Creating Views

Objectives

After completing this lesson, you should be able to do the following:
• Describe a view
• Create, alter the definition of, and drop a view
• Retrieve data through a view
• Insert, update, and delete data through **a view**
• Create and use an inline view
• Perform top-n analysis

Database Objects

Table	**Basic unit of storage; composed of rows and columns**
View	**Logically represents subsets of data from one or more tables**
Sequence	**Generates primary key values**
Index	**Improves the performance of some queries**
Synonym	**Alternative name for an object**

What Is a View?
You can present logical subsets or combinations of data by creating views of tables. A view is a logical table based on a table or another view. A view contains no data of its own but is like a window through which data from tables can be viewed or changed. The tables on which a view is based are called base tables. The view is stored as a SELECT statement in the data dictionary.

Advantages of Views
• Views restrict access to the data because the view can display selective columns from the table.
• Views can be used to make simple queries to retrieve the results of complicated queries. For example, views can be used to query information from multiple tables without the user knowing how to write a join statement.
• Views provide data independence for ad hoc users and application programs. One view can be used to retrieve data from several tables.
• Views provide groups of users access to data according to their particular criteria.
For more information, see Oracle9i SQL Reference, “CREATE VIEW.”

Simple Views versus Complex Views
There are two classifications for views: simple and complex. The basic difference is related to the DML
(INSERT, UPDATE, and DELETE) operations.
• A simple view is one that:
- Derives data from only one table
- Contains no functions or groups of data
- Can perform DML operations through the view
• A complex view is one that:
- Derives data from many tables
- Contains functions or groups of data
- Does not always allow DML operations through the view

Creating a View
You can create a view by embedding a subquery within the CREATE VIEW statement.
In the syntax:
OR REPLACE re-creates the view if it already exists
FORCE creates the view regardless of whether or not the base tables exist
NOFORCE creates the view only if the base tables exist (This is the default.)
view
is the name of the view
alias
specifies names for the expressions selected by the view's query (The number of aliases must match the number of expressions selected by the view.)
subquery is a complete SELECT statement (You can use aliases for the columns in the SELECT list.)
WITH CHECK OPTION specifies that only rows accessible to the view can be inserted or
updated
constraint is the name assigned to the CHECK OPTION constraint
WITH READ ONLY ensures that no DML operations can be performed on this view

CREATE VIEW salvu50

AS SELECT employee_id ID_NUMBER, last_name NAME, salary*12 ANN_SALARY
FROM employees
WHERE department_id = 50;
View created.
• Select the columns from this view by the given **alias names.**
SELECT * FROM salvu50;

Retrieving Data from a View
You can retrieve data from a view as you would from any table. You can display either the contents of the entire view or just specific rows and columns.

Modifying a View
With the OR REPLACE option, a view can be created even if one exists with this name already, thus replacing the old version of the view for its owner. This means that the view can be altered without dropping, re-creating, and regranting object privileges.
Note: When assigning column aliases in the CREATE VIEW clause, remember that the aliases are listed in the same order as the columns in the subquery.
Performing DML Operations on a View

You can perform DML operations on data through a view if those operations follow certain rules.
You can remove a row from a view unless it contains any of the f ollowing:
• Group functions
• A GROUP BY clause
• The DISTINCT keyword
• The pseudocolumn ROWNUM keyword
You can modify data through a view unless it contains any of the conditions mentioned in the previous slide
or columns defined by expressions: for example, SALARY * 12.

You can add data through a view unless it contains • Group functions • A GROUP BY clause • The DISTINCT keyword • The pseudocolumn ROWNUM keyword
or there are NOT NULL columns, without default values, in the base table that are not selected by the view. All required values must be present in the view. Remember that you are adding values directly into the underlying table through the view.

Using the WITH CHECK OPTION Clause
It is possible to perform referential integrity checks through views. You can also enforce constraints at the database level. The view can be used to protect data integrity, but the use is very limited. The WITH CHECK OPTION clause specifies that INSERTs and UPDATEs performed through the view cannot create rows which the view cannot select, and therefore it allows integrity constraints and data validation checks to be enforced on data being inserted or updated.
If there is an attempt to perform DML operations on rows that the view has not selected, an error is displayed, with the constraint name if that has been specified.

Denying DML Operations
You can ensure that no DML operations occur on your view by creating it with the WITH READ ONLY option. The example in the slide modifies the EMPVU10 view to prevent any DML operations on the view.

Removing a View
You use the DROP VIEW statement to remove a view. The statement removes the view definition from the database. Dropping views has no effect on the tables on which the view was based. Views or other applications based on deleted views become invalid.

Inline Views
An inline view is created by placing a subquery in the FROM clause and giving that subquery an alias. The subquery defines a data source that can be referenced in the main query. In the following example, the inline view b returns the details of all department numbers and the maximum salary for each department from the EMPLOYEES table. The WHERE a.department_id = b.department_id AND a.salary < b.maxsal clause of the main query displays employee names, salaries, department numbers, and maximum salaries for all the employees who earn less than the maximum salary in their department.

```
SELECT  a.last_name, a.salary, a.department_id, b. maxsal
FROM    employees a, (SELECT   department_id, max(salary) maxsal FROM  employees
GROUP BY department_id) b
WHERE   a.department_id = b.department_id  AND     a.salary < b.maxsal;
```

Top-n Analysis
Top-n queries are useful in scenarios where the need is to display only the n top-most or the n bottommost records from a table based on a condition. This result set can be used for further analysis. For example using top-n analysis you can perform the following types of queries:
• The top three earners in the company
• The four most recent recruits in the company
• The top two sales representatives who have sold the maximum number of products
• The top three products that have had the maximum sales in the last six months

Performing Top-n Analysis
Top-n queries use a consistent nested query structure with the elements descr ibed below:
• A subquery or an inline view to generate the sorted list of data. The subquery or the inline view includes the ORDER BY clause to ensure that the ranking is in the desired

order. For results retrieving the largest values, a DESC parameter is needed.
• An outer query to limit the number of rows in the final result set. The outer query includes the following components:
- The ROWNUM pseudocolumn, which assigns a sequential value starting with 1 to each of the
rows returned from the subquery.
- A WHERE clause, which specifies the n rows to be returned. The outer WHERE clause must
use a < or <= operator.

Example of Top-n Analysis

To display the top three earner names and salaries from the EMPLOYEES table.
SELECT ROWNUM as RANK, last_name, salary FROM (SELECT last_name,salary FROM employees ORDER BY salary DESC)
WHERE ROWNUM <= 3;

Other Database Objects

Objectives

After completing this lesson, you should be able to do the following:
• Create, maintain, and use sequences
• Create and maintain indexes
• Create private and public synonyms

What Is a Sequence?
A sequence is a user created database object that can be shared by multiple users to generate unique integers. A typical usage for sequences is to create a primary key value, which must be unique for each row. The sequence is generated and incremented (or decremented) by an internal Oracle routine. This can be a time-saving object because it can reduce the amount of application code needed to write a sequence-generating routine. Sequence numbers are stored and generated independently of tables. Therefore, the same sequence can be used for multiple tables.

Creating a Sequence
Automatically generate sequential numbers by using the CREATE SEQUENCE statement.
In the syntax:
sequence is the name of the sequence generator
INCREMENT BY n specifies the interval between sequence numbers where n is an integer (If this clause is omitted, the sequence increments by 1.)
START WITH n specifies the first sequence number to be generated (If this clause is omitted, the sequence starts with 1.)
MAXVALUE n specifies the maximum value the sequence can generate
NOMAXVALUE specifies a maximum value of 10^27 for an ascending sequence and -1 for a descending sequence (This is the default option.)
MINVALUE n specifies the minimum sequence value
NOMINVALUE specifies a minimum value of 1 for an ascending sequence and -(10^26) for a descending sequence (This is the default option.)
CYCLE | NOCYCLE specifies whether the sequence continues to generate values after reaching its maximum or minimum value (NOCYCLE is the default option.)
CACHE n | NOCACHE specifies how many values the Oracle Server preallocates and

keep in memory (By default, the Oracle Server caches 20 values.)
CREATE SEQUENCE dept_deptid_seq INCREMENT BY 10 START WITH 120 MAXVALUE 9999 NOCACHE NOCYCLE;

Sequence created.

The example creates a sequence named DEPT_DEPTID_SEQ to be used for the DEPARTMENT_ID column of the DEPARTMENTS table. The sequence starts at 120, does not allow caching, and does not cycle.
Do not use the CYCLE option if the sequence is used to generate primary key values, unless you have a reliable mechanism that purges old rows faster than the sequence cycles.
Note: The sequence is not tied to a table. Generally, you should name the sequence after its intended use; however the sequence can be used anywhere, regardless of its name.

Confirming Sequences

Once you have created your sequence, it is documented in the data dictionary. Because a sequence is a database object, you can identify it in the USER_OBJECTS data dictionary table. You can also confirm the settings of the sequence by selecting from the USER_SEQUENCES data dictionary view.

Using a Sequence

After you create your sequence, it generates sequential numbers for use in your tables. Reference the sequence values by using the NEXTVAL and CURRVAL pseudocolumns.

NEXTVAL and CURRVAL Pseudocolumns

The NEXTVAL pseudocolumn is used to extract successive sequence numbers from a specified sequence. You must qualify NEXTVAL with the sequence name. When you reference sequence.NEXTVAL, a new sequence number is generated and the current sequence number is placed in CURRVAL. The CURRVAL pseudocolumn is used to refer to a sequence number that the current user has just generated. NEXTVAL must be used to generate a sequence number in the current user' s session before CURRVAL can be referenced. You must qualify CURRVAL with the sequence name. When sequence.CURRVAL is referenced, the last value returned to that user's process is displayed.

Rules for Using NEXTVAL and CURRVAL

You can use NEXTVAL and CURRVAL in the following contexts:
• The SELECT list of a SELECT statement that is not part of a subquery
• The SELECT list of a subquery in an INSERT statement
• The VALUES clause of an INSERT statement
• The SET clause of an UPDATE statement
You cannot use NEXTVAL and CURRVAL in the following contexts:
• The SELECT list of a view
• A SELECT statement with the DISTINCT keyword
• A SELECT statement with GROUP BY, HAVING, or ORDER BY clauses
• A subquery in a SELECT, DELETE, or UPDATE statement
• The DEFAULT expression in a CREATE TABLE or ALTER TABLE statement

Caching Sequence Values
Cache sequences in memory to provide faster access to those sequence values. The cache is populated the first time you refer to the sequence. Each request for the next sequence value is retrieved from the cached sequence. After the last sequence value is used, the next request for the sequence pulls another cache of sequences into memory.

Gaps in the Sequence
Although sequence generators issue sequential numbers without gaps, this action occurs independent of a commit or rollback. Therefore, if you roll back a statement containing a sequence, the number is lost.Another event that can cause gaps in the sequence is a system crash. If the sequence caches values in the memory, then those values are lost if the system crashes. Because sequences are not tied directly to tables, the same sequence can be used for multiple tables. If you do so, each table can contain gaps in the sequential numbers.

Viewing the Next Available Sequence Value without Incrementing It
If the sequence was created with NOCACHE, it is possible to view the next available sequence value without incrementing it by querying the USER_SEQUENCES table.

Altering a Sequence
If you reach the MAXVALUE limit for your sequence, no additional values from the sequence are allocated and you will receive an error indicating that the sequence exceeds the MAXVALUE. To continue to use the sequence, you can modify it by using the ALTER SEQUENCE statement.
Syntax
ALTER SEQUENCE sequence
[INCREMENT BY n]
[{MAXVALUE n | NOMAXVALUE}]
[{MINVALUE n | NOMINVALUE}]
[{CYCLE | NOCYCLE}]
[{CACHE n | NOCACHE}];
In the syntax:
sequence is the name of the sequence generator

Guidelines for Modifying Sequences
• You must be the owner or have the ALTER privilege for the sequence in order to modify it.
• Only future sequence numbers are affected by the ALTER SEQUENCE statement.
• The START WITH option cannot be changed using ALTER SEQUENCE. The sequence must be dropped and re-created in order to restart the sequence at a different number.
• Some validation is performed. For example, a new MAXVALUE that is less than the current sequence number cannot be imposed.

Removing a Sequence
To remove a sequence from the data dictionary, use the DROP SEQUENCE statement. You must be the owner of the sequence or have the DROP ANY SEQUENCE privilege to remove it.
Syntax
DROP SEQUENCE sequence;
In the syntax:
sequence is the name of the sequence generator
For more information, see Oracle9i SQL Reference, “DROP SEQUENCE.”

More Is Not Always Better
More indexes on a table does not mean faster queries. Each DML operation that is committed on a table
with indexes means that the indexes must be updated. The more indexes you have associated with a table,
the more effort the Oracle server must make to update all the indexes after a DML operation.

When to Create an Index
Therefore, you should create indexes only if:
• The column contains a wide range of values
• The column contains a large number of null values
• One or more columns are frequently used together in a WHERE clause or join condition
• The table is large and most queries are expected to retrieve less than 2 to 4% of the rows
Remember that if you want to enforce uniqueness, you should defi ne a unique constraint in the table
definition. Then a unique index is created automatically.

When Not to Create an Index

It is usually not worth creating an index if:
• The table is small
• The columns are not often used as a condition in **the query**
• Most queries are expected to retrieve more than 2 **to 4% of the rows in the table**
• The table is updated frequently
• The indexed columns are referenced as part of an **expression**

Using SET Operators

Objectives

After completing this lesson, you should be able to do the following:
• Describe SET operators
• Use a SET operator to combine multiple queries into a single **query**
• Control the order of rows returned

The SET Operators
The SET operators combine the results of two or more component queries into one re sult. Queries containing SET operators are called compound queries.

Operator	**Returns**
UNION	All distinct rows selected by either query
UNION ALL	All rows selected by either query, including all duplicates
INTERSECT	All distinct rows selected by both queries
MINUS	All distinct rows that are selected by the first SELECT statement and that are not selected in the second SELECT statement

All SET operators have equal precedence. If a SQL statement contains multiple SET operators, the Oracle server evaluates them from left (top) to right (bottom) if no parentheses explicitly specify another order. You should use parentheses to specify the order of evaluation explicitly in queries that use the INTERSECT operator with other SET operators.

Tables Used in This Lesson

Two tables are used in this lesson. They are the EMPLOYEES table and the JOB_HISTORY table. The EMPLOYEES table stores the employee details. For the human resource records, this table stores a unique identification number and email address for each employee . The details of the employee's job identification number, salary, and manager are also stored. Some of the employees earn a commission in addition to their salary; this information is tracked too. The company organizes the roles of employees into jobs. Some of the employees have been with the company for a long time and have switched to different jobs. This is monitored using the JOB_HISTORY table. When an employee switches jobs, the details of the start date and end date of the former job, the job identification number and department are recorded in the JOB_HISTORY table.

There have been instances in the company of people who have held the same position more than once during their tenure with the company. For example, consider the employee Taylor, who joined the company on 24-MAR-1998. Taylor held the job title SA_REP for the period 24-MAR-98 to 31-DEC-98 and the job title SA_MAN for the period 01-JAN -99 to 31-DEC-99. Taylor moved back into the job title of SA_REP, which is his current job title.

Similarly consider the employee Whalen, who joined the company on 17-SEP-1987. Whalen held the job title AD_ASST for the period 17-SEP-87 to 17-JUN-93 and the job title AC_ACCOUNT for the period 01-JUL-94 to 31-DEC-98. Taylor moved back into the job title of AD_ASST, which is his current job title.

The UNION SET Operator

The UNION operator returns all rows selected by either query. Use the UNION operator to return all rows from multiple tables and eliminate any duplicate rows.

Guidelines

• The number of columns and the data types of the columns being selected must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.
• UNION operates over all of the columns being selected.
• NULL values are not ignored during duplicate checking.
• The IN operator has a higher precedence than the UNION operator.
• By default, the output is sorted in ascending order of the first column of the SELECT clause.

Display the current and previous job details of all employees. Display each employee only once.

```
SELECT employee_id, job_id FROM   employees
UNION
SELECT employee_id, job_id FROM   job_history;
```


Using the UNION SET Operator

The UNION operator eliminates any duplicate records. If there are records that occur both in the EMPLOYEES and the JOB_HISTORY tables and are identical, the records will be displayed only once.

The UNION ALL Operator
Use the UNION ALL operator to return all rows from multiple queries.
Guidelines
• Unlike UNION, duplicate rows are not eliminated and the output is not sorted by default.
• The DISTINCT keyword cannot be used.

Note: With the exception of the above, the guidelines for UNION and UNION ALL are the same.
The INTERSECT Operator
Use the INTERSECT operator to return all rows common to multiple queries.
Guidelines
• The number of columns and the data types of the columns being selected by the SELECT statements in the queries must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.
• Reversing the order of the intersected tables does not alter the result.
• INTERSECT does not ignore NULL values.

Display the employee IDs and job IDs of employees who are currently in a job title that they have held once before during their tenure with the company
SELECT employee_id, job_id FROM employees
INTERSECT

SELECT employee_id, job_id FROM job_history;
In the example, the query returns only the records that have the same values in the selected columns in both tables.

The MINUS Operator
Use the MINUS operator to return rows returned by the first query that are not present in the second query (the first SELECT statement MINUS the second SELECT statement).
Guidelines
• The number of columns and the data types of the columns being selected by the SELECT statements in the queries must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.
• All of the columns in the WHERE clause must be in the SELECT clause for the MINUS operator to work.

Display the employee IDs of those employees who have not changed their jobs even once.
SELECT employee_id FROM employees
MINUS
SELECT employee_id FROM job_history;
In the example , the employee IDs in the JOB_HISTORY table are subtracted from those in the EMPLOYEES table. The results set displays the employees remaining after the subtraction; they are represented by rows that exist in the EMPLOYEES table but do not exist in the JOB_HISTORY table. These are the records of the employees who have not changed their jobs even once.

SET Operator Guidelines
• The expressions in the select lists of the queries must match in number and datatype. Queries that use UNION, UNION ALL, INTERSECT, and MINUS SET operators in their WHERE clause must have the same number and type of columns in their SELECT list.
• The ORDER BY clause:

- Can appear only at the very end of the statement
- Will accept the column name, an alias, or the positional notation
• The column name or alias, if used in an ORDER BY clause, must be from the first SELECT list.
• SET operators can be used in subqueries.

The Oracle Server and SET Operators
When a query uses SET operators, the Oracle Server eliminates duplicate rows automatically except in the case of the UNION ALL operator. The column names in the output are decided by the column list in the first SELECT statement. By default, the output is sorted in ascending order of the first column of the SELECT clause.
The corresponding expressions in the select lists of the component queries of a compound query must match in number and datatype. If component queries select character data, the data type of the return values are determined as follows:
• If both queries select values of datatype CHAR, the returned values have datatype CHAR.
• If either or both of the queries select values of datatype VARCHAR2, the returned values
have datatype VARCHAR2.

Matching the SELECT Statements

Using the UNION operator, display the department ID, location, and hire date for all employees.
SELECT department_id, TO_NUMBER(null) location, hire_date FROM employees
UNION
SELECT department_id, location_id, TO_DATE(null) FROM departments;
Matching the SELECT Statements

As the expressions in the select lists of the queries must match in number , you can use dummy columns and the data type conversion functions to comply with this rule. In the slide, the name location is given as the dummy column heading. The TO_NUMBER function is used in the first query to match the NUMBER data type of the LOCATION_ID column retrieved by the second query. Similarly, the TO_DATE function in the second query is used to match the DATE datatype of the HIRE_DATE column retrieved by the second query.

Controlling the Order of Rows
By default, the output is sorted in ascending order on the first column. You can use the ORDER BY clause to change this.

Using ORDER BY to Order Rows
The ORDER BY clause can be used only once in a compound query. If used, the ORDER BY clause must be placed at the end of the query. The ORDER BY clause accepts the column name, an alias, or the positional notation.
Note: Consider a compound query where the UNION SET operator is used more than once. In this case, the ORDER BY clause can use only positions rather than explicit expressions.

Advanced Subqueries

Objectives

After completing this lesson, you should be able to do the following:
• Write a multiple-column subquery
• Describe and explain the behavior of subqueries when **null values are retrieved**
• Write a subquery in a FROM clause
• Use scalar subqueries in SQL
• Describe the types of problems that can be solved with **correlated subqueries**
• Write correlated subqueries
• Update and delete rows using correlated subqueries
• Use the EXISTS and NOT EXISTS operators
• Use the WITH clause

What Is a Subquery?
A subquery is a SELECT statement that is embedded in a clause of another SQL statement, called the parent statement.The subquery (inner query) returns a value that is used by the parent statement. Using a nested subquery is equivalent to performing two sequential queries and using the result of the inner query as the search value in the outer query (main query).
Subqueries can be used for the following purposes:
• To provide values for conditions in WHERE, HAVING, and START WITH clauses of SELECT statements
• To define the set of rows to be inserted into the target table of an INSERT or CREATE TABLE statement
• To define the set of rows to be included in a view or snapshot in a CREATE VIEW or CREATE SNAPSHOT statement
• To define one or more values to be assigned to existing rows in an UPDATE statement
• To define a table to be operated on by a containing query. (You do this by placing the subquery in the FROM clause. This can be done in INSERT, UPDATE, and DELETE statements as well.)

Note: A subquery is evaluated once for the entire parent statement.
Subqueries
You can build powerful statements out of simple ones by using subqueries. Subqueries can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself or some other table. Subqueries are very useful for writing SQL statements that need values based on one or more unknown conditional values.

Multiple-Column Subqueries
So far you have written single-row subqueries and multiple-row subqueries where only one column is returned by the inner SELECT statement and this is used to evaluate the expression in the parent select statement. If you want to compare two or more columns, you must write a compound WHERE clause using logical operators. Using multiple-column subqueries, you can combine duplicate WHERE conditions into a single WHERE clause.

Syntax

```
SELECT column, column , ...
FROM  table
WHERE (column, column, ...) IN
(SELECT column, column, ...
FROM   table
WHERE  condition);
```

Pairwise Versus Nonpairwise Comparisons

Column comparisons in a multiple-column subquery can be pairwise comparisons or nonpairwise comparisons.

In the example, a pairwise comparison was executed in the WHERE clause. Each candidate row in the SELECT statement must have both the same MANAGER_ID column and the DEPARTMENT_ID as the employee with the EMPLOYEE_ID 178 or 174.

A multiple-column subquery can also be a nonpairwise comparison. In a nonpairwise comparison, each of the columns from the WHERE clause of the parent SELECT statement are individually compared to multiple values retrieved by the inner select statement. The individual columns can match any of the values retrieved by the inner select statement. But collectively, all the multiple conditions of the main SELECT statement must be satisfied for the row to be displayed.

Display the details of the employees who are managed by the same manager and work in the same department
as the employees with EMPLOYEE_ID 178 or 174.

```
SELECT employee_id, manager_id, department_id FROM employees
WHERE  (manager_id, department_id) IN (SELECT manager_id, department_id
FROM  employees WHERE  employee_id IN (178,174)) AND employee_id NOT IN
(178,174);
```

Pairwise Comparison Subquery

The example in the slide is that of a multiple-column subquery because the subquery returns more than one column. It compares the values in the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPLOYEES table with the values in the MANAGER_ID column and the DEPARTMENT_ID
column for the employees with the EMPLOYEE_ID 178 or 174.First, the subquery to retrieve the MANAGER_ID and DEPARTMENT_ID values for the employees with the EMPLOYEE_ID 178 or 174 is executed. These values are compared with the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPLOYEES table. If the values match, the row is displayed. In the output, the records of the employees with the EMPLOYEE_ID 178 or 174 will not be displayed.

Display the details of the employees who are managed by the same manager as the employees with EMPLOYEE_ID 174 or 141 and work in the same department as the employees with EMPLOYEE_ID 174 or 141.

```
SELECT  employee_id, manager_id, department_id FROM   employees
WHERE  manager_id IN (SELECT manager_id FROM  employees WHERE
employee_id IN (174,141))
AND  department_id IN (SELECT department_id FROM  employees WHERE
employee_id IN (174,141)) AND employee_id NOT IN(174,141);
```

Nonpairwise Comparison Subquery

The example shows a nonpairwise comparison of the columns. It displays the EMPLOYEE_ID, MANAGER_ID, and DEPARTMENT_ID of any employee whose manager ID matches any of the manager IDs of employees whose employee IDs are either 174 or 141 and DEPARTMENT_ID match any of the department IDs of employees whose employee IDs are either 174 or 141. First, the subquery to retrieve the MANAGER_ID values for the employees with the EMPLOYEE_ID 174 or 141 is executed. Similarly, the second subquery to retrieve the DEPARTMENT_ID values for the employees with the EMPLOYEE_ID 174 or 141 is executed. The retrieved values of the MANAGER_ID and DEPARTMENT_ID columns are compared with the MANAGER_ID and DEPARTMENT_ID column for each row in the EMPLOYEES table. If the MANAGER_ID column of the row in the EMPLOYEES table matches with any of the values of the MANAGER_ID retrieved by the inner subquery and if the DEPARTMENT_ID column of the row in the EMPLOYEES table matches with any of the values of the DEPARTMENT_ID retrieved by the second subquery, the record is displayed.

Using a Subquery in the FROM Clause

You can use a subquery in the FROM clause of a SELECT statement, which is very similar to how views are used. A subquery in the FROM clause of a SELECT statement is also called an inline view. A subquery in the FROM clause of a SELECT statement defines a data source for that particular SELECT statement, and only that SELECT statement.

Scalar Subqueries in SQL

A subquery that returns exactly one column value from one row is also referred to as a scalar subquery. Multiple-column subqueries written to compare two or more columns, using a compound WHERE clause and logical operators, do not qualify as scalar subqueries.

The value of the scalar subquery expression is the value of the select list item of the subquery. If the subquery returns 0 rows, the value of the scalar subquery expression is NULL. If the subquery returns more than one row, the Oracle Server returns an error. The Oracle Server has always supported the usage of a scalar subquery in a SELECT statement. The usage of scalar subqueries has been enhanced in Oracle9i.

You can now use scalar subqueries in:

- Condition and expression part of DECODE and CASE
- All clauses of SELECT except GROUP BY
- In the left-hand side of the operator in the SET clause and WHERE clause of UPDATE statement

However, scalar subqueries are not valid expressions in the following places:

- As default values for columns and hash expressions for clusters
- In the RETURNING clause of DML statements
- As the basis of a function-based index
- In GROUP BY clauses , CHECK constraints , WHEN conditions
- HAVING clauses
- In START WITH and CONNECT BY clauses
- In statements that are unrelated to queries, such as CREATE PROFILE

Scalar Subqueries in CASE Expressions

```
SELECT employee_id, last_name, (CASE  20 WHEN department_id = (SELECT
department_id FROM departments WHERE location_id = 1800) THEN 'Canada'
ELSE 'USA' END) location FROM employees;
```

Scalar Subqueries in ORDER BY Clause

```
SELECT employee_id, last_name FROM employees e
ORDER BY (SELECT department_name FROM departments d
WHERE e.department_id = d.department_id);
```

Scalar Subqueries: Examples

The first example in the slide demonstrates that scalar subqueries can be used in CASE expressions. The inner query returns the value 20, which is the department ID of the department whose location ID is 1800. The CASE expression in the outer query uses the result of the inner query to display the employee ID, last names, and a value of Canada or USA, depending on whether the department ID of the record retrieved by the outer query is 20 or not.

Correlated Subqueries

The Oracle Server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a SELECT, UPDATE, or DELETE statement.

Nested Subqueries Versus Correlated Subqueries

With a normal nested subquery, the inner SELECT query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each candidate row considered by the outer query. In other words, the inner query is driven by the outer query.

Nested Subquery Execution

- The inner query executes first and finds a value.
- The outer query executes once, using the value from the inner query.

Correlated Subquery Execution

- Get a candidate row (fetched by the outer query).
- Execute the inner query using the value of the candidate row.
- Use the values resulting from the inner query to qualify or disqualify the candidate.
- Repeat until no candidate row remains.

A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. In other words, you use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement. The Oracle Server performs a correlated subquery when the subquery references a column from a table in the parent query.

Note: You can use the ANY and ALL operators in a correlated subquery.

Find all employees who earn more than the average salary in their department.

SELECT last_name, salary, department_id FROM employees outer
WHERE salary > (SELECT AVG(salary) FROM employees WHERE department_id = outer.department_id);

The example determines which employees earn more than the average salary of their department. In this case, the correlated subquery specifically c omputes the average salary for each department.Because both the outer query and inner query use the EMPLOYEES table in the FROM clause, an alias is given to EMPLOYEES in the outer SELECT statement, for clarity. Not only does the alias make the entire SELECT statement more readable, but without the alias the query would not work properly, because the inner statement would not be able to distinguish the inner table column from the outer table column.

The EXISTS Operator

With nesting SELECT statements, all logical operators are valid. In addition, you can use the EXISTS operator. This operator is frequently used with correlated subqueries to test whether a value retrieved by the outer query exists in the results set of the values retrieve d by the inner query. If the subquery returns
at least one row, the operator returns TRUE. If the value does not exist, it returns FALSE. Accordingly, NOT EXISTS tests whether a value retrieved by the outer query is not a part of the results set of the values retrieved by the inner query.

Find employees who have at least one person reporting to them.

SELECT employee_id, last_name, job_id, department_id FROM employees outer
WHERE EXISTS (SELECT 'X' FROM employees WHERE manager_id = outer.employee_id);

The EXISTS operator ensures that the search in the inner query does not continue when at least one match is found for the manager and employee number by the condition: WHERE manager_id = outer.employee_id.
Note that the inner SELECT query does not need to return a specific value, so a constant can be selected. **From a performance standpoint, it is faster to select a constant than a column. Note: Having EMPLOYEE_ID in the SELECT clause of the inner query causes a table scan for that column. Replacing it with the literal X, or any constant, improves performance. This is more efficient than using the IN operator.**
A IN construct can be used as an alternative for a EXISTS operator, as shown in the following example:

SELECT employee_id,last_name,job_id,department_id FROM employees
WHERE employee_id IN (SELECT manager_id FROM employees WHERE manager_id IS NOT NULL);

Find all departments that do not have any employees.

SELECT department_id, department_name FROM departments d
WHERE NOT EXISTS (SELECT 'X' FROM employees WHERE department_id = d.department_id);
Using the NOT EXISTS Operator

Alternative Solution
A NOT IN construct can be used as an alternative for a NOT EXISTS operator, as shown in the following example.
SELECT department_id, department_name FROM departments
WHERE department_id NOT IN (SELECT department_id FROM employees);
However, NOT IN evaluates to FALSE if any member of the set is a NULL value. Therefore, your query will not return any rows even if there are rows in the departmen ts table that satisfy the WHERE condition.

UPDATE table1 alias1
SET column = (SELECT expression FROM table2 alias2 WHERE alias1.column = *alias2.column);*

Use a correlated subquery to update rows in one table based on rows from another table.

Correlated UPDATE
In the case of the UPDATE statement, you can use a correlated subquery to update rows in one table based on rows from another table.

Correlated UPDATE

• Denormalize the EMPLOYEES table by adding a column to store the department name.
• Populate the table by using a correlated update.

```
ALTER TABLE employees
ADD(department_name VARCHAR2(14));
UPDATE employees e
SET department_name = (SELECT department_name FROM departments d WHERE
e.department_id = d.department_id);
```

DELETE FROM table1 alias1 WHERE column operator
(SELECT expression FROM table2 alias2 WHERE alias1.column = alias2.column);

Use a correlated subquery to delete rows in one table based on rows from another table.

Correlated DELETE
In the case of a DELETE statement, you can use a correlated subquery to delete only those rows that also exist in another table. If you decide that you will maintain only the last four job history records in the JOB_HISTORY table, then when an employee transfers to a fifth job, you delete the oldest JOB_HISTORY row by looking up the JOB_HISTORY table for the MIN(START_DATE)for the employee. The following code

illustrates how the preceding operation can be performed using a correlated DELETE:
DELETE FROM job_history JH
WHERE employee_id = (SELECT employee_id FROM employees E WHERE JH.employee_id = E.employee_id AND START_DATE =
(SELECT MIN(start_date) FROM job_history JH WHERE JH.employee_id = E.employee_id) AND 5 > (SELECT COUNT(*)
FROM job_history JH WHERE JH.employee_id = E.employee_id
GROUP BY EMPLOYEE_ID
HAVING COUNT(*) >= 4));

The WITH clause
Using the WITH clause, you can define a query block before using it in a query. The WITH clause (formally known as subquery_factoring_clause) enables you to reuse the same query block in a SELECT statement when it occurs more than once within a complex query. This is particularly useful when a query has many references to the same query block and there are joins and aggregations. Using the WITH clause, you can reuse the same query when it is high cost to evaluate the query block and it occurs more than once within a complex query. Using the WITH clause, the Oracle Server retrieves the results of a query block and stores it in the user's temporary tablespace. This can improve performance.

WITH Clause Benefits
• Makes the query easy to read
• Evaluates a clause only once, even if it appears multiple times in the query, thereby enhancing performance

Using the WITH clause, write a query to display the department name and total salaries for those departments whose total salary is greater than the average salary across departments.

WITH Clause: Example
The problem would require the following intermediate calculations:
1. Calculate the total salary for every department, and store the result using a WITH clause.
2. Calculate the average salary across departments, and store the result using a WITH clause.
3. Compare the total salary calculated in the first step with the a verage salary calculated in the second step. If the total salary for a particular department is greater than the average salary across departments, display the department name and the total salary for that department.

WITH dept_costs AS (SELECT department_name, SUM(salary) AS dept_total
FROM employees, departments
WHERE employees.department_id = departments.department_id
GROUP BY department_name),
AS
avg_cost (SELECT SUM(dept_total)/COUNT(*) AS dept_avg dept_costs FROM)
SELECT * FROM WHERE dept_total > (SELECT FROM)
ORDER BY department_name;
dept_costs
dept_avg
WITH Clause: Example

The SQL code in the slide is an example of a situation in which you can improve performance and write SQL more simply by using the WITH clause. The query creates the query names DEPT_COSTS and AVG_COST and then uses them in the body of the main query. Internally, the WITH clause is resolved either as an in-line view or a temporary table. The optimizer chooses the appropriate resolution depending on the cost or benefit of temporarily storing the results of the WITH clause.

Note: A subquery in the FROM clause of a SELECT statement is also called an in-line view.

The WITH Clause Usage Notes

• It is used only with SELECT statements.

• A query name is visible to all WITH element query blocks (including their subquery blocks) defined after it and the main query block itself (including its subquery blocks).

• When the query name is the same as an existing table name, the parser searches from the inside out, the query block name takes precedence over the table name.

• The WITH clause can hold more than one query. Each query is then separated by a comma.

Hierarchical Retrieval

Objectives

After completing this lesson, you should be able to do the following:

• Interpret the concept of a hierarchical query

• Create a tree-structured report

• Format hierarchical data

• Exclude branches from the tree structure

Sample Data from the EMPLOYEES Table

Using hierarchical queries, you can retrieve data based on a natural hierarchical relationship between rows in a table. A relational database does not store records in a hierarchical way. However, where a hierarchical relationship exists between the rows of a single table, a process called tree walking enables the hierarchy to
be constructed. A hierarchical query is a method of reporting, in order, the branches of a tree. Imagine a family tree with the eldest members of the family found close to the base or trunk of the tree and the youngest members representing branches of the tree. Branches can have their own branches, and so on. A hierarchical query is possible when a relationship exists betw een rows in a table. For example, in the slide, you see that employees with the job IDs of AD_VP, ST_MAN, SA_MAN, and MK_MAN report directly to the president of the company. We know this because the MANAGER_ID column of these records contain the employee ID 100, which belongs to the president (AD_PRES).

Note: Hierarchical trees are used in various fields such as human gene alogy (family trees), livestock (breeding purposes), corporate management (management hierarchies), manufacturing (product assembly), evolutionary research (species development), and scientific research.

Natural Tree Structure
The EMPLOYEES table has a tree structure representing the management reporting line. The hierarchy can be created by looking at the relationship between equivalent values in the EMPLOYEE_ID and MANAGER_ID columns. This relationship can be exploited by joining the table to itself. The MANAGER_ID column contains the employee number of the employee's manager.The parent-child relationship of a tree structure enables you to control:
• The direction in which the hierarchy is walked
• The starting point inside the hierarchy

Hierarchical Queries

SELECT [LEVEL], column, expr...
FROM table
[WHERE condition(s)]
[START WITH condition(s)]
[CONNECT BY PRIOR condition(s)];

WHERE condition:

expr comparison_operator expr
Keywords and Clauses
Hierarchical queries can be identified by the presence of the CONNECT BY and START WITH clauses.
In the syntax:

SELECT Is the standard SELECT clause.
LEVEL For each row returned by a hierarchical query, the LEVEL pseudocolumn returns 1 for a root row, 2 for a child of a root, and so on.
FROM table Specifies the table, view, or snapshot containing the columns. You can select from only one table.
WHERE Restricts the rows returned by the query without affecting other rows of the hierarchy.

condition Is a comparison with expressions.
START WITH Specifies the root rows of the hierarchy (where to start). This clause is required for a true hierarchical query.
CONNECT BY Specifies the columns in which the relationship between parent and child PRIOR rows exist. This clause is required for a hierarchical query.
The SELECT statement cannot contain a join or query from a view that contains a join.

Walking the Tree
The row or rows to be used as the root of the tree are determined by the START WITH clause. The START WITH clause can be used in conjunction with any valid condition.

Examples
Using the EMPLOYEES table, start with King, the president of the company.
... START WITH manager_id IS NULL
Using the EMPLOYEES table, start with employee Kochhar. A START WITH condition

can contain a
subquery.
... START WITH employee_id = (SELECT employee_id FROM employees WHERE last_name = 'Kochhar')
If the START WITH clause is omitted, the tree walk is started with all of the rows in the table as root rows. If a WHERE clause is used, the walk is started with all the rows that satisfy the WHERE condition. This no longer reflects a true hierarchy.
The direction of the query, whether it is from parent to child or from child to parent, is determined by the CONNECT BY PRIOR column placement. The PRIOR operator refers to the parent row. To find the children of a parent row, the Oracle Server evaluates the PRIOR expression for the parent row and the other expressions for each row in the table. Rows for which the condition is true are the children of the parent. The Oracle Server always selects children by evaluating the CONNECT BY condition with respect to a current parent row.

Examples
Walk from the top down using the EMPLOYEES table. Define a hierarchical relationship in which the EMPLOYEE_ID value of the parent row is equal to the MANAGER_ID value of the child row.
... CONNECT BY PRIOR employee_id = manager_id
Walk from the bottom up using the EMPLOYEES table.
... CONNECT BY PRIOR manager_id = employee_id
The PRIOR operator does not necessarily need to be coded immediately following the CONNECT BY Thus, the following CONNECT BY PRIOR clause gives the same result as the one in the preceding example.
... CONNECT BY employee_id = PRIOR manager_id

Note: The CONNECT BY clause cannot contain a subquery.
SELECT employee_id, last_name, job_id, manager_id FROM employees
START WITH employee_id = 101
CONNECT BY PRIOR manager_id = employee_id;
In the following example, EMPLOYEE_ID values are evaluated for the parent row and MANAGER_ID, and SALARY values are evaluated for the child rows. The PRIOR operator applies only to the EMPLOYEE_ID value.
... CONNECT BY PRIOR employee_id = manager_id AND salary > 15000;
To qualify as a child row, a row must have a MANAGER_ID value equal to the EMPLOYEE_ID value of the parent row and must have a SALARY value greater than $15,000.

Ranking Rows with the LEVEL Pseudocolumn
You can explicitly show the rank or level of a row in the hierarchy by using the LEVEL pseudocolumn. This will make your report more readable. The forks where one or more branches split away from a larger branch are called nodes, and the very end of a branch is called a leaf, or leaf node.
The LEVEL Pseudocolumn Value Level
1 A root node
2 A child of a root node
3 A child of a child, and so on
Note: A root node is the highest node within an inverted tree. A child node is any nonroot node. A parent node is any node that has children. A leaf node is any node without children. The number of levels returned by a hierarchical query may be limited by available user memory.

Formatting Hierarchical Reports Using LEVEL and LPAD

Create a report displaying company management levels, beginning with the highest level and indenting

each of the following levels.

```
COLUMN org_chart FORMAT A12
SELECT LPAD(last_name, LENGTH(last_name)+(LEVEL*2) -2,'_') AS org_chart
FROM   employees
START WITH last_name='King'
CONNECT BY PRIOR employee_id=manager_id
```

Formatting Hierarchical Reports Using LEVEL

The nodes in a tree are assigned level numbers from the root. Use the LPAD function in conjunction with the pseudocolumn LEVEL to display a hierarchical report as an indented tree.In the example • LPAD(ciar1,n [,char2]) returns char1, left -padded to length n with the sequence of characters in char2. The argument n is the total length of the return value as it is displayed on screen.

• LPAD(last_name, LENGTH(last_name)+(LEVEL*2) -2,'_')defines the display format.

• char1 is the LAST_NAME , n the total length of the return value, is length of the LAST_NAME +(LEVEL*2)-2 ,and char2 is '_'.

In other words, this tells SQL to take the LAST_NAME and left-pad it with the '_' character till the length of the resultant string is equal to the value determined by LENGTH(last_name)+(LEVEL*2) - 2.

For King, LEVEL = 1. Hence, (2 * 1) - 2 = 2 - 2 = 0. So King does not get padded with any '_' character and is displayed in column 1.

For Kochhar, LEVEL = 2. Hence, (2 * 2) - 2 = 4 - 2 = 2 . So Kochhar gets padded with 2 '_' characters and is displayed indented.

The rest of the records in the EMPLOYEES table are displayed similarly.

ORG_CHART
King
__Kochhar
____Whalen
____Higgins
______Gietz
__De Haan
____Hunold
______Ernst
______Lorent z
__Mourgos
____Rajs
____Davies
____Matos
____Vargas
__Zlotkey
____Abel
____Taylor
____Grant
__Hartstein
____Fay

20 rows selected.

Pruning Branches

You can use the WHERE and CONNECT BY clauses to prune the tree; that is, to control which nodes or rows are displayed. The predicate you use acts as a Boolean condition.

Examples

Starting at the root, walk from the top down, and eliminate employee Higgins in the result, but process the child rows.

```
SELECT  department_id, employee_id,last_name, job_id, salary
FROM    employees
WHERE   last_name  != 'Higgins'
START   WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id;
```

Starting at the root, walk from the top down, and eliminate employee Higgins and all child rows.

```
SELECT  department_id, employee_id,last_name, job_id, salary
FROM    employees
START WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id
AND last_name != 'Higgins';
```

Oracle 9i Extensions to DML and DDL Statements

Objectives

After completing this lesson, you should be able to do the following:
• Describe the features of multitable inserts
• Use the following types of multitable inserts
- Unconditional INSERT
- Pivoting INSERT
- Conditional ALL INSERT
- Conditional FIRST INSERT
• Create and use external tables
• Name the index at the time of creating a primary **key constraint**

Overview of Multitable INSERT Statements
In a multitable INSERT statement, you insert computed rows derived from the rows returned from the evaluation of a subquery into one or more tables.
Multitable INSERT statements can play a very useful role in a data warehouse scenario. You need to load your data warehouse regularly so that it can serve its purpose of facilitating business analysis. To do this, data from one or more operational systems needs to be extracted and copied into the warehouse. The process of extracting data from the source system and bringing it into the data warehouse is commonly called ETL, which stands for extraction, transformation, and loading.During extraction, the desired data has to be identified and extracted from many different sources, such as database systems and applications. After extraction, the data has to be physically transported to the target system or an intermediate system for further processing. Depending on the chosen way of transportation, some transformations can be done during this process. For example, a SQL statement that directly accesses a remote target through a gateway can concatenate two columns as part of the SELECT statement. Once data is loaded into an Oracle9i, database, data transformations can be executed using SQL operations. With Oracle9i multitable INSERT statements is one of the techniques for implementing SQL data transformations.

Multitable INSERTS statement offer the benefits of the INSERT ... SELECT statement when multiple tables are involved as targets. Using functionality prior to Oracle9i, you had to deal with n independent INSERT ... SELECT statements, thus processing the same source data n times and increasing the transformation workload n times. As with the existing INSERT ... SELECT statement, the new statement can be parallelized and used with the direct-load mechanism for faster performance.
Each record from any input stream, such as a nonrelational database table, can now be converted into multiple records for more relational database table environment. To implement this functionality before Oracle9i, you had to write multiple INSERT statements.

Types of Multitable INSERT Statements
Oracle 9i introduces the following types of multitable INSERT statements:
• Unconditional INSERT
• Conditional ALL INSERT
• Conditional FIRST INSERT
• Pivoting INSERT
You use different clauses to indicate the type of INSERT to be executed.

Multitable INSERT Statements

Syntax

INSERT [ALL] [conditional_insert_clause]
[insert_into_clause values_clause] (subquery)

conditional_insert_clause

[ALL] [FIRST]
[WHEN condition THEN] [insert_into_clause values_clause]
[ELSE] [insert_into_clause values_clause]

Unconditional INSERT: ALL into_clause
Specify ALL followed by multiple insert_into_clauses to perform an unconditional multitable insert. The Oracle Server executes each insert_into_clause once for each row returned by the
subquery.

Conditional INSERT: conditional_insert_clause
Specify the conditional_insert_clause to perform a conditional multitable insert. The Oracle server filters each insert_into_clause through the corresponding WHEN condition, which determines whether that insert_into_clause is executed. A single multitable insert statement can contain up to 127 WHEN clauses.

Conditional INSERT: ALL
If you specify ALL, the Oracle server evaluates each WHEN clause regardless of the results of the evaluation of any other WHEN clause. For each WHEN clause whose condition evaluates to true, the Oracle server executes the corresponding INTO clause list.

Conditional FIRST: INSERT
If you specify FIRST, the Oracle Server evaluates each WHEN clause in the order in which it appears in the statement. If the first WHEN clause evaluates to true, the Oracle Server executes the corresponding INTO clause and skips subsequent WHEN clauses for the given row.

Conditional INSERT: ELSE Clause
For a given row, if no WHEN clause evaluates to true:
• If you have specified an ELSE, clause the Oracle Server executes the INTO clause list associated with the ELSE clause.
• If you did not specify an ELSE clause, the Oracle Server takes no action for that row.

Restrictions on Multitable INSERT Statements
• You can perform multitable inserts only on tables, not on views or materialized views.
• You cannot perform a multitable insert into a remote table.
• You cannot specify a table collection expression when performing a multitable insert.
• In a multitable insert, all of the insert_into_clauses cannot combine to specify more than 999 target columns.

• Select the EMPLOYEE_ID, HIRE_DATE, SALARY, and **MANAGER_ID values from the EMPLOYEES table for**
those employees whose EMPLOYEE_ID is greater than 200.

• Insert these values into the SAL_HISTORY and **MGR_HISTORY tables using a multitable INSERT.**

INSERT ALL
INTO sal_history VALUES(EMPID,HIREDATE,SAL)
INTO mgr_history VALUES(EMPID,MGR,SAL)
SELECT employee_id , hire_date ,
salary , manager_id
EMPID
HIREDATE
MGR
SAL
FROM employees
WHERE employee_id > 200;
Unconditional INSERT ALL
The example inserts rows into both the SAL_HISTORY and the MGR_HISTORY tables. The SELECT statement retrieves the details of employee ID, hire date, salary, and manager ID of those employees whose employee ID is greater than 200 from the EMPLOYEES table. The details of the employee ID, hire date, and salary are inserted into the SAL_HISTORY table. The details of employee ID, manager ID and salary are inserted into the MGR_HISTORY table.
This INSERT statement is referred to as an unconditional INSERT, as no further restriction is applied to the rows that are retrieved by the SELECT statement. All the rows retrieved by the SELECT statement are inserted into the two tables, SAL_HISTORY and MGR_HISTORY. The VALUES clause in the INSERT statements specifies the columns from the SELECT statement that have to be inserted into each of the tables. Each row returned by the SELECT statement results in two inserts, one for the SAL_HISTORY table and one for the MGR_HISTORY table.

• Select the EMPLOYEE_ID, HIRE_DATE, SALARY and **MANAGER_ID values from the EMPLOYEES table for those employees whose EMPLOYEE_ID is greater than 200.**

• If the SALARY is greater than $10,000, insert these **values into the SAL_HISTORY table using a conditional multitable INSERT statement.**
• If the MANAGER_ID is greater than 200, insert these **values into the MGR_HISTORY table using a conditional multitable INSERT statement.**
Conditional INSERT ALL
INSERT ALL
WHEN SAL > 10000 THEN
INTO sal_history VALUES(EMPID,HIREDATE,SAL)
WHEN MGR > 200 THEN
INTO mgr_history VALUES(EMPID,MGR,SAL)
SELECT employee_id ,hire_date ,
salary , manager_id
EMPID
HIREDATE
MGR
SAL
FROM employees
WHERE employee_id > 200;
The example is similar to the example on the previous slide as it inserts rows into both the SAL_HISTORY and the MGR_HISTORY tables. The SELECT statement retrieves the details of employee ID, hire date, salary, and manager ID of those employees whose

employee ID is greater than 200 from the EMPLOYEES table. The details of employee ID, hire date, and salary are inserted into the SAL_HISTORY table. The details of employee ID, manager ID, and salary are inserted into the MGR_HISTORY table. This INSERT statement is referred to as a conditional ALL INSERT, as a further restriction is applied to the rows that are retrieved by the SELECT statement. From the rows that are retrieved by the SELECT statement, only those rows in which the value of the SAL column is more than 10000 are inserted in the SAL_HISTORY table, and similarly only those rows where the value of the MGR column is more than 200 are inserted in the MGR_HISTORY table.

• Select the DEPARTMENT_ID , SUM(SALARY) and **MAX(HIRE_DATE) from the EMPLOYEES table.**
• If the SUM(SALARY) is greater than $25,000 then **insert these values into the SPECIAL_SAL, using a conditional FIRST multitable INSERT.**
• If the first WHEN clause evaluates to true, the **subsequent WHEN clauses for this row should be skipped.**
• For the rows that do not satisfy the first WHEN **condition,insert into the HIREDATE_HISTORY_00,**
or HIREDATE_HISTORY_99, or HIREDATE_HISTORY tables, based on the value in the HIRE_DATE column using a conditional multitable INSERT.

```
INSERT FIRST
WHEN SAL > 25000 THEN
INTO special_sal VALUES(DEPTID, SAL)
WHEN HIREDATE like ('%00%') THEN
INTO hiredate_history_00 VALUES(DEPTID,HIREDATE)
WHEN HIREDATE like ('%99%') THEN
INTO hiredate_history_99 VALUES(DEPTID, HIREDATE)
ELSE
INTO hiredate_history VALUES(DEPTID, HIREDATE)
SELECT department_id DEPTID, SUM(salary) SAL,
MAX(hire_date) HIREDATE
FROM employees
GROUP BY department_id;
```

The example inserts rows into more than one table, using one single INSERT statement. The SELECT statement retrieves the details of department ID, total salary, and maximum hire date for every department in the EMPLOYEES table. This INSERT statement is referred to as a conditional FIRST INSERT, as an exception is made for the departments whose total salary is more than $25,000. The condition WHEN ALL > 25000 is evaluated first. If the total salary for a department is more than $25,000, then the record is inserted into the SPECIAL_SAL table irrespective of the hire date. If this first WHEN clause evaluates to true, the Oracle
server executes the corresponding INTO clause and skips subsequent WHEN clauses for this row.
For the rows that do not satisfy the first WHEN condition (WHEN SAL > 25000), the rest of the conditions are evaluated just as a conditional INSERT statement, and the records retrieved by the SELECT statement are inserted into the HIREDATE_HISTORY_00 , or HIREDATE_HISTORY_99 , or HIREDATE_HISTORY tables, based on the value in the HIREDATE column.

• Suppose you receive a set of sales records from a **nonrelational database table, SALES_SOURCE_DATA in the following format:**
EMPLOYEE_ID, WEEK_ID, SALES_MON, SALES_TUE, SALES_WED, SALES_THUR, SALES_FRI
• You would want to store these records in the **SALES_INFO table in a more typical relational format:**
EMPLOYEE_ID, WEEK, SALES
• Using a pivoting INSERT, convert the set of sales **records from the nonrelational database table to relational format.**

Pivoting INSERT

Pivoting is an operation in which you need to build a transformation such that each record from any input stream, such as, a nonrelational database table, must be converted into multiple records for a more relational database table environment. In order to solve the problem mentioned in the slide, you need to build a transformation such that each record from the original nonrelational database table, SALES_SOURCE_DATA, is converted into five records for the data warehouse's SALES_INFO table. This operation is commonly referred to as pivoting.

```
INSERT ALL
INTO sales_info VALUES (employee_id,week_id,sales_MON)
INTO sales_info VALUES (employee_id,week_id,sales_TUE)
INTO sales_info VALUES (employee_id,week_id,sales_WED)
INTO sales_info VALUES (employee_id,week_id,sales_THUR)
INTO sales_info VALUES (employee_id,week_id, sales_FRI)
SELECT EMPLOYEE_ID, week_id, sales_MON, sales_TUE,
sales_WED, sales_THUR,sales_FRI
FROM sales_source_data;
```

In the example , the sales data is received from the nonrelational database table SALES_SOURCE_DATA, which is the details of the sales performed by a sales representative on each day of a week, for a week with a particular week ID.

External Tables

An external table is a read-only table whose metadata is stored in the database but whose data is stored outside the database. Using the Oracle9 external table feature, you can use external data as a virtual table. This data can be queried and joined directly and in parallel without requiring the external data to be first loaded in the database. You can use SQL, PL/SQL, and Java to query the data in an external table. The main difference between external tables and regular tables is that externally organized tables are read-only. No DML operations (UPDATE/INSERT/DELETE) are possible, and no indexes can be created on them.

The means of defining the metadata for external tables is through the CREATE TABLE ... ORGANIZATION EXTERNAL statement. This external table definition can be thought of as a view that is used for running any SQL query against external data without requiring that the external data first be loaded into the database.

The Oracle Server provides two major access drivers for external tables. One, the loader access driver, or ORACLE_LOADER, is used for reading of data from external files using the Oracle loader technology. This access driver allows the Oracle Server to access data from any data source whose format can be interpreted
by the SQL*Loader utility. The other Oracle provided access driver, the import/export access driver, or ORACLE_ INTERNAL , can be used for both the importing and exporting of data using a platform independent format.

Creating an External Table

You create external tables using the ORGANIZATION EXTERNAL clause of the CREATE TABLE statement. You are not in fact creating a table. Rather, you are creating metadata in the data dictionary that you can use to access external data. The ORGANIZATION clause lets you specify the order in which the data rows of the table are stored. By specifying EXTERNAL in the ORGANIZATION clause, you indicate that the table is a read-only table located outside the database.TYPE access_driver_type indicates the access driver of the external table. The access driver is the Application Programming Interface (API) that interprets the external data for the database. If you do not specify TYPE, Oracle uses the default access driver, ORACLE_LOADER. The REJECT LIMIT clause lets you specify how many conversion errors can occur during a query of the external data before an Oracle error is returned and the query is aborted. The default value is 0. DEFAULT DIRECTORY lets you specify one or more default directory objects corresponding to directories on the file system where the external data sources m ay reside. efault directories can also be used by the access driver to store auxiliary files such as error logs. Multiple default directories are permitted to facilitate load balancing on multiple disk drives. The optional ACCESS PARAMETERS clause lets you assign values to the parameters of the specific access driver for this external table. Oracle does not interpret anything in this clause. It is up to the access driver to interpret this information in the context of the exter nal data. The LOCATION clause lets you specify one external locator for each external data source. Usually the location_specifier is a file, but it need not be. Oracle does not interpret this clause. It is up to the
access driver to interpret this information in the context of the external data.

Create a DIRECTORY object that corresponds to the directory on the file system where the external data
source resides.

CREATE DIRECTORY emp_dir AS '/flat_files' ;

Example of Creating an External Table

Use the CREATE DIRECTORY statement to create a directory object. A directory object specifies an alias for a directory on the server's file system where an external data source resides. You can use directory names when referring to an external data source, rather than har d-code the operating system pathname, for
greater file management flexibility. You must have CREATE ANY DIRECTORY system privileges to create directories. When you create a directory, you are automatically granted the READ object privilege and can grant READ privileges to other users and roles. The DBA can also grant this privilege to other users and roles.

Syntax

CREATE [OR REPLACE] DIRECTORY AS 'path_name';

In the syntax:

OR REPLACE Specify OR REPLACE to re-create the directory database object if it already exists. You can use this clause to change the definition of an
existing directory without dropping, re-creating, and regranting database object privileges previously granted on the directory. Users who had previously been granted privileges on a redefined directory can still access the directory without being regranted the privileges directory Specify the name of the directory object to be created. The maximum length of directory is 30 bytes. You cannot qualify a directory object with a schema name. 'path_name' Specify the full pathname of the operating system directory on the server where the files are located. The single quotes are required, with the result that the path name is case sensitive.

CREATE TABLE oldemp (
empno NUMBER, empname CHAR(20), birthdate DATE)
ORGANIZATION EXTERNAL
(TYPE ORACLE_LOADER
DEFAULT DIRECTORY emp_dir
ACCESS PARAMETERS
(RECORDS DELIMITED BY NEWLINE
BADFILE 'bad_emp'
LOGFILE 'log_emp'
FIELDS TERMINATED BY ','
(empno CHAR,
empname CHAR,
birthdate CHAR date_format date mask "dd-mon-yyyy"))
LOCATION ('emp1.txt'))
PARALLEL 5
REJECT LIMIT 200;

Example of Creating an External Table (continued)

Assume that there is a flat file that has records in the following format:
10,jones,11-Dec-1934
20,smith,12-Jun-1972
Records are delimited by new lines, and the fields are all terminated by a ",". The name of the file is:
/flat_files/emp1.txt
To convert this file as the data source for an external table, w hose metadata will reside in the database, you
need to perform the following steps:
1. Create a directory object emp_dir as follows:
CREATE DIRECTORY emp_dir AS '/flat_files' ;
2. Run the CREATE TABLE command shown
The example illustrates the table specification to create an external table for the file: / flat_files/emp1.txt
In the example, the TYPE specification is given only to illustrate its use. If not specified, ORACLE_LOADER is the default access driver. The ACCESS PARAMETERS provide values to parameters of the specific access driver and are interpreted by the access driver, not by the Oracle Server.The PARALLEL clause enables five parallel execution servers to simultaneously scan the external data sources (files) when executing the INSERT INTO TABLE statement. For example, if PARALLEL=5 were specified, then more that one parallel execution server could be working on a data source. Because external tables can be very large, for performance reasons it is advisable to specify the PARALLEL clause, or a parallel hint for the query.
The REJECT LIMIT clause specifies that if more than 200 conversion errors occur during a query of the external data, the query is aborted and an error returned. These conversion errors can arise when the access driver tries to transform the data in the data file to match the external table definition. Once the CREATE TABLE command executes successfully, the external table OLDEMP can be described, queried upon like a relational table.

Querying External Table

An external table does not describe any data that is stored in the database. Nor does it describe how data is stored in the external source. Instead, it describes how the external table layer needs to present the data to the server. It is the responsibility of the access

driver and the external table layer to do the necessary transformations required on the data in the data file so that it matches the external table definition. When the database server needs to access data in an external source, it calls the appropriate access driver to get the data from an external source in a form that the database server expects.It is important to remember that the description of the data in the data source is separate from the definition of the external table. The source file can contain more or fewer fields than there are columns in the table. Also, the data types for fields in the data source can be differ ent from the columns in the table. The access driver takes care of ensuring the data from the data source is processed so that it matches the definition of the external table.