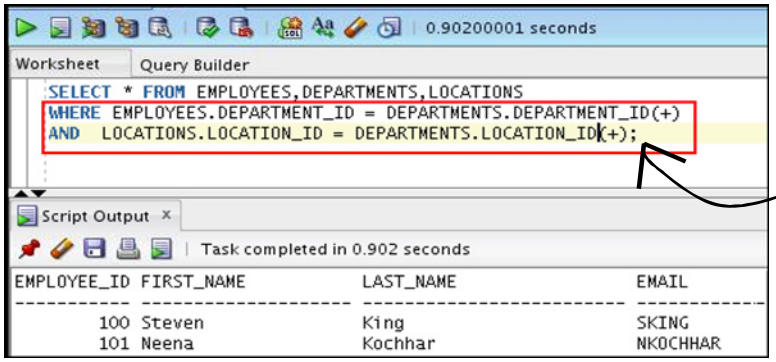
Secure Files: Beginning with Oracle Database 12*c*, SecureFiles are the default for large objects (LOB) storage. If no storage type is explicitly specified, new LOB columns use the SecureFiles LOB storage. When the compatible `init.ora` parameter is set to 12.0.0.0.0 or higher, the default value for `db_securefile` is `PREFERRED`. To change this behavior, set the value to `PERMITTED`.

SecureFiles provide optimal performance while storing unstructured data in the database. Making SecureFiles the default for unstructured data helps to ensure that the database delivers the best possible performance when managing unstructured data.

LEFT OUTER JOIN Extension

- Oracle Database 12*c* provides an extension of the Oracle native LEFT OUTER JOIN syntax that allows multiple tables on the left hand side of an outer join.
- Example:

```
SELECT * FROM A, B, D
WHERE A.c1 = B.c2(+) and D.c3 = B.c4(+);
```



ORACLE

Oracle Database 12*c* provides an extension of the Oracle native LEFT OUTER JOIN syntax that allows multiple tables on the left hand side of an outer join. In earlier releases of Oracle Database, in a query that performed outer joins of more than two pairs of tables, a single table could be the null-generated table for only one other table, and having multiple tables on the left hand side of an outer join was illegal and resulted in an ORA-01417 error. The only way to execute such a query was to translate it into ANSI syntax. Beginning with Oracle Database 12*c*, a single table can be the null-generated table for multiple tables.

In the code example in the slide, the null-generated table B is outer-joined to two tables: A and D.

The benefits of extending the left outer join are as follows:

- It allows merging of multiple table views, which originate from a user query or are generated during conversion from the ANSI LEFT OUTER JOIN syntax on the left outer join.
- Merging of views allows more join reordering and therefore, more optimal execution plans.
- Views are merged heuristically, without undergoing cost-based transformation.
- It reduces the burden of formulating queries in terms of views or the LEFT OUTER JOIN syntax.

Summary

In this lesson, you should have learned how to:

- Use the `FETCH` clause to limit the rows returned by a query
- Create an invisible column in a table
- Utilize the enhanced DDL capabilities by using the `ONLINE` keyword
- Use the additional new features available in the SQL language

In this lesson, you should have learned how to use the `FETCH` clause to limit the rows returned by a query, create an invisible column in a table, utilize enhanced DDL capabilities by using the `ONLINE` keyword, and also make use of some additional new features that are available in the SQL language.

3 Data Type Enhancements

Objectives

After completing this lesson, you should be able to:

- Identify the increased length limits of data types and configure the database for extended data types
- Describe the `SQL IDENTITY` column
- Use the SQL column enhancements defaulting on explicit `NULL` and using a sequence
- Describe the additional features that are available in the `ANYDATA` and `GATEWAY` data types

In this lesson, you learn about the increased length limits of data types and configuring a database for an extended data type, the `SQL IDENTITY` column, SQL column enhancements defaulting on explicit `NULL` and using a sequence, and some additional new data type features such as `ANYDATA` and `GATEWAY` that are available in Oracle Database 12*c*.

Lesson Agenda

- Increased length limits of data types
- Using the `SQL IDENTITY` Column
- SQL column enhancements
- `ANYDATA` type enhancements
- Gateway enhancements

Increased Length Limits of Data Types

- Maximum size for the VARCHAR2, NVARCHAR2, and RAW data types has been increased from 4000 to 32767 bytes.

The VARCHAR2 and NVARCHAR2 columns with a declared column length of 4000 bytes or less and RAW columns with a declared column length of 2000 bytes or less are stored inline.

The VARCHAR2 and NVARCHAR2 columns with a declared column length of greater than 4000 bytes and RAW columns with a declared column length of greater than 2000 bytes are called extended character data type columns and are stored out- of- line.

- MAX_STRING_SIZE controls the maximum size of the extended data types in SQL.

```
MAX_STRING_SIZE = { STANDARD | EXTENDED }
```


In Oracle Database 12*c*, you can specify a maximum size of 32767 bytes for the VARCHAR2, NVARCHAR2, and RAW data types. This enables users to store longer strings in the database. Before this release, the maximum size was 4000 bytes for the VARCHAR2 and NVARCHAR2 data types and 2000 bytes for the RAW data type. The declared length of the VARCHAR2, NVARCHAR2, or RAW column affects how the column is internally stored.

- VARCHAR2 and NVARCHAR2 columns with a declared column length of 4000 bytes or less and RAW columns with a declared column length of 2000 bytes or less are stored inline.
- VARCHAR2 and NVARCHAR2 columns with a declared column length of greater than 4000 bytes and RAW columns with a declared column length of greater than 2000 bytes are called “extended character data type columns” and are stored out-of-line.

MAX_STRING_SIZE controls the maximum size of the extended data types in SQL:

- STANDARD means the length limit of the data types used before Oracle Database 12*c*.
- EXTENDED means the 32767 bytes limit in Oracle Database 12*c*.

You must set the COMPATIBLE initialization parameter to 12.0.0.0 or later to set MAX_STRING_SIZE = EXTENDED.

Extended character data types have the following restrictions:

- Not supported in clustered tables and index-organized tables
- No intra-partition parallel DDL operations, UPDATE operation, and DELETE DML operation
- No intra-partition, parallel, and direct-path inserts on tables that are stored in a tablespace that is managed with Automatic Segment Space Management (ASSM)

Configuring the Database for Extended Data Types

1. Shut down the database instance.
2. Restart the database in `UPGRADE` mode.
3. Change the setting of `MAX_STRING_SIZE` to `EXTENDED`;

```
SQL> ALTER SYSTEM SET MAX_STRING_SIZE = EXTENDED;
```

4. Run the `$ORACLE_HOME/rdbms/admin/utl32k.sql` script as `SYSDBA`.
5. Restart the database instance.

Notes

- You cannot change the value from `EXTENDED` to `STANDARD`.
- In a RAC environment, shut down all instances.

ORACLE

You cannot create a table with extended character data type columns until you configure the database with the instance parameter `MAX_STRING_SIZE` and set it to `EXTENDED`. This value is also enforced to be the same on all RAC nodes in a RAC environment.

Follow the steps described in the slide.

Notes

- `UPGRADE` mode starts the database in `OPEN UPGRADE` mode and sets the system initialization parameters to specific values that are required to enable the database upgrade scripts to be run. For more information, refer to the *Oracle Database Upgrade Guide* at https://docs.oracle.com/cd/B19306_01/server.102/b14238/toc.htm.
- The `utl32k.sql` script increases the maximum size of the `VARCHAR2`, `NVARCHAR2`, and `RAW` columns for the views where required. The script does not increase the maximum size of the `VARCHAR2`, `NVARCHAR2`, and `RAW` columns in some views because of the way the SQL for those views is written.

Extended Data Types: Example

- **CREATE TABLE**: You can specify a 32k column during CREATE TABLE by simply specifying the appropriate length in the data type specification.
- 32k columns are allowed only if the table is not:
  - A clustered table
  - An index-organized table
- Example:

```
CREATE TABLE long_varchar(id int,vc varchar2(32767));
```

Extended Data Types: Example

ALTER TABLE MODIFY (column ...):

- You can modify the size of existing VARCHAR2, NVARCHAR2, and RAW columns with the ALTER TABLE MODIFY (column ...) statement.
- Example:

```
ALTER TABLE t MODIFY (varchar_column VARCHAR2(32767));
```

ALTER TABLE ADD (column...):

- You can add a 32k column to an existing table via ALTER TABLE ADD DDL if the table satisfies the conditions required for 32k types.
- Example:

```
ALTER TABLE t ADD (long_varchar_column VARCHAR2(12345));
```

ORACLE

You can modify the size of existing VARCHAR2, NVARCHAR2, and RAW columns with the ALTER TABLE MODIFY (column ...) statement. However, Oracle recommends that you do not excessively increase the size of an existing VARCHAR2 column beyond 4000 bytes. The following are the reasons:

- Row chaining may occur.
- Any data that is stored inline must be read in its entirety, whether or not a column is selected. The extended character data type columns that are stored inline can therefore negatively impact performance.
- To migrate to the new out-of-line storage of the extended character data type columns, you must re-create the tables. The inline storage of the column is not preserved during any type of table reorganization.

You can also add a 32k column to an existing table via ALTER TABLE ADD DDL if the table satisfies the conditions required for 32k types, as described earlier. The slide shows an example of extended data types.

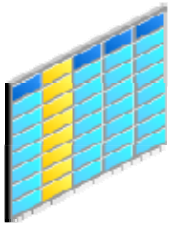
Note that the table is not a clustered table or an index-organized table.

Lesson Agenda

- Increased length limits of data types
- Using the `SQL IDENTITY` column
- SQL column enhancements
- `ANYDATA` type enhancements
- Gateway enhancements

SQL Identity Column

- The SQL table column definition syntax has been enhanced to support the `ANSI IDENTITY` keyword.
- The identity column is assigned an increasing or decreasing integer value from a sequence generator for each subsequent `INSERT` statement.
- You can use `identity_clause` to specify an identity column.
- You can use the `identity_options` clause to configure the sequence generator.
- The definition of an identity column may specify `ALWAYS` or `BY DEFAULT`.



The SQL table column definition syntax has been enhanced to support the `ANSI IDENTITY` keyword. The identity column is assigned an increasing or decreasing integer value from a sequence generator for each subsequent `INSERT` statement. This provides a standards-based approach to the declaration of automatically incrementing columns, simplifying application development and making the migration of DDL to Oracle database simpler. You can use `identity_clause` to specify an identity column.

`ALWAYS`: If you specify `ALWAYS`, Oracle Database always uses the sequence generator to assign a value to the column. If you attempt to explicitly assign a value to the column by using `INSERT` or `UPDATE`, an error is returned. The column is not updatable, so the default value is the only possible value for the column.

`BY DEFAULT`: If you specify `BY DEFAULT`, Oracle Database uses the sequence generator to assign a value to the column by default, but you can also explicitly assign a specified value to the column. If you specify `ON NULL`, Oracle Database uses the sequence generator to assign a value to the column when a subsequent `INSERT` statement attempts to assign a value that evaluates to `NULL`.

The details of the sequence generator are hidden from the user and automatically maintained. Thus, identity columns provide a simple way to default a numeric value from a sequence generator.

SQL Identity Column: Syntax

You specify the column definition syntax in CREATE TABLE or ALTER TABLE.

Syntax:

```
column definition::=
column datatype
[SORT]
[DEFAULT expr | identity ]
[ENCRYPT encryption spec ]
[ ( { inline_constraint }…)
   | inline_ref_constraint
]
Identity::=
GENERATED [ ALWAYS | BY DEFAULT [ON NULL] ]AS IDENTITY
[( <options> )]
```


You specify the column definition syntax in CREATE TABLE or ALTER TABLE. You can use the identity_options clause to configure the sequence generator.

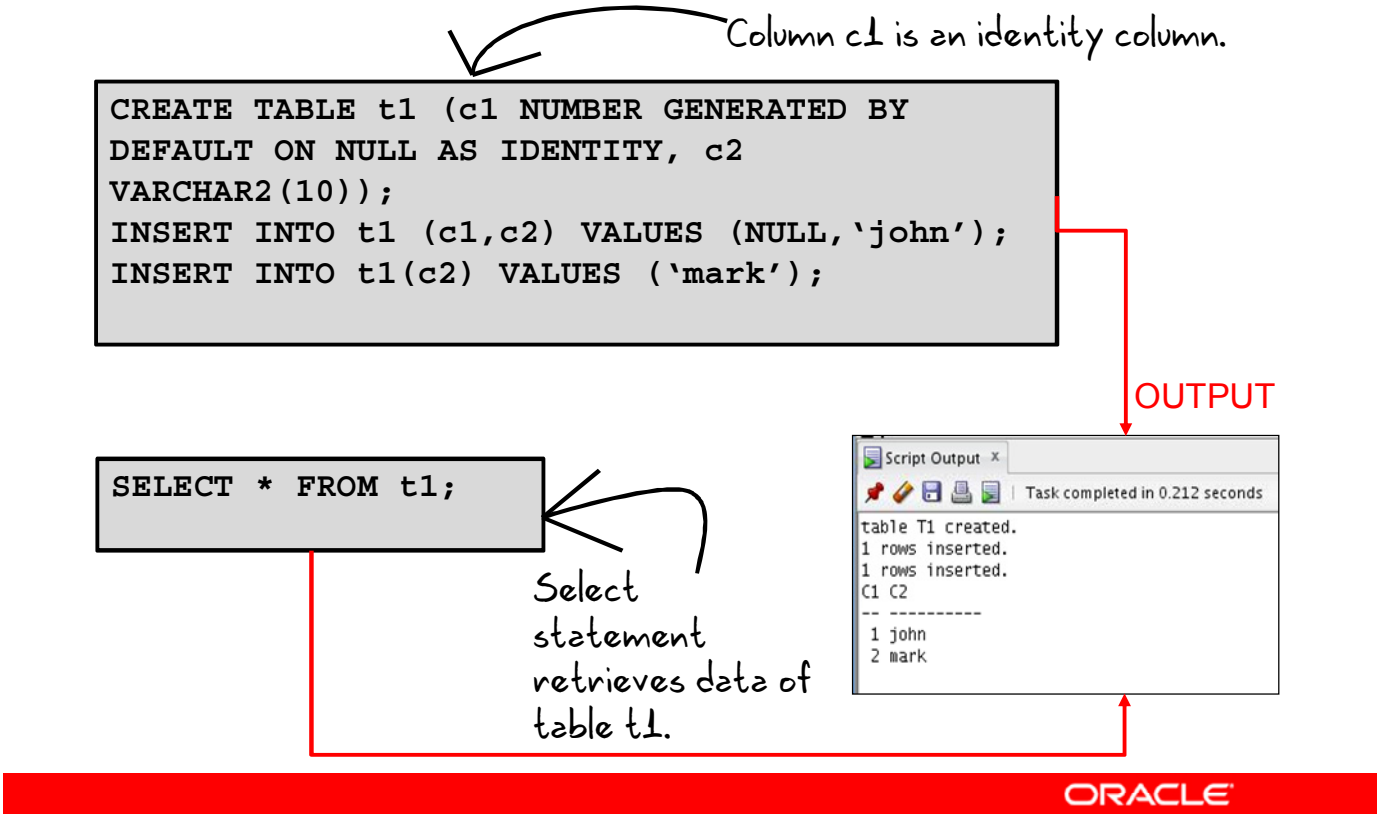
The options are shown as follows:

```
[START WITH (<sequence generator start value> | LIMIT VALUE)
   | INCREMENT BY <sequence generator increment>
   | ( MAXVALUE <sequence generator max value> | NO MAXVALUE )
   | ( MINVALUE <sequence generator min value> | NO MINVALUE )
   | ( CYCLE | NO CYCLE )
   | (CACHE integer | NOCACHE)
   | (ORDER | NOORDER)]+
```

SQL IDENTITY columns have the following restrictions:

- You can specify only one identity column per table.
- If you specify identity_clause, you must specify a numeric data type for the data type in the column definition clause. You cannot specify a user-defined data type.
- If you specify identity_clause, you cannot specify the DEFAULT clause in the column definition clause. When you specify identity_clause, the NOT NULL constraint and NOT DEFERRABLE constraint states are implicitly specified. If you specify an inline constraint that conflicts with NOT NULL and NOT DEFERRABLE, an error is raised.

SQL Identity Column: Example



The code example in the slide shows how to create the `SQL IDENTITY` column in the `CREATE TABLE` statement. Note that the column `c1` is an identity column and that the `SELECT` statement retrieves data from the table `t1`.

Quiz

The definition of the `IDENTITY` column may specify a `table_reference` or a `column_reference`.

a. True
b. False

Answer: a

Practice 3-2 Overview: Using the SQL Identity Column

This practice covers creating a table with multiple columns (`number` and `varchar2` data types) and specifying the number column with the `IDENTITY` keyword to automatically generate a sequence of numbers for that column.

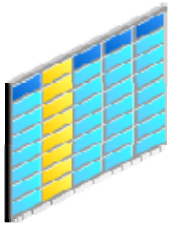
Lesson Agenda

- Increased length limits of data types
- Using the SQL Identity Column
- SQL column enhancements
- `ANYDATA` type enhancements
- Gateway enhancements

SQL Column Enhancements

The SQL column enhancements are:

- SQL column defaulting on explicit `NULL`
- SQL column defaulting by using a sequence



Beginning with Oracle Database 12*c*, the SQL column enhancements are:

- SQL column defaulting on explicit `NULL`
- SQL column defaulting by using a sequence

SQL Column Defaulting on Explicit NULL

- The SQL syntax is enhanced to allow SQL column defaulting on a NOT NULL column when a user specifies a null value for the column in the VALUES clause or the subquery of a SQL insert DML.
- For column defaulting on explicit NULL, you add the ON NULL keyword after the DEFAULT keyword in a CREATE or ALTER TABLE DDL.
- If you specify the ON NULL clause, Oracle Database assigns the DEFAULT column value when a subsequent INSERT statement attempts to assign a value that evaluates to NULL.

The SQL syntax is enhanced so that it allows SQL column defaulting on a NOT NULL column when a user specifies an expression that evaluates to NULL for the column in the VALUES clause or subquery of a SQL insert DML. Column defaulting on explicit NULL is achieved by adding the ON NULL keyword after the DEFAULT keyword in a CREATE or ALTER TABLE DDL.

If you specify the ON NULL clause, Oracle Database assigns the DEFAULT column value when a subsequent INSERT statement attempts to assign a value that evaluates to NULL. When you specify ON NULL, the NOT NULL constraint and NOT DEFERRABLE constraint states are implicitly specified. If you specify an inline constraint that conflicts with NOT NULL and NOT DEFERRABLE, an error is raised.

Notes

- **Identity column:** If you specify the ON NULL clause, Oracle Database uses the sequence generator to assign a value to the column when a subsequent INSERT statement attempts to assign a value that evaluates to NULL.
- **Regular column:** If you specify ON NULL, Oracle Database assigns the DEFAULT column value when a subsequent INSERT statement attempts to assign a value that evaluates to NULL.

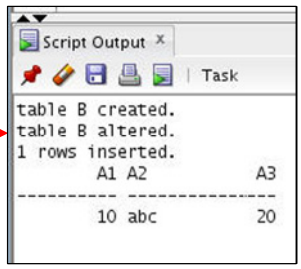
SQL Column Defaulting on Explicit NULL: Example

- The column definition syntax is specified in CREATE or ALTER TABLE:

```
column definition::=
column datatype [SORT]
 [ DEFAULT [ON NULL] expr ]
 [ ENCRYPT encryption spec ]
 [ ( {inline constraint}…)
   | inline_ref_constraint
 ]
```

- Example:

```
CREATE TABLE b(a1 NUMBER DEFAULT ON NULL 10
NOT NULL, a2 VARCHAR2(30));
ALTER TABLE b ADD(a3 NUMBER DEFAULT ON NULL 20
NOT NULL);
INSERT INTO b (a1, a2) VALUES (NULL,'abc');
SELECT * FROM b;
```



The column definition syntax is specified in CREATE or ALTER TABLE. The code example in the slide creates table b, where the a1 column is defined with a DEFAULT ON NULL column value of 10. Therefore, if a subsequent INSERT statement attempts to assign a NULL value to a1, the value of 10 is assigned instead.

SQL Column Defaulting by Using a Sequence

- The SQL syntax for column defaults now allows `<sequence>.nextval`, `<sequence>.currval` as a SQL column defaulting expression for numeric columns, where `<sequence>` is an Oracle Database sequence.
- The `DEFAULT` expression can include the sequence pseudocolumns `CURRVAL` and `NEXTVAL`, as long as the sequence exists and you have the necessary privileges to access it.
- To perform subsequent inserts that use the `DEFAULT` expression, you must have the `INSERT` privilege on the table and the `SELECT` privilege on the sequence.

The SQL syntax for column defaults is enhanced so that it allows `<sequence>.nextval`, `<sequence>.currval` as a SQL column defaulting expression for numeric columns, where `<sequence>` is an Oracle Database sequence. The `DEFAULT` expression can include the sequence pseudocolumns, `CURRVAL` and `NEXTVAL`, as long as the sequence exists and you have the necessary privileges to access it.

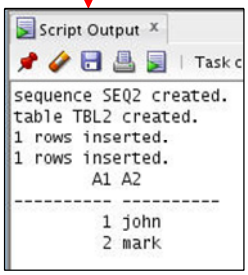
To perform subsequent inserts that use the `DEFAULT` expression, you must have the `INSERT` privilege on the table and the `SELECT` privilege on the sequence. If the sequence is dropped later, subsequent `INSERT` statements where the `DEFAULT` expression is used will result in an error. If you do not fully qualify the sequence by specifying the sequence owner, for example, `SCOTT.SEQ1`, Oracle Database defaults the sequence owner to be the user who issues the `CREATE TABLE` statement. For example, if user `MARY` creates `SCOTT.TABLE` and refers to a sequence that is not fully qualified, such as `SEQ2`, the column uses the `MARY.SEQ2` sequence. The synonyms on sequences undergo a full name resolution and are stored as a fully qualified sequence in the data dictionary; this is true for public and private synonyms.

If you specify the `DEFAULT` clause for a column, the default value is stored as metadata but the column itself is not populated with data. However, subsequent queries that specify the new column are rewritten so that the default value is returned in the result set. Therefore, adding new columns with `DEFAULT` values no longer requires the default value to be stored in all existing records. This not only enables schema modification in sub-seconds, independent of the existing data volume, it also consumes no space.

SQL Column Defaulting by Using a Sequence: Example

```
CREATE SEQUENCE seq2 START WITH 1;
CREATE TABLE tbl2 (a1 NUMBER DEFAULT
seq2.NEXTVAL NOT NULL, a2
VARCHAR2(10));
INSERT INTO tbl2 (a2) VALUES ('john');
INSERT INTO tbl2 (a2) VALUES ('mark');
SELECT * FROM tbl2;
```

Sequence seq2 is created, which starts from 1.



The code in the slide is an example of a SQL column defaulting by using a sequence. Note that the sequence `s1` is created, which starts from `1`.

Quiz

The `DEFAULT` expression can include the sequence pseudocolumns, `CURRVAL` and `NEXTVAL`, as long as the sequence exists and you have the privileges necessary to access it.

a. True
b. False

Answer: a

Quiz

SQL column defaulting on explicit `NULL` is achieved by adding:

a. `ON NULL`

b. `ON NOT NULL`

c. `NULL`

Answer: a

Practice 3-3 Overview: SQL Column Enhancements

This practice covers the following topics:

- Creating a table with a numeric column and defaulting the column value to 10 on null and observing that when a null value is inserted into the table for this column, the column value defaults to 10
- Creating a sequence that starts with the value `1` and also creating a table that uses the sequence in its column. When null values are inserted into the table for the column by using the sequence, the column is assigned the default sequence values.

Lesson Agenda

- Increased length limits of data types
- Using the SQL Identity Column
- SQL column enhancements
- `ANYDATA` type enhancements
- Gateway enhancements

ANYDATA Data Type Enhancements

The ANYDATA data type:

- Contains an instance of a given type with data, as well as a description, of the type
- Can be used as a table column data type
- Allows storing heterogeneous values in a single column
- Can includes values of SQL built-in types, as well as user-defined types
- Supports embedded XML and CLOB
- Is supported by data pump with embedded CLOB and XMLTYPE
- Works with JDBC without additional need for data transformation

The ANYDATA data type contains an instance of a given type with data, as well as a description, of the type. ANYDATA can be used as a table column data type and allows you to store heterogeneous values in a single column. The values can be of SQL built-in types as well as user-defined types. The enhancements described increase the flexibility of the Oracle ANYDATA implementation and it can now be used with database editions.

The ANYDATA data type supports embedded XML and CLOB. The size of the XML or CLOB is not expected to be more than a few KB. Data pump supports the ANYDATA data type with embedded CLOB and XMLTYPE. ANYDATA works with JDBC without additional need for data transformation.

ANYDATA to Support Embedded XML and CLOB

Storage of LOBs or XMLType inside an ANYDATA column is possible if you specifically request the storage by using the following ALTER TABLE syntax:

```
ALTER TABLE <table_name> MODIFY OPAQUE TYPE
<anydata_column_name> STORE (<types to be stored as unpacked>)
UNPACKED;
```

```
ALTER TABLE [ schema. ] table
  [ alter_table_properties
 | column_clauses
 | constraint_clauses
 | alter_table_partitioning
 | alter_external_table
 | move_table_clause
 | modify_opaque_type   ]    [ enable_disable_clause
 | { ENABLE | DISABLE } { TABLE LOCK | ALL TRIGGERS }   ] ...
  ;
```

In Oracle Database 11*g* R2, the storage of LOBs or XMLType inside an ANYDATA column, either directly or as an attribute of an abstract data type (ADT), was not possible.

As of Oracle Database 12*c*, users can store LOBs or XMLTYPE inside an ANYDATA column by using the ALTER TABLE syntax. This statement creates regular ADT columns for the specified types and associates them with the ANYDATA column. These columns are hidden and are totally opaque to the user. You insert or read from the ANYDATA column just like before, and you manipulate the images by using the ANYDATA functions and procedures.

ALTER TABLE also creates a new hidden column to store the type of data.

Gateway Enhancements

Gateway support for large VARCHAR

- Gateways previously mapped non-Oracle VARCHAR data that was longer than 4000 bytes to the Oracle LONG data type.
- The limit is now 32,767 bytes.

Oracle Database Gateways address the need for disparate data access. In a heterogeneously distributed environment, Gateways make it possible to integrate with any number of non-Oracle systems from an Oracle application. They enable integration with data stores such as IBM DB2, Microsoft SQL Server and Excel, transaction managers such as IBM CICS, and message queuing systems such as IBM WebSphere MQ.

In Oracle Database 11*g*, Gateways mapped non-Oracle VARCHAR data longer than 4,000 bytes to the Oracle LONG data type. In Oracle Database 12*c*, the limit is now 32,767 bytes.

Summary

In this lesson, you should have learned how to:

- Increase and configure the length limits of data types
- Use the `SQL IDENTITY` column
- Use the SQL enhancement features
- Use the `ANYDATA` data type and Gateway enhancements

In this lesson, you should have learned how to increase and configure the length limits of data types, use the SQL Identity column, use the SQL enhanced features, and also use the `ANYDATA` data type and Gateway enhancements.

4

PL/SQL Enhancements

Objectives

After completing this lesson, you should be able to:

- Describe fine-grained access control for packaged procedures
- Define the Invoker's Rights function that can be result cached
- Grant roles to PL/SQL packages and stand-alone stored subprograms
- Identify the PL/SQL functions that can run faster in SQL
- Identify some additional PL/SQL feature enhancements

ORACLE

In this lesson, you learn about using fine-grained access control for packaged procedures, defining the Invoker's Rights function that can be result cached, granting roles to PL/SQL packages and stand-alone stored subprograms, identifying the PL/SQL functions that can run faster in SQL, and identifying some additional PL/SQL feature enhancements that are available in Oracle Database 12*c*.

Lesson Agenda

- Fine-grained access control for package procedures
- Result cache with the Invoker's Rights program unit
- Granting roles to PL/SQL packages and stand-alone stored subprograms
- The PL/SQL functions that run faster in SQL
- Additional PL/SQL enhancements

What Is a White List?

- A “white list” is a list of programs that are allowed to access a program unit.
- The white list clause can be used in conjunction with an `AUTHID` clause in any order.
- Only one “white list” clause is allowed on a PL/SQL unit.
- The `ACCESSIBLE BY` clause in a PL/SQL database object allows you to specify a white list of PL/SQL units that can access the PL/SQL database object.

```
white_list_clause ::= ACCESSIBLE BY ( accessor_list )
  accessor_list     ::= accessor | accessor_list , accessor
  accessor          ::= [ accessor_kind ] accessor_name
  accessor_name     ::= identifier | identifier . identifier
  accessor_kind     ::= PROCEDURE | FUNCTION | PACKAGE | TRIGGER | TYPE
```


- A “white list” is a list of programs that are allowed to access a program unit. It is a new clause that is added near the beginning of a PL/SQL database object. A PL/SQL unit that includes a white list is known as a definer and a unit that accesses a definer is known as an invoker.
- The `ACCESSIBLE BY` clause of the package specification allows you to specify a “white list” of PL/SQL units that can access the package. Some examples of situations where you can use this clause are as follows:
  - When you implement a PL/SQL application that contains a package that provides the application programming interface (API) and helper packages to do the work. You want clients to have access to the API, but not to the helper packages. In such a case, you can omit the `ACCESSIBLE BY` clause from the API package specification and include it in each helper package specification, where you specify that only the API package can access the helper package.
  - You create a utility package to provide services to some, but not all, PL/SQL units in the same schema. To restrict the use of the package to the intended units, you list them in the `ACCESSIBLE BY` clause in the package specification.
- The `ACCESSIBLE BY` clause specifies each `accessor` (PL/SQL unit) that can invoke the function. An `accessor` can appear more than once in the `ACCESSIBLE BY` clause, but the `ACCESSIBLE BY` clause can appear only once in the function.

CREATE PROCEDURE Statement: Using the ACCESSIBLE BY Clause

- The syntax for using the ACCESSIBLE BY clause when creating a procedure is:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(parameter1 [mode] datatype1,
   parameter2 [mode] datatype2, ...)]
AUTHID [CURRENT_USER| DEFINER]
ACCESSIBLE BY (accessor)
IS|AS
  [local_variable_declarations; ...]
BEGIN
      -- actions;
END [procedure_name];
```

where:

- accessor := [accessor_ kind] accessor_ name
- accessor_kind := [PROCEDURE | FUNCTION | PACKAGE | TRIGGER | TYPE]

ORACLE

The code example in the slide shows the syntax of using the ACCESSIBLE BY clause in the CREATE PROCEDURE statement.

An accessor_name that consists of a single identifier is regarded as the name of a PL/SQL database object or trigger in the same schema as the object in which the white list appears.

An accessor_name that consists of an identifier pair with an internal period is regarded as the schema-qualified name of a PL/SQL database object or of a trigger.

The accessor_kind is optional, except when the accessor_name is intended to name a trigger. Trigger names do not appear in the normal PL/SQL namespace.

The ACCESSIBLE BY clause can be used with the following statements:

- CREATE FUNCTION
- CREATE PACKAGE
- CREATE TYPE
- ALTER TYPE

Quiz

The `ACCESSIBLE BY` clause in a PL/SQL database object allows you to specify a white list of PL/SQL units that can access the PL/SQL database object.

a. True
b. False

Answer: a

The `ACCESSIBLE BY` clause in a PL/SQL database object allows you to specify a white list of PL/SQL units, thereby allowing you to specify the PL/SQL units that can invoke the PL/SQL database object.

Practice 4-1 Overview: Using the `ACCESSIBLE BY` Clause in PL/SQL Database Objects

This practice covers creating a procedure that has the `ACCESSIBLE BY` clause with a package and a procedure as parameters that can access the procedure.

Lesson Agenda

- Fine-grained access control for packaged procedures
- Result cache with the Invoker's Rights program unit
- Granting roles to PL/SQL packages and stand-alone stored subprograms
- The PL/SQL functions that run faster in SQL
- Additional PL/SQL enhancements

Defining Invoker's Right and Definer's Right

- A unit whose AUTHID value is CURRENT_USER is called an Invoker's Rights unit or IR unit.
- A unit whose AUTHID value is DEFINER is called a Definer's Rights unit or DR unit.
- The PL/SQL units and schema objects for which you cannot specify an AUTHID value work as follows:

View Name	Column Name
Anonymous block	Invoker' s Right unit
BEQUEATH CURRENT_USER view	Similar to an Invoker's Right unit
BEQUEATH DEFINER view	Definer's Right unit
Trigger	Definer's Right unit

ORACLE

The AUTHID property of a stored PL/SQL unit affects the name resolution and privilege checking of SQL statements that the unit issues at run time. The AUTHID property does not affect compilation, and has no meaning for units that have no code, such as collection types.

AUTHID property values are exposed in the static data dictionary view *_PROCEDURES. For units for which AUTHID has meaning, the view shows the value CURRENT_USER or DEFINER; for other units, the view shows NULL.

The table in the slide shows the behavior of the PL/SQL units and schema units for which you cannot specify an AUTHID.

Now you will see how a PL/SQL function is declared with both AUTHID CURRENT_USER and RESULT CACHE.

Defining a PL/SQL Function with the CURRENT_USER and RESULT_CACHE Clauses

```
CREATE OR REPLACE FUNCTION get_hire_date (emp_id NUMBER)
 RETURN VARCHAR
 RESULT_CACHE
 AUTHID CURRENT_USER
IS
  date_hired DATE;
BEGIN
  SELECT hire_date INTO date_hired
    FROM HR.EMPLOYEES
      WHERE EMPLOYEE_ID = emp_id;
  RETURN TO_CHAR(date_hired);
END;
```

Creating a PL/SQL function by using the AUTHID CURRENT_USER and RESULT CACHE

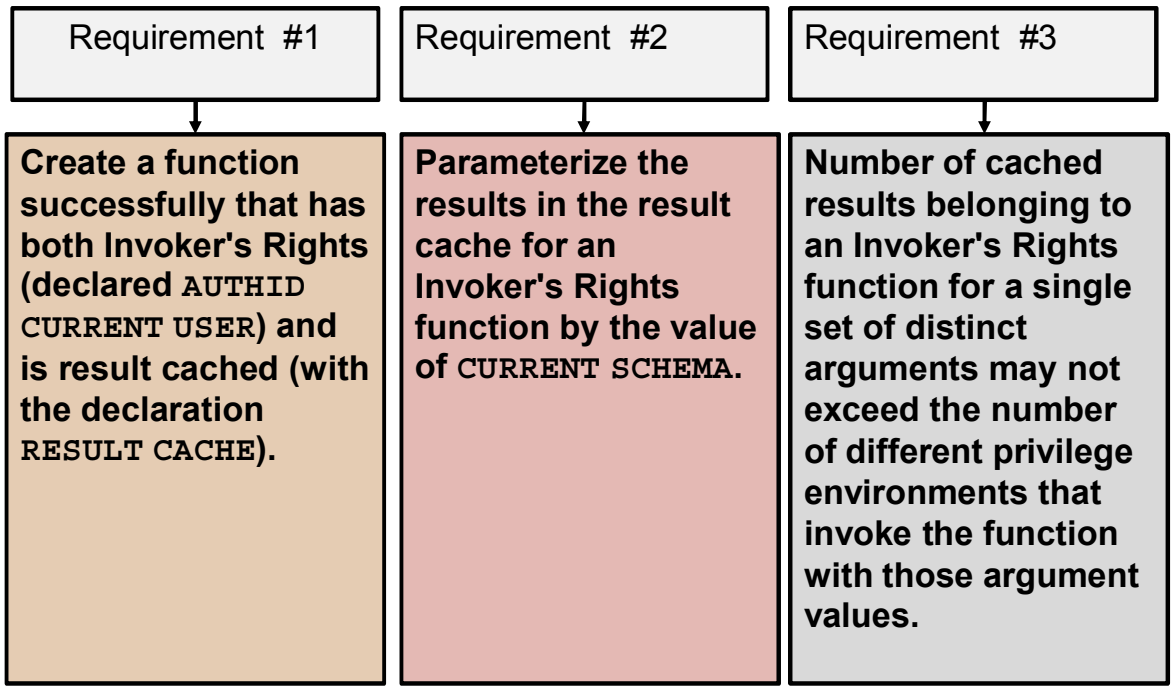
```
SQL> select get_hire_date(206) from dual;

GET_HIRE_DATE(206)
--------------------------------------------------------------------------------
07-JUN-02

SQL> select hire_date, employee_id from employees where employee_id = 206;

HIRE_DATE EMPLOYEE_ID
--------- -----------
07-JUN-02         206
```

ORACLE

As of 12*c*, the Invoker's Rights function can be result cached. The code example in the slide depicts the creation of a function `get_hire_date` that takes an `employee id` as parameter and returns the hire date. Notice that the function is result cached and the `AUTHID` property is set to Invoker's Right.

The `get_hire_date` function uses the `TO_CHAR` function to convert a `DATE` item to a `VARCHAR` item. The function `get_hire_date` does not specify a format mask, so the format mask defaults to the one that `NLS_DATE_FORMAT` specifies.

Requirements for Result-Caching Invoker's Right



There are a couple of requirements that ensure that the Invoker's Rights function is result cached.

You can create a function successfully that has both Invoker's Rights (declared `AUTHID CURRENT USER`) and is result cached (with the declaration `RESULT CACHE`).

The results in the result cache for an Invoker's Rights function must be parameterized by the value of `CURRENT SCHEMA` in the invoking environment if any name resolution that is sensitive to the value of this setting occurs while the result is being computed.

The number of cached results that belong to an Invoker's Rights function for a single set of distinct argument values may not exceed the number of different privilege environments that invoke the function with those argument values. This effectively places an upper bound on the amount of space that an Invoker's Rights function may consume per set of distinct argument values. If any privilege-sensitive operation is performed while building a cache result for an Invoker's Rights function, the result must somehow be parameterized by that privilege.

Use of the BEQUEATH Clause in the CREATE VIEW Statement

The BEQUEATH clause:

- Enables an Invoker's Rights function to be executed by using the rights of the user that is issuing the SQL that references the view
- Can be set to CURRENT_USER in the CREATE VIEW statement
- Example:

```
CREATE VIEW MY_OBJECTS_VIEW BEQUEATH CURRENT_USER AS
 SELECT GET_OBJS_FUNCTION;
```


To enable an Invoker's Rights function to be executed by using the rights of the user that is issuing the SQL that references the view, in the CREATE VIEW statement, you can set the BEQUEATH clause to CURRENT_USER.

If you plan to issue a SQL query or DML statement against the view, the view owner must be granted the INHERIT PRIVILEGES privilege on the invoking user or the view owner must have the INHERIT ANY PRIVILEGES privilege. If not, when a SELECT query or DML statement involves a BEQUEATH CURRENT_USER view, the runtime system raises the error ORA-06598: insufficient INHERIT PRIVILEGES privilege.

The example in the slide shows how to use BEQUEATH CURRENT_USER to set the view's function to be executed by using Invoker's Rights.

If you want the function within the view to be executed by using the view owner's rights, you should either omit the BEQUEATH clause or set it to DEFINER.

For example:

```
CREATE VIEW my_objects_view BEQUEATH DEFINER AS
SELECT OBJECT_NAME FROM USER_OBJECTS;
```

Practice 4-2 Overview: Invoker's Rights Function That Can Be Result Cached

This practice covers creating a function that can be result cached and has the `AUTHID` property set to Invoker's Rights.

- This function takes `emp_id` as the input parameter and returns the `hire_date` of the `empid` as the result.
- To view the result, invoke this function by passing a value for `empid`.

Lesson Agenda

- Fine-grained access control for packaged procedures
- Result cache with the Invoker's Rights program unit
- Granting roles to PL/SQL packages and stand-alone stored subprograms
- The PL/SQL functions that run faster in SQL
- Additional PL/SQL enhancements

Granting Roles to PL/SQL Packages and Stand-Alone Stored Subprograms

- You can grant roles to individual PL/SQL packages and stand-alone stored subprograms by:
  - Creating an Invoker's Rights unit (instead of a Definer's Right unit)
  - Granting the Invoker's Rights unit roles
- The IR unit runs with the privileges of both the invoker and the roles.

In Oracle Database Release 11*g*, a Definer's Rights (DR) unit always ran with the privileges of the definer and an Invoker's Rights (IR) unit always ran with the privileges of the invoker. If you wanted to create a PL/SQL unit that all users could invoke, even if their privileges were lower than yours, it had to be a DR unit. The DR unit always ran with all your privileges, regardless of which user invoked it.

As of Oracle Database Release 12*c*, you can grant roles to individual PL/SQL packages and stand-alone stored subprograms. Instead of a DR unit, you can create an IR unit, and then grant it roles. The IR unit runs with the privileges of both the invoker and the roles, but without any additional privileges that you have.

You grant roles to an IR unit, so that users with privileges lower than yours can run the unit with only the privileges that are needed to do so. There is no reason to grant roles to a DR unit, because its invokers run it with all your privileges.

Using the `SQL GRANT` command, you can grant roles to PL/SQL packages and stand-alone stored subprograms. Roles granted to a PL/SQL unit do not affect compilation. They affect the privilege checking of SQL statements that the unit issues at run time. The unit runs with the privileges of both its own roles and any other currently enabled roles.

Lesson Agenda

- Fine-grained access control for packaged procedures
- Result cache with the Invoker's Rights program unit
- Granting roles to PL/SQL packages and stand-alone stored subprograms
- The PL/SQL functions that run faster in SQL
- Additional PL/SQL enhancements

PL/SQL Functions That Run Faster in SQL

- The PL/SQL functions that are declared and defined in the `WITH` clauses of SQL `SELECT` statements
- The PL/SQL functions that are defined with “UDF Pragma”

As of Oracle Database 12*c*, the two kinds of PL/SQL functions listed in the slide might run faster in SQL.

PL/SQL Function Using the WITH SQL Clause: Example

```
WITH
  FUNCTION get_domain(url VARCHAR2) RETURN VARCHAR2 IS
    pos BINARY_INTEGER;
    len BINARY_INTEGER;
  BEGIN
    pos := INSTR(url, 'www.');
    len := INSTR(SUBSTR(url, pos + 4), '.') - 1;
     RETURN SUBSTR(url, pos + 4, len);
  END;
SELECT DISTINCT get_domain(catalog_url)
   FROM product_information;
/
```


The example in the slide declares and defines a PL/SQL function `get_domain` in the `WITH` clause. The `get_domain` function returns the domain name from a URL string, assuming that the URL string has the "`www`" prefix immediately preceding the domain name, and the domain name is separated by dots on the left and the right. The `SELECT` statement uses `get_domain` to find distinct catalog domain names from the `orders` table in the `oe` schema.

Lesson Agenda

- Fine-grained access control for packaged procedures
- Result cache with the Invoker's Rights program unit
- Granting roles to PL/SQL packages and stand-alone stored subprograms
- The PL/SQL functions that run faster in SQL
- Additional PL/SQL enhancements

Granting the INHERIT PRIVILEGES Privilege to Other Users

- By default, all users are granted INHERIT PRIVILEGES ON USER *newuser* TO PUBLIC when their accounts are created.
- Syntax:

```
GRANT INHERIT PRIVILEGES ON USER invoking_user TO
   procedure_owner;
```

- Example:

```
GRANT INHERIT PRIVILEGES ON USER jward TO ebrown;
```

  - Here, the statement enables any Invoker's Rights procedure that ebrown writes, or will write in the future, to access jward's privileges when jward runs it.

By default, all users are granted INHERIT PRIVILEGES ON USER newuser TO PUBLIC when their accounts are created or when accounts that were created earlier are upgraded to the current release.

The syntax for the grant of the INHERIT PRIVILEGES privilege is as follows:

```
GRANT INHERIT PRIVILEGES ON USER invoking_user TO procedure_owner;
```

In this specification:

- invoking_user is the user who runs the Invoker's Rights procedure. This user must be a database user account.
- procedure_owner is the user who owns the Invoker's Rights procedure. This value must be a database user account. As an alternative to granting the INHERIT PRIVILEGES privilege to the procedure's owner, you can grant the privilege to a role that is in turn granted to the procedure.

The following users or roles must have the INHERIT PRIVILEGES privilege granted to them by users who will run their Invoker's Rights procedures:

- Users or roles who own the Invoker's Rights procedures
- Users or roles who own the BEQUEATH CURRENT_USER views

Example:

```
GRANT INHERIT PRIVILEGES ON USER jward TO ebrown;
```

The statement enables any Invoker's Rights procedure that `ebrown` writes, or will write in the future, to access `jward`'s privileges when `jward` runs it.

The invoking user can revoke the `INHERIT PRIVILEGE` privilege from other users, and then grant it to only those users that are trusted.

Example:

```
REVOKE INHERIT PRIVILEGES ON USER jward FROM ebrown;
```

Granting the INHERIT ANY PRIVILEGES Privilege to Other Users

With INHERIT ANY PRIVILEGES

- By default, the user SYS has the INHERIT ANY PRIVILEGES system privilege and can grant this privilege to other database users or roles.
- A user's Invoker's Rights procedures have access to the privileges of the invoking user.
- Example:

```
GRANT INHERIT ANY PRIVILEGES TO ebrown;
```

  - Here, the INHERIT ANY PRIVILEGES privilege has been granted to the user ebrown.

By default, the user SYS has the INHERIT ANY PRIVILEGES system privilege and can grant this privilege to other database users or roles.

As with all ANY privileges, grant this privilege only to trusted users or roles. After a user or role has been granted the INHERIT ANY PRIVILEGES privilege, this user's Invoker's Rights procedures have access to the privileges of the invoking user. You can find the users who have been granted the INHERIT ANY PRIVILEGES privilege by querying the DBA_SYS_PRIVS data dictionary view.

The example in the slide shows how to grant the INHERIT ANY PRIVILEGES privilege to a Trusted Procedure Owner.

Note: Be careful about revoking the INHERIT ANY PRIVILEGES privilege from powerful users. For example, suppose the user SYSTEM has created a set of Invoker's Rights procedures. If you revoke INHERIT ANY PRIVILEGES from SYSTEM, other users cannot run that user's procedures, unless they have specifically been granted the INHERIT PRIVILEGE privilege.

Additional PL/SQL Enhancements

Before Oracle Database 12*c*	As of Oracle Database 12*c*
NA	A `DATABASE` event trigger can be created on a pluggable database.
A `LIBRARY` object could be defined only by using the explicit path, where a `DIRECTORY` object was intended as the single point of maintenance for file system paths.	A `LIBRARY` object can be defined as a `DIRECTORY` object and with the `CREDENTIAL` clause.
A view always behaved like a Definer's Rights (DR) unit.	A view can be either `BEQUEATH DEFINER` (the default), which behaves like a DR unit, or `BEQUEATH CURRENT_USER`, which behaves somewhat like an Invoker's Rights (IR) unit.

ORACLE

- `DATABASE` triggers on PDBs
  - As of Oracle Database 12*c*: You can create a `DATABASE` event trigger on a pluggable database.
- `LIBRARY` can be defined as a `DIRECTORY` object and with `CREDENTIAL`.
  - Before Oracle Database 12*c*: You could define a `LIBRARY` object only by using an explicit path, even in versions of Oracle Database where the `DIRECTORY` object was intended as the single point of maintenance for file system paths. When running a subprogram stored in a library, the `extproc` agent always impersonated the owner of the Oracle Database installation.
  - As of Oracle Database 12*c*: You can define a `LIBRARY` object by using either an explicit path or a `DIRECTORY` object. Using a `DIRECTORY` object improves the security and portability of an application that uses external procedures. When you define a `LIBRARY` object, you can use the `CREDENTIAL` clause to specify the operating system user that the `extproc` agent impersonates when running a subprogram stored in the library.
- `BEQUEATH CURRENT_USER` Views
  - Before Oracle Database 12*c*: A view always behaved like a Definer's Rights (DR) unit.

Additional PL/SQL Enhancements

Before Oracle Database 12*c*	As of Oracle Database 12*c*
An IR unit always ran with the privileges of its invoker.	An IR unit can run with the privileges of its invoker only if its owner has either the `INHERIT PRIVILEGES` privilege on the invoker or the `INHERIT ANY PRIVILEGES` privilege.
A schema object was editionable if its type was editionable in the database and its owner was editions-enabled.	A schema object is editionable if its type is editionable in the schema that owns it and it has the `EDITIONABLE` property.
Diagnostic code could identify only the name of the current PL/SQL unit (with the predefined inquiry directive `$$PLSQL_UNIT`) and the number of the source line on which the predefined inquiry directive `$$PLSQL_LINE` appeared in that unit.	The additional predefined inquiry directives, `$$PLSQL_UNIT_OWNER` and `$$PLSQL_UNIT_TYPE`, allow diagnostic code to identify the owner and type of the current PL/SQL unit.

  - As of Oracle Database 12*c*: A view can be either `BEQUEATH DEFINER` (the default), which behaves like a DR unit, or `BEQUEATH CURRENT_USER`, which behaves somewhat like an Invoker's Rights (IR) unit.
- `INHERIT PRIVILEGES` and `INHERIT ANY PRIVILEGES` Privileges
  - Before Oracle Database 12*c*: An IR unit always ran with the privileges of its invoker. If its invoker had higher privileges than its owner, the IR unit might perform operations that are unintended by, or forbidden to, its owner.
  - As of Oracle Database 12*c*: An IR unit can run with the privileges of its invoker only if its owner has either the `INHERIT PRIVILEGES` privilege on the invoker or the `INHERIT ANY PRIVILEGES` privilege.
- Objects, not types, are editioned or non-editioned.
  - Before Oracle Database 12*c*: A schema object was editionable if its type was editionable in the database and its owner was editions-enabled. An editions-enabled user could not own a non-editioned object of an editionable type.
  - As of Oracle Database 12*c*: A schema object is editionable if its type is editionable in the schema that owns it and it has the `EDITIONABLE` property.

- An editions-enabled user can own a non-editioned object of a type that is editionable in the database if the type is non-editionable in the schema or the object has the NONEDITIONABLE property. Therefore, "CREATE [OR REPLACE] Statements" and "ALTER Statements" enable you to specify EDITIONABLE or NONEDITIONABLE.

- Predefined Inquiry Directives: $$PLSQL_UNIT_OWNER and $$PLSQL_UNIT_TYPE
  - Before Oracle Database 12*c*: Diagnostic code could identify only the name of the current PL/SQL unit (with the predefined inquiry directive $$PLSQL_UNIT) and the number of the source line on which the predefined inquiry directive $$PLSQL_LINE appeared in that unit.
  - As of Oracle Database 12*c*: The additional predefined inquiry directives, $$PLSQL_UNIT_OWNER and $$PLSQL_UNIT_TYPE, allow diagnostic code to identify the owner and the type of the current PL/SQL unit.

Summary

In this lesson, you should have learned how to:

- Create fine-grained access controls for packaged procedures
- Result cache with the Invoker's Right program unit
- Grant roles to PL/SQL packages and stand-alone stored subprograms
- Access PL/SQL functions that can run faster in SQL
- Use some additional PL/SQL features that are available in Oracle Database 12*c*

ORACLE

In this lesson, you should have learned how to create fine-grained access controls for packaged procedures, define the Invoker's Rights function that can be result cached, grant roles to PL/SQL packages and stand-alone stored subprograms, access the PL/SQL functions that can run faster in SQL, and also use some additional PL/SQL features that are available in Oracle Database 12*c*.

5

Data Warehousing Enhancements

Objectives

After completing this lesson, you should be able to:

- Maintain multi partition maintenance operations
- Analyze partial global indexes for partitioned tables
- Maintain an asynchronous global index
- Determine SQL for pattern matching
- Perform Synchronous Materialized View Refresh
- Identify ways to improve query performance against OLAP Cubes

ORACLE

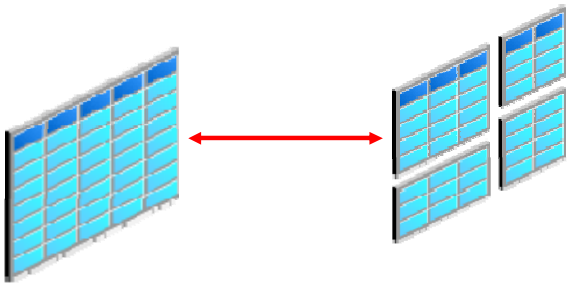
In this lesson, you learn about multi partition maintenance operations, analyzing partial global indexes for partitioned tables, maintaining an asynchronous global index, determining SQL for pattern matching, and identifying ways to improve query performance against OLAP Cubes.

Lesson Agenda

- Multi Partition Maintenance Operations
- Partial Global Indexes For Partitioned Tables
- Asynchronous Global Index Maintenance
- SQL for Pattern Matching
- Synchronous Materialized View Refresh
- Improving Query Performance Against OLAP Cubes

Multi Partition Maintenance Operations

- Split partition and merge partition operations allow you to roll out data into smaller partitions or to roll up data into a larger partition.
- Before Oracle Database 12*c*, Oracle allowed splitting into and merging of only two partitions by using the `ALTER TABLE` split and merge partition DDLs.



Split partition and merge partition operations allow you to roll out data into smaller partitions or to roll up data into a larger partition. Before Oracle Database 12*c*, Oracle allowed splitting into and merging of only two partitions by using the alter table split and merge partition DDLs. As a result, to split a partition into N partitions or to merge N partitions into one, (N-1) DDLs must be issued. For instance, you may want to roll up data for the previous year by merging all the monthly partitions. Because only two partitions can be merged at a time, 11 alter table merge partition DDLs must be issued. This is not only cumbersome, but also expensive due to multiple data reads and writes.

An alternative approach to merging multiple range partitions is to load data from N source partitions into a new non-partitioned table by using the `CREATE TABLE … AS SELECT` (CTAS) statement. Next, you drop the first (`N-1`) source partition and exchange the `N-th` source partition with the new table. The `N-th` source partition now contains data from the `N` partitions that were to be merged and may be renamed to the target partition name. This approach reduces data movement compared to issuing (`N-1`) merge partition DDLs. However, this is not as straight forward from the usability perspective. For example, when rolling up monthly partitions for the last year, the customer issues a CTAS, 11 drop partition DDLs and an exchange partition DDL to achieve the same result with lesser data movement.

Adding Multiple Partitions

- Add multiple new partitions with the `ADD PARTITION` clause of the `ALTER TABLE` statement.
- Local and global index operations are the same as when adding a single partition.
- Add multiple Range partitions that are listed in ascending order of their upper bound values to the high end of a Range-partitioned or Composite Range–partitioned table.
- Add multiple List partitions to a table by using new sets of partition values if the `DEFAULT` partition does not exist.

You can add multiple new partitions with the `ADD PARTITION` clause of the `ALTER TABLE` statement. When adding multiple partitions, local and global index operations are the same as when adding a single partition. Note that both `ADD PARTITION` and `ADD PARTITIONS` are synonymous.

You can add multiple Range partitions that are listed in ascending order of their upper bound values to the high end (after the last existing partition) of a Range-partitioned or Composite Range–partitioned table, provided the `MAXVALUE` partition is not defined. Similarly, you can add multiple List partitions to a table by using new sets of partition values if the `DEFAULT` partition does not exist.

Creating a Range-Partitioned Table

```
CREATE TABLE sales
( prod_id NUMBER(6)
, cust_id NUMBER
, time_id DATE
, channel_id CHAR(1)
, promo_id NUMBER(6)
, quantity_sold NUMBER(3)
, amount_sold NUMBER(10,2)
)
PARTITION BY RANGE (time_id)
( PARTITION sales_q1_2006 VALUES LESS THAN (TO_DATE('01-APR-2006','dd-
    MON-yyyy'))
TABLESPACE tsa
, PARTITION sales_q2_2006 VALUES LESS THAN (TO_DATE('01-JUL-2006','dd-
    MON-yyyy'))
TABLESPACE tsb
, PARTITION sales_q3_2006 VALUES LESS THAN (TO_DATE('01-OCT-2006','dd-
    MON-yyyy'))
TABLESPACE tsc
, PARTITION sales_q4_2006 VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-
    MON-yyyy'))
TABLESPACE tsd
);
```


The slide example creates a `sales` table with four partitions, one for each quarter of `2006`. Each partition is given a name: `sales_q1_2006`, `sales_q2_2006`, `sales_q3_2006`, and `sales_q4_2006`. The `time_id` column is the partitioning column, whereas its value represents the partitioning key of a specific row. The `VALUES LESS THAN` clause determines the partition bound: rows with partitioning key values that compare less than the ordered list of values specified by the clause are stored in the partition. Each partition is contained in a separate tablespace: `tsa`, `tsb`, `tsc`, and `tsd`.

For example, a row with `time_id=17-MAR-2006` would be stored in partition `sales_q1_2006`.

Adding Multiple Partitions

You can add multiple partitions by using a single SQL statement by specifying the individual partitions as follows:

```
ALTER TABLE sales ADD
PARTITION sales_q1_2007 VALUES LESS THAN
(TO_DATE('01-APR-2007','dd-MON-yyyy')),
PARTITION sales_q2_2007 VALUES LESS THAN
(TO_DATE('01-JUL-2007','dd-MON-yyyy')),
PARTITION sales_q3_2007 VALUES LESS THAN
(TO_DATE('01-OCT-2007','dd-MON-yyyy')),
PARTITION sales_q4_2007 VALUES LESS THAN
(TO_DATE('01-JAN-2008','dd-MON-yyyy'));
```


You can add multiple partitions by using a single statement by specifying the individual partitions. For example, in the slide example, you add multiple partitions to the Range-partitioned `sales` table that was created earlier named `sales_q1_2007`, `sales_q2_2007`, `sales_q3_2007`, and `sales_q4_2007`.

Merging Multiple Range Partitions

- You can merge the contents of two or more partitions or subpartitions into one new partition or subpartition.

```
ALTER TABLE sales_orders
MERGE PARTITIONS p01, p02, p03, p04 INTO PARTITION p0;
```

- When merging Range partitions:
  - The partitions must be adjacent
  - The partitions must be specified in the ascending order of their partition bound values
  - The new partition inherits the partition upper bound of the highest of the original partitions

ORACLE

You can merge the contents of two or more partitions or subpartitions into one new partition or subpartition. You can then drop the original partitions or subpartitions with the `MERGE PARTITIONS` and `MERGE SUBPARTITIONS` clauses of the `ALTER TABLE` SQL statement as shown in the slide. Before running the code example in the slide, it is assumed that a table named `sales_orders` with four partitions: `p01`, `p02`, `p03`, and `p04`, is already created.

When merging multiple range partitions, the partitions must be adjacent and specified in the ascending order of their partition bound values. The new partition inherits the partition upper bound of the highest of the original partitions.

Merging Multiple Range Partitions: Examples

- Merge four partitions of the Range-partitioned `sales` table.

```
ALTER TABLE sales ADD
PARTITION sales_q1_2009 VALUES LESS THAN
(TO_DATE('01-APR-2009','dd-MON-yyyy')),
PARTITION sales_q2_2009 VALUES LESS THAN
(TO_DATE('01-JUL-2009','dd-MON-yyyy')),
PARTITION sales_q3_2009 VALUES LESS THAN
(TO_DATE('01-OCT-2009','dd-MON-yyyy')),
PARTITION sales_q4_2009 VALUES LESS THAN
(TO_DATE('01-JAN-2010','dd-MON-yyyy'))
/
ALTER TABLE sales
MERGE PARTITIONS sales_q1_2009, sales_q2_2009, sales_q3_2009,
    sales_q4_2009
INTO PARTITION sales_2009;
```

- If you are using Range partitioning, you can rewrite the previous statement as follows:

```
ALTER TABLE sales
MERGE PARTITIONS sales_q1_2009 TO sales_q4_2009
INTO PARTITION sales_2009;
```

ORACLE

In the first example in the slide, you add the four partitions for the year 2009 to the `sales` Range-partitioned table, and then merge these partitions. These four partitions that correspond to the four quarters of the year 2009 are merged into a single partition containing the entire sales data for the year.

If you are using Range partitioning, you can use the second slide example to rewrite the first statement by using the `TO` keyword.

Merging List, System, and Range Partitions

- There is no enforced partition order when merging multiple partitions for List partitioning and System partitioning.
- When merging multiple List partitions, the resulting partition value list is the union of the set of partition value lists of all the partitions to be merged.
- A `DEFAULT` List partition that is merged with other List partitions results in a `DEFAULT` partition.
- You can specify the lowest and the highest partitions to be merged when merging multiple Range partitions with the `TO` keyword:

```
ALTER TABLE sales_orders
MERGE PARTITIONS p01 TO p04 INTO PARTITION p0;
```


There is no enforced partition order when merging multiple partitions for List partitioning and System partitioning.

When merging multiple List partitions, the resulting partition value list is the union of the set of partition value lists of all the partitions to be merged. A `DEFAULT` List partition that is merged with other list partitions results in a `DEFAULT` partition.

You can specify the lowest and the highest partitions to be merged when merging multiple Range partitions with the `TO` syntax. All partitions between the specified partitions, including those specified, are merged into the target partition. You cannot use this syntax for List and System partitions.

For example, the slide example merges partitions `p01` through `p04` into partition `p0`.

Dropping Multiple Partitions

- Remove multiple partitions or subpartitions from a Range or List partitioned table with the following clauses of the `ALTER TABLE` statement:
  - `DROP PARTITION`
  - `DROP SUBPARTITION`

```
-- The 2008 partitions must be first added.
ALTER TABLE sales DROP PARTITIONS
sales_q1_2008, sales_q2_2008, sales_q3_2008,
    sales_q4_2008;
```

```
SQL> ALTER TABLE sales DROP PARTITIONS
sales_q1_2008, sales_q2_2008, sales_q3_2008, sales_q4_2008;

Table altered.

SQL>
```

- Local and global index operations are the same as when dropping a single partition.

You can remove multiple partitions or subpartitions from a Range or List partitioned table with the `DROP PARTITION` and `DROP SUBPARTITION` clauses of the `ALTER TABLE` statement. For example, the statement in the slide drops multiple partitions from the Range-partitioned table, `sales`. Note that you cannot drop all the partitions of a table.

When you drop multiple partitions, local and global index operations are the same as when you drop a single partition.

Splitting into Multiple Partitions

- You can redistribute the contents of one partition into multiple partitions with the `SPLIT PARTITION` clause of the `ALTER TABLE` statement.
- When splitting multiple partitions, the segment that is associated with the current partition is discarded.
- Each new partition obtains a new segment and inherits all unspecified physical attributes from the current source partition.
- You can use the extended split syntax to specify a list of new partition descriptions, similar to the create partitioned table SQL statements, instead of using the `AT` or `VALUES` clauses.

ORACLE

You can redistribute the contents of one partition into multiple partitions with the `SPLIT PARTITION` clause of the `ALTER TABLE` statement. When splitting multiple partitions, the segment that is associated with the current partition is discarded. Each new partition obtains a new segment and inherits all unspecified physical attributes from the current source partition.

You can use the extended split syntax to specify a list of new partition descriptions, similar to the create partitioned table SQL statements, instead of using the `AT` or `VALUES` clauses. Additionally, the Range or List values clause for the last new partition description is derived based on the high bound of the source partition and the bound values specified for the first (N-1) new partitions resulting from the split.

Splitting into Multiple Partitions: Examples

```
ALTER TABLE sales_orders SPLIT PARTITION p0 INTO
(PARTITION p01 VALUES LESS THAN (25),
 PARTITION p02 VALUES LESS THAN (50),
 PARTITION p03 VALUES LESS THAN (75),
 PARTITION p04);
```

```
-- Before running this example, you need to re-run the
-- script that merges partitions p01 to p04 into p0.

ALTER TABLE sales_orders SPLIT PARTITION p0 INTO
(PARTITION p01 VALUES LESS THAN (25),
PARTITION p02);
```

```
SELECT PARTITION_NAME
FROM USER_TAB_PARTITIONS
WHERE TABLE_NAME = 'SALES_ORDERS';SQL>   2    3

PARTITION_NAME
------------------------------------------------------------
P01
P02

SQL>
```


The first slide example demonstrates splitting a partition `p0` into multiple partitions, namely `p01`, `p02`, `p03`, and `p04`.

In the second slide example, partition `p02` has the high bound of the original partition `p0`.

In the third code example, the query shows the two new partitions.

Splitting into Multiple Partitions: Rules

- To split a Range partition into N partitions:
  - (N-1) values of the partitioning key column must be specified within the range of the partition at which to split the partition
- To split a List partition into N partitions:
  - (N-1) lists of literal values must be specified
  - Each list defines the first (N-1) partitions into which rows with corresponding partitioning key values are inserted
- When splitting a DEFAULT List partition or a MAXVALUE Range partition into multiple partitions:
  - The first (N-1) new partitions are created by using the literal value lists or the high bound values that are specified
  - The Nth new partition resulting from the split has the DEFAULT value or MAXVALUE

ORACLE

To split a Range partition into N partitions, (N-1) values of the partitioning key column must be specified within the range of the partition at which to split the partition. The new non-inclusive upper bound values specified must be in ascending order. The high bound of Nth new partition is assigned the value of the high bound of the partition that is being split. The names and physical attributes of the N new partitions resulting from the split can be optionally specified.

To split a List partition into N partitions, (N-1) lists of literal values must be specified, each of which defines the first (N-1) partitions into which rows with corresponding partitioning key values are inserted. The remaining rows of the original partition are inserted into the Nth new partition whose value list contains the remaining literal values from the original partition. No two value lists can contain the same partition value. The (N-1) value lists that are specified cannot contain all the partition values of the current partition because the Nth new partition would be empty. Also, the new (N-1) value lists cannot contain any partition values that do not exist for the current partition.

When splitting a DEFAULT List partition or a MAXVALUE Range partition into multiple partitions, the first (N-1) new partitions are created by using the literal value lists or the high bound values that are specified, whereas the Nth new partition resulting from the split has the DEFAULT value or MAXVALUE. The SPLIT_TABLE_SUBPARTITION clause is extended similarly to allow the split of a Range or List subpartition into N new subpartitions.

Splitting into Multiple Partitions: Examples

```
ALTER TABLE sales2 SPLIT
PARTITION sales2_q4_2007 INTO
  (PARTITION sales2_q1_2008 VALUES LESS THAN
  (TO_DATE('01-APR-2008','dd-MON-yyyy')),
  PARTITION sales2_q2_2008 VALUES LESS THAN
  (TO_DATE('01-JUL-2008','dd-MON-yyyy')),
  PARTITION sales2_q3_2008 VALUES LESS THAN
  (TO_DATE('01-OCT-2008','dd-MON-yyyy')),
  PARTITION sales2_q4_2008 VALUES LESS THAN
  (TO_DATE('01-JAN-2009','dd-MON-yyyy')),
  PARTITION rest);
```

```
CREATE TABLE list_customers
(cust_id number, cust_name varchar2(100), region varchar2(100))
PARTITION BY LIST (region)
  (PARTITION europe VALUES
    ('GERMANY','FRANCE','ITALY','GREECE','SPAIN'),
   PARTITION rest_of_world VALUES (DEFAULT))
/
ALTER TABLE list_customers SPLIT PARTITION Europe INTO
(PARTITION western_europe VALUES ('GERMANY', 'FRANCE'),
 PARTITION southern_europe VALUES ('ITALY'),
 PARTITION rest_europe);
```

ORACLE

The first slide example splits the `sales_Q4_2007` partition of the partitioned by Range table `sales2` into four partitions that correspond to the quarters of the next year and a partition contains the rest. In this example, the partition `rest` implicitly becomes the high bound of the split partition.

In the second slide example, the sample partitioned table, `list_customers`, is first created. The table is then partitioned by List, and splits the partition, `Europe`, into three partitions: `western_europe`, `southern_europe`, and `rest_europe`.

Truncating Multiple Partitions

- Truncate multiple partitions from a Range or List partitioned table with the `ALTER TABLE ... TRUNCATE PARTITION` statement.
- The corresponding partitions of local indexes are truncated in the operation.
- Global indexes must be rebuilt unless the `UPDATE INDEXES` clause is specified.
- Example:

```
ALTER TABLE sales TRUNCATE PARTITIONS
sales_q1_2008, sales_q2_2008,sales_q3_2008,sales_q4_2008;
```

ORACLE

Use the `ALTER TABLE ... TRUNCATE PARTITION` statement to remove all rows from a table partition. Truncating a partition is similar to dropping a partition, except that the partition is emptied of its data, but not physically dropped.

The corresponding partitions of local indexes are truncated in the operation.

Global indexes must be rebuilt unless the `UPDATE INDEXES` clause is specified.

The slide example truncates four partitions in the Range-partitioned `sales` table.

Practice 5-1 Overview: Multi Partition Maintenance Operations

This practice covers the following topics:

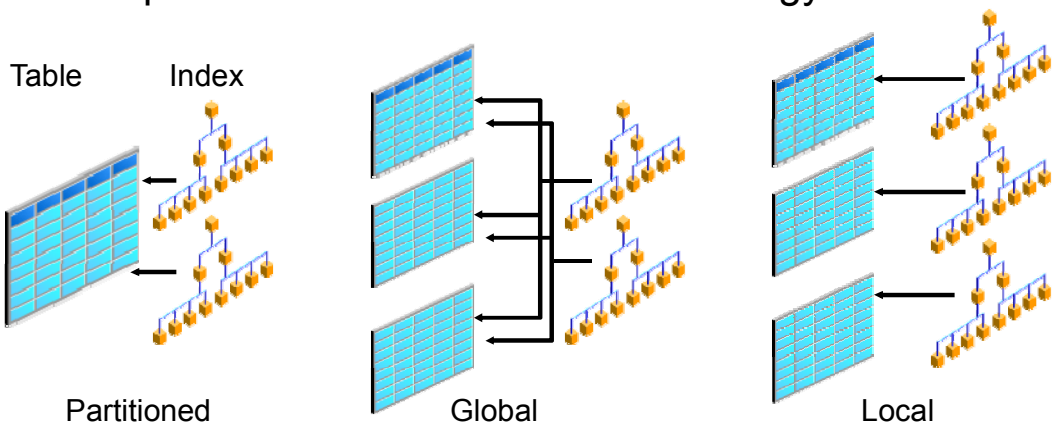
- Creating a range partitioned table and adding multiple partitions
- Merging multiple range partitions
- Splitting a partition into multiple partitions

Lesson Agenda

- Maintaining Multiple Partitions
- Partial Global Indexes For Partitioned Tables
- Asynchronous Global Index Maintenance
- SQL for Pattern Matching
- Synchronous Materialized View Refresh
- Improving Query Performance Against OLAP Cubes

Partitioned Indexes: Review

- Indexes can be partitioned similar to tables.
- A partitioned index can exist on a nonpartitioned table.
- A global partitioned index uses a different partition strategy than that of a table.
- A local partitioned index follows the strategy of the table.



ORACLE

The same management benefits and performance improvement that can be achieved by partitioning tables is achieved by partitioning indexes. When an index is partitioned, each partition is a complete and separate index. There is no master index to decide which index to use because this piece of information is inherent in the partition information. The syntax that is used to specify the partitioning of an index is very similar to that used to partition a table.

Global partitioned indexes can be defined on a nonpartitioned table. Local indexes can be defined only on partitioned tables. A local index has the same partitioning key as the table partitioning key. You can use only range and hash partitioning for global indexes.

Local indexes are automatically maintained; that is, changes made on the partitions on the table are automatically repeated on the local indexes; however, depending on the DDL, you may have to use the update indexes clause to maintain an associated local index during partition maintenance operations.

A local index segment is always built for the equivalent data (table) segment. Therefore, for composite partitioned tables, you have only local subpartitioned index segments and nothing on the logical partition top-level.

Partial Indexes for Partitioned Tables

- Local and global indexes can be created on a subset of the partitions of a table, enabling more flexibility in index creation.
- This feature supports global indexes that include or index a certain subset of table partitions or subpartitions and exclude the others.
- This feature is supported by using a *default* table indexing property.
- When a table is created or altered, a default indexing property can be specified for the table or its partitions.

Indexes typically occupy considerable space since each table in the database could have many indexes defined on it.

Global indexes index all table partitions and the index structure has no correlation with the table partitioning. Before Oracle Database 12*c*, Oracle did not provide the ability to index a subset of partitions and to exclude the others. There is no operation that a DBA may use before Oracle Database 12*c* to exclude non-active partitions from a global index.

In Oracle Database 12*c*, Partial Global Indexes save space and improve performance during loads and queries.

Local and global indexes can be created on a subset of the partitions of a table, enabling more flexibility in index creation.

This feature supports global indexes that include or index a certain subset of table partitions or subpartitions and exclude the others. This operation is supported by using a default table indexing property. When a table is created or altered, a default indexing property can be specified for the table or its partitions.

Partial Index Creation on a Table

When an index is created as PARTIAL on a table:

- Local indexes:
  - An index partition is created as usable if indexing is enabled for the table partition, and as unusable otherwise
  - You can override this behavior by using the USABLE or UNUSABLE properties at the index or index partition level
- Global indexes:
  - Include only those partitions for which indexing is enabled and exclude the others
  - FULL is the default if neither FULL nor PARTIAL is specified

- **Local indexes:** An index partition is created as usable if indexing is enabled for the table partition and as unusable otherwise. You can override this behavior by specifying the USABLE or UNUSABLE property at the index or index partition level.
- **Global indexes:** Include only those partitions for which indexing is enabled and exclude the others. FULL is the default if neither FULL nor PARTIAL is specified; that is, Oracle indexes all partitions (similar to the behavior of Oracle Database releases before Oracle Database 12*c*).

This feature is not supported for unique indexes, or for indexes that are used to enforce unique constraints.

- In a local index, all keys in a particular index partition refer only to rows stored in a single underlying table partition. A local index is created by specifying the LOCAL attribute.
- In a global partitioned index, the keys in a particular index partition may refer to rows stored in multiple underlying table partitions or subpartitions. A global index can be range or hash partitioned, although it can be defined on any type of partitioned table. A global index is created by specifying the GLOBAL attribute.

Full Index Creation on a Table

```
CREATE TABLE orders2 (
order_id NUMBER(12),
order_date DATE CONSTRAINT order_date_nn_2 NOT NULL,
order_mode VARCHAR2(8),
customer_id NUMBER(6) CONSTRAINT order_customer_id_nn_2 NOT
  NULL,
order_status NUMBER(2),
order_total NUMBER(8,2),
sales_rep_id NUMBER(6),
promotion_id NUMBER(6),
CONSTRAINT order_mode_lov_2 CHECK (order_mode in
  ('direct','online')),
CONSTRAINT order_total_min_2 CHECK (order_total >= 0)
)
INDEXING OFF
PARTITION BY RANGE (ORDER_DATE)
(PARTITION ord_p1 VALUES LESS THAN (TO_DATE('01-MAR-2013','DD-
  MON-YYYY')),
PARTITION ord_p2 VALUES LESS THAN (TO_DATE('01-JUL-2013','DD-
  MON-YYYY')),
. . .
```


An index can be created FULL to decouple it from the table properties.

For example, the slide example creates a table ORDERS2 where all partitions are to be excluded from all partial global indexes, and where all partitions are to have the corresponding unusable local index partitions in the PARTIAL local indexes. This applies to automatically created partitions as well. INDEXING ON is the default.

Note: The local index partitions (in PARTIAL local indexes) that correspond to table partitions with INDEXING OFF are created as unusable.

In Oracle Database 12*c*, the CREATE TABLE syntax is extended to support specification of the default indexing attributes clause. indexing_clause is as follows:

```
[INDEXING { ON | OFF }]
```

You can use ALTER TABLE . . MODIFY DEFAULT ATTRIBUTES clause to change the default property of INDEXING ON/OFF. This changes the default for future partitions (and subpartitions).

Specifying the INDEXING Clause at the Partition and Subpartition Levels

```
CREATE TABLE orders (
-- column_clauses as described in previous example
order_id NUMBER(12),
order_date DATE CONSTRAINT order_date_nn NOT NULL,
. . .
CONSTRAINT order_total_min CHECK (order_total >= 0))
)
INDEXING OFF
PARTITION BY RANGE (ORDER_DATE)
(PARTITION ord_p1 VALUES LESS THAN (TO_DATE('01-MAR-2013','DD-MON-
    YYYY')) INDEXING ON,
PARTITION ord_p2 VALUES LESS THAN (TO_DATE('01-JUL-2013','DD-MON-YYYY'))
    INDEXING OFF,
PARTITION ord_p3 VALUES LESS THAN (TO_DATE('01-OCT-2013','DD-MON-YYYY'))
    INDEXING ON,
PARTITION ord_p4 VALUES LESS THAN (TO_DATE('01-JAN-2014','DD-MON-
    YYYY')),
PARTITION ord_p5 VALUES LESS THAN (TO_DATE('01-MAR-2014','DD-MON-
YYYY')));
```


The INDEXING clause may also be specified at the partition and subpartition levels.

The slide example creates a table, ORDERS, with the following items:

- The partitions ORD_P1 and ORD_P3 are included in all partial global indexes.
- The local index partitions (for indexes that are created as PARTIAL) that correspond to the preceding two table partitions are created as usable by default.
- Other partitions are excluded from all partial global indexes and are created as unusable in the local indexes (for indexes created PARTIAL).

Note that the partitions, ORD_P4 and ORD_P5, are not included in all partial global indexes because the default INDEXING property for the table is OFF, and INDEXING ON/OFF has not been specified at the partition level for these partitions; therefore, ORD_P4 and ORD_P5 partitions inherit the table properties.

Creating a Local or Global Index

An index, local or global, may be created to follow the table indexing properties by specifying the `INDEXING PARTIAL` clause.

```
CREATE INDEX ORDERS_GIDX_ORDERTOTAL ON ORDERS
(ORDER_TOTAL)
GLOBAL INDEXING PARTIAL;
```


The `ORDERS_GIDX_ORDERTOTAL` index in the slide example is created to index only those partitions that have `INDEXING ON` and to exclude the remaining partitions. Local indexes may also be created `PARTIAL` as discussed earlier.

Affected Data Dictionary Views: Overview

Data Dictionary View	Description
DBA|ALL|USER_PART_TABLES	Display object-level partitioning information for the partitioned tables.
DBA|ALL|USER_TAB_PARTITIONS	Display partition-level partitioning information, partition storage parameters, and partition statistics.
DBA|ALL|USER_TAB_SUBPARTITIONS	Display subpartition-level partitioning information, subpartition storage parameters, and subpartition statistics.
DBA|ALL|USER_INDEXES	Display indexes.
DBA|ALL|USER_IND_PARTITIONS	Display for each index partition, the partition-level partitioning information, the storage parameters for the partition, and various partition statistics.

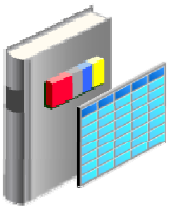
ORACLE

The following is a quick review of some of the affected views.

- The ALL_PART_TABLES view displays the object-level partitioning information for the partitioned tables that are accessible to the current user. The USER_PART_TABLES view displays the object-level partitioning information for the partitioned tables owned by the current user (No OWNER column). The DBA_PART_TABLES view displays the object-level partitioning information for all partitioned tables in the database.
- The USER_TAB_PARTITIONS view describes the partition-level partitioning information, partition storage parameters, and partition statistics generated by the DBMS_STATS package for all partitions owned by the current user.
- The USER_TAB_SUBPARTITIONS view describes, for each table subpartition owned by the current user, the subpartition name, the name of the table and partition to which it belongs, and its storage attributes.
- The USER_INDEXES view describes the indexes owned by the current user. To gather statistics for this view, you use the DBMS_STATS package.
- The USER_IND_PARTITIONS view displays, for each index partition owned by the current user, the partition-level partitioning information, the storage parameters for the partition, and various partition statistics generated by the DBMS_STATS package.

Data Dictionary View Changes

- Table Indexing Property:
  - The column DEF_INDEXING is added to the following view:
    - DBA|ALL|USER_PART_TABLES
  - The column INDEXING is added to the following views:
    - DBA|ALL|USER_TAB_PARTITIONS
    - DBA|ALL|USER_TAB_SUBPARTITIONS
  - This INDEXING column has one of two values: ON or OFF.
- Partial Global Indexes as an Index Level Property:
  - A new column INDEXING is added to the USER_INDEXES view.
  - This column can be set to the following values:
    - FULL
    - PARTIAL



The updates to the data dictionary views that were listed in the previous slide are as follows:

Table Indexing Property

- The column DEF_INDEXING is added to the DBA|ALL|USER_PART_TABLES views.
- The column INDEXING is added to the DBA|ALL|USER_TAB_PARTITIONS and DBA|ALL|USER_TAB_SUBPARTITIONS views.
- The INDEXING column has one of two values, ON or OFF, specifying whether indexing is enabled or disabled.

Partial Global Indexes as an Index Level Property

- A new column, INDEXING, is added to the USER_INDEXES view.
- This column's value is FULL if the index is full or PARTIAL if the index is partial.

Asynchronous Global Index Maintenance

- The partition maintenance operations `DROP PARTITION` and `TRUNCATE PARTITION` that support the updating of global indexes are optimized by:
  - Restricting index maintenance to only metadata
- Maintenance operations on indexes can be performed with the automatic scheduler job to clean up all global indexes:
  - `SYS.PMO_DEFERRED_GIDX_MAINT_JOB`

ORACLE

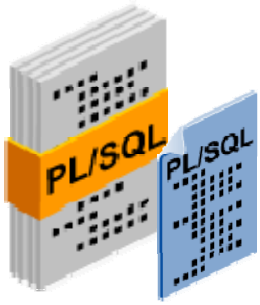
The partition maintenance operations `DROP PARTITION` and `TRUNCATE PARTITION` that support the updating of global indexes are optimized by restricting index maintenance to only metadata. This functionality enables maintenance of a list of data object numbers in metadata, where index entries corresponding to the drop and truncated objects that are invalid are ignored.

Maintenance operations on indexes can be performed with the automatic scheduler job `SYS.PMO_DEFERRED_GIDX_MAINT_JOB` to clean up all global indexes. This job is scheduled to run at `2:00 A.M.` on a daily basis by default. You can run this job at any time by using `DBMS_SCHEDULER.RUN_JOB` if you want to proactively clean up the indexes.

You can also modify the job to run with a different schedule based on your specific requirements. However, Oracle recommends that you do not drop the job.

DBMS_PART Package

- The DBMS_PART package enables you to maintain and manage operations on partitioned objects.
- The new package contains the CLEANUP_GIDX procedure:
  - This procedure identifies and cleans up the global indexes to ensure efficiency in terms of storage and performance.



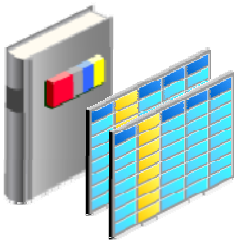
DBMS_PART.CLEANUP_GIDX
package procedure

The DBMS_PART package provides an interface for maintenance and management operations on partitioned objects. As a consequence of prior partition maintenance operations with asynchronous global index maintenance, global indexes can contain entries pointing to data segments that no longer exist. These stale index rows will not cause any correctness issues during any operation on the table or index, whether these are queries, DMLs, DDLs or analyze statements.

The DBMS_PART.CLEANUP_GIDX package procedure identifies and cleans up these global indexes to ensure efficiency in terms of storage and performance.

Global Index Maintenance Optimization During Partition Maintenance

- Partial Global Index Optimization
  - A new column, ORPHANED_ENTRIES, is added to the following data dictionary views:
    - DBA|ALL|USER_INDEXES
    - DBA|ALL|USER_IND_PARTITIONS
  - This column specifies whether or not a global index (partition) contains stale entries due to deferred index maintenance during one of the following operations:
    - DROP/TRUNCATE PARTITION
    - MODIFY PARTITION INDEXING OFF
  - The column will have one of three values:
    - YES, NO, or NA

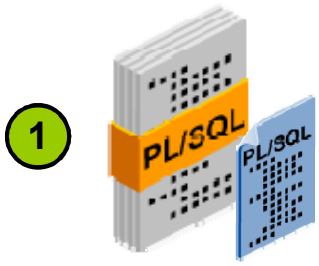


Partial Global Index Optimization

- The column, ORPHANED_ENTRIES, is added to the dictionary views USER_INDEXES and USER_IND_PARTITIONS. This column specifies whether or not a global index (partition) contains stale entries due to deferred index maintenance during DROP/TRUNCATE PARTITION or MODIFY PARTITION operations.
- The column can have one of three values:
  - YES: The index (partition) contains orphaned entries.
  - NO: The index (partition) does not contain any orphaned entries.
  - N/A: The property is not applicable. This is the case for local indexes or indexes on non-partitioned tables.

Forcing an Index Cleanup: Additional Methods

You can also force cleanup of an index that needs maintenance by using one of the following options:



DBMS_PART.CLEANUP_GIDX
package procedure

ALTER INDEX REBUILD [PARTITION]

ALTER INDEX COALESCE [PARTITION] CLEANUP

You can also force cleanup of an index that needs maintenance by using one of the following options:

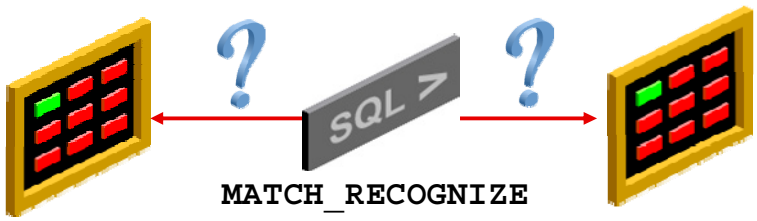
- `DBMS_PART.CLEANUP_GIDX`: This PL/SQL package procedure gathers the list of global indexes in the system that may require cleanup, and runs the operations that are necessary to restore the indexes to a clean state.
- `ALTER INDEX REBUILD [PARTITION]`: This SQL statement rebuilds the entire index or index partition as was done before Oracle Database 12.1 releases; the resulting index (partition) does not contain any stale entries.
- `ALTER INDEX COALESCE [PARTITION] CLEANUP`: This SQL statement cleans up any orphaned entries in index blocks.

Lesson Agenda

- Maintaining Multiple Partitions
- Partial Global Indexes For Partitioned Tables
- Asynchronous Global Index Maintenance
- SQL for Pattern Matching
- Synchronous Materialized View Refresh
- Improving Query Performance Against OLAP Cubes

Pattern Matching: Overview

- Logically partition and order the data that is used in the `MATCH_RECOGNIZE` clause with its `PARTITION BY` and `ORDER BY` clauses.
- Define patterns of rows to seek by using the `PATTERN` clause of the `MATCH_RECOGNIZE` clause.
- Specify the logical conditions required to map a row to a row pattern variable in the `DEFINE` clause.
- Define measures, which are expressions that are usable in the `MEASURES` clause of the SQL query.



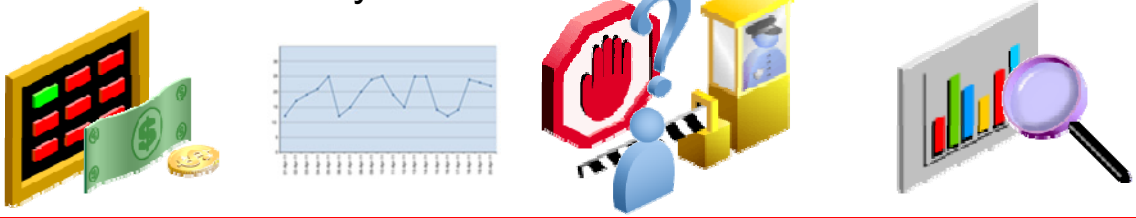
Row pattern matching in native SQL improves application and development productivity and query efficiency for row-sequence analysis. The syntax incorporates regular expressions and full conditional logic, thus enabling precise and flexible pattern definition. Whatever the domain (for example, financial market prices, internet clicks, or security sensor output), applications analyzing row sequences can benefit from `MATCH_RECOGNIZE`.

Recognizing patterns in a sequence of rows has been a capability that was widely desired, but not easily possible with SQL until now. There were many workarounds, but these were difficult to write, hard to understand, and inefficient to execute. With Oracle Database 12*c*, Release 1, you can use the `MATCH_RECOGNIZE` clause to perform pattern matching in SQL to do the following:

- Logically partition and order the data that is used in the `MATCH_RECOGNIZE` clause with its `PARTITION BY` and `ORDER BY` clauses.
- Define patterns of rows to seek by using the `PATTERN` clause of the `MATCH_RECOGNIZE` clause. These patterns use the regular expressions syntax, which is a powerful and expressive feature that can be applied to the pattern variables that you define.
- Specify the logical conditions required to map a row to a row pattern variable in the `DEFINE` clause.
- Define measures, which are expressions that are usable in the `MEASURES` clause of the SQL query.

Why Use Pattern Matching?

- To recognize patterns across multiple rows in applications where all kinds of business processes are driven by sequences of events
- Examples:
  - Security application where unusual behavior must be detected
  - Financial applications seeking:
    - Pricing patterns
    - Trading volume and other behavior
- Fraud detection applications
- Sensor data analysis

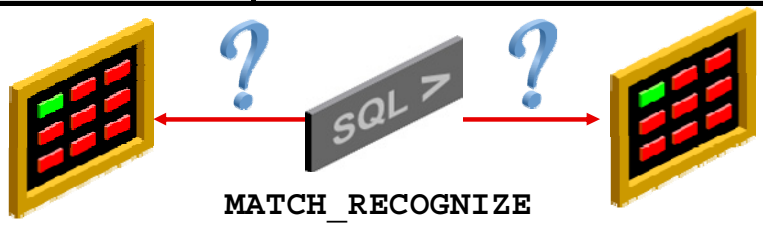


The ability to recognize patterns found across multiple rows is important for many types of applications. Examples include all kinds of business processes that are driven by sequences of events, such as financial applications seeking patterns of pricing, trading volume, and other behavior; security applications where unusual behavior must be detected; fraud detection applications; and sensor data analysis. One term that describes this general area is complex event processing, and pattern matching is a powerful aid to this activity.

For additional information about pattern matching, see the *Oracle Database Data Warehousing Guide 12c Release 1 (12.1)*.

Tasks and Keywords in Pattern Matching

Task/Keyword	Description
PARTITION BY	Logically divides rows into groups at a high level
ORDER BY	Logically orders the rows in a partition
ONE ROW PER MATCH	Returns one summary row of output for each match
ALL ROWS PER MATCH	Returns one detail row of output for each row of each match
MEASURES	Defines row pattern measure columns



Some of the tasks and keywords used in pattern matching are:

- **PARTITION BY:** Typically, you want to divide your input data into logical groups for analysis. In the stock examples, you want to divide pattern matching so that it applies to just one stock at a time. You do that with PARTITION BY, which specifies that the rows of the input table are to be partitioned by one or more columns. Matches are found within partitions and do not cross partition boundaries. If there is no PARTITION BY, all rows of the row pattern input table constitute a single row pattern partition.
- **ORDER BY:** This is used to specify the order of rows within a row pattern partition. If the order of two rows in a row pattern partition is not determined by ORDER BY, the result of MATCH_RECOGNIZE is non-deterministic: it may not give consistent results each time the query is run.
- **[ONE ROW | ALL ROWS] PER MATCH]:** Sometimes, you may need summary data about the matches, whereas at other times, you may need details. You can do this as follows:
  - ONE ROW PER MATCH gives you one row of output for each match.
  - ALL ROWS PER MATCH gives you one row of output for each row of each match. This is the default.
- **MEASURES:** The MEASURES clause defines a list of columns for the pattern output table. Each pattern measure column is defined with a column name whose value is specified by a corresponding pattern measure expression.

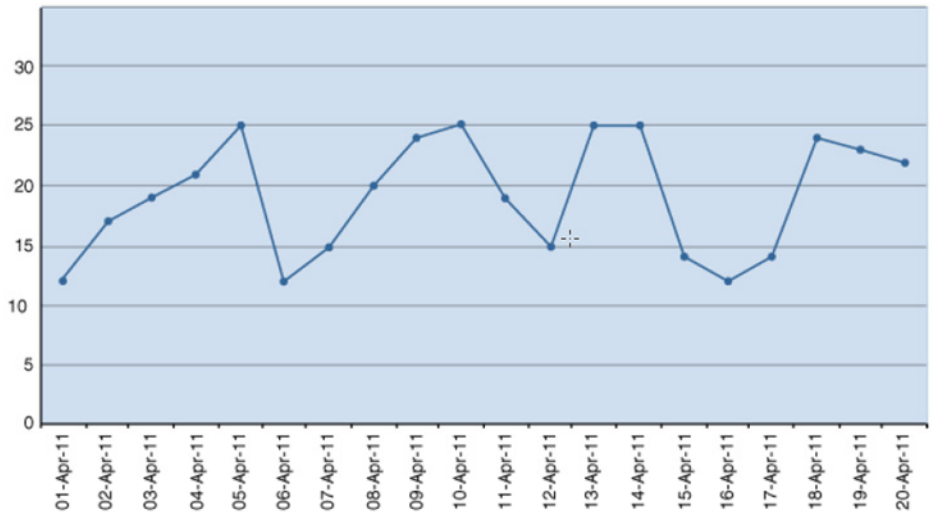
Tasks and Keywords in Pattern Matching

Task/Keyword	Description
PATTERN	Defines which pattern variables must be matched, the sequence in which they must be matched, and the quantity of rows which must be matched
DEFINE	Specifies the conditions that define a pattern variable
AFTER MATCH	Restarts the matching process after a match is found
MATCH_NUMBER	Finds which pattern variable applies to which rows
CLASSIFIER	Identifies which component of a pattern applies to a specific row

- **PATTERN:** This specifies the pattern to be recognized in the ordered sequence of rows in a partition. Each variable name in a pattern corresponds to a Boolean condition, which is specified later by using the DEFINE component of the syntax. The PATTERN clause is used to specify a regular expression. The regular expression is enclosed in parentheses.
- **DEFINE:** Because the PATTERN clause depends on pattern variables, you must have a clause to define these variables. They are specified in the DEFINE clause (required), and are used to specify the conditions that define a pattern variable.
- **AFTER MATCH:** After the query finds a match, it is vital that it begins looking for the next match at exactly the right point. Do you want to find matches where the end of the earlier match overlaps the start of the next match? Or do you want some other variation? Pattern matching provides great flexibility in specifying the restart point.
- **MATCH_NUMBER:** You might have a large number of matches for your pattern inside a given row partition. How do you tell all these matches apart? This is done with the MATCH_NUMBER function. Matches within a row pattern partition are numbered sequentially, starting with 1 in the order in which they are found. Note that match numbering starts over again at 1 in each row pattern partition, because there is no inherent ordering between the row pattern partitions.

- **CLASSIFIER:** Along with knowing which `MATCH_NUMBER` you are seeing, you may want to know which component of a pattern applies to a specific row. This is done by using the `CLASSIFIER` function. The classifier of a row is the pattern variable that the row is mapped to by a row pattern match. The classifier of a row that is not mapped by a row pattern match is null. The `CLASSIFIER` function returns a character string whose value is the classifier of a row.

Pattern Match Example: Stock Chart



ORACLE

Pattern matching enables you to identify price patterns, such as **V-shapes** and **W-shapes**, as illustrated in the slide diagram, along with performing many types of calculations. For example, your calculations might include the count of observations or the average value on a downward or an upward slope.

The diagram in the slide example uses the stock price, which you can load into your database with the `CREATE` and `INSERT` statements that are shown in the next slide. The query finds all cases where stock prices dipped to a bottom price, and then rose. This is generally called a **V-shape**.

Pattern Match Example: Simple V Shape with One Row Output Per Match

```
CREATE TABLE ticker (SYMBOL VARCHAR2(10), tstamp DATE, price NUMBER);
```

```
INSERT INTO Ticker VALUES('ACME', '01-Apr-11', 12);
INSERT INTO Ticker VALUES('ACME', '02-Apr-11', 17);
INSERT INTO Ticker VALUES('ACME', '03-Apr-11', 19);
INSERT INTO Ticker VALUES('ACME', '04-Apr-11', 21);
INSERT INTO Ticker VALUES('ACME', '05-Apr-11', 25);
INSERT INTO Ticker VALUES('ACME', '06-Apr-11', 12);
INSERT INTO Ticker VALUES('ACME', '07-Apr-11', 15);
INSERT INTO Ticker VALUES('ACME', '08-Apr-11', 20);
INSERT INTO Ticker VALUES('ACME', '09-Apr-11', 24);
INSERT INTO Ticker VALUES('ACME', '10-Apr-11', 25);
INSERT INTO Ticker VALUES('ACME', '11-Apr-11', 19);
INSERT INTO Ticker VALUES('ACME', '12-Apr-11', 15);
INSERT INTO Ticker VALUES('ACME', '13-Apr-11', 25);
INSERT INTO Ticker VALUES('ACME', '14-Apr-11', 25);
INSERT INTO Ticker VALUES('ACME', '15-Apr-11', 14);
INSERT INTO Ticker VALUES('ACME', '16-Apr-11', 12);
INSERT INTO Ticker VALUES('ACME', '17-Apr-11', 14);
INSERT INTO Ticker VALUES('ACME', '18-Apr-11', 24);
INSERT INTO Ticker VALUES('ACME', '19-Apr-11', 23);
INSERT INTO Ticker VALUES('ACME', '20-Apr-11', 22);
```

ORACLE

The slide example uses the stock price graph that was shown in the preceding slide. You can load the example into your database by using the `CREATE` and `INSERT` statements.

The `ticker` table created in the slide is used in the code example in the next slide.

Pattern Match Example: Simple V Shape with One Row Output Per Match

```
SELECT *
FROM Ticker MATCH_RECOGNIZE (
PARTITION BY symbol
ORDER BY tstamp
MEASURES STRT.tstamp AS start_tstamp,
LAST(DOWN.tstamp) AS bottom_tstamp,
LAST(UP.tstamp) AS end_tstamp
ONE ROW PER MATCH
AFTER MATCH SKIP TO LAST UP
PATTERN (STRT DOWN+ UP+)
DEFINE
DOWN AS DOWN.price < PREV(DOWN.price),
UP AS UP.price > PREV(UP.price)
) MR
ORDER BY MR.symbol, MR.start_tstamp;
```

```
SYMBOL     START_TST BOTTOM_TS END_TSTAM
---------- --------- --------- ---------
ACME       05-APR-11 06-APR-11 10-APR-11
ACME       10-APR-11 12-APR-11 13-APR-11
ACME       14-APR-11 16-APR-11 18-APR-11
```

There are only three rows because only one row per match is chosen to be reported.

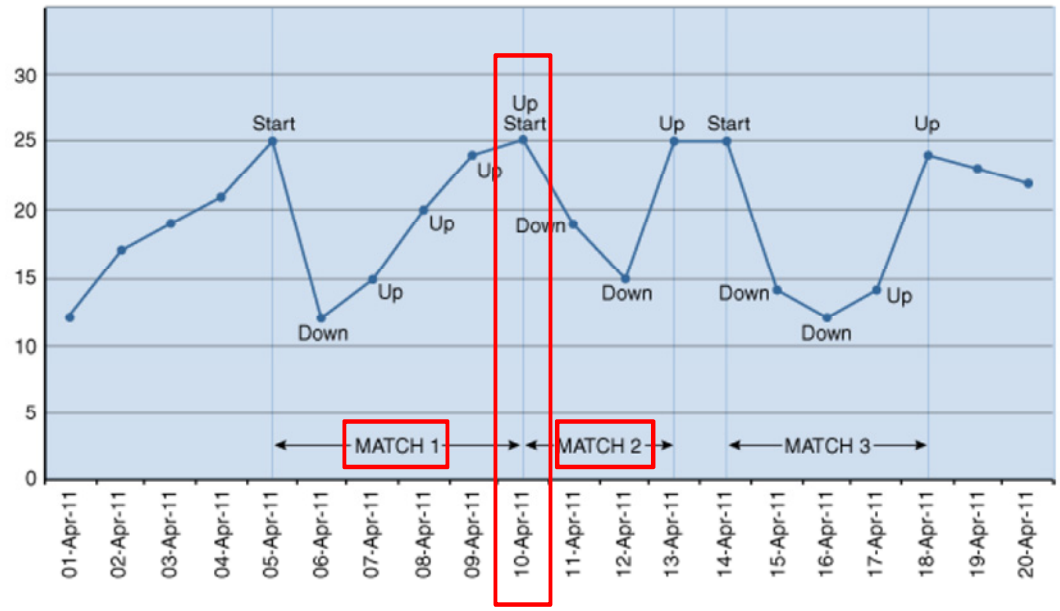
Before examining the slide query, take a look at the output. There are only three rows because only one row per match was chosen to be reported, and three matches were found. `MATCH_RECOGNIZE` lets you choose between showing one row per match or all rows per match. In this example, the shorter output of ***one row per match*** is used. An explanation of each line in the `MATCH_RECOGNIZE` clause is as follows:

- `PARTITION BY` divides the data from the `ticker` table into logical groups where each group contains one stock symbol.
- `ORDER BY` orders the data within each logical group by `tstamp`.
- `MEASURES` defines three measures: the time stamp at the beginning of a **V-shape** (`start_tstamp`), the time stamp at the bottom of a **V-shape** (`bottom_tstamp`), and the time stamp at the end of a **V-shape** (`end_tstamp`). The `bottom_tstamp` and `end_tstamp` measures use the `LAST()` function to ensure that the values retrieved are the final values of the time stamp within each pattern match.
- `ONE ROW PER MATCH` means that for every pattern match found, there is one row of output.
- `AFTER MATCH SKIP TO LAST UP` means that whenever a match is found, the search is restarted at the row, which is the last row of the **UP** pattern variable. A pattern variable is a variable used in a `MATCH_RECOGNIZE` statement, and is defined in the `DEFINE` clause that is described as follows.

`PATTERN (STRT DOWN+ UP+)` indicates that the search pattern has three pattern variables: `STRT`, `DOWN`, and `UP`. The "`+`" after `DOWN` and `UP` indicates that at least one row must be mapped to each of the pattern variables. The pattern defines a regular expression, which is a highly expressive way to search for patterns.

`DEFINE` gives the conditions that must be met for a row to map to the row pattern variables `STRT`, `DOWN`, and `UP`. Because there is no condition for `STRT`, any row can be mapped to `STRT`. A pattern variable with no condition is used as a starting point for testing for matches. Both `DOWN` and `UP` take advantage of the `PREV()` function, which lets them compare the price in the current row to the price in the preceding row. `DOWN` is matched when a row has a lower price than the row that preceded it, so it defines the downward (left) leg of the V-shape. A row can be mapped to `UP` if the row has a higher price than the row that preceded it.

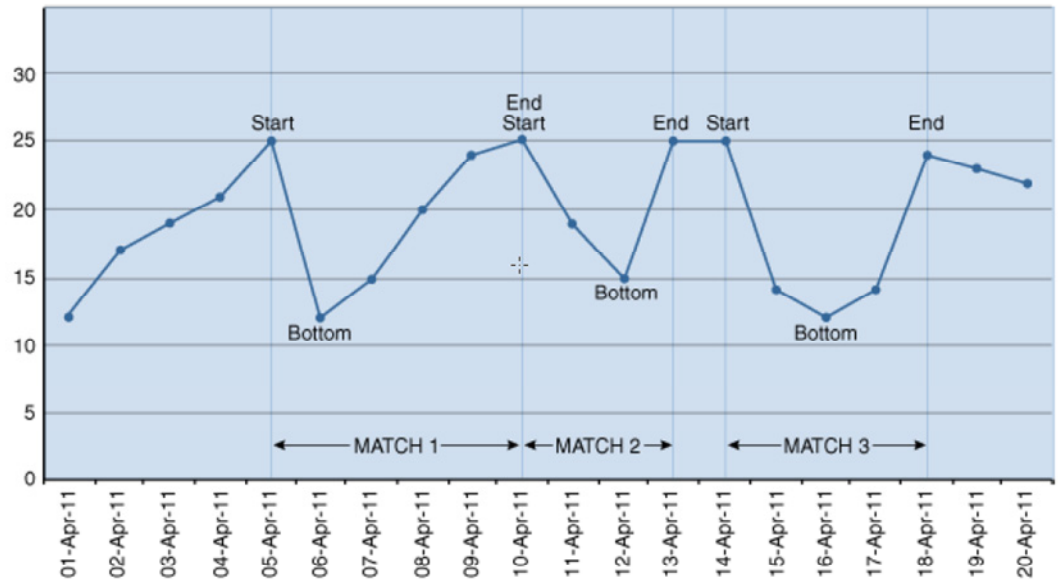
Example: Which Dates Are Mapped to Which Pattern Variables



The slide diagram helps you to better understand the results returned from the query on the previous page. The diagram shows the dates mapped to specific pattern variables, as specified in the `PATTERN` clause. After the mapping of pattern variables to dates is available, this information is used by the `MEASURES` clause to calculate the measure values. The slide diagram labels every date mapped to a pattern variable. The mapping is based on the pattern specified in the `PATTERN` clause and the logical conditions specified in the `DEFINE` clause. The thin vertical lines show the borders of the three matches that were found for the pattern. In each match, the first date has the `STRT` pattern variable mapped to it (labeled as “Start”), followed by one or more dates mapped to the `DOWN` pattern variable, and finally, one or more dates mapped to the `UP` pattern variable. Because `AFTER MATCH SKIP TO LAST UP` was specified in the query, two adjacent matches can share a row. This means that a single date can have two variables mapped to it. For example, 10-April has both the pattern variables `UP` and `STRT` mapped to it:

April 10 is the end of Match 1 and the start of Match 2.

Examples: Which Dates Do the Measures Correspond To?



In the slide diagram, the labels are solely for measures defined in the `MEASURES` clause of the query: `START` (`start_tstamp` in the query), `BOTTOM` (`bottom_tstamp` in the query), and `END` (`end_tstamp` in the query). As shown in the diagram in the preceding slide, the thin vertical lines show the borders of the three matches that were found for the pattern. Every match has a Start date, a Bottom date, and an End date. As with the diagram in the preceding slide, the date **10-April** is found in two matches.

It is the `END` measure for *Match 1* and the `START` measure for *Match 2*. The labeled dates in the slide diagram show the dates that correspond to the measure definitions, which are in turn based on the pattern variable mappings shown in the diagram in the preceding slide.

Note that the dates labeled in the slide diagram correspond exactly to the nine dates shown earlier in the output of the example. The first row of the output has the dates shown in Match 1, the second row of the output has the dates shown in Match 2, and the third row of the output has the dates shown in Match 3.

How Data Is Processed in Pattern Matching By Using MATCH_RECOGNIZE

1. The row pattern input table is partitioned according to the PARTITION BY clause.
2. Each row pattern partition is ordered according to the ORDER BY clause.
3. Each ordered row pattern partition is searched for matches to the PATTERN.
4. A match at the earliest row is sought, considering the rows in a row pattern partition in the order specified by the ORDER BY clause.
5. After a match is found, row pattern matching calculates the row pattern measure columns, which are expressions defined by the MEASURES clause.

MATCH_RECOGNIZE performs the following steps:

1. The row pattern input table is partitioned according to the PARTITION BY clause. Each row pattern partition consists of the set of rows of the row pattern input table that are equal on the row pattern partitioning columns.
2. Each row pattern partition is ordered according to the ORDER BY clause.
3. Each ordered row pattern partition is searched for matches to the PATTERN.
4. Pattern matching operates by seeking a match at the earliest row, considering the rows in a row pattern partition in the order specified by the ORDER BY clause. Pattern matching in a sequence of rows is an incremental process, with one row after another examined to see if it fits the pattern. With the incremental processing model, at any step until the complete pattern has been recognized, you only have a partial match and you do not know what rows might be added in the future, or what variables those future rows might be mapped to. If no match is found at the earliest row, the search moves to the next row in the partition, checking to see if a match can be found starting with that row.
5. After a match is found, row pattern matching calculates the row pattern measure columns, which are expressions defined by the MEASURES clause.

How Data Is Processed in Pattern Matching By Using MATCH_RECOGNIZE

6. Using ONE ROW PER MATCH, pattern matching generates one row for each match that is found.
7. The AFTER MATCH SKIP clause determines where row pattern matching resumes within a row pattern partition after a non-empty match has been found.

6. Using ONE ROW PER MATCH, as shown in the first example in the slide, pattern matching generates one row for each match that is found. If you use ALL ROWS PER MATCH, every row that is matched is included in the pattern match output.
7. The AFTER MATCH SKIP clause determines where row pattern matching resumes within a row pattern partition after a non-empty match has been found. In the example in the slide, row pattern matching resumes at the next row after the rows mapped by a match (AFTER MATCH SKIP TO LAST UP).

Pattern Match for a Simple V-Shape with All Rows Output Per Match

```
-- The first line in the example is used to improve
-- formatting if you are using SQL*Plus.
COLUMN var_match FORMAT A4
SELECT *
FROM Ticker MATCH_RECOGNIZE (
PARTITION BY symbol
ORDER BY tstamp
MEASURES STRT.tstamp AS start_tstamp,
FINAL LAST(DOWN.tstamp) AS bottom_tstamp,
FINAL LAST(UP.tstamp) AS end_tstamp,
MATCH_NUMBER() AS match_num,
CLASSIFIER() AS var_match
ALL ROWS PER MATCH
AFTER MATCH SKIP TO LAST UP
PATTERN (STRT DOWN+ UP+)
DEFINE
DOWN AS DOWN.price < PREV(DOWN.price),
UP AS UP.price > PREV(UP.price)
) MR
ORDER BY MR.symbol, MR.match_num, MR.tstamp;
```


This query is similar to the earlier query, except for the items in the `MEASURES` clause, the change to `ALL ROWS PER MATCH`, and the change to the `ORDER BY` clause at the end of the query. In the `MEASURES` clause, the following additions exist:

- `MATCH_NUMBER() AS match_num`: Because this example returns multiple rows per match, you want to know which rows are members of which match. `MATCH_NUMBER` assigns the same number to each row of a specific match. For instance, all the rows in the first match found in a row pattern partition are assigned the `MATCH_NUMBER` value of `1`. Note that match numbering starts over again at 1 in each row pattern partition.
- `CLASSIFIER() AS var_match`: The `CLASSIFIER` function is used to know which rows map to which variable. In this example, some rows map to the `STRT` variable, some rows the `DOWN` variable, and others to the `UP` variable.
- `FINAL LAST()`: This syntax was added to the `bottom_tstamp` and `end_tstamp` measures. By specifying `FINAL` and using the `LAST()` function for `bottom_tstamp`, every row inside each match shows the same date for the bottom of its **V-shape**.

Similarly, applying `FINAL LAST()` to the `end_tstamp` measure makes every row in each match show the same date for the end of its **V-shape**. Without this syntax, the dates shown would be the running value for each row.

Changes made in two other lines:

- `ALL ROWS PER MATCH`: Whereas the earlier example gave you a summary with just one row of each match by using the line `ONE ROW PER MATCH`, this example gives you every row of each match.
- `ORDER BY` on the last line: This is changed to take advantage of `MATCH_NUM` to view all rows in the same match together and in chronological order.

Pattern Match for a Simple V-Shape with All Rows Output Per Match

```
SYMBOL     TSTAMP    START_TST BOTTOM_TS END_TSTAM  MATCH_NUM VAR_      PRICE
---------- --------- --------- --------- --------- ---------- ---- ----------
ACME       05-APR-11 05-APR-11 06-APR-11 10-APR-11          1 STRT         25
ACME       06-APR-11 05-APR-11 06-APR-11 10-APR-11          1 DOWN         12
ACME       07-APR-11 05-APR-11 06-APR-11 10-APR-11          1 UP           15
ACME       08-APR-11 05-APR-11 06-APR-11 10-APR-11          1 UP           20
ACME       09-APR-11 05-APR-11 06-APR-11 10-APR-11          1 UP           24
ACME       10-APR-11 05-APR-11 06-APR-11 10-APR-11          1 UP           25
ACME       10-APR-11 10-APR-11 12-APR-11 13-APR-11          2 STRT         25
ACME       11-APR-11 10-APR-11 12-APR-11 13-APR-11          2 DOWN         19
ACME       12-APR-11 10-APR-11 12-APR-11 13-APR-11          2 DOWN         15
ACME       13-APR-11 10-APR-11 12-APR-11 13-APR-11          2 UP           25
ACME       14-APR-11 14-APR-11 16-APR-11 18-APR-11          3 STRT         25
ACME       15-APR-11 14-APR-11 16-APR-11 18-APR-11          3 DOWN         14
ACME       16-APR-11 14-APR-11 16-APR-11 18-APR-11          3 DOWN         12
ACME       17-APR-11 14-APR-11 16-APR-11 18-APR-11          3 UP           14
ACME       18-APR-11 14-APR-11 16-APR-11 18-APR-11          3 UP           24

15 rows selected.
```

Note that the row for April 10 appears twice because it is in two pattern matches: It is the last day of the first match and the first day of the second match.

ORACLE

The slide shows the output of the code example in the preceding slide.

Note that the row for **April 10** appears twice because it is in two pattern matches: It is the last day of the first match *and* the first day of the second match.

Pattern Match for a W-Shape

```
SELECT *
FROM Ticker MATCH_RECOGNIZE (
PARTITION BY symbol
ORDER BY tstamp
MEASURES STRT.tstamp AS start_tstamp,
UP.tstamp AS end_tstamp
ONE ROW PER MATCH
AFTER MATCH SKIP TO LAST UP
PATTERN (STRT DOWN+ UP+ DOWN+ UP+)
DEFINE
DOWN AS DOWN.price < PREV(DOWN.price),
UP AS UP.price > PREV(UP.price)
) MR
ORDER BY MR.symbol, MR.start_tstamp;
```

SYMBOL	START_TST	END_TSTAM
ACME	05-APR-11	13-APR-11

This query is identical to the first query, except for the line defining the `PATTERN`. The regular expression has been modified to seek the pattern `DOWN`, followed by `UP`, two consecutive times: `PATTERN (STRT DOWN+ UP+ DOWN+ UP+)`. This pattern specification means that the query can only match a W-shape where the two V-shapes have no separation between them. For instance, if there is a flat interval with the price unchanging, and that interval occurs between two V-shapes, the pattern will not match that data.

Nesting FIRST and LAST Within PREV and NEXT

- FIRST and LAST provide navigation within the set of rows that are already mapped to a particular pattern variable.
- PREV and NEXT provide navigation by using a physical offset from a particular row.
- You can combine these navigation types by nesting FIRST or LAST within PREV or NEXT:
  - `PREV (LAST (A.Price + A.Tax, 1), 3)`

Pattern matching is an extremely rich feature that offers many capabilities beyond the scope of this lesson. In this slide and the next few slides, you note four capabilities or features that add to its power. FIRST and LAST provide navigation within the set of rows that are already mapped to a particular pattern variable; PREV and NEXT provide navigation by using a physical offset from a particular row. These navigation types may be combined by nesting FIRST or LAST within PREV or NEXT. In the slide example, A must be a pattern variable. It is required to have a row pattern column reference, and all pattern variables in the compound operator must be equivalent (A, in this example).

Handling Empty Matches or Unmatched Rows

ALL ROWS PER MATCH has three options:

- ALL ROWS PER MATCH *SHOW EMPTY MATCHES*
- ALL ROWS PER MATCH *OMIT EMPTY MATCHES*
- ALL ROWS PER MATCH *WITH UNMATCHED ROW*

ALL ROWS PER MATCH has three suboptions:

- ALL ROWS PER MATCH SHOW EMPTY MATCHES: With this option, an empty match generates a single row in the row pattern output table.
- ALL ROWS PER MATCH OMIT EMPTY MATCHES: With this option, an empty match is omitted from the row pattern output table.
- ALL ROWS PER MATCH WITH UNMATCHED ROWS: This option shows both empty matches and unmatched rows.

For details on the full capabilities in pattern matching, see the chapter titled “Pattern Matching” in the *Oracle Database 12c Data Warehousing* Guide.

Some Additional Capabilities of Pattern Matching

- Exclude portions of the pattern from the output.
- Handle patterns where multiple orderings are satisfactory matches by using the `PERMUTE` syntax.
  - For example, `PATTERN (PERMUTE (A, B, C))` is equivalent to an alternation of all permutations of three pattern variables `A`, `B`, and `C`, as follows:
  - `PATTERN (A B C | A C B | B A C | B C A | C A B | C B A)`

Exclude Portions of the Pattern from the Output

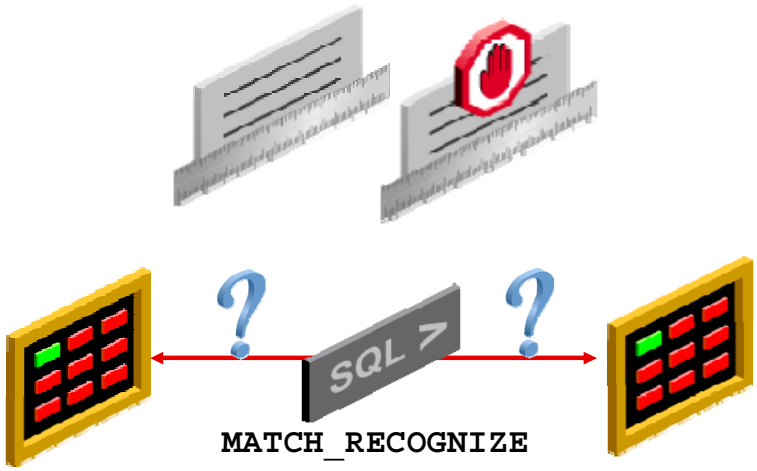
When using `ALL ROWS PER MATCH` with either the `OMIT EMPTY MATCHES` or `SHOW EMPTY MATCHES` options, rows matching a portion of the `PATTERN` may be excluded from the row pattern output table. The excluded portion is bracketed between `{-` and `-}` in the `PATTERN` clause.

Handling Patterns Where Multiple Orderings Are Satisfactory Matches

The `PERMUTE` syntax may be used to express a pattern that is a permutation of simpler patterns as shown in the slide example.

Rules and Restrictions in Pattern Matching

- Input table requirements
- Prohibited nesting in the MATCH_RECOGNIZE clause
- Concatenated pattern matching
- Aggregate restrictions



Input Table Requirements: The row pattern input table is the input argument to MATCH_RECOGNIZE. You can use a table or view, or a named query (defined by using the WITH clause). The row pattern input table can also be a derived table (also known as **in-line view**).

Prohibited Nesting in the MATCH_RECOGNIZE Clause: The following kinds of nesting are prohibited in pattern matching:

- Nesting one pattern matching clause within another
- Outer references in MEASURES or DEFINE. This means that pattern matching may not reference any table in an outer query block, except the row pattern input table.
- Correlated subqueries in MEASURES or DEFINE. Also, subqueries in MEASURES or DEFINE cannot reference pattern variables.
- Pattern matching in recursive queries
- SELECT FOR UPDATE with MATCH_RECOGNIZE

Concatenated Pattern Matching: It is not prohibited to feed the output of one pattern matching as the input of another.

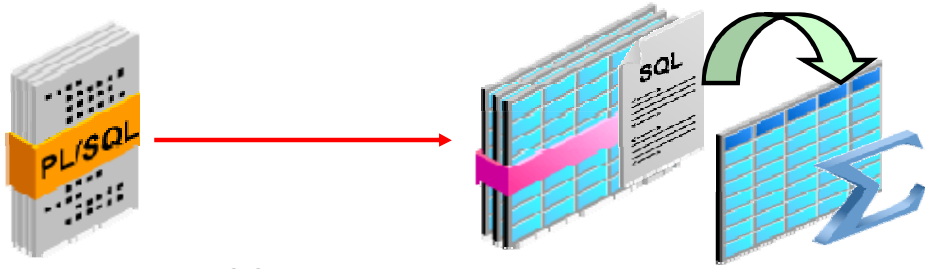
Aggregate Restrictions: Only the following aggregate functions can be used in the MEASURES and DEFINE clauses: COUNT, SUM, AVG, MAX, and MIN.

Lesson Agenda

- Maintaining Multiple Partitions
- Partial Global Indexes For Partitioned Tables
- Asynchronous Global Index Maintenance
- SQL for Pattern Matching
- Synchronous Materialized View Refresh
- Improving Query Performance Against OLAP Cubes

Synchronous Refresh: Overview

- You can now keep your base tables in sync with the materialized views (MV) that are built on them.
  - This is particularly suited for data warehouses where the loading of data is tightly controlled.
  - You can avoid the problem of stale MVs impacting performance.
- Synchronous Refresh is a new refresh method introduced in Oracle Database Release 12*c*.



`DBMS_SYNC_REFRESH` PL/SQL package

Materialized views (MV) can now be refreshed simultaneously with its base tables by leveraging partitioning and the logical dependencies between tables and the corresponding materialized views. This means that the amount of time a materialized view is stale (meaning its data is not up-to-date) is minimized, thus increasing its availability.

Synchronous refresh presents a new paradigm for maintaining tables and MVs in a data warehouse. Using this refresh method, tables and MVs are refreshed at the same time. This is suited for data warehouses where the loading of data is tightly controlled. You can also avoid the problem of stale MVs, which impacts performance.

In other cases, changes are applied to the base tables and the MVs are refreshed separately with one of the other refresh methods such as log-based incremental (fast) refresh, `PCT` refresh if it is applicable, or a complete refresh.

The synchronous refresh method combines some elements of the log-based incremental (fast) refresh and `PCT` refresh methods but it is applicable only to `ON DEMAND` MVs unlike the other two methods. When you use synchronous refresh, there are three major differences between it and the other refresh methods:

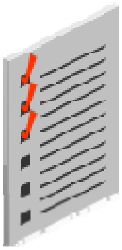
- You must register the tables and MVs with it.
- You must specify the changes to data according to some formally specified rules.

- Synchronous refresh works by dividing the refresh operation into two steps: preparation and execution. This approach provides it important advantages over the other methods.

The APIs for synchronous refresh are defined in a new package that is introduced in Oracle Database 12*c* called `DBMS_SYNC_REFRESH`.

Key Requirements and Partitioning Types for Synchronous Refresh

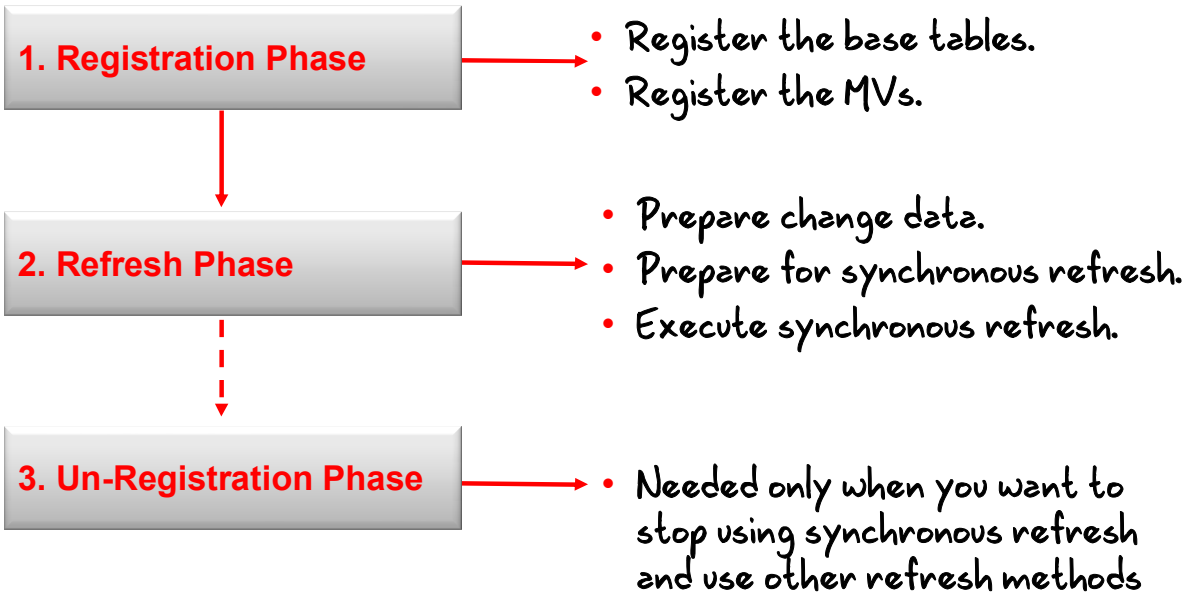
- The MVs must also be partitioned along the same dimension as the fact table.
- The partition key of the fact table should functionally determine the partition key of the MV.
- Synchronous refresh supports two types of partitioning on fact tables and MVs:
  - Range partitioning
  - Composite partitioning



A key requirement of synchronous refresh is that the MVs must also be partitioned along the same dimension as the fact table. In addition, the partition key of the fact table should functionally determine the partition key of the MV. The term “functionally determine” means that the partition key of the MV must be derivable from the partition key of the fact table based on a foreign-key constraint relationship. This condition is satisfied if the partition key of the MV is the same as the fact table or related by joins from the fact table to the dimension table as in a star or snowflake schema. For example, if the fact table is partitioned by a date column such as `TIME_KEY`, the MV can be partitioned by `TIME_KEY` or `MONTH` or `YEAR`.

Synchronous refresh supports two types of partitioning on fact tables and MVs: range partitioning and composite partitioning, with the top-level partitioning type being range.

Synchronous Refresh Phases: Overview



A synchronous refresh has three main processes:

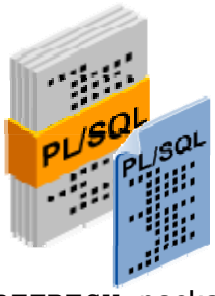
- In the **Registration** phase, you register the base tables and the MVs.
- In the **Refresh** phase, you prepare the change data, prepare for synchronous refresh, and execute the synchronous refresh.
- The **Un-registration** phase is needed only if you want to stop using synchronous refresh and use other available refresh methods instead.

Notes

- For additional detailed information about the Synchronous Refresh feature, see the Oracle Data Warehousing reference guide.
- The `rdbms/demo` Oracle installation directory contains several demo scripts such as `syncref_run.sql`. This script demonstrates the Synchronous Refresh operations by using staging logs and registered-partition-operations for loading change data. This script includes the following scripts: `syncref_setup.sql`, `utlsrt.sql`, `syncref_cst.sql`, and `syncref_cleanup.sql`. You must copy all the scripts into your working directory before running `syncref_run.sql`. The expected log output from this script is stored in `syncref_run.log`.

Synchronous Refresh Preparation and Execution APIs

- The APIs for synchronous refresh are defined in a new package called DBMS_SYNC_REFRESH.
- The DBMS_SYNC_REFRESH package contains the following procedures for synchronous refresh:
  - REGISTER_MVIEWS
  - PREPARE_STAGING_LOG
  - REGISTER_PARTITION_OPERATION
  - PREPARE_REFRESH
  - EXECUTE_REFRESH
  - ABORT_REFRESH



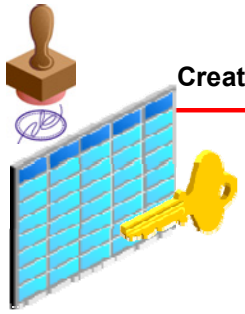
The DBMS_SYNC_REFRESH package procedures

The APIs for synchronous refresh are defined in a new package that was introduced in Oracle Database 12*c* called DBMS_SYNC_REFRESH.

The DBMS_SYNC_REFRESH package contains the following procedures for synchronous refresh:

- REGISTER_MVIEWS
- PREPARE_STAGING_LOG
- REGISTER_PARTITION_OPERATION
- PREPARE_REFRESH
- EXECUTE_REFRESH
- ABORT_REFRESH

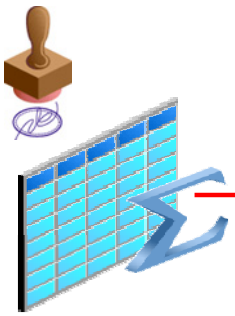
Synchronous Refresh: Registration Phase



Create staging log.

```
CREATE MV LOG ON FACT
FOR SYNCHRONOUS REFRESH USING ST_FACT
```

1. Register the base tables.



```
EXECUTE
DBMS_SYNC_REFRESH.REGISTER_MVIEWS('MV1');
```

Execute the `DBMS_SYNC_REFRESH.REGISTER_MVIEWS` procedure.

2. Register the MVs.

ORACLE

Synchronous refresh provides a couple of methods to register tables and MVs.

Tables are registered with synchronous refresh by creating a staging log on them. You can create a staging log with the `CREATE MV LOG` statement. The statement's syntax has been extended in this release to create staging logs, as well as the familiar MV logs that are used for the traditional incremental refresh.

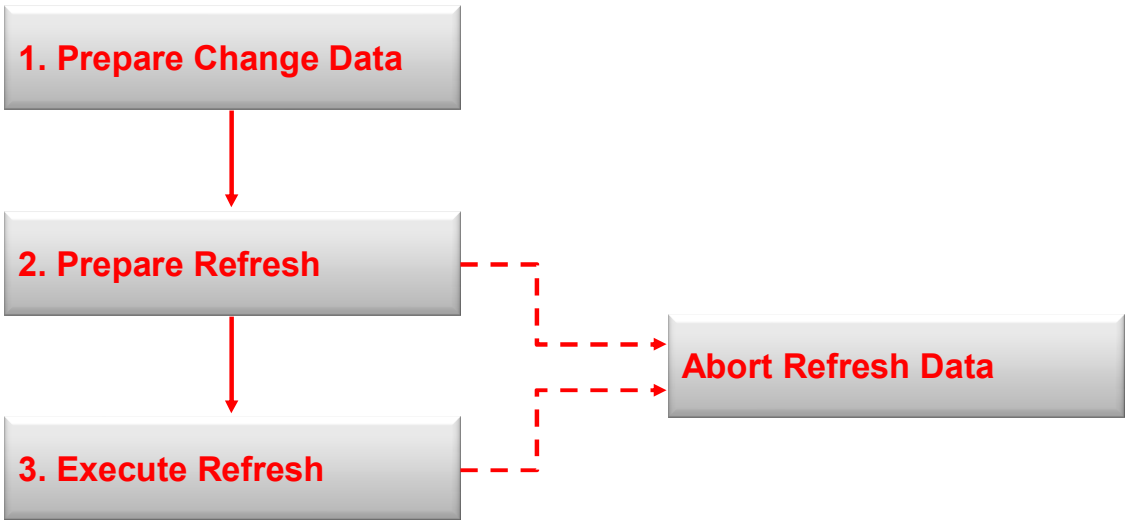
After you create a staging log on a table, it is registered with synchronous refresh and can be modified only by using the synchronous refresh procedures. In other words, a table with a staging log defined on it is registered with synchronous refresh and you cannot modify it.

Next, you register the MVs. MVs are registered with synchronous refresh by using the `REGISTER_MVIEWS` procedure in the `DBMS_SYNC_REFRESH` package. For example, you can register an MV with the following statement:

```
EXECUTE DBMS_SYNC_REFRESH.REGISTER_MVIEWS('MV1');
```

The `REGISTER_MVIEWS` procedure implicitly creates groups of related objects called sync refresh groups. A sync refresh group consists of all related MVs and tables that must be refreshed together as a single entity because they are dependent on one another.

Synchronous Refresh: Refresh Phase



The **Refresh** phase can be used repeatedly to perform synchronous refresh. The following are the three main steps in this phase:

1. Prepare the change data for the refresh operation.
2. Perform the first step of the refresh operation. You execute the `DBMS_SYNC_REFRESH.PREPARE_REFRESH` procedure. This can potentially be a long-running operation because it prepares and loads the outside tables. If this procedure raises a user error, you can execute the `ABORT_REFRESH` procedure to restore the tables and MVs to the state before the refresh operation, fix the problem, and retry the refresh operation, starting from the beginning.
3. Perform the second and last step of the refresh operation. You execute the `DBMS_SYNC_REFRESH.EXECUTE_REFRESH` procedure. This should normally run very fast because it usually consists of a series of partition-exchange operations. If this procedure raises a user error, you can execute the `ABORT_REFRESH` procedure to restore the tables and MVs to the state before the refresh operation, fix the problem, and retry the refresh operation, starting from the beginning.

Refresh Phase: Preparing the Change Data

There are two ways to specify the change data:

- Provide the change data in an outside table and register it with `REGISTER_PARTITION_OPERATION`.
- Provide the change data in staging logs and process the logs with `PREPARE_STAGING_LOG`.
  - You must run `PREPARE_STAGING_LOG` for every table before executing the refresh operation on that table.

In this step of the Refresh phase, you prepare the change data for the refresh operation.

There are two ways to specify the change data:

- You can use the `REGISTER_PARTITION_OPERATION` procedure to provide the change data directly. This method is applicable only to fact tables. For each fact table partition that is changed, you need to provide an outside table containing the data for that partition. You can provide the change data in an outside table, and then register it with the `DBMS_SYNC_REFRESH.REGISTER_PARTITION_OPERATION` procedure. For information about using the `REGISTER_PARTITION_OPERATION` procedure method to specify change data, see the ***Oracle Data Warehousing*** Guide.
- You can provide the change data in staging logs, and then process the staging logs with the `DBMS_SYNC_REFRESH.PREPARE_STAGING_LOG` procedure before proceeding to the next step.

In this lesson, only specifying the change data by using staging logs is covered.

Specifying Change Data with Staging Logs

- You can create staging logs by using a DDL statement, and you may alter them to be materialized view logs.
- Load changes into the staging logs in a specified format.
- Each row has a staging log key to uniquely identify it. The staging log will consist of:
  - All the columns in the base table
  - An additional control column `DMLTYPE$$` of type `CHAR(2)`

Specify in pairs.

Task/Keyword	Description
I	Row being inserted
D	Row being deleted
UO	The old values of the row to be updated
UN	The new values of the row to be updated
I	Row being inserted

ORACLE

In synchronous refresh, staging logs play a role that is analogous to materialized view logs in incremental refresh. They are created with a DDL and may be altered to being materialized view logs. Unlike materialized view logs, however, you are responsible for loading changes into the staging logs in a specified format. Each row in the staging log needs to have a key to identify it uniquely; this key is called the staging log key.

You are responsible for populating the staging log, which consists of all the columns in the base table and an additional control column, `DMLTYPE$$`, of type `CHAR(2)`. The control column must have the value `I` to denote a row being inserted, `D` for delete, and `UO` and `UN` for the old and new values of the row being updated, respectively. `UO` and `UN` must be specified in pairs, which means that you must specify the old values followed by the new values of the row being updated.

For information about and examples on staging log rules and keys, see the *Oracle Database Data Warehousing Guide 12c Release 1 (12.1)*.

Specifying Change Data with Staging Logs

- The staging log is:
  - Validated by `PREPARE_STAGING_LOG`
  - Consumed by the synchronous refresh operations `PREPARE_REFRESH` and `EXECUTE_REFRESH`
- During validation by `PREPARE_STAGING_LOG`, if errors are detected, they are captured in an exceptions table.
- You can query the view `USER_SR_STLOG_EXCEPTIONS` to get details on the exceptions.
- The staging log for all the tables in the group must be processed with `PREPARE_STAGING_LOG` before calling `PREPARE_REFRESH` for the sync refresh groups, even if a table has no change data and its staging log is empty.

ORACLE

The staging log is validated by `PREPARE_STAGING_LOG` and consumed by the synchronous refresh operations, `PREPARE_REFRESH` and `EXECUTE_REFRESH`.

During validation by `PREPARE_STAGING_LOG`, if errors are detected, they are captured in an exceptions table. You can query the view `USER_SR_STLOG_EXCEPTIONS` to get details on the exceptions.

The staging log for all the tables in the group must be processed with `PREPARE_STAGING_LOG` before calling `PREPARE_REFRESH` for the sync refresh groups, even if a table has no change data and its staging log is empty.

Populating the Staging Logs with Change Data: Example

```
-- Populate the staging logs with the change data

insert into  st_time (dmltype$$, time_key, month, year, quarter)
values ('I', '11-FEB-1998', 199802, 1998, 19982);

insert into st_store (dmltype$$, store_key, store_number, store_name, zipcode)
values ('I',   5, 5, 'Store 5', '03060');

insert into st_store (dmltype$$, store_key, store_number, store_name, zipcode)
values ('I',   6, 6, 'Store 6', '03062');

insert into st_store (dmltype$$, store_key, store_number, store_name, zipcode)
values ('UO',  4, 4, 'Store 4', '03062');

insert into st_store (dmltype$$, store_key, store_number, store_name, zipcode)
values ('UN',  4, 4, 'Stor4NewNam', '03062');

insert into st_store (dmltype$$, store_key, store_number, store_name, zipcode)
values ('D',   3, 3, 'Store 3', '03060');
```

ORACLE

The examples in this slide and the next slide show the section of the `syncref_run.sql` script from the Synchronous Refresh demo (available in the `home/oracle/labs/demos` folder) where the staging logs are populated with the change data for the `fact`, `time`, and `store` tables. Examine the insert statements in the slide:

1. The first insert into `st_time`, the staging log for the `time` dimension table, represents an `INSERT` operation as indicated by the value of the `dmltype$$` column: `I`. A new row is added to the `time` dimension base table with a time key of '`11-FEB-1998`'.
2. The next two inserts into the `st_store` staging log represent an `INSERT` of two new rows in the `store` dimension table: `Stores 5` and `6`.
3. The next two inserts into `st_store` represent an update operation:
   a. In the first insert, the value of `dmltype$$` is `UO` for update old. It specifies the old values of the row to be updated in the `store` table where the store # is `4` and the store name is `Store 4`.
   b. In the second insert, the value of `dmltype$$` is `UN` for update new. This indicates that it is an update operation. It specifies the new values of the row to be updated. The store name is changed from `Store 4` to `Store4NewNam`.
4. The next insert represents a delete operation, as indicated by `D` in the `dmltype$$` column. Here, the row for `store 3` is deleted.

Populating the Staging Logs with Change Data: Example

```
insert into st_fact (dmltype$$, time_key, store_key, dollar_sales, unit_sales)
values('I', '11-FEB-1998', 1, 100, 100);

insert into st_fact (dmltype$$, time_key, store_key, dollar_sales, unit_sales)
values('I', '11-FEB-1998', 2, 200, 200);

insert into st_fact (dmltype$$, time_key, store_key, dollar_sales, unit_sales)
values('D', '10-FEB-1998', 3, 300, 300);

insert into st_fact (dmltype$$, time_key, store_key, dollar_sales, unit_sales)
values('I', '11-FEB-1998', 5, 500, 500);

insert into st_fact (dmltype$$, time_key, store_key, dollar_sales, unit_sales)
values('I', '10-FEB-1998', 6, 600, 600);

insert into st_fact (dmltype$$, time_key, store_key, dollar_sales, unit_sales)
values('UO', '10-FEB-1998', 4, 400, 400);

insert into st_fact (dmltype$$, time_key, store_key, dollar_sales, unit_sales)
values('UN', '10-FEB-1998', 4, 401, 401);
```


5. The next two inserts insert two new rows into the fact table as indicated by `I` in the `dmltype$$` column.
   a. In the first insert, a new row with '`11-FEB-1998`' for `time_key`, `1` for store #, and `100` for `dollar_sales` and `unit_sales` is added to the fact table.
   b. In the second insert, a row with '`11-FEB-1998`' for `time_key`, `2` for store #, and `200` for `dollar_sales` and `unit_sales` is added to the fact table.
6. Next, as indicated by `D` in the `dmltype$$` column, a row is deleted from the fact table. This is the row where the `time` value is `February 10, 1998`, the store # is `3`, and `dollar_sales` and `unit_sales` are `300`.
7. The next two inserts insert two new rows in the fact table:
   a. For the first row, `time_key` is '`11-FEB-1998`', `store_key` is `5`, and `dollar_sales` and `unit_sales` are `500`.
   b. For the second row, `time_key` is '`10-FEB-1998`', `store_key` is `6`, and `dollar_sales` and `unit_sales` are `600`.
8. The last two inserts represent an update operation as indicated by the `UO` and `UN` pair of values in the `dmltype$$` column. Basically, the value `400` is being replaced with `401` for `dollar_sales` and `unit_sales`.

Using the PREPARE_STAGING_LOG Procedure: Example

```
SQL> execute dbms_sync_refresh.prepare_staging_log('syncref_user', 'fact');

PL/SQL procedure successfully completed.

SQL> execute dbms_sync_refresh.prepare_staging_log('syncref_user', 'time');

PL/SQL procedure successfully completed.

SQL> execute dbms_sync_refresh.prepare_staging_log('syncref_user', 'store');

PL/SQL procedure successfully completed.
```

DBMS_SYNC_REFRESH. *PREPARE_STAGING_LOG* procedure

ORACLE

In the slide example, the staging log for the `fact`, `time`, and `store` base tables from the Synchronous Refresh demo is successfully processed with the `PREPARE_STAGING_LOG` procedure.

Synchronous Refresh Groups

- Changes to a table and its MVs are loaded and refreshed simultaneously.
- Database objects must be registered for tables and MVs to be maintained by synchronous refresh.
- All tables that are related by constraints must be refreshed simultaneously to ensure data integrity.
- Some of the tables registered for synchronous refresh have several MVs built on top of them, in which case all those MVs must also be refreshed simultaneously.
- Oracle Database 12*c* automatically generates the minimal sets of tables and MVs, which must be refreshed together. These sets are called *Synchronous Refresh Groups.*

The distinguishing feature of synchronous refresh is that changes to a table and its MVs are loaded and refreshed simultaneously—thus the name synchronous refresh. In order for the tables and MVs to be maintained by synchronous refresh, the objects must be registered. Tables are registered for synchronous refresh when staging logs are created on them and MVs are registered with the `REGISTER_MVIEWS` procedure.

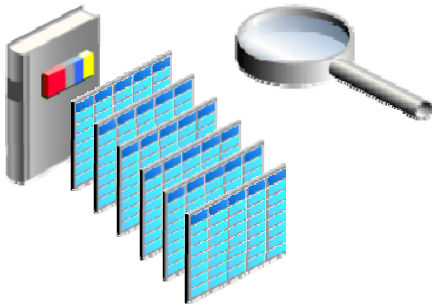
Synchronous refresh supports the refresh of MVs built on multiple tables, with changes in one or more of them. Tables that are related by constraints must all necessarily be refreshed simultaneously to ensure data integrity. Furthermore, it is possible that some of the tables that are registered for synchronous refresh have several MVs built on top of them, in which case all those MVs must also be refreshed simultaneously.

Instead of having you keep track of these dependencies and issue the refresh commands on the right set of tables, Oracle automatically generates the minimal sets of tables and MVs, which must necessarily be refreshed together. These sets are called Synchronous Refresh Groups or just sync refresh groups. Each sync refresh group is identified by a `GROUP_ID`.

The three procedures that are related to performing synchronous refresh (`PREPARE_REFRESH`, `EXECUTE_REFRESH`, and `ABORT_REFRESH`) take as input either a single group ID or a list of group IDs that identify the sync refresh groups.

Catalog Views: Overview

Catalog View	Category
USER_SR_OBJ	Object views
USER_SR_OBJ_STATUS	Object status views
USER_SR_GRP_STATUS	Group status views
USER_SR_STLOG_STATS	Staging log statistics views
USER_SR_STLOG_EXCEPTIONS	Staging log exceptions views



Some of the useful catalog views that you can use with Synchronous Refresh are as follows:

- The object (USER_SR_OBJ) views contain information about the objects (tables and MVs) that are registered for synchronous refresh.
- The object status (USER_SR_OBJ_STATUS) views contain information about the status of the objects in a group after the refresh operations.
- The group status (USER_GRP_STATUS) views contain information about the status of the refresh operations of the sync refresh groups. Synchronous refresh is done in two stages: prepare and execute. The OPERATION field in these views indicates the current step.
- The staging log statistics (USER_SR_STLOG_STATS) views contain information about the statistics in the staging logs when a user performs a PREPARE_STAGING_LOG operation.
- The staging log exceptions (USER_SR_STLOG_EXCEPTIONS) views contain information about the exceptions found when a user performs a PREPARE_STAGING_LOG operation.
- The partition operations (USER_SR_PARTN_OPS) views contain information about the partitions prepared and registered by a user with the REGISTER_PARTITION_OPERATION command.

Using the USER_SR_STLOG_STATS Catalog View: Example

```
select table_name, staging_log_name, num_inserts, num_deletes, num_updates
from user_sr_stlog_stats
order by table_name;
```

```
TABLE_NAME STAGING_LOG_NAME NUM_INSERTS NUM_DELETES NUM_UPDATES
---------- ---------------- ----------- ----------- -----------
FACT       ST_FACT                    4           1           1
STORE      ST_STORE                   2           1           1
TIME       ST_TIME                    1           0           0

3 rows selected.
```


In the slide example, the status of the staging logs after preparation in the Synchronous Refresh demo is being checked. Notice that the number of inserts, deletes, and updates are non-null values even after preparing the staging logs by using the PREPARE_STAGING_LOG procedure. You will see later that the number of inserts, deletes, and updates are null values only after you successfully run the EXECUTE_REFRESH procedure. This indicates that the staging logs have been processed.

USER_SR_GRP_STATUS Catalog View

Displays the status of the whole group:

Field	Description	Values
OPERATION	Current refresh operation being run on the group	• PREPARE • EXECUTE
STATUS	Status of the operation	• RUNNING • COMPLETE • ERROR-SOFT • ERROR-HARD • ABORT • PARTIAL

```
select operation, status from user_sr_grp_status
where group_id = dbms_sync_refresh.get_group_id('MV1');
```

```
OPERATION  STATUS
---------- ----------------
PREPARE    COMPLETE

1 row selected.
```


The DBMS_SYNC_REFRESH package provides three APIs to control the refresh execution process. The user initiates the synchronous refresh with PREPARE_REFRESH, which plans the entire refresh operation and performs the bulk of the computational work for refresh, followed by EXECUTE_REFRESH, which carries out the refresh. The third API that is provided is ABORT_REFRESH, which is used to recover from errors if either of these procedures fails.

The USER_SR_GRP_STATUS and USER_SR_OBJ_STATUS catalog views contain all the information about the status of these refresh operations for current groups:

- The USER_SR_GRP_STATUS shows the status of the group as a whole.
  - The OPERATION field indicates the current refresh operation run on the group: PREPARE or EXECUTE.
  - The STATUS field indicates the status of the operation: RUNNING, COMPLETE, ERROR-SOFT, ERROR-HARD, ABORT, or PARTIAL. These values are explained in the next slide. The group is identified by its group ID.

The code example in the slide checks the status of the group itself after the PREPARE_REFRESH operation. The output shows that the operation column is set to PREPARE and the status is set to COMPLETE.

USER_SR_OBJ_STATUS Catalog View

Displays the status of each individual object:

Field	Description	Values
TYPE	Object type	• TABLE • MVIEW
STATUS	Status of the object	• NOT PROCESSED • ABORT • COMPLETE

```
select name, type, status from user_sr_obj_status
where group_id = dbms_sync_refresh.get_group_id('MV1')
order by type, name;
```

```
NAME            TYPE       STATUS
--------------- ---------- ----------------
MV1             MVIEW      COMPLETE
FACT            TABLE      COMPLETE
STORE           TABLE      COMPLETE
TIME            TABLE      COMPLETE

4 rows selected.
```

ORACLE

USER_SR_OBJ_STATUS shows the status of each individual object.

- The object is identified by its owner, name, type (TABLE or MVIEW), and group ID.
- The STATUS field can be NOT PROCESSED, ABORT, or COMPLETE. These keywords are explained in detail later.

The code example in the slide checks the status of the objects in the group after the EXECUTE_REFRESH operation. The STATUS column is set to COMPLETE for the base tables and the MV.

How PREPARE_REFRESH Sets the STATUS Fields

- When you launch a new PREPARE_REFRESH job:
  - The group's STATUS is set to RUNNING
  - The STATUS of the objects in the group is set to NOT PROCESSED
- When the PREPARE_REFRESH procedure completes:
  - The status of the objects remains unchanged
  - The group's status is changed to one of following values:
    - COMPLETE: The job completed successfully.
    - ERROR_SOFT: The job encountered the ORA-01536: space quota exceeded for tablespace '%s' error.
    - ERROR_HARD: The job encountered any error other than ORA-01536.

When you launch a new PREPARE_REFRESH job:

- The group's STATUS is set to RUNNING. If the STATUS is RUNNING even after PREPARE_REFRESH finishes, this means that an error has occurred.
- The STATUS of the objects in the group is set to NOT PROCESSED. The NOT PROCESSED status of the objects in the group signifies that the data of the objects has not been modified by the PREPARE_REFRESH job. The data modification will occur only in the EXECUTE_REFRESH step, at which time the status will be changed as appropriate.

When the PREPARE_REFRESH job finishes, the status of the objects remains unchanged, but the group's status is changed to one of the following three values:

- COMPLETE if the job completed successfully
- ERROR_SOFT if the job encountered the ORA-01536: space quota exceeded for tablespace '%s' error. You can fix this error by increasing the space quota for the appropriate tablespace, and then resume PREPARE_REFRESH or abort the refresh with ABORT_REFRESH.
- ERROR_HARD if the job encountered any error than ORA-01536. This status may be related to running out of resources because PREPARE_REFRESH can be resource intensive.

How PREPARE_REFRESH Sets the STATUS Fields

- If STATUS is RUNNING even after PREPARE_REFRESH finishes, this means that an error has occurred. Contact Oracle Support Services for assistance.
- If STATUS of PREPARE_REFRESH at the end is not COMPLETE:
  - You cannot proceed to the EXECUTE_REFRESH step
  - You can proceed to the *Unregistration* phase, and then maintain the objects in the groups by using other refresh methods
- You can launch a new PREPARE_REFRESH job only when the status of the previous refresh operation on the group is COMPLETE or ABORT.

ORACLE

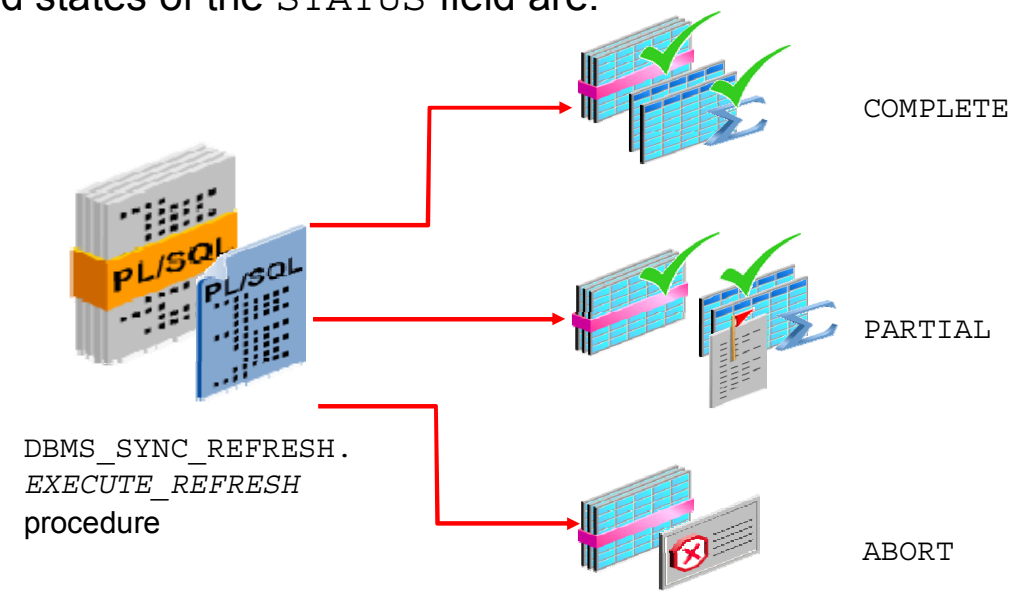
If STATUS is RUNNING even after PREPARE_REFRESH completes, this means that an error has occurred. Contact Oracle Support Services for assistance.

If STATUS of PREPARE_REFRESH at the end is not COMPLETE, you cannot proceed to the EXECUTE_REFRESH step. In this case, you can proceed to the *Unregistration* phase, and then maintain the objects in the groups by using other refresh methods.

You can launch a new PREPARE_REFRESH job only when the previous refresh operation on the group (if any) completes successfully or is aborted.

How EXECUTE_REFRESH Sets the STATUS Fields

In the case of the EXECUTE_REFRESH procedure, the possible end states of the STATUS field are:



The EXECUTE_REFRESH procedure divides the groups of objects in the sync refresh group into subgroups, each of which is refreshed atomically. The first subgroup consists of the base tables. Each MV in the sync refresh group is placed in a separate subgroup and refreshed atomically.

In the case of EXECUTE_REFRESH, the possible end states of the STATUS field are COMPLETE, PARTIAL, and ABORT.

Possible End State for STATUS Field for EXECUTE_REFRESH

Status Field	Description
COMPLETE	The base tables and all MVs have refreshed successfully.
PARTIAL	• All the base tables have refreshed successfully and some, but not all MVs have refreshed successfully. • The data in the tables and MVs that have refreshed successfully is consistent in both; the other MVs are stale and need a complete refresh.

COMPLETE

PARTIAL

ABORT

In the case of EXECUTE_REFRESH, the possible end states of the STATUS field are:

- COMPLETE: This state is reached if the base tables and all the MVs refresh successfully.
- PARTIAL: If all the base tables refresh successfully and some, but not all MVs refresh successfully, this state is reached. The data in the tables and MVs that have refreshed successfully is consistent in both; the other MVs are stale and need a complete refresh. If this happens, there should be an error associated with the failure. You can try the following:
  - Retry execution of EXECUTE_REFRESH. In such a case, EXECUTE_REFRESH will retry the refresh of the failed MVs with another refresh method such as PCT-refresh or COMPLETE refresh.
  - Invoke ABORT_REFRESH to abort the MVs. This will roll back changes to all MVs and base tables. They will all have the same data as was in the original state before any of the changes in the staging logs or registered partition operations were applied to them.
- ABORT: This indicates that the refresh of the base tables subgroup has failed; the data in the tables and MVs is consistent but unchanged.

When the refresh fails, there should be an error associated with the failure. If it is a user error, such as a constraint violation, you can fix the problem and retry the synchronous refresh operation from the beginning; that is, `PREPARE_STAGING_LOG` for each table in the group, `PREPARE_REFRESH`, and then `EXECUTE_REFRESH`. Otherwise, you should contact Oracle Support Services.

Synchronous Refresh Demo Files and Location

- Download the Synchronous Refresh demo files from the `rdbms/demo` Oracle Database installation directory.
- The demo contains five scripts and one log file.
- The main script is named `syncref_run.sql` and it calls the following scripts:
  - `syncref_setup.sql`
  - `utlsrt.sql`
  - `syncref_cst.sql`
  - `syncref_cleanup.sql`
- The `syncref_run.log` log file contains the commands and the output of the `syncref_run.sql` script.

You can download the Synchronous Refresh demo files from the `rdbms/demo` Oracle Database installation directory. The demo consists of five scripts and one log file. The main script is named `syncref_run.sql`. This script demonstrates the Synchronous Refresh operations by using staging logs and registered-partition-operations for loading change data among other things. This script calls the following other scripts:

- `syncref_setup.sql`
- `utlsrt.sql`
- `syncref_cst.sql`
- `syncref_cleanup.sql`

The `syncref_run.log` log file contains the spooled output of the `syncref_run.sql` script. You can use this log file to compare its contents with your own generated log file.

Running the Demo Scripts and Viewing the Log File

To use the Synchronous Refresh demo, perform the following:

Copy the five scripts and one log file to your working directory, and then run them in SQL*Plus in the following order:

1. Spool the output to a log file as follows:
   - `spool myrun.log`
2. Run the `syncref_run.sql` script as follows:
   - `@syncref_run.sql`
3. Turn spooling off as follows:
   - `spool off`
4. Check whether your generated log file and the `syncref_run.log` file match as follows:
   - `diff myrun.log syncref_run.log`

ORACLE

To use the demo scripts, copy the scripts and the log file to your working directory, and then run them in `SQL*Plus` in the following order:

1. Spool the output by using the `spool` command as follows: `spool myrun.log`. You can give the log file any name that you like.
2. Run the `syncref_run.sql` script by using the @ command, `@syncref_run.sql`. This script does most of the work and calls the other scripts, including the `syncref_setup.sql` script. The `syncref_setup.sql` script creates the default tablespace for the new user and also creates tablespaces for the fact table and the MV partitions. Finally, it creates a new user named `syncref_user`, grants the user some privileges, and echoes the commands in the script to the spooled file.
3. Turn spooling off by using the `spool off` command.
4. After you run the scripts in your environment, you should get the same results in `myrun.log` as in the downloaded `syncref_run.log`. You can check whether the two log files match by using the following command at the operating system prompt: `diff myrun.log syncref_run.log`

Watch the demonstration of using synchronous refresh with staging logs in the `home/oracle/labs/demos` folder.

In the Synchronous Refresh: Part 1 demo, you examine some of the contents of the generated log file, `syncref_run.log`, which contains the commands from the `syncref_run.sql` script, along with the output of such commands. The main focus of this demo is to examine the steps that are required to perform Synchronous Refresh by using only staging logs.

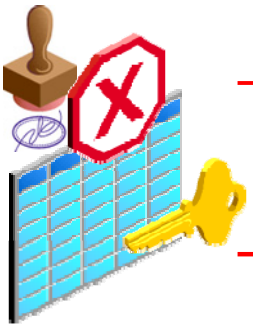
In the Synchronous Refresh: Part 2 demo, you examine the remaining steps that are required to perform synchronous refresh by using staging logs, including populating the staging logs with the change data, preparing the staging logs, checking the status of the staging logs after the preparation, displaying the base tables and MV contents before the refresh operation, performing the prepare refresh operations, displaying the status from the catalog views after each Prepare Refresh, and displaying the status of the objects after Prepare Refresh.

Synchronous Refresh: Unregistration Phase

Execute the
DBMS_SYNC_REFRESH.*UNREGISTER_MVIEWS*
Procedure.

```
EXECUTE
DBMS_SYNC_REFRESH.UNREGISTER_MVIEWS('MV1');
```

1. Unregister the MVs.



Drop the base tables staging logs.

```
DROP MV LOG ON fact;
```

or,

Alter the staging logs to ordinary logs.

```
ALTER MV LOG ON fact
FOR FAST REFRESH;
```

2. Unregister the base tables.

If you choose to stop using synchronous refresh and use other refresh methods, you must unregister the MVs and the base tables in the following order:

1. First, unregister the MVs. For example, to unregister the single MV named `MV1`, execute the following statement:
   `EXECUTE DBMS_SYNC_REFRESH.UNREGISTER_MVIEWS`
2. Next, unregister the base tables. You can unregister the base tables by performing either of the following:
   a. Dropping their staging logs
   b. Altering the staging logs to ordinary logs. If you convert the staging logs to ordinary MV logs with the `ALTER MATERIALIZED LOG … FOR FAST REFRESH` statement, the MV can be maintained with the other fast-refresh methods.

Constraint Violations Detected at EXECUTE_REFRESH Time

- In the synchronous refresh method:
  - Change data is loaded into tables and materialized views at the same time to keep them in sync
  - Foreign-key constraint relationships checking is deferred to the Refresh execution phase
  - Therefore, if you enter inconsistent data that leads to constraint violations, you will see the error only when you run the EXECUTE_REFRESH procedure with an ABORT status
  - You need to identify and fix the problem in the change data, and then begin the sync refresh phase again
- You can find examples of constraint violations in the *Oracle Data Warehousing Guide*.

In the synchronous refresh method, change data is loaded into tables and materialized views at the same time to keep them in sync. In the other refresh methods, change data is loaded into tables first and foreign-key constraint relationships are checked at that time, but in the synchronous refresh method, this checking is deferred to the refresh execution time. Therefore, if you enter inconsistent data that leads to constraint violations, you will see the error only when you run the EXECUTE_REFRESH procedure. To be successful in using synchronous refresh, you should be aware of such errors and learn how to recognize and correct them.

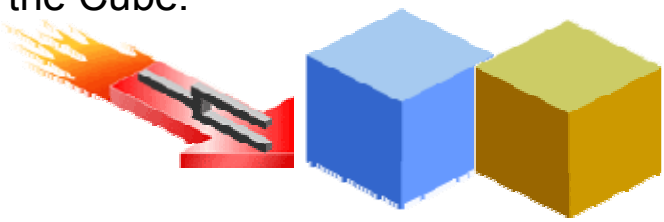
You can find examples of foreign-key constraint violations in the *Oracle Database Data Warehousing Guide 12c Release 1 (12.1).* In such cases, the final status of the EXECUTE_REFRESH job will be ABORT and you need to identify and fix the problem in the change data and begin the synchronous refresh phase again.

Lesson Agenda

- Maintaining Multiple Partitions
- Partial Global Indexes For Partitioned Tables
- Asynchronous Global Index Maintenance
- SQL for Pattern Matching
- Synchronous MV Refresh
- Improving Query Performance Against OLAP Cubes

Cube Join

- Cube Join is an optimization that prevents massive over-fetching from Cubes when they are joined to sources that originate from relational tables.
  - When Cubes are joined to a base table with a Hash Join, the entire Cube is frequently fetched.
  - This includes having to dynamically calculate the aggregate values.
  - The rows that do not match the base table are then discarded.
- Cube Join improves performance to prevent fetching these aggregate values from the Cube.



Oracle Cubes allow SQL users to transparently access advanced analytic calculations. Recent hardware advances, in particular in Oracle Exadata machine, present numerous opportunities for Cube query performance enhancements. This feature leverages these hardware improvements by fully and appropriately utilizing the available hardware.

The Cube Statistics Support feature simplifies the administration of Oracle instances that include Oracle Cubes and online analytical processing (OLAP) dimensions.

Cube Join is an optimization that prevents massive over-fetching from Cubes when they are joined to sources that originate from relational tables.

Cubes contain both leaf-level and aggregate-level data. Fact tables, on the other hand, typically contain only values from a single level. When Cubes are joined to a base table with a Hash Join, the entire Cube is frequently fetched. This includes having to dynamically calculate the aggregate values, only to discard all those rows when they do not match the base table. There is big performance improvement when Cube Join is used to prevent fetching these aggregate values from the Cube.

Query Joins That Can Benefit from Cube Joins

- There are many types of queries that can benefit from Cube Join:
  - A query joining a Cube to a relational table
  - A query joining two or more Cubes
- Cube Join also improves many of the refresh expressions generated for MVs.

There are several types of queries that can benefit from Cube Join:

- Joining a Cube to a relational table, either directly or indirectly via query rewrite to an Analytic Workspace (AW) MV, would see many of these benefits. Cube Join will prevent over-fetching all of these needless values from the Cube.
- Another query type that can benefit from Cube Join is a join between Cubes. If two Cubes are joined, and OLAP Join Pushing cannot eliminate the join, Cube Join can significantly improve the query. Any predicates on the left side Cube will be implicitly applied to the right side Cube. Furthermore, if any dimension values do not exist on the left, such as no sale of shoes in the queried time period, those values will never be looped or fetched from the Cube on the right even if shoes do have data in that Cube.
- Cube Join will also improve many of the refresh expressions generated for MVs. This includes the `MERGE` command that implicitly joins the MV container table to the MV logs and the source tables. These join operations are very slow for some aggregation MVs, particularly when a small fraction of the Cube is actually changed.

Effect on Alternative Joins

During compilation, there are no added instructions when a query is executed in either of the following situations:

- If all the necessary conditions for Cube Join are not met
- If Cube Join does not have the best cost

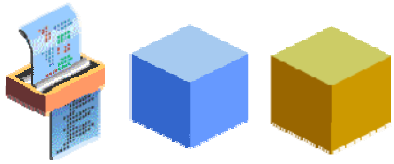
Cube Join has minimal impact on other join types. During compilation, there are no added instructions when a query is executed in either of the following situations:

- If all the necessary conditions for Cube Join are not met
- If Cube Join does not have the best cost

Cube Join Memory Usage

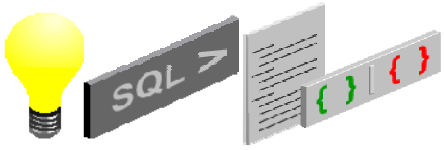
During Cube Join execution:

- Data from the left side of the join will end up being pulled into an LRU-managed UGA pool of memory that OLAP uses for large objects
- This pool starts relatively small and grows to larger sizes as required to maintain a good hit rate
- It is bounded by the free memory within `memory_target`
- If the data is paged out, it goes to a temporary segment
- Cube Join will not engage if the predicted size of the left side grows too big as compared to the available memory



Query Statements Hints and Initialization Parameter File Parameters

- SQL Query Statements Hints
  - Use the `NO_USE_CUBE` hint to disable the Cube Join.
  - Use the `USE_CUBE` hint to force the optimizer to allow Cube Join to be used by a query.
  - These hints are included in the plan outline.
  - You can use the `CUBE_SJ` and `CUBE_AJ` hints to cause Cube anti-joins and Cube semi-joins to be chosen when possible.
- Initialization Parameter File "`init.ora`" Parameters
  - There is a new internal initialization parameter, `_cube_join_enabled` (default setting is `TRUE`).
  - You can set this parameter to `FALSE` to disable Cube Joins.



Sample Explain Plan

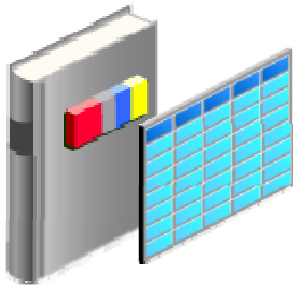
```
-------------------------------------------------------------------------
| Id  | Operation               | Name            | Rows  | Bytes |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |                 |   576 | 22464 |
|   1 |  SORT ORDER BY          |                 |   576 | 22464 |
|*  2 |   CUBE JOIN             |                 |   576 | 22464 |
|*  3 |    TABLE ACCESS FULL    | CHANNEL_DIM_VAR |     1 |    34 |
|*  4 |    MAT_VIEW CUBE ACCESS | CB$UNITS_CUBE   |   576 |  2880 |
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
2 - access("CHANNEL_DIM_VAR"."CHANNEL_ID_VAR"="CB$UNITS_CUBE"."CHANNEL")
3 - filter("CHANNEL_DIM_VAR"."CHANNEL_DSC_FRENCH"='Catalogue')
4 - filter("SYS_GID"=1014)
```


The sample explain plan in the slide shows the new Cube Join row source. The only difference between this plan and the plan from earlier releases is that it reads `CUBE JOIN` where it previously read `HASH JOIN`.

New OLAP Statistics in the V$SQL_PLAN_MONITOR View

The V$SQL_PLAN_MONITOR view contains the following new OLAP statistics:

- OLAP Page Pool Size
- ***OLAP Page Pool Size Change***
- OLAP Page Pool Hits
- OLAP Page Pool Misses
- OLAP AggFunc Calculation Count
- OLAP Rows Read
- OLAP Null Suppressed Rows
- OLAP Rows Failed Filter
- ***OLAP Cube Join Size***
- ***OLAP Cube Join Key Count***
- ***OLAP Cube Join Non-Key Count***



V$SQL_PLAN_MONITOR
data dictionary view

SQL Monitor will record the execution statistics that are unique to the OLAP row sources such as Cube Scan, AW Scan, Cube MV scan, and Cube Join. These statistics will appear in the V$SQL_PLAN_MONITOR view in the OTHERSTAT_n_VALUE column, with their names found in V$SQL_MONITOR_STATNAME. The DBMS_SQLTUNE.REPORT_SQL_MONITOR function is commonly used to view the SQL Monitor results in a Database Active report.

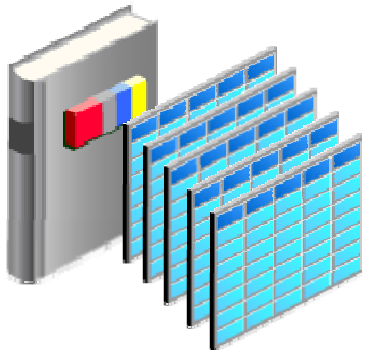
Most of the statistics in this slide are the same as that used by the session and system statistics. The only exceptions are the following unique ones, which are in bold italic font in the slide:

- **OLAP Page Pool Size Change:** This is based on an unsigned integer. It will show 0 (zero) or the magnitude of increase in OLAP Page Pool Size during the execution of the row source. It will be zero if the page pool shrinks.
- **OLAP Cube Join Size:** This represents the size of the OLAP row cache workspace that is allocated to process a Cube Join. The largest size allocated is reported if the row source is restarted during query execution. This is the LRU-managed UGA pool of memory.
- **OLAP Cube Join Key Count:** This is the count of Cube Join scan keys. It is the number of columns joined to the Cube.
- **OLAP Cube Join Non-Key Count:** This is the count of non-join columns of the left-hand side (LHS) relation. It is the number of related columns to the join column that is participating in the Cube Join.

New OLAP Statistics in Views

New OLAP statistics are included in the following views:

- V$SYSSTAT
- V$STATNAME
- V$SESSTAT
- V$SYS_TIME_MODEL
- V$SESS_TIME_MODEL



The V$SYSSTAT view displays system statistics, including OLAP kernel statistics. To find the name of the statistic that is associated with each statistic number (the STATISTIC# column), query the V$STATNAME view. For the NAME column (statistic name), you can get a complete listing of statistic names by querying the V$STATNAME view.

The V$STATNAME view displays decoded statistic names for the statistics shown in the V$SESSTAT and V$SYSSTAT tables. On some platforms, the NAME and CLASS columns contain additional operating system–specific statistics.

The V$SESSTAT view displays user session statistics, including OLAP kernel statistics.

The V$SYS_TIME_MODEL view and the directly related system views will have the OLAP ENGINE CPU TIME and OLAP ENGINE ELAPSED TIME rows.

OLAP Table Function Statistics

- OLAP Row Source Rows Processed
- OLAP Full Limit
- OLAP Fast Limit
- OLAP Custom Member Limit
- OLAP Unique Key Attribute Limit
- OLAP GID Limit
- OLAP INHIER Limit
- OLAP Row Id Limit
- OLAP Level Relation Limit
- OLAP All Member Limit
- OLAP Limit Time

ORACLE

- **OLAP Row Source Rows Processed:** The number of rows processed by the OLAP row source
- **OLAP Full Limit:** The number of times an OLAP table function issues a full limit
- **OLAP Fast Limit:** The number of times an OLAP table function issues a fast limit
- **OLAP Custom Member Limit:** The number of times an OLAP table function issues a custom member limit
- **OLAP Unique Key Attribute Limit:** The number of times an OLAP table function issues a unique key attribute limit
- **OLAP GID Limit:** The number of times an OLAP table function issues a Cube Grouping ID (CGID) limit. Typically, this type of limit occurs for query rewrite transformations that resolve to a Cube-organized MV.
- **OLAP INHIER Limit:** The number of times an OLAP table function issues an in-hierarchy limit. This type of limit can occur when you use Cube dimension hierarchy views.
- **OLAP Row Id Limit:** A counter that is incremented each time an OLAP Table Function issues a row ID limit
- **OLAP Level Relation Limit:** A counter that is incremented each time a level relation limit is issued

- **`OLAP All Member Limit`:** A counter that is incremented each time an OLAP All Member limit is issued
- **`OLAP Limit Time`:** Elapsed time value of the sum of all OLAP Limit operations performed during the last call to the OLAP table function

OLAP Data Loading Statistics

- OLAP Import Rows Pushed
- OLAP Import Rows Loaded
- OLAP Row Load Time

- **OLAP Import Rows Pushed:** A count of OLAP Import Rows Pushed. This count is of interest during Cube build operations and provides the count of rows encountered from a source cursor.
- **OLAP Import Rows Loaded:** A count of OLAP Import Rows Loaded. This count is useful during Cube build operations. This is the count of rows of the source cursor that are actually loaded into an AW. The difference between the pushed and loaded rows gives the count of rejected rows.
- **OLAP Row Load Time:** Elapsed time spent loading rows into an Analytic Workspace (AW) during Cube build and OLAP SQL import operations. This time statistic can be used along with the system time model “OLAP engine elapsed time” statistic to better measure time spent running OLAP engine routines that involve loading data into AWs from a SQL source.

V$STATNAME View: Example

```
SQL> SELECT NAME
FROM V$STATNAME
WHERE NAME LIKE 'OLAP%'
ORDER BY NAME;
  2    3    4
NAME
----------------------------------------------------------------
OLAP Aggregate Function Calc
OLAP Aggregate Function Logical NA
OLAP Aggregate Function Precompute
OLAP Custom Member Limit
OLAP Engine Calls
OLAP Fast Limit
OLAP Full Limit
OLAP GID Limit
OLAP INHIER Limit
OLAP Import Rows Loaded
OLAP Import Rows Pushed
OLAP Limit Time
OLAP Paging Manager Cache Changed Page
OLAP Paging Manager Cache Hit
OLAP Paging Manager Cache Miss
OLAP Paging Manager Cache Write
OLAP Paging Manager New Page
OLAP Paging Manager Pool Size
OLAP Perm LOB Read
OLAP Row Id Limit
OLAP Row Load Time
OLAP Row Source Rows Processed
OLAP Session Cache Hit
OLAP Session Cache Miss
OLAP Temp Segment Read
OLAP Temp Segments
OLAP Unique Key Attribute Limit

27 rows selected.
```


In the slide example, the query against the view `V$STATNAME` shows the output of the new OLAP statistics.

V$SESS_TIME_MODEL and V$SYS_TIME_MODEL Statistics

The V$SESS_TIME_MODEL and V$SYS_TIME_MODEL views now have the following two OLAP statistics:

- OLAP engine elapsed time:
  - Amount of time spent performing OLAP session transactions
- OLAP engine CPU time:
  - Amount of CPU time spent on OLAP session transactions

The V$SESS_TIME_MODEL and V$SYS_TIME_MODEL views now have the following two OLAP statistics:

- **OLAP engine elapsed time:** Amount of time spent performing OLAP session transactions. This includes time spent on database user-level calls, SQL statement execution, and PL/SQL execution within the OLAP transaction.
- **OLAP engine CPU time:** Amount of CPU time spent on OLAP session transactions. This includes the time spent on database user-level calls, SQL statement execution, and PL/SQL execution within the OLAP transaction.

Summary

In this lesson, you should have learned how to:

- Maintain multiple partitions
- Utilize partial global indexes for partitioned tables
- Maintain an asynchronous global index
- Determine SQL for pattern matching
- Perform synchronous materialized view refresh
- Identify ways to improve query performance against OLAP Cubes

ORACLE

In this lesson, you learned how to:

- Add multiple new partitions with the `ADD PARTITION` clause of the `ALTER TABLE` statement
- Merge the contents of two or more partitions or subpartitions into one new partition or subpartition with the `MERGE PARTITIONS` and `MERGE SUBPARTITIONS` clauses of the `ALTER TABLE SQL` statement
- Remove multiple partitions or subpartitions from a range- or list-partitioned table with the `DROP PARTITION` and `DROP SUBPARTITION` clauses of the `ALTER TABLE` statement
- Redistribute the contents of one partition into multiple partitions with the `SPLIT PARTITION` clause of the `ALTER TABLE statement`
- Create local and global indexes on a subset of the partitions of a table, which enables faster `DROP` and `TRUNCATE` partition operations because index maintenance can be delayed to off-peak time
- Use the `DBMS_PART` package to maintain and manage operations on partitioned objects
- Perform maintenance operations on indexes to clean up all global indexes with the automatic scheduler job, `SYS.PMO_DEFERRED_GIDX_MAINT_JOB`
- Recognize patterns in a sequence of rows by using the `MATCH_RECOGNIZE` clause in native SQL, which executes efficiently

- Use the new synchronous refresh `DBMS_SYNC_REFRESH` package to keep your base tables in sync with the materialized views (MVs) that are built on them and to avoid the problem of stale MVs, which impacts performance
- Use Cube Join to improve query performance against `OLAP` Cubes
- Identify the new `OLAP` statistics in the `V$SQL_PLAN_MONITOR` view and several other views