Structured Query Language (Part 2)

Capabilities of SQL SELECT Statements

- A SELECT statement retrieves information from the database. Using a SELECT statement, you can do the following:
  - **Projection**: You can use the projection capability in SQL to choose the columns in a table that you want returned by your query.
  - **Selection**: You can use the selection capability in SQL to choose the rows in a table that you want returned by a query.
  - **Joining**: You can use the join capability in SQL to bring together data that is stored in different tables by creating a link between them.

Arithmetic Expressions

- Create expressions with number and date data by using arithmetic operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
</tr>
</tbody>
</table>
Using Arithmetic Operators

- Example
  - SELECT last_name, salary, salary + 300
  FROM employees;

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>SALARY+300</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>12000</td>
<td>13000</td>
</tr>
<tr>
<td>Jones</td>
<td>17200</td>
<td>17500</td>
</tr>
<tr>
<td>Brown</td>
<td>17320</td>
<td>17550</td>
</tr>
<tr>
<td>Wood</td>
<td>7650</td>
<td>8000</td>
</tr>
<tr>
<td>Dean</td>
<td>4100</td>
<td>4200</td>
</tr>
<tr>
<td>Adams</td>
<td>5000</td>
<td>5500</td>
</tr>
</tbody>
</table>

Operator Precedence

- Multiplication and division take priority over addition and subtraction.
- Operators of the same priority are evaluated from left to right.
- Parentheses are used to force prioritized evaluation and to clarify statements.

Using Parentheses

- Example
  - SELECT last_name, salary, 12*(salary+100)
  FROM employees;

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>12*(SALARY+100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>12000</td>
<td>156000</td>
</tr>
<tr>
<td>Jones</td>
<td>17200</td>
<td>219600</td>
</tr>
<tr>
<td>Brown</td>
<td>17320</td>
<td>218760</td>
</tr>
<tr>
<td>Wood</td>
<td>7650</td>
<td>92700</td>
</tr>
<tr>
<td>Dean</td>
<td>4100</td>
<td>49200</td>
</tr>
<tr>
<td>Adams</td>
<td>5000</td>
<td>60000</td>
</tr>
<tr>
<td>Evans</td>
<td>6000</td>
<td>72000</td>
</tr>
<tr>
<td>Gates</td>
<td>7000</td>
<td>84000</td>
</tr>
<tr>
<td>Clark</td>
<td>8000</td>
<td>96000</td>
</tr>
<tr>
<td>Jones</td>
<td>9000</td>
<td>108000</td>
</tr>
<tr>
<td>Smith</td>
<td>10000</td>
<td>120000</td>
</tr>
</tbody>
</table>

Using Arithmetic Operators

- Example
  - SELECT last_name, salary, salary + 300
  FROM employees;

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>SALARY+300</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>12000</td>
<td>13000</td>
</tr>
<tr>
<td>Jones</td>
<td>17200</td>
<td>17500</td>
</tr>
<tr>
<td>Brown</td>
<td>17320</td>
<td>17550</td>
</tr>
<tr>
<td>Wood</td>
<td>7650</td>
<td>8000</td>
</tr>
<tr>
<td>Dean</td>
<td>4100</td>
<td>4200</td>
</tr>
<tr>
<td>Adams</td>
<td>5000</td>
<td>5500</td>
</tr>
</tbody>
</table>

Operator Precedence

- Multiplication and division take priority over addition and subtraction.
- Operators of the same priority are evaluated from left to right.
- Parentheses are used to force prioritized evaluation and to clarify statements.

Using Parentheses

- Example
  - SELECT last_name, salary, 12*(salary+100)
  FROM employees;

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>12*(SALARY+100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>12000</td>
<td>156000</td>
</tr>
<tr>
<td>Jones</td>
<td>17200</td>
<td>219600</td>
</tr>
<tr>
<td>Brown</td>
<td>17320</td>
<td>218760</td>
</tr>
<tr>
<td>Wood</td>
<td>7650</td>
<td>92700</td>
</tr>
<tr>
<td>Dean</td>
<td>4100</td>
<td>49200</td>
</tr>
<tr>
<td>Adams</td>
<td>5000</td>
<td>60000</td>
</tr>
<tr>
<td>Evans</td>
<td>6000</td>
<td>72000</td>
</tr>
<tr>
<td>Gates</td>
<td>7000</td>
<td>84000</td>
</tr>
<tr>
<td>Clark</td>
<td>8000</td>
<td>96000</td>
</tr>
<tr>
<td>Jones</td>
<td>9000</td>
<td>108000</td>
</tr>
<tr>
<td>Smith</td>
<td>10000</td>
<td>120000</td>
</tr>
</tbody>
</table>
Duplicate Rows

- The default display of queries is all rows, including duplicate rows.
- Example
  
  ```sql
  SELECT department_id
  FROM employees;
  ```

Eliminating Duplicate Rows

- Eliminate duplicate rows by using the DISTINCT keyword in the SELECT clause.
- Example
  
  ```sql
  SELECT DISTINCT department_id
  FROM employees;
  ```

Limiting Rows Using a Selection

- Restrict the rows returned by using the WHERE clause.
- The WHERE clause follows the FROM clause.
- Syntax:
  
  ```sql
  SELECT * [DISTINCT] column | expression [alias] ... 
  FROM table
  [WHERE condition(s)];
  ```
Using the WHERE Clause

- Example
  
  ```sql
  SELECT employee_id, last_name, job_id, department_id
  FROM employees
  WHERE department_id = 90;
  ```

Character Strings and Dates

- Character strings and date values are enclosed in single quotation marks.
- Character values are case sensitive, and date values are format sensitive.
- The default date format is DD-MON-RR.
- Example
  
  ```sql
  SELECT last_name, job_id, department_id
  FROM employees
  WHERE last_name = 'Goyal';
  ```

Comparison Conditions

- Comparison conditions are used in conditions that compare one expression to another value or expression.
- Syntax:
  
  ```sql
  WHERE expr operator value
  ```

Using Comparison Conditions

- Example
  
  ```sql
  SELECT last_name, salary
  FROM employees
  WHERE salary <= 3000;
  ```
### Other Comparison Conditions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BETWEEN ...AND...</td>
<td>Between two values (inclusive)</td>
</tr>
<tr>
<td>IN(set)</td>
<td>Match any of a list of values</td>
</tr>
<tr>
<td>LIKE</td>
<td>Match a character pattern</td>
</tr>
<tr>
<td>IS NULL</td>
<td>Is a null value</td>
</tr>
</tbody>
</table>

### Using the BETWEEN Condition
- Use the BETWEEN condition to display rows based on a range of values.
- Example
  - SELECT `last_name`, `salary` FROM `employees` WHERE `salary` BETWEEN 2500 AND 3500;

### Using the IN Condition
- Use the IN membership condition to test for values in a list.
- Example
  - SELECT `employee_id`, `last_name`, `salary`, `manager_id` FROM `employees` WHERE `manager_id` IN (100, 101, 201);

### Using the LIKE Condition
- Use the LIKE condition to perform wildcard searches of valid search string values.
- Search conditions can contain either literal characters or numbers:
  - % denotes zero or many characters.
  - _ denotes one character.
- You can use the ESCAPE identifier to search for the actual % and _ symbols.

---

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>3000</td>
</tr>
<tr>
<td>Smith</td>
<td>2500</td>
</tr>
<tr>
<td>Jones</td>
<td>3500</td>
</tr>
<tr>
<td>Brown</td>
<td>2000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Tony</td>
<td>Young</td>
<td>3000</td>
</tr>
<tr>
<td>101</td>
<td>John</td>
<td>Jones</td>
<td>2500</td>
</tr>
<tr>
<td>102</td>
<td>Mary</td>
<td>Smith</td>
<td>3500</td>
</tr>
<tr>
<td>103</td>
<td>Tony</td>
<td>Brown</td>
<td>2000</td>
</tr>
</tbody>
</table>
Using the LIKE Condition

- Example
  - SELECT last_name
    FROM employees
    WHERE last_name LIKE '_o%';

Using the NULL Conditions

- Test for nulls with the IS NULL operator.
- Example
  - SELECT last_name, job_id, commission_pct
    FROM employees
    WHERE commission_pct IS NULL;

Logical Conditions

- A logical condition combines the result of two component conditions to produce a single result based on them or inverts the result of a single condition.
- A row is returned only if the overall result of the condition is true.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Returns TRUE if both component conditions are true</td>
</tr>
<tr>
<td>OR</td>
<td>Returns TRUE if either component condition is true</td>
</tr>
<tr>
<td>NOT</td>
<td>Returns TRUE if the following condition is false</td>
</tr>
</tbody>
</table>

Using the AND Operator

- AND requires both conditions to be true.
- Both conditions must be true for any record to be selected.
- All character searches are case sensitive.
- Character strings must be enclosed in quotation marks.
AND Truth Table

- The following table shows the results of combining two expressions with AND:

<table>
<thead>
<tr>
<th>AND</th>
<th>TRUE</th>
<th>FALSE</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>NULL</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Example of AND Operator

- Example
  - SELECT employee_id, last_name, job_id, salary
    FROM employees
    WHERE salary >= 10000
    AND job_id LIKE '%MAN%';

Using the OR Operator

- OR requires either condition to be true.
- Either condition can be true for any record to be selected.

The OR Truth Table

- The following table shows the results of combining two expressions with OR:

<table>
<thead>
<tr>
<th>OR</th>
<th>TRUE</th>
<th>FALSE</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>TRUE</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Example of OR Operator

- Example
  - SELECT employee_id, last_name, job_id, salary
    FROM employees
    WHERE salary >= 10000
    OR job_id LIKE '%MAN%';

Using the NOT Operator

- The NOT operator can also be used with other SQL operators, such as BETWEEN, LIKE, and NULL.
  - ... WHERE job_id NOT IN ('AC_ACCOUNT', 'AD_VP')
  - ... WHERE salary NOT BETWEEN 10000 AND 15000
  - ... WHERE last_name NOT LIKE '%A%'
  - ... WHERE commission_pct IS NOT NULL

The NOT Truth Table

- The following table shows the result of applying the NOT operator to a condition:

<table>
<thead>
<tr>
<th>NOT</th>
<th>TRUE</th>
<th>FALSE</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TRUE</td>
<td>FALSE</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Example of NOT Operator

- Example
  - SELECT last_name, job_id
    FROM employees
    WHERE job_id NOT IN (IT_PROG, ST_CLERK, SA_REP);
**Rules of Precedence**

- Override rules of precedence by using parentheses.

<table>
<thead>
<tr>
<th>Order</th>
<th>Evaluated</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Arithmetic operators</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Concatenation operator</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Comparison conditions</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>IS [NOT] NULL, LIKE [NOT] IN</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>[NOT] BETWEEN</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>NOT logical condition</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>AND logical condition</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>OR logical condition</td>
</tr>
</tbody>
</table>

**Rules of Precedence**

- Use parentheses to force priority.

```sql
SELECT last_name, job_id, salary
FROM employees
WHERE job_id = 'SA_REP'
    OR job_id = 'AD_REP'
    AND salary > 15000;
```

**ORDER BY Clause**

- The order of rows returned in a query result is undefined.
- The ORDER BY clause can be used to sort the rows.
- If you use the ORDER BY clause, it must be the last clause of the SQL statement.
- You can specify an expression, an alias, or column position as the sort condition.
- Sort rows with the ORDER BY clause
  - ASC: ascending order (the default order)
  - DESC: descending order
- The ORDER BY clause comes last in the SELECT statement.

```sql
SELECT last_name, job_id, salary
FROM employees
WHERE job_id = 'SA_REP'
    OR job_id = 'AD_REP'
    AND salary > 15000;
```
Default Ordering of Data

- The default sort order is ascending:
  - Numeric values are displayed with the lowest values first: for example, 1 – 999.
  - Date values are displayed with the earliest value first: for example, 01-JAN-92 before 01-JAN-95.
  - Character values are displayed in alphabetical order: for example, A first and Z last.
  - Null values are displayed last for ascending sequences and first for descending sequences.

- Reversing the Default Order
  - To reverse the order in which rows are displayed, specify the DESC keyword after the column name in the ORDER BY clause.

Sorting in Ascending Order

- Example
  - SELECT last_name, job_id, department_id, hire_date
    FROM employees
    ORDER BY hire_date;

Sorting in Descending Order

- Example
  - SELECT last_name, job_id, department_id, hire_date
    FROM employees
    ORDER BY hire_date DESC;

Sorting by Multiple Columns

- The order of ORDER BY list is the order of sort.
- You can sort by a column that is not in the SELECT list.

- Example
  - SELECT last_name, department_id, salary
    FROM employees
    ORDER BY department_id, salary DESC;
SQL Functions

Functions are a very powerful feature of SQL and can be used to do the following:
- Perform calculations on data
- Modify individual data items
- Manipulate output for groups of rows
- Format dates and numbers for display
- Convert column data types

SQL functions sometimes take arguments and always return a value.

Two Types of SQL Functions

- Single-Row Functions
  - These functions operate on single rows only and return one result per row. There are different types of single-row functions.
- Multiple-Row Functions
  - Functions can manipulate groups of rows to give one result per group of rows. These functions are known as group functions.
Single-Row Functions

- Single row functions:
  - Manipulate data items
  - Accept arguments and return one value
  - Act on each row returned
  - Return one result per row
  - May modify the data type
  - Can be nested
  - Accept arguments which can be a column or an expression
    - function_name ([arg1, arg2,...])

Nesting Functions

- Single-row functions can be nested to any level.
- Nested functions are evaluated from deepest level to the least deep level.

\[ F_3(F_2(F_1(col, arg1), arg2), arg3) \]

Obtaining Data from Multiple Tables
Cartesian Products

- A Cartesian product is formed when:
  - A join condition is omitted
  - A join condition is invalid
  - All rows in the first table are joined to all rows in the second table
- To avoid a Cartesian product, always include a valid join condition in a WHERE clause.

Generating a Cartesian Product

Types of Join

<table>
<thead>
<tr>
<th>Oracle Proprietary Joins (8i and prior):</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Equi join</td>
</tr>
<tr>
<td>• Nonequi join</td>
</tr>
<tr>
<td>• Outer join</td>
</tr>
<tr>
<td>• Self join</td>
</tr>
</tbody>
</table>

SQL: 1999 Compliant Joins:

- Cross joins
- Natural joins
- Using clause
- Full or two sided outer joins
- Arbitrary join conditions for outer joins

Defining Joins

- When data from more than one table in the database is required, a join condition is used. Rows in one table can be joined to rows in another table according to common values existing in corresponding columns, that is, usually primary and foreign key columns.
- To display data from two or more related tables, write a simple join condition in the WHERE clause.
- Syntax:
  - SELECT table1.column, table2.column
    FROM table1, table2
    WHERE table1.column1 = table2.column2;
Guidelines

- When writing a SELECT statement that joins tables, precede the column name with the table name for clarity and to enhance database access.
- If the same column name appears in more than one table, the column name must be prefixed with the table name.
- To join n tables together, you need a minimum of n-1 join conditions. For example, to join four tables, a minimum of three joins is required. This rule may not apply if your table has a concatenated primary key, in which case more than one column is required to uniquely identify each row.

What Is an Equijoin?

- Equijoins are also called Simple Joins or Inner Joins.

Retrieving Records with Equijoins

Example

```
SELECT employees.employee_id, employees.last_name, employees.department_id, departments.department_id, departments.location_id
FROM employees, departments
WHERE employees.department_id = departments.department_id;
```

Additional Search Conditions Using the AND Operator

- In addition to the join, you may have criteria for your WHERE clause to restrict the rows under consideration for one or more tables in the join.
**Additional Search Conditions Using the AND Operator**

- Example
  - SELECT last_name, employees.department_id, department_name
    FROM employees, departments
    WHERE employees.department_id = departments.department_id
    AND last_name = 'Matos';

**Qualifying Ambiguous Column Names**

- Use table prefixes to qualify column names that are in multiple tables.
- Improve performance by using table prefixes.
- Distinguish columns that have identical names but reside in different tables by using column aliases.

**Using Table Aliases**

- Qualifying column names with table names can be very time consuming, particularly if table names are lengthy.
- You can use table aliases instead of table names.
- Just as a column alias gives a column another name, a table alias gives a table another name.
- Table aliases help to keep SQL code smaller, therefore using less memory.

**Using Table Aliases**

- Simplify queries by using table aliases
- Improve performance by using table prefixes
- Example
  - SELECT e.employee_id, e.last_name, e.department_id, d.department_id, d.location_id
    FROM employees e, departments d
    WHERE e.department_id = d.department_id;
**Guidelines of Using Table Aliases**

- Table aliases can be up to 30 characters in length, but the shorter they are the better.
- If a table alias is used for a particular table name in the FROM clause, then that table alias must be substituted for the table name throughout the SELECT statement.
- Table aliases should be meaningful.
- The table alias is valid only for the current SELECT statement.

**Joining More than Two Tables**

- To join n tables together, you need a minimum of n-1 join conditions.

**Nonequijoins**

- A nonequijoin is a join condition containing something other than an equality operator.

**Retrieving Records with Nonequijoins**

- Example
  ```sql
  SELECT e.last_name, e.salary, j.grade_level
  FROM employees e, job_grades j
  WHERE e.salary BETWEEN j.lowest_sal AND j.highest_sal;
  ```
Self Joins

Joining a Table to Itself

Example

```sql
SELECT worker.last_name
FROM employees worker, employees manager
WHERE worker.manager_id = manager.employee_id;
```

Using a Subquery to Solve a Problem

Who has a salary greater than Abel’s?

Main Query:

```
Which employees have salaries greater than Abel’s salary?
```

Subquery:

```
What is Abel’s salary?
```
What is a Subquery?

- A subquery is a SELECT statement that is embedded in a clause of another SQL statement, called the parent statement.

Using a Subquery

- A subquery is a SELECT statement that is embedded in a clause of another SELECT statement.
- You can build powerful statements out of simple ones by using subqueries.
- They 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.

Subquery Syntax

- The subquery (inner query) executes once before the main query.
- The result of the subquery is used by the main query (outer query).
- Syntax
  - SELECT select_list
    FROM table
    WHERE expr operator
      (SELECT select_list
       FROM table);
Note on Subquery

- Comparison conditions fall into two classes: single-row operators (>, =, >=, <, <>, <=) and multiplerow operators (IN, ANY, ALL).
- The subquery is often referred to as a nested SELECT, sub-SELECT, or inner SELECT statement.
- The subquery generally executes first, and its output is used to complete the query condition for the main or outer query.

Guidelines for Using Subqueries

- Enclose subqueries in parentheses.
- Place subqueries on the right side of the comparison condition.
- The ORDER BY clause in the subquery is not needed unless you are performing top-n analysis.
- Use single-row operators with single-row subqueries and use multiple-row operators with multiple-row subqueries.

Using a Subquery

```sql
SELECT last_name
FROM employees
WHERE salary > (SELECT salary
FROM employees
WHERE last_name = 'Abel');
```

Types of Subqueries

**Single-row subquery**

```
Main query
Subquery
returns ST_CLERK
```

**Multiple-row subquery**

```
Main query
Subquery
returns ST_CLERK
SA_MAN
```
Types of Subqueries

- Single-row subqueries
  - Queries that return only one row from the inner SELECT statement
- Multiple-row subqueries
  - Queries that return more than one row from the inner SELECT statement
- There are also multiple-column subqueries: queries that return more than one column from the inner SELECT statement.

Single-Row Subqueries

- Example
  - SELECT last_name, job_id
    FROM employees
    WHERE job_id =
      (SELECT job_id
       FROM employees
       WHERE employee_id = 141);

Single-Row Subqueries

- Return only one row
- Use single-row comparison operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>

Executing Single-Row Subqueries

```
SELECT last_name, job_id, salary
FROM employees
WHERE job_id =
  (SELECT job_id
   FROM employees
   WHERE employee_id = 141);

AND salary >
  (SELECT salary
   FROM employees
   WHERE employee_id = 141);
```
Using Group Functions in a Subquery

- You can display data from a main query by using a group function in a subquery to return a single row.
- The subquery is in parentheses and is placed after the comparison condition.

```
SELECT last_name, job_id, salary
FROM employees
WHERE salary = (SELECT MIN(salary)
                 FROM employees);
```

Errors with Subqueries

- One common error with subqueries is more than one row returned for a single-row subquery.
- To correct this error, change the = operator to IN.

```
SELECT employee_id, last_name
FROM employees
WHERE salary = (SELECT MIN(salary)
                FROM employees
                GROUP BY employee_id)
```

The HAVING Clause with Subqueries

- The Oracle Server executes subqueries first.
- The Oracle Server returns results into the HAVING clause of the main query.

```
SELECT department_id, MIN(salary)
FROM employees
GROUP BY department_id
HAVING MIN(salary) > 1000;
```

Problems with Subqueries

- A common problem with subqueries is that no rows are returned by the inner query.

```
SELECT last_name, job_id
FROM employees
WHERE job_id = (SELECT job_id
                 FROM employees
                 WHERE last_name = 'Baker');
```

ERROR at line 1
ORA-01427: single-row subquery returns more than one row
Multiple-Row Subqueries

- Subqueries that return more than one row are called multiple-row subqueries.
- You use a multiple-row operator, instead of a single-row operator, with a multiple-row subquery. The multiple-row operator expects one or more values.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>Equal to any member in the list</td>
</tr>
<tr>
<td>ANY</td>
<td>Compare value to each value returned by the subquery</td>
</tr>
<tr>
<td>ALL</td>
<td>Compare value to every value returned by the subquery</td>
</tr>
</tbody>
</table>

Using the IN Operator in Multiple-Row Subqueries

- Each row of the main query is compared to values from a multiple-row and multiple-column subquery.

Using the ANY Operator in Multiple-Row Subqueries

- The ANY operator (and its synonym the SOME operator) compares a value to each value returned by a subquery.

Using the ALL Operator in Multiple-Row Subqueries

- The ALL operator compares a value to every value returned by a subquery.
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.

### Null Values in a Subquery

```sql
SELECT emp_last_name
FROM employees emp
WHERE emp.employee_id NOT IN
(SELECT emp.manager_id
FROM employees emp);
```

no rows selected

### Using a Subquery in the FROM Clause

```sql
SELECT a.last_name, a.salary, a.department_id, b.salary
FROM employees a, (SELECT department_id,
AVG(salary) salary
FROM employees
GROUP BY department_id) b
WHERE a.department_id = b.department_id
AND a.salary > b.salary;
```

null values in a subquery

### What are Group Functions?

**EMPLOYEES**

- The maximum salary in the employees table.

null values in a subquery

- The maximum salary in the employees table.

null values in a subquery

- The maximum salary in the employees table.
Group Functions

- Group functions operate on sets of rows to give one result per group.
- Unlike single-row functions, group functions operate on sets of rows to give one result per group.
- These sets may be the whole table or the table split into groups.

Types of Group Functions

- Each of the functions accepts an argument.
- The following table identifies the options that you can use in the syntax:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG(DISTINCT</td>
<td>ALL</td>
</tr>
<tr>
<td>COALESCE(*</td>
<td>DISTINCT</td>
</tr>
<tr>
<td>MAX(DISTINCT</td>
<td></td>
</tr>
<tr>
<td>MIN(DISTINCT</td>
<td></td>
</tr>
<tr>
<td>SUM(DISTINCT</td>
<td></td>
</tr>
<tr>
<td>VARIANCE(DISTINCT</td>
<td></td>
</tr>
</tbody>
</table>

Guidelines for Using Group Functions

- DISTINCT makes the function consider only non-duplicate values; ALL makes it consider every value including duplicates. The default is ALL and therefore does not need to be specified.
- The data types for the functions with an expr argument may be CHAR, VARCHAR2, NUMBER, or DATE.
- All group functions ignore null values. To substitute a value for null values, use the NVL, NVL2, or COALESCE functions.
- The Oracle Server implicitly sorts the result set in ascending order when using a GROUP BY clause. To override this default ordering, DESC can be used in an ORDER BY clause.

Group Functions Syntax

- Syntax
  - SELECT [column,] group_function(column), ...
  - FROM table
  - [WHERE condition]
  - [GROUP BY column]
  - [ORDER BY column];
Using the AVG and SUM Functions

- You can use AVG and SUM for numeric data.
- Example
  - SELECT AVG(salary), MAX(salary),
    MIN(salary), SUM(salary)
  FROM employees
  WHERE job_id LIKE '%REP%';

Using the MIN and MAX Functions

- You can use MIN and MAX for any data type.
- Example
  - SELECT MIN(hire_date), MAX(hire_date)
  FROM employees;

Using the COUNT Function

- The COUNT function has three formats:
  - COUNT(*)
  - COUNT(expr)
  - COUNT(DISTINCT expr)
Using the \texttt{COUNT(*)} Function

- Example
  - \texttt{SELECT COUNT(*)}
    \texttt{FROM employees}
    \texttt{WHERE department\_id = 50;}

Using the \texttt{COUNT(expr)} Function

- \texttt{COUNT(expr)} returns the number of rows with non-null values for the expr.
- Example
  - \texttt{SELECT COUNT(commission\_pct)}
    \texttt{FROM employees}
    \texttt{WHERE department\_id = 80;}

Using the \texttt{COUNT (DISTINCT expr)} Function

- Use the \texttt{DISTINCT} keyword to suppress the counting of any duplicate values within a column.
- \texttt{COUNT(DISTINCT expr)} returns the number of distinct non-null values of the expr.
- Example
  - \texttt{SELECT COUNT(DISTINCT department\_id)}
    \texttt{FROM employees;}

Group Functions and Null Values

- All group functions ignore null values in the column.
- The average is calculated based only on the rows in the table where a valid value is stored in column.
- Example
  - \texttt{SELECT AVG(commission\_pct)}
    \texttt{FROM employees;
Creating Groups of Data

You can use the GROUP BY clause to divide the rows in a table into groups. You can then use the group functions to return summary information for each group.

Syntax:

\[
\text{SELECT column, group_function(column) FROM table [WHERE condition] [GROUP BY group_by_expression] [ORDER BY column];}
\]

In the syntax: `group_by_expression` specifies columns whose values determine the basis for grouping rows.

Guidelines

- If you include a group function in a SELECT clause, you cannot select individual results as well, unless the individual column appears in the GROUP BY clause. You receive an error message if you fail to include the column list in the GROUP BY clause.
- Using a WHERE clause, you can exclude rows before dividing them into groups.
- You must include the columns in the GROUP BY clause.
- You cannot use a column alias in the GROUP BY clause.
- By default, rows are sorted by ascending order of the columns included in the GROUP BY list. You can override this by using the ORDER BY clause.

Using the GROUP BY Clause

- When using the GROUP BY clause, make sure that all columns in the SELECT list that are not group functions are included in the GROUP BY clause.
  - The FROM clause specifies the tables that the database must access.
  - The WHERE clause specifies the rows to be retrieved. Because there is no WHERE clause, all rows are retrieved by default.
  - The GROUP BY clause specifies how the rows should be grouped.
  - The GROUP BY column does not have to be in the SELECT clause.
Example of GROUP BY Clause

- Example
  - SELECT department_id, AVG(salary)
    FROM employees
    GROUP BY department_id
    ORDER BY AVG(salary);

Using the GROUP BY Clause on Multiple Columns

- You can return summary results for groups and subgroups by listing more than one GROUP BY column.
- You can determine the default sort order of the results by the order of the columns in the GROUP.
  - The SELECT clause specifies the column to be retrieved.
  - The FROM clause specifies the tables that the database must access.
  - The GROUP BY clause specifies how you must group the rows.

Using the GROUP BY Clause on Multiple Columns

- Example
  - SELECT department_id dept_id, job_id, SUM(salary)
    FROM employees
    GROUP BY department_id, job_id;
Illegal Queries Using Group Functions

- Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP BY clause.

```
SELECT department_id, COUNT(last_name)
FROM employees;
```

Column missing in the GROUP BY clause

```
SELECT department_id, COUNT(last_name)
FROM employees;
```

ERROR at line 1:
<ER-05937: not a single-group group function>

Illegal Queries Using Group Functions

- You cannot use the WHERE clause to restrict groups.
- You use the HAVING clause to restrict groups.
- You cannot use group functions in the WHERE clause.

```
SELECT department_id, AVG(salary)
FROM employees
WHERE AVG(salary) > 8000
GROUP BY department_id;
```

WHERE AVG(salary) > 8000
ERROR at line 1:
<ER-05914: group function is not allowed here>

Excluding Group Results

You use the HAVING clause to specify which groups are to be displayed, and thus, you further restrict the groups on the basis of aggregate information.

- Syntax
  - `SELECT column, group_function
   FROM table
   [WHERE condition]
   [GROUP BY group_by_expression]
   [HAVING group_condition]
   [ORDER BY column];`
  - In the syntax, `group_condition` restricts the groups of rows returned to those groups for which the specified condition is true.

```
SELECT department_id, MAX(salary)
FROM employees
GROUP BY department_id
HAVING MAX(salary) > 10000;
```

The maximum salary per department when it is greater than $10,000.
Excluding Group Results: The HAVING clause

- The Oracle Server performs the following steps when you use the HAVING clause:
  - Rows are grouped.
  - The group function is applied to the group.
  - The groups that match the criteria in the HAVING clause are displayed.
- The HAVING clause can precede the GROUP BY clause, but it is recommended that you place the GROUP BY clause first because that is more logical. Groups are formed and group functions are calculated before the HAVING clause is applied to the groups in the SELECT list.

Using the HAVING Clause

- Example
  - SELECT department_id, MAX(salary)
    FROM employees
    GROUP BY department_id
    HAVING MAX(salary)>10000;

Nesting Group Functions

- Example
  - SELECT MAX(AVG(salary))
    FROM employees
    GROUP BY department_id;

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.

Using the UNION Operator

- Example
  - SELECT employee_id, job_id
    FROM employees
  UNION
    SELECT employee_id, job_id
    FROM job_history;

DEPARTMENTS Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPARTMENT_ID</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>DEPARTMENT_NAME</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>MANAGER_ID</td>
<td>NUMBER</td>
</tr>
<tr>
<td>LOCATION_ID</td>
<td>NUMBER</td>
</tr>
</tbody>
</table>

Appendix

Table Descriptions and Data
### DEPARTMENTS Table

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>MANAGER_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
<td>300</td>
<td>1700</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
<td>301</td>
<td>1800</td>
</tr>
<tr>
<td>30</td>
<td>Shipping</td>
<td>302</td>
<td>1400</td>
</tr>
<tr>
<td>40</td>
<td>IT</td>
<td>303</td>
<td>1400</td>
</tr>
<tr>
<td>50</td>
<td>Sales</td>
<td>500</td>
<td>3600</td>
</tr>
<tr>
<td>60</td>
<td>Executive</td>
<td>100</td>
<td>1700</td>
</tr>
<tr>
<td>70</td>
<td>Accounting</td>
<td>305</td>
<td>1700</td>
</tr>
<tr>
<td>80</td>
<td>Contracts</td>
<td></td>
<td>1700</td>
</tr>
</tbody>
</table>

8 rows selected.

### EMPLOYEES Table

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>EMAIL</th>
<th>PHONE_NUMBER</th>
<th>HIRE_DATE</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
<th>MANAGER_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smith</td>
<td>Johnson</td>
<td><a href="mailto:Smith@abc.com">Smith@abc.com</a></td>
<td>123-456-7890</td>
<td>2002-01-01</td>
<td>E01</td>
<td>D01</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Brown</td>
<td>Lee</td>
<td><a href="mailto:Brown@xyz.com">Brown@xyz.com</a></td>
<td>987-654-3210</td>
<td>2003-02-02</td>
<td>E02</td>
<td>D02</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Davis</td>
<td>White</td>
<td><a href="mailto:Davis@def.com">Davis@def.com</a></td>
<td>456-789-0123</td>
<td>2004-03-03</td>
<td>E03</td>
<td>D03</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Johnson</td>
<td>Williams</td>
<td><a href="mailto:Johnson@ghi.com">Johnson@ghi.com</a></td>
<td>321-456-7890</td>
<td>2005-04-04</td>
<td>E04</td>
<td>D04</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>Smith</td>
<td>Jones</td>
<td><a href="mailto:Smith@jkl.com">Smith@jkl.com</a></td>
<td>654-321-0987</td>
<td>2006-05-05</td>
<td>E05</td>
<td>D05</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>Brown</td>
<td>Cooper</td>
<td><a href="mailto:Brown@mno.com">Brown@mno.com</a></td>
<td>789-654-1230</td>
<td>2007-06-06</td>
<td>E06</td>
<td>D06</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>Davis</td>
<td>Miller</td>
<td><a href="mailto:Davis@pqr.com">Davis@pqr.com</a></td>
<td>897-564-0123</td>
<td>2008-07-07</td>
<td>E07</td>
<td>D07</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>Johnson</td>
<td>Turner</td>
<td><a href="mailto:Johnson@stu.com">Johnson@stu.com</a></td>
<td>980-765-4512</td>
<td>2009-08-08</td>
<td>E08</td>
<td>D08</td>
<td>9</td>
</tr>
</tbody>
</table>

8 rows selected.