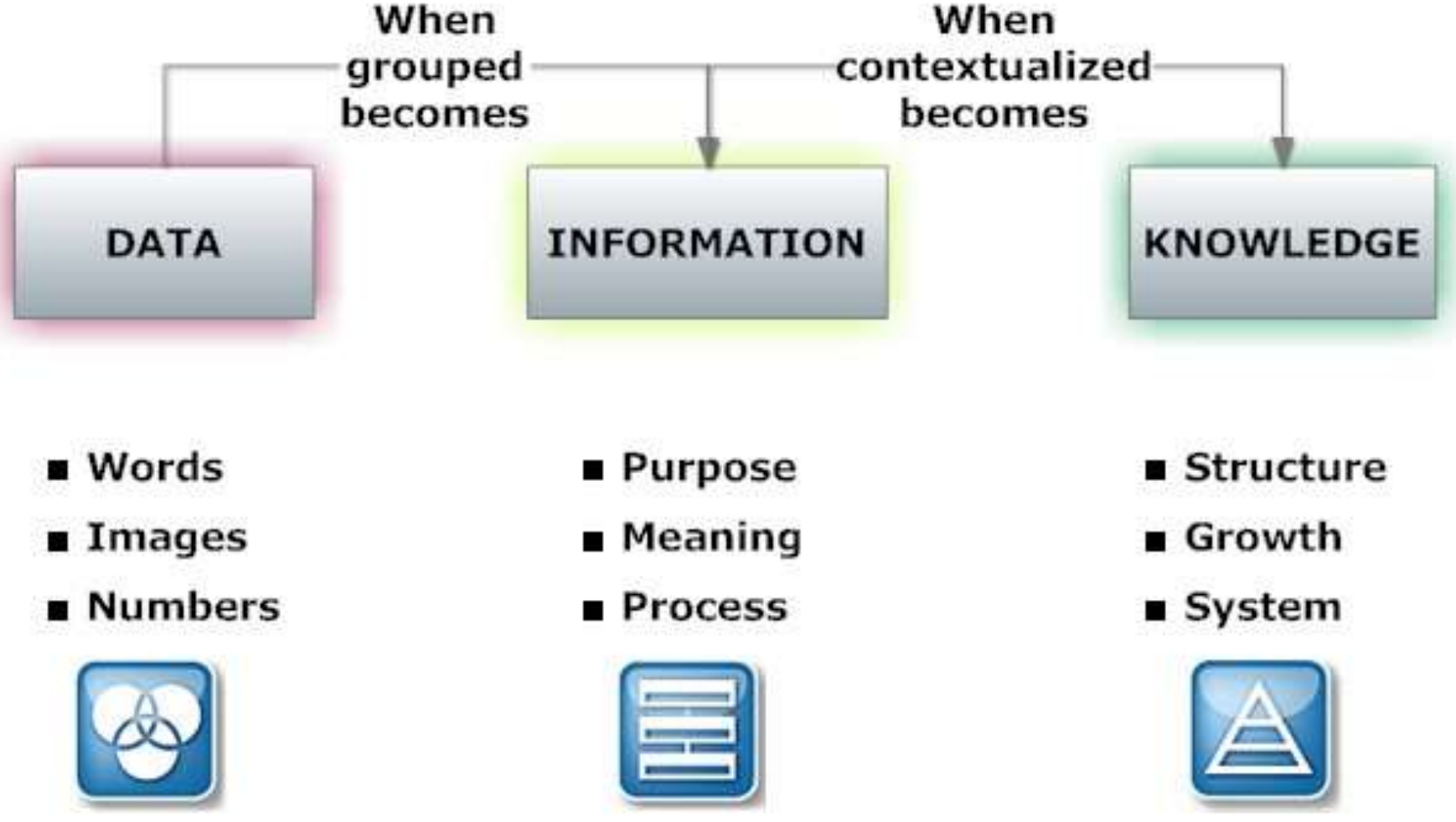
# Chapter 3

Structured Query Language:
Creating and manipulating data

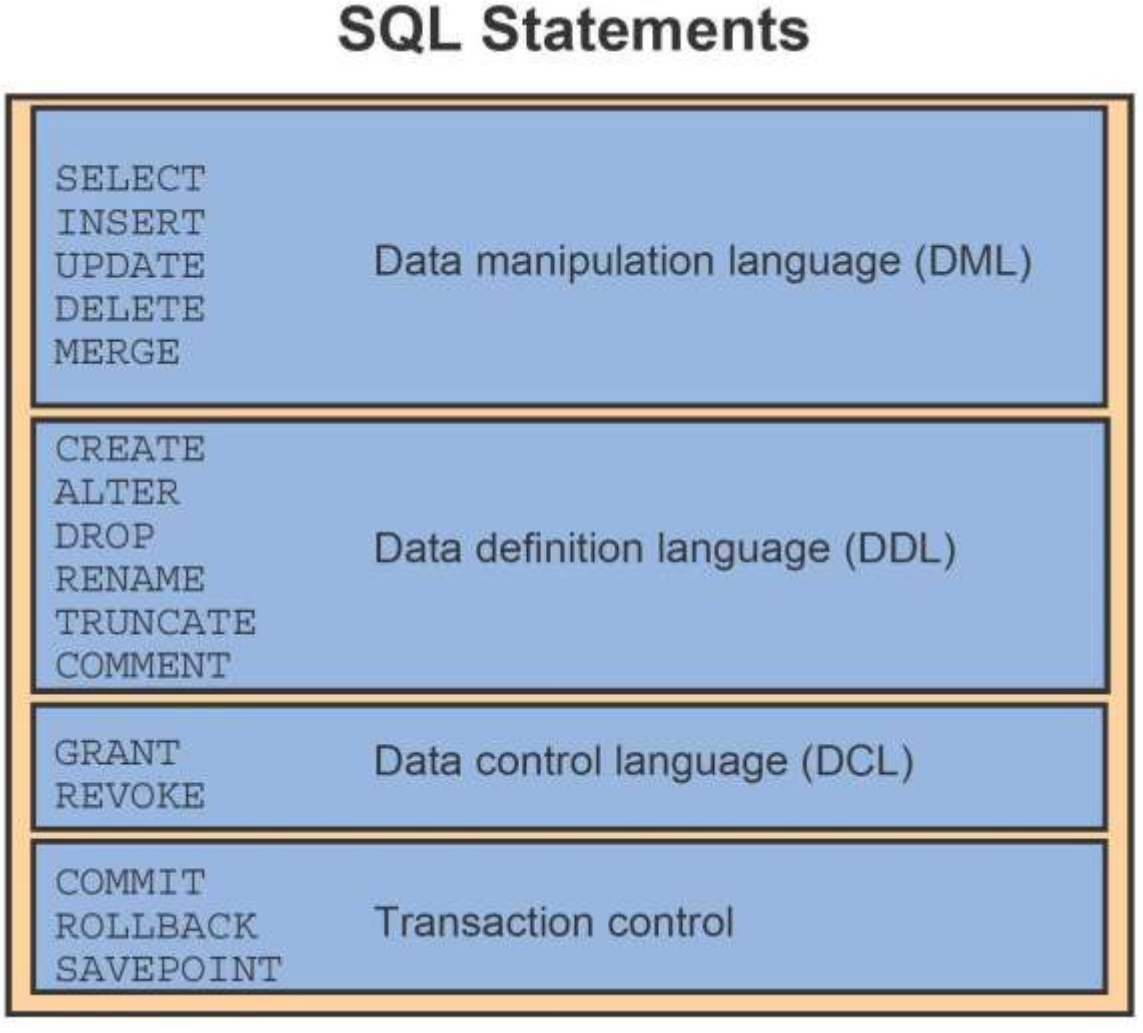
- In chapter 1, we looked at how databases are used in various business applications

  - Create and process forms
  - Process user queries
  - Create and process reports
  - Execute application logic
  - Control application

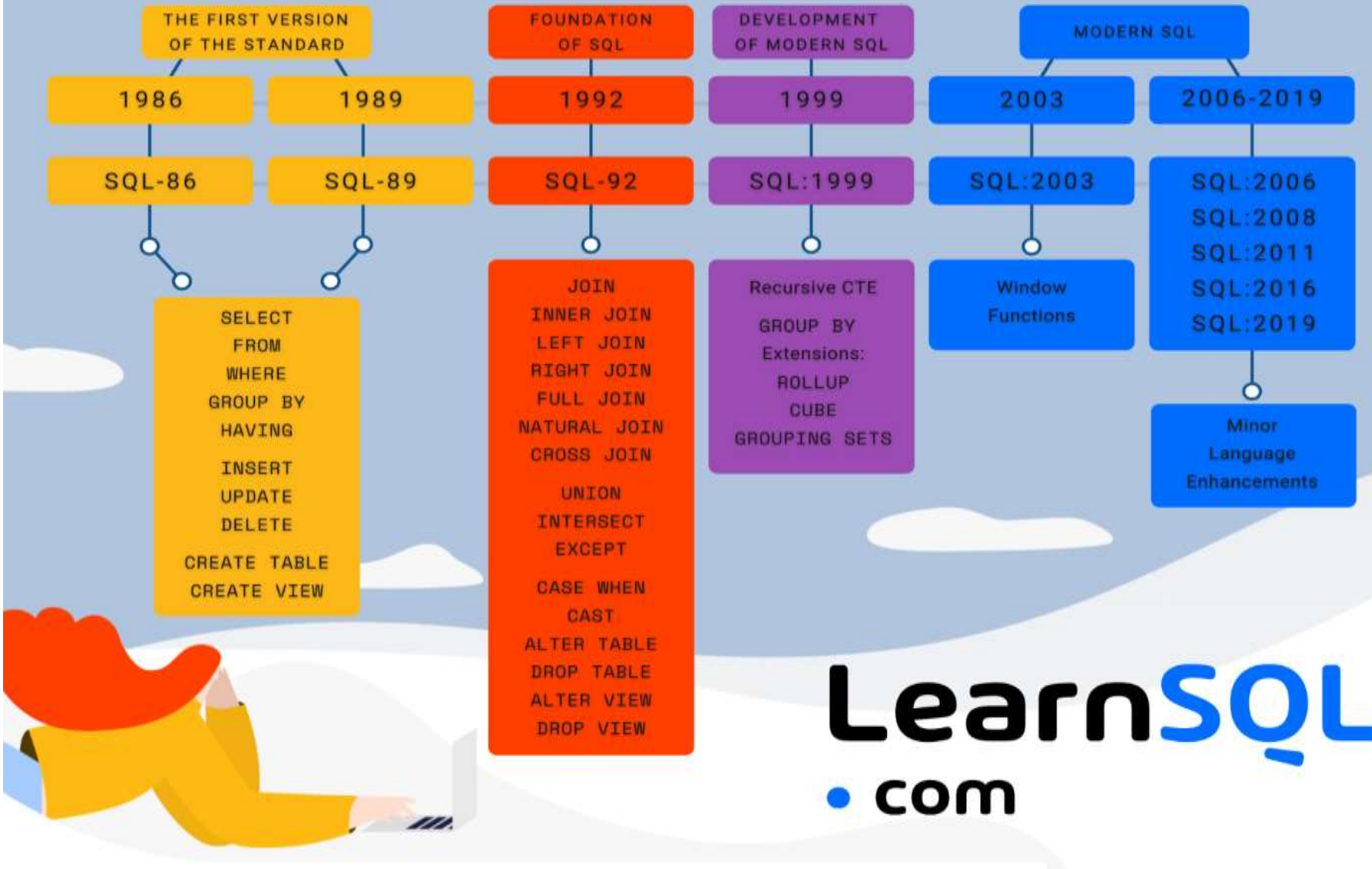
- In chapter 2, we see how databases are helpful in decision making



- In this chapter, we learn how to create, load data and extract information from a database
- This is accomplish using Structured Query Language (SQL)

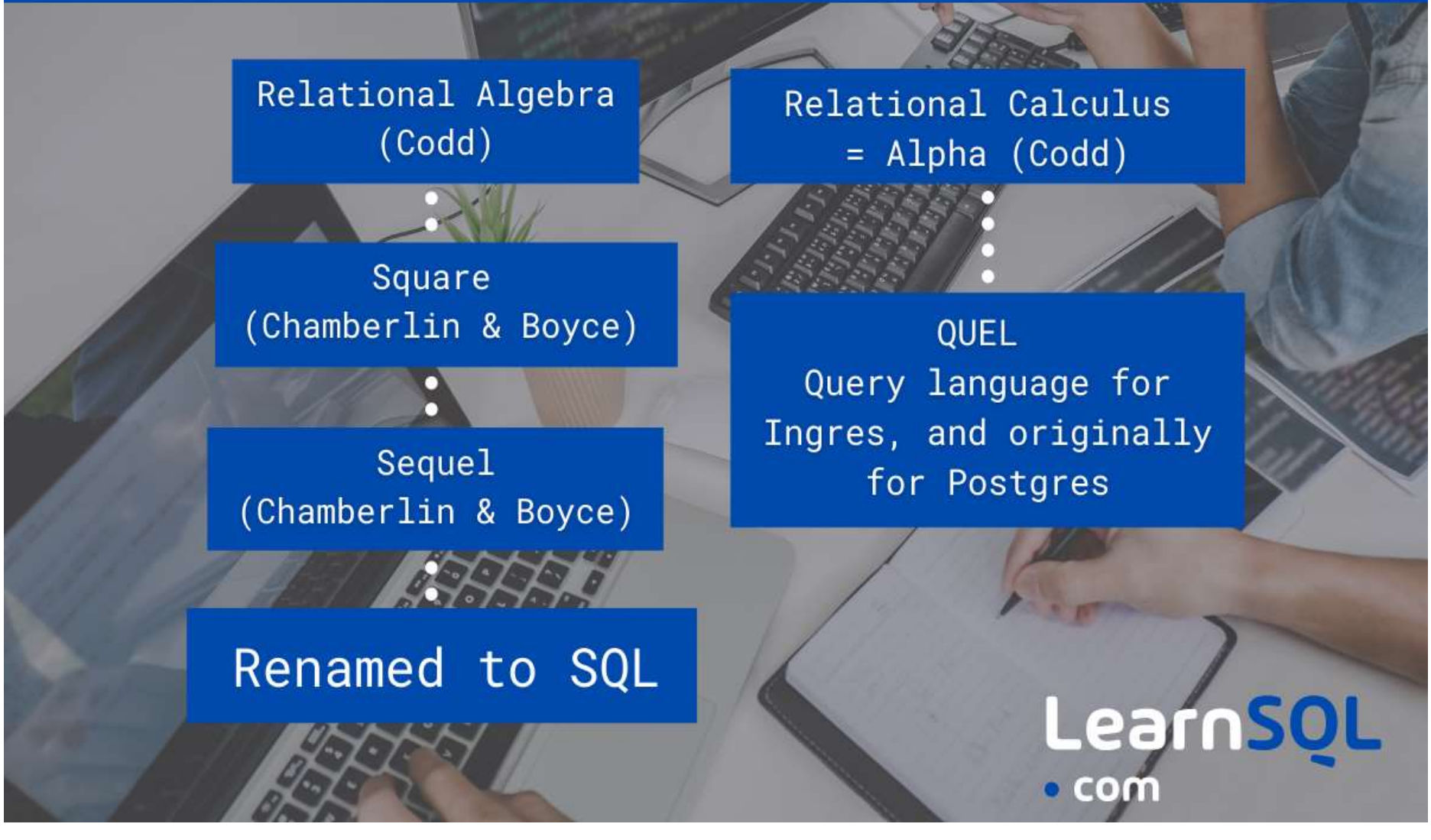


# The History of SQL Standards




LearnSQL.com

# SQL FAMILY TREE



$$\pi_{e.name} ( \sigma_{e.salary > m.salary} ( \rho_e(employee) \bowtie_{e.manager = m.name} \rho_m(employee) ) )$$

(a) Relational Algebra version

RANGE employee e;
RANGE employee m;
GET w (e.name): $\exists m((e.manager = m.name) \wedge (e.salary > m.salary))$

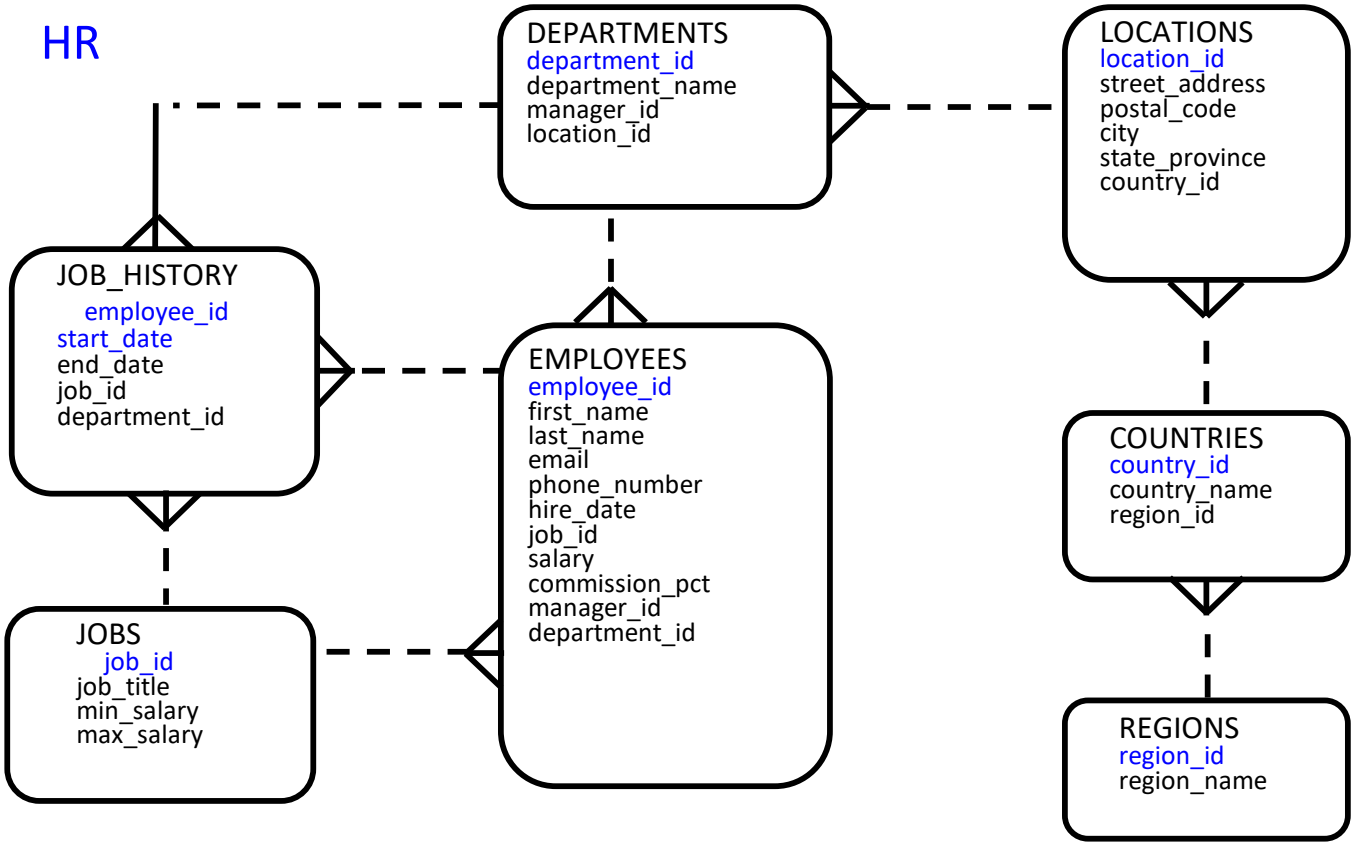
(b) Relational Calculus version

```
select e.name
from employee e, employee m
where e.manager = m.name and e.salary > m.salary
```

(c) Sequel (SQL) version

**Figure 2. Three versions of the query, "Find names of employees who earn more than their managers."**

# The HR Entity Relationship Diagram




Note:

Please create the HR database to facilitate your learning for Tutorials 3-5.

Execute the Create_HR.txt and In_HR.txt script.

# Basic SELECT Statement

- SELECT identifies the columns to be displayed
- FROM identifies the table that contains those columns

```
SELECT {*|[DISTINCT] column|expression [alias],...}
FROM    table;
```

# Selecting All Columns

- All columns of a table can be displayed by placing an * after keyword SELECT

```
SELECT *
FROM   departments;
```

| DEPARTMENT_ID | DEPARTMENT_NAME | MANAGER_ID | LOCATION_ID |
|---|---|---|---|
| 10 | Administration | 200 | 1700 |
| 20 | Marketing | 201 | 1800 |
| 50 | Shipping | 124 | 1500 |
| 60 | IT | 103 | 1400 |
| 80 | Sales | 149 | 2500 |
| 90 | Executive | 100 | 1700 |
| 110 | Accounting | 205 | 1700 |

# Selecting Specific Columns - Projection

- You can use the SELECT statement to display specific columns of the table by indicating the column names in the order you would like to see them, separated by commas

```
SELECT department_id, location_id
FROM   departments;
```

| DEPARTMENT_ID | LOCATION_ID |
|---|---|
| 10 | 1700 |
| 20 | 1800 |
| 50 | 1500 |
| 60 | 1400 |
| 80 | 2500 |

...

# Database Foundations

## 6-7

## Restricting Data Using WHERE

ORACLE
Academy

# Objectives

- This lesson covers the following objectives:
  - Limit rows with:
    - WHERE clause
    - Comparison operators using =, <=,>=, <>,>,<, !=,^=, BETWEEN,IN,LIKE and NULL conditions
    - Logical conditions using AND,OR and NOT operators
  - Describe the rules of precedence for operators in an expression

# Limiting Rows Using Selection (WHERE)

**EMPLOYEES**

| EMPLOYEE_ID | FIRST_NAME | LAST_NAME | JOB_ID | SALARY | DEPARTMENT_ID |
|---|---|---|---|---|---|
| 100 | Steven | King | AD_PRES | 24000 | 90 |
| 101 | Neena | Kochhar | AD_VP | 17000 | 90 |
| 102 | Lex | De Haan | AD_VP | 17000 | 90 |
| 200 | Jennifer | Whalen | AD_ASST | 4400 | 10 |
| 205 | Shelley | Higgins | AC_MGR | 12000 | 110 |
| 206 | William | Gietz | AC_ACCOUNT | 8300 | 110 |
| 149 | Eleni | Zlotkey | SA_MAN | 10500 | 80 |
| 174 | Ellen | Abel | SA_REP | 11000 | 80 |

...

**"retrieve all employees in department 90"**

| EMPLOYEE_ID | FIRST_NAME | LAST_NAME | JOB_ID | SALARY | DEPARTMENT_ID |
|---|---|---|---|---|---|
| 100 | Steven | King | AD_PRES | 24000 | 90 |
| 101 | Neena | Kochhar | AD_VP | 17000 | 90 |
| 102 | Lex | De Haan | AD_VP | 17000 | 90 |

# Limiting the Rows That Are Selected

- Restrict the rows that are returned by using the WHERE clause:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM   table
[WHERE logical expression(s)];
```

- If the logical expression evaluates to true, the row meeting the condition is returned
- The WHERE clause follows the FROM clause

# Using the WHERE Clause

- Retrieve all employees in department 90

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

| EMPLOYEE_ID | LAST_NAME | JOB_ID | DEPARTMENT_ID |
|---|---|---|---|
| 100 | King | AD_PRES | 90 |
| 101 | Kochhar | AD_VP | 90 |
| 102 | De Haan | AD_VP | 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

```
SELECT last_name, job_id, department_id
FROM   employees
WHERE  last_name = 'Whalen';
```

# Character Strings and Dates

- The default date display format is DD-Mon-YYYY

```
SELECT last_name
FROM   employees
WHERE  hire_date = '29-Jan-2000';
```

# Using the ORDER BY Clause

- Sort the retrieved rows with the ORDER BY clause:
  - ASC: Ascending order (default)
  - DESC: Descending order
- The ORDER BY clause comes last in the SELECT statement:

```
SELECT   last_name, job_id, department_id, hire_date
FROM     employees
ORDER BY hire_date ;
```

# ORDER BY Clause

- Numeric values are displayed lowest to highest
- Date values are displayed with the earliest value first
- Character values are displayed in alphabetical order
- Null values are displayed last in ascending order and first in descending order
- NULLS FIRST specifies that NULL values should be returned before non-NULL values
- NULLS LAST specifies that NULL values should be returned after non-NULL values

# Sorting

- Sorting in descending order:

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

- Sorting by column alias:

```
SELECT   employee_id, last_name, salary*12 annsal
FROM     employees
ORDER BY annsal ;
```

# Sorting

- Sorting by using the column's numeric position:

```
SELECT   last_name, job_id, department_id, hire_date
FROM     employees
ORDER BY 3;
```

- Sorting by multiple columns:

```
SELECT last_name, department_id, salary
FROM   employees
ORDER BY department_id, salary DESC;
```

# Order of Execution

- The order of execution of a SELECT statement is as follows:
  - FROM clause:
    - locates the table that contains the data
  - WHERE clause:
    - restricts the rows to be returned
  - SELECT clause:
    - selects from the reduced data set the columns requested
  - ORDER BY clause:
    - orders the result set

# Comparison Operators

| Operator | Meaning |
|---|---|
| = | Equal to |
| > | Greater than |
| >= | Greater than or equal to |
| < | Less than |
| <= | Less than or equal to |
| <> | Not equal to |
| BETWEEN…AND… | Between two values (inclusive) |
| IN (set) | Match any of a list of values |
| LIKE | Match a character pattern |
| IS NULL | Is a null value |

# Using Comparison Operators

- Retrieve records from the EMPLOYEES table where the salary is less than or equal to $3,000

```
SELECT last_name, salary
FROM   employees
WHERE  salary <= 3000;
```

| LAST_NAME | SALARY |
|---|---|
| Matos | 2600 |
| Vargas | 2500 |

# Range Conditions: BETWEEN Operator

- Use the BETWEEN operator to display rows based on a range of values:

```
SELECT last_name, salary
FROM   employees
WHERE  salary BETWEEN 2500 AND 3500 ;
```

Lower limit

Upper limit

| LAST_NAME | SALARY |
|---|---|
| Rajs | 3500 |
| Davies | 3100 |
| Matos | 2600 |
| Vargas | 2500 |

** Note – when using BETWEEN lower value must be specified first

# Membership Conditions: IN Operator

- Use the IN operator to test for values in a list:

```
SELECT employee_id, last_name, salary, manager_id
FROM   employees
WHERE  manager_id IN (100, 101, 201) ;
```

| EMPLOYEE_ID | LAST_NAME | SALARY | MANAGER_ID |
|---|---|---|---|
| 101 | Kochhar | 17000 | 100 |
| 102 | De Haan | 17000 | 100 |
| 149 | Zlotkey | 10500 | 100 |
| 124 | Mourgos | 5800 | 100 |
| 201 | Hartstein | 13000 | 100 |

...

** Note: items in list can be in any order

# Membership Conditions: NOT IN Operator

- Use the NOT IN operator to test for values not in a list:

```
SELECT employee_id, last_name, salary, manager_id
FROM   employees
WHERE  department_id NOT IN (60, 90, 100) ;
```

| EMPLOYEE_ID | LAST_NAME | SALARY | MANAGER_ID |
|---|---|---|---|
| 200 | Whalen | 4400 | 101 |
| 205 | Higgins | 12000 | 101 |
| 206 | Gietz | 8300 | 205 |
| 149 | Zlotkey | 10500 | 100 |
| 174 | Abel | 11000 | 149 |
| 176 | Taylor | 8600 | 149 |

...

# Pattern Matching: LIKE Operator

- Use the LIKE operator to perform wildcard searches of valid search string values
- Search conditions can contain literal characters or numbers:
  - % denotes zero or more characters
  - _ denotes one character

```
SELECT   first_name
FROM     employees
WHERE    first_name LIKE 'S%' ;
```

| FIRST_NAME |
| --- |
| Shelley |
| Steven |

# Combining Wildcard Characters

- You can combine the two wildcard characters (%, _) with literal characters for pattern matching:

```
SELECT last_name
FROM   employees
WHERE  last_name LIKE '_o%' ;
```

| LAST_NAME |
|---|
| Kochhar |
| Lorentz |
| Mourgos |

# Combining Wildcard Characters

- You can use the ESCAPE identifier to search for the actual % and _ symbols

```
SELECT employee_id, last_name, job_id
FROM   employees
WHERE  job_id LIKE '%SA\_%' ESCAPE '\';
```

- This will return records with SA_ in their job_id

| EMPLOYEE_ID | LAST_NAME | JOB_ID |
|---|---|---|
| 149 | Zlotkey | SA_MAN |
| 174 | Abel | SA_REP |
| 176 | Taylor | SA_REP |
| 178 | Grant | SA_REP |

# Using the NULL Conditions

- Test for nulls with the IS NULL or IS NOT NULL operators:

```
SELECT last_name, manager_id
FROM   employees
WHERE  manager_id IS NULL;
```

| LAST_NAME | MANAGER_ID |
|---|---|
| King | - |

- You cannot test with = because a null cannot be equal or unequal to any value

# Defining Conditions Using the Logical Operators

- A logical condition combines the result of two component conditions to produce a single result based on those conditions or if using NOT it inverts the result of a single condition

| Operator | Meaning |
|---|---|
| AND | Returns TRUE if both component conditions are TRUE |
| OR | Returns TRUE if either component condition is TRUE |
| NOT | Returns TRUE if the condition is FALSE<br>Returns FALSE if the condition is TRUE |

# Using the AND Operator

- AND requires both component conditions to be true:

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

| EMPLOYEE_ID | LAST_NAME | JOB_ID | SALARY |
|---|---|---|---|
| 149 | Zlotkey | SA_MAN | 10500 |
| 201 | Hartstein | MK_MAN | 13000 |

- Note: All character searches are case sensitive and must be enclosed in quotation marks

# Using the OR Operator

- OR requires either component condition to be true:

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

| EMPLOYEE_ID | LAST_NAME | JOB_ID | SALARY |
|---|---|---|---|
| 100 | King | AD_PRES | 24000 |
| 101 | Kochhar | AD_VP | 17000 |
| 102 | De Haan | AD_VP | 17000 |
| 205 | Higgins | AC_MGR | 12000 |
| 149 | Zlotkey | SA_MAN | 10500 |

...

# Using the NOT Operator

- NOT reverses the value of the condition:

```
SELECT last_name, job_id
FROM employees
WHERE job_id NOT IN ('IT_PROG', 'ST_CLERK', 'SA_REP') ;
```

| LAST_NAME | JOB_ID |
|---|---|
| King | AD_PRES |
| Kochhar | AD_VP |
| De Haan | AD_VP |
| Whalen | AD_ASST |
| Higgins | AC_MGR |
| Gietz | AC_ACCOUNT |
| Zlotkey | SA_MAN |

...

# Rules of Precedence

| Precedence | Operator |
|---|---|
| 1 | Arithmetic operators |
| 2 | Concatenation operator |
| 3 | Comparison conditions |
| 4 | IS [NOT] NULL, LIKE, [NOT] IN |
| 5 | [NOT] BETWEEN |
| 6 | Not equal to |
| 7 | NOT logical operator |
| 8 | AND logical operator |
| 9 | OR logical operator |

**Use parentheses to override rules of precedence**

# Rules of Precedence

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

Precedence of the AND Operator

| LAST_NAME | JOB_ID | SALARY |
|---|---|---|
| King | AD_PRES | 24000 |
| Abel | SA_REP | 11000 |
| Taylor | SA_REP | 8600 |
| Grant | SA_REP | 7000 |

# Rules of Precedence

```
SELECT last_name, job_id, salary
FROM employees
WHERE (job_id = 'SA_REP'
OR     job_id = 'AD_PRES')
AND    salary > 15000;
```

Parentheses

| LAST_NAME | JOB_ID | SALARY |
|---|---|---|
| King | AD_PRES | 24000 |

# Arithmetic Expressions

- Create expressions with number and date data by using arithmetic operators
- Column names, numeric constants and arithmetic operators can be used in an arithmetic expression

| Operator | Description |
|---|---|
| + | Add |
| - | Subtract |
| * | Multiply |
| / | Divide |

- Arithmetic operators can be used in any clause of a SQL statement except FROM

# Using Arithmetic Operators

- Here the addition operator is used to calculate a salary increase of $300 for all employees
- SALARY + 300 is displayed as the column heading

```
SELECT last_name, salary, salary + 300
FROM   employees;
```

| LAST_NAME | SALARY | SALARY+300 |
|---|---|---|
| King | 24000 | 24300 |
| Kochhar | 17000 | 17300 |
| De Haan | 17000 | 17300 |
| Whalen | 4400 | 4700 |

...

# Operator Precedence

- Use parentheses to reinforce the standard order of precedence and to improve clarity

```
SELECT last_name, salary, 12*salary+100
FROM   employees;
```

| LAST_NAME | SALARY | 12*SALARY+100 |
|---|---|---|
| King | 24000 | 288100 |
| Kochhar | 17000 | 204100 |
| De Haan | 17000 | 204100 |
| Whalen | 4400 | 52900 |
| Higgins | 12000 | 144100 |
| Gietz | 8300 | 99700 |

...

ORACLE Academy

# Operator Precedence

- You can override the rules of precedence by using parentheses to specify the order in which the operators are to be executed

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

| LAST_NAME | SALARY | 12*(SALARY+100) |
|---|---|---|
| King | 24000 | 289200 |
| Kochhar | 17000 | 205200 |
| De Haan | 17000 | 205200 |
| Whalen | 4400 | 54000 |
| Higgins | 12000 | 145200 |
| Gietz | 8300 | 100800 |
| Zlotkey | 10500 | 127200 |

...

# Defining a Null Value

- Null is a value that is unavailable, unassigned, unknown, or inapplicable
- Null is not the same as zero or a blank space

```
SELECT last_name, job_id, salary, commission_pct
FROM   employees;
```

| LAST_NAME | JOB_ID | SALARY | COMMISSION_PCT |
|---|---|---|---|
| King | AD_PRES | 24000 | - |
| Gietz | AC_ACCOUNT | 8300 | - |
| Zlotkey | SA_MAN | 10500 | .2 |
| Abel | SA_REP | 11000 | .3 |
| Taylor | SA_REP | 8600 | .2 |
| Grant | SA_REP | 7000 | .15 |
| Mourgos | ST_MAN | 5800 | - |

...

# Null Values in Arithmetic Expressions

- Any arithmetic expression containing a null value will evaluate to null

```
SELECT last_name, 12*salary*commission_pct
FROM   employees;
```

| LAST_NAME | 12*SALARY*COMMISSION_PCT |
|---|---|
| King | - |
| Gietz | - |
| Zlotkey | 25200 |
| Abel | 39600 |
| Taylor | 20640 |
| Grant | 12600 |
| Mourgos | - |

...

# Defining a Column Alias

- A column alias:
  - Renames a column heading
  - Is useful with calculations
  - Immediately follows the column name (There can also be the optional AS keyword between the column name and the alias.)
  - Requires double quotation marks if it contains spaces or special characters or if it is case-sensitive, the default is all uppercase

# Using Column Aliases

- Keyword AS is optional
- Column names appear uppercase by default

```
SELECT last_name AS name,
commission_pct comm
FROM   employees;
```

| NAME | COMM |
|---|---|
| King | - |
| Kochhar | - |
| Whalen | - |
| Higgins | - |
| ..... | |

- Column names enclosed in parenthesis will appear as entered

```
SELECT last_name "Name" ,
  salary*12 "Annual Salary"
FROM   employees;
```

| Name | Annual Salary |
|---|---|
| King | 288000 |
| Kochhar | 204000 |
| Whalen | 204000 |
| Higgins | 52800 |
| ..... | |

ORACLE Academy

# Concatenation Operator

- Links columns or character strings to other columns
- Is represented by two vertical bars (||)
- Creates a column that is a character expression

```
SELECT  last_name||job_id AS "Employees"
FROM    employees;
```

- Concatenating a NULL with a character results in a character string

| Employees |
| --- |
| KingAD_PRES |
| KochharAD_VP |
| De HaanAD_VP |
| WhalenAD_ASST |

# Literal Character Strings

- A literal is a character, a number, or a date that is included in the SELECT statement
- Date and character literal values must be enclosed within single quotation marks
- Each character string is output once for each row returned

# Using Literal Character Strings

- In the example, the last name and job_id for each employee are concatenated with a literal to give the returned rows more meaning

```
SELECT last_name ||' is a '||job_id
       AS "Employee Details"
FROM   employees;
```

| Employee Details |
|---|
| King is a AD_PRES |
| Kochhar is a AD_VP |
| De Haan is a AD_VP |
| Whalen is a AD_ASST |
| Higgins is a AC_MGR |
| Gietz is a AC_ACCOUNT |

# Duplicate Rows

- The default display of queries is all rows, including duplicate rows

```
SELECT department_id
FROM   employees;
```

| DEPARTMENT_ID |
|---|
| 90 |
| 90 |
| 90 |
| 10 |
| 110 |
| 110 |
| 80 |
| 80 |
| 80 |

# Duplicate Rows

- To eliminate duplicate rows in the result, include the DISTINCT keyword in the SELECT clause immediately after the SELECT keyword

```
SELECT DISTINCT department_id
FROM   employees;
```

| DEPARTMENT_ID |
|---|
| - |
| 90 |
| 20 |
| 110 |
| 80 |
| 50 |

...

# SELECT Statement

- Used for queries on single or multiple tables
- Clauses of the SELECT statement:
  - **SELECT**
    - List the columns (and expressions) that should be returned from the query
  - **FROM**
    - Indicate the table(s) or view(s) from which data will be obtained
  - **WHERE**
    - Indicate the conditions under which a row will be included in the result
  - **GROUP BY**
    - Indicate categorization of results
  - **HAVING**
    - Indicate the conditions under which a category (group) will be included
  - **ORDER BY**
    - Sorts the result according to specified criteria

# Summary

- In this lesson, you should have learned how to:
  - Write and execute SELECT statement that:
    - Uses arithmetic and concatenation operators
    - Uses literal character strings
    - Eliminate duplicate rows
    - Use the `WHERE` clause to restrict rows of output:
      - Use the comparison conditions
      - Use the `BETWEEN`, `IN`, `LIKE`, and `NULL` operators
      - Apply the logical `AND`, `OR`, and `NOT` operators
    - Use the `ORDER BY` clause to sort rows of output

```
SELECT   *|{[DISTINCT] column|expression [alias],...}
FROM     table
[WHERE   condition(s)]
[ORDER BY {column, expr, alias} [ASC|DESC]] ;
```

# Displaying Data from Multiple Tables

SQL provides join conditions that enable information to be queried from separate tables and combined in one report.

# ON Clause Example

- In this example, the ON clause is used to join the employees table with the jobs table

```
SELECT last_name, job_title
FROM employees e JOIN jobs j
  ON (e.job_id = j.job_id);
```

- A join ON clause is required when the common columns have different names in the two tables

| LAST_NAME | JOB_TITLE |
|---|---|
| King | President |
| Kochhar | Administration Vice President |
| De Haan | Administration Vice President |
| Whalen | Administration Assistant |
| Higgins | Accounting Manager |
| Gietz | Public Accountant |
| Zlotkey | Sales Manager |
| Abel | Sales Representative |
| Taylor | Sales Representative |
| ... | |

# ON Clause Example

```
SELECT last_name, job_title
FROM employees e JOIN jobs j
  ON (e.job_id = j.job_id);
```

- When using an ON clause on columns with the same name in both tables, you need to add a qualifier (either the table name or alias) otherwise an error will be returned
- The example above uses table aliases as a qualifier e.job_id = j.job_id, but could also have been written using the table names (employees.job_id = jobs.job_id

| LAST_NAME | JOB_TITLE |
|---|---|
| King | President |
| Kochhar | Administration Vice President |
| De Haan | Administration Vice President |
| Whalen | Administration Assistant |
| Higgins | Accounting Manager |
| Gietz | Public Accountant |
| Zlotkey | Sales Manager |
| Abel | Sales Representative |
| Taylor | Sales Representative |
| ... | |

# Retrieving Records with the ON Clause

- You can also use the ON clause to join columns that have different names or data types

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e JOIN departments d
ON     (e.department_id = d.department_id);
```

| EMPLOYEE_ID | LAST_NAME | DEPARTMENT_ID | DEPARTMENT_ID | LOCATION_ID |
|---|---|---|---|---|
| 200 | Whalen | **10** | **10** | 1700 |
| 201 | Hartstein | **20** | **20** | 1800 |
| 202 | Fay | **20** | **20** | 1800 |
| 124 | Mourgos | **50** | **50** | 1500 |
| 141 | Rajs | **50** | **50** | 1500 |
| 142 | Davies | **50** | **50** | 1500 |

ORACLE Academy

# Creating Three-Way Joins with the ON Clause

- There must be 2 join statements when joining 3 tables as shown:

```
SELECT  employee_id, city, department_name
FROM    employees e JOIN departments d
ON      d.department_id = e.department_id
JOIN    locations l
ON      d.location_id = l.location_id;
```

| EMPLOYEE_ID | CITY | DEPARTMENT_NAME |
|---|---|---|
| 201 | Toronto | Marketing |
| 202 | Toronto | Marketing |
| 149 | Oxford | Sales |
| 174 | Oxford | Sales |
| 176 | Oxford | Sales |
| 103 | Southlake | IT |

...

# Applying Additional Conditions to a Join

- Use the AND clause or the WHERE clause to apply additional conditions:

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e JOIN departments d
ON     (e.department_id = d.department_id)
AND    e.manager_id = 149 ;
```

Or

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e JOIN departments d
ON     (e.department_id = d.department_id)
WHERE   e.manager_id = 149 ;
```

# Joining a Table to Itself

**EMPLOYEES (WORKER)**

| EMPLOYEE_ID | LAST_NAME | MANAGER_ID |
|---|---|---|
| 100 | King | - |
| 101 | Kochhar | **100** |
| 102 | De Haan | **100** |
| 200 | Whalen | **101** |
| 205 | Higgins | **101** |
| 206 | Gietz | **205** |
| 149 | Zlotkey | **100** |
| 174 | Abel | **149** |
| 176 | Taylor | **149** |
| 201 | Hartstein | **100** |
| 202 | Fay | **201** |

...

**EMPLOYEES (MANAGER)**

| EMPLOYEE_ID | LAST_NAME |
|---|---|
| **100** | King |
| **101** | Kochhar |
| **102** | De Haan |
| **200** | Whalen |
| **205** | Higgins |
| **206** | Gietz |
| **149** | Zlotkey |
| **174** | Abel |
| **176** | Taylor |
| **201** | Hartstein |
| **202** | Fay |

...

**MANAGER_ID in the WORKER table is equal to EMPLOYEE_ID in the MANAGER table**

ORACLE Academy

# Self-Joins Using the ON Clause

- The ON clause can also be used to join columns that have different names, within the same table or in a different table

```
SELECT  worker.last_name emp, manager.last_name mgr
FROM    employees worker JOIN employees manager
ON      (worker.manager_id = manager.employee_id);
```

| EMP | MGR |
|---|---|
| Kochhar | King |
| De Haan | King |
| Zlotkey | King |
| Mourgos | King |
| Hartstein | King |

...

# Nonequijoins

- The JOB_GRADES table defines the LOWEST_SAL and HIGHEST_SAL range of values for each GRADE_LEVEL

```
SELECT   *
FROM     job_grades;
```

| GRADE_LEVEL | LOWEST_SAL | HIGHEST_SAL |
|---|---|---|
| A | 1000 | 2999 |
| B | 3000 | 5999 |
| C | 6000 | 9999 |
| D | 10000 | 14999 |
| E | 15000 | 24999 |
| F | 25000 | 40000 |

# Nonequijoins

- Therefore, the GRADE_LEVEL column can be used to assign grade levels to each employee based on their salary

**EMPLOYEES**

| EMPLOYEE_ID | SALARY |
|---|---|
| 100 | 24000 |
| 101 | 17000 |
| 102 | 17000 |
| 200 | 4400 |
| 205 | 12000 |
| 206 | 8300 |
| 149 | 10500 |
| 174 | 11000 |
| 176 | 8600 |
| 178 | 7000 |

**JOB_GRADES**

| GRADE_LEVEL | LOWEST_SAL | HIGHEST_SAL |
|---|---|---|
| A | 1000 | 2999 |
| B | 3000 | 5999 |
| C | 6000 | 9999 |
| D | 10000 | 14999 |
| E | 15000 | 24999 |
| F | 25000 | 40000 |

# Retrieving Records with Nonequijoins

- This example creates a nonequijoin to evaluate an employee's salary grade. The salary must be between any pair of the low and high salary ranges

```
SELECT e.last_name, e.salary, j.grade_level
FROM employees e JOIN job_grades j
ON e.salary BETWEEN j.lowest_sal AND j.highest_sal;
```

| LAST_NAME | SALARY | GRADE_LEVEL |
|---|---|---|
| Vargas | 2500 | A |
| Matos | 2600 | A |
| Davies | 3100 | B |
| Rajs | 3500 | B |
| Lorentz | 4200 | B |
| Whalen | 4400 | B |

...

# Cartesian Products

- A Cartesian product is when all combinations of rows are displayed. All rows in the first table are joined to all rows in the second table

```
SELECT last_name, department_name
FROM   employees, departments;
```

- A Cartesian product is formed when a join condition is omitted or invalid
- Always include a valid join condition if you want to avoid a Cartesian product

```
SELECT last_name, department_name
FROM   employees e, departments d
ON     (e.department_id = d.department_id);
```

# Generating a Cartesian Product

**EMPLOYEES (40 rows)**

| EMPLOYEE_ID | LAST_NAME | DEPARTMENT_ID |
|---|---|---|
| 100 | King | 90 |
| 149 | Zlotkey | 80 |
| 103 | Ernst | 60 |
| ... | | |

**DEPARTMENTS (9 rows)**

| DEPARTMENT_ID | DEPARTMENT_NAME | LOCATION_ID |
|---|---|---|
| 60 | IT | 1400 |
| 80 | Sales | 2500 |
| 90 | Executive | 1700 |
| ... | | |

**Cartesian product:**

**40 x 9 = 360 rows**

| EMPLOYEE_ID | LAST_NAME ... | DEPARTMENT_ID | DEPARTMENT_NAME | LOCATION_ID |
|---|---|---|---|---|
| 100 | King | 90 | **Administration** | **1700** |
| 101 | Kochhar | 90 | **Administration** | **1700** |
| 102 | De Haan | 90 | **Administration** | **1700** |
| 200 | Whalen | 10 | **Administration** | **1700** |
| 205 | Higgins | 110 | **Administration** | **1700** |
| 206 | Gietz | 110 | **Administration** | **1700** |
| 149 ... | Zlotkey | 80 | **Administration** | **1700** |
| ... | | | | |

ORACLE Academy

# Summary

- In this lesson, you should have learned how to:
  - Write SELECT statement to access data from more than one table using equijoins and non-equijoins
  - Use a self-join to join a table to itself
  - Generate a Cartesian product )cross join) of all rows from two or more tables

# Reporting Aggregated Data Using the Group Functions

# Types of Group Functions

- In SQL, the following group functions can operate on a whole table or on a specific grouping of rows.
- Each function returns ONE result
  - **AVG**
  - **COUNT**
  - **MAX**
  - **MIN**
  - **SUM**


| Function | Description |
|---|---|
| AVG([DISTINCT\|ALL]*n*) | Average value of *n*, ignoring null values |
| COUNT({*\|[DISTINCT\|ALL]*expr*}) | Number of rows, where *expr* evaluates to |

# GROUP Functions List

- MIN: Used with columns that store any data type to return the minimum value
- MAX: Used with columns that store any data type to return the maximum value

| DEPT_ID | SALARY |
|---|---|
| 90 | 24000 |
| 90 | 17000 |
| 90 | 17000 |
| 60 | 9000 |
| 60 | 6000 |
| 60 | 4200 |
| 50 | 5800 |
| 50 | 3500 |
| 50 | 3100 |
| 50 | 2600 |
| 50 | 2500 |
| ... | ... |
| | 7000 |
| 10 | 4400 |

```
SELECT MAX(salary)
FROM employees;
```

| MAX (SALARY) |
|---|
| 24000 |

# GROUP Functions List

- SUM: Used with columns that store numeric data to find the total or sum of values
- AVG: Used with columns that store numeric data to compute the average

| DEPT_ID | SALARY |
|---|---|
| 90 | 24000 |
| 90 | 17000 |
| 90 | 17000 |
| 60 | 9000 |
| 60 | 6000 |
| 60 | 4200 |
| 50 | 5800 |
| 50 | 3500 |
| 50 | 3100 |
| 50 | 2600 |
| 50 | 2500 |
| ... | ... |
| | 7000 |
| 10 | 4400 |

```
SELECT MAX(salary)
FROM employees;
```

| MAX (SALARY) |
|---|
| 24000 |

# GROUP Functions List

- COUNT: Returns the number of rows
- VARIANCE: Used with columns that store numeric data to calculate the spread of data around the mean
  - For example, if the average grade for the class on the last test was 82% and the student's scores ranged from 40% to 100%, the variance of scores would be greater than if the student's scores ranged from 78% to 88%
- STDDEV: Similar to variance, standard deviation measures the spread of data
  - For two sets of data with approximately the same mean, the greater the spread, the greater the standard deviation





Standard deviation and variance measure the spread of data around the mean or average. For the purpose of this course, it is important to be able to recognize and use them as group functions. Understanding how they work is beyond the scope of this course.

# GROUP Functions SELECT Clause

- Group functions are written in the SELECT clause:

```
SELECT  column,
group_function(column),
..
FROM table
WHERE condition
GROUP BY column;
```

- What are Group Functions?
- Group Functions operate on sets of rows to give one result per group


ORACLE Academy



The WHERE clause can be included to perform a group function on a sub-set of the table, for example WHERE department_id = 90.

The GROUP BY clause will be discussed in a later lesson.

# GROUP Function Cautions

- Important things you should know about group functions:
  - Group functions cannot be used in the WHERE clause:

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

ORA-00934: group function is not allowed here







We will cover how to resolve this issue in a later lesson.

# GROUP Function examples

- MIN: Used with columns that store any data type to return the minimum value

| Example: | Result |
|---|---|
| SELECT MIN(life_expect_at_birth)<br>AS "Lowest Life Exp"<br>FROM wf_countries; | 32.62 |
| SELECT MIN(country_name)<br>FROM wf_countries; | Anguilla |
| SELECT MIN(hire_date)<br>FROM employees; | 17-Jun-1987 |





Example 1 returns the lowest number in the life_expect_at_birth column.

Example 2 uses a character data column, and returns the county whose name is first in the alphabetic list of country names.

Example 3 uses a date data type column and returns the earliest hire date.

# GROUP Function examples

- MAX: Used with columns that store any data type to return the maximum value

| Example: | Result |
|---|---|
| SELECT MAX(life_expect_at_birth)<br>AS "Highest Life Exp"<br>FROM wf_countries; | 83.51 |
| SELECT MAX(country_name)<br>FROM wf_countries | Western Sahara |
| SELECT MAX(hire_date)<br>FROM employees; | 29-Jan-2000 |





Example 1 returns the highest number in the life_expect_at_birth column.

Example 2 uses a character data column, and returns the county whose name is last in the alphabetic list of country names.

Example 3 uses a date data type column and returns the most recent hire date.

# GROUP Function examples

- SUM: Used with columns that store numeric data to find the total or sum of values

| Example: | Result |
|---|---|
| SELECT SUM(area)<br>FROM wf_countries | 241424 |
| SELECT SUM (salary)<br>FROM employees<br>WHERE department_id = 90; | 58000 |

ORACLE Academy







You can restrict the group function to a subset of the table using a WHERE clause.

Example 1 returns the total (sum) of all the areas of countries in region 29 (Caribbean).

Example 2 returns the total salary for employees in department 90.

# GROUP Function examples

- AVG: Used with columns that store numeric data to compute the average

| Example: | Result |
|---|---|
| SELECT AVG(area)<br>FROM wf_countries<br>WHERE region_id = 29; | 9656.96 |
| SELECT ROUND(AVG(salary), 2)<br>FROM employees<br>WHERE department_id = 90; | 19333.33 |





Example 1 returns the average of all the areas of countries in region 29 (Caribbean).

Example 2 returns the average salary for employees in department 90, rounded to two decimal places.

- VARIANCE: Used with columns that store numeric data to calculate the spread of data around the mean
- STDDEV: Similar to variance, standard deviation measures the spread of data

| Example: | Result |
|---|---|
| SELECT ROUND(VARIANCE(life_expect_at_birth),4)<br>FROM wf_countries; | 143.2394 |
| SELECT ROUND(STDDEV(life_expect_at_birth), 4)<br>FROM wf_countries; | 11.9683 |

# GROUP Function and NULL

- Group functions ignore NULL values
- In the example below, the null values were not used to find the average commission_pct

| LAST_NAME | COMMISSION_PCT |
|---|---|
| King | - |
| Kochhar | - |
| De Haan | - |
| Whalen | - |
| Higgins | - |
| Gietz | - |
| Zlotkey | .2 |
| Abel | .3 |
| Taylor | .2 |
| Grant | .15 |
| Mourgos | - |
| ... | ... |

```
SELECT AVG(commission_pct)
FROM employees;
```

| AVG(COMMISSION_PCT) |
|---|
| .2125 |

ORACLE Academy







The employees table has 20 rows. Only 4 employees have a commission_pct, the other 16 rows contain NULL. The average is calculated by finding the SUM of the not-null rows, and dividing by the COUNT of the not null rows.

This topic will be covered in more depth in the next lesson.

# More Than One Group Function

- You can have more than one group function in the SELECT clause, on the same or different columns

```
SELECT MAX(salary), MIN(salary), MIN(employee_id)
FROM employees
WHERE department_id = 60;
```

| MAX(SALARY) | MIN(SALARY) | MIN(EMPLOYEE_ID) |
|---|---|---|
| 9000 | 4200 | 103 |

# Rules for Group Functions

- Group functions ignore null values
- Group functions cannot be used in the WHERE clause
- MIN, MAX and COUNT can be used with any data type; SUM, AVG, STDDEV, and VARIANCE can be used only with numeric data types

# Summary

- In this lesson, you should have learned how to:
  - Define and give an example of the seven group functions: SUM, AVG, COUNT, MIN, MAX, STDDEV, VARIANCE
  - Construct and execute a SQL query using group functions
  - Construct and execute group functions that operate only with numeric data types

- Group functions:
  - Types and syntax
  - Use `AVG`, `SUM`, `MIN`, `MAX`, `COUNT`
  - Use `DISTINCT` keyword within group functions
  - `NULL` values in a group function
- Grouping rows:
  - `GROUP BY` clause
  - `HAVING` clause
- Nesting group functions

# GROUP BY Use

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

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

| DEPARTMENT_ID | AVG(SALARY) |
|---|---|
| 10 | 4400 |
| 20 | 9500 |
| 50 | 3500 |
| 60 | 6400 |
| 80 | 10033.3333333333333333 |
| 90 | 19333.3333333333333333 |
| 110 | 10150 |
| - | 7000 |

# GROUP BY Use

- In the SELECT statement shown, the rows are being grouped by department_id
- The AVG function is then applied to each group

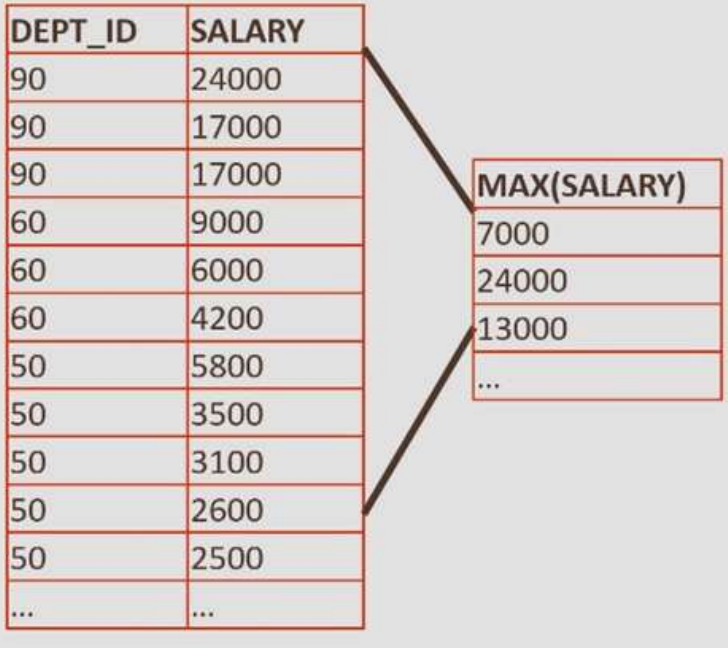
```
SELECT department_id, AVG(salary)
FROM employees
GROUP BY department_id
ORDER BY department_id;
```

| DEPARTMENT_ID | AVG(SALARY) |
|---|---|
| 10 | 4400 |
| 20 | 9500 |
| 50 | 3500 |
| 60 | 6400 |
| 80 | 10033.3333333333333333 |
| 90 | 19333.3333333333333333 |
| 110 | 10150 |
| - | 7000 |

# GROUP BY Example

- What if we wanted to find the maximum salary of employees in each department?
- We use a GROUP BY clause stating which column to use to group the rows

```
SELECT MAX(salary)
FROM employees
GROUP BY department_id;
```



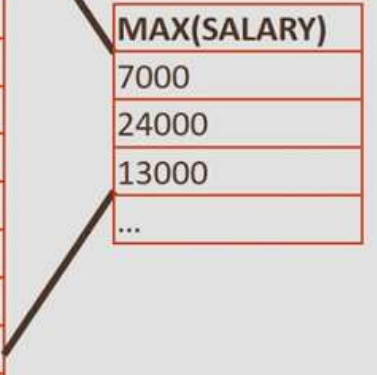
| DEPT_ID | SALARY |
|---|---|
| 90 | 24000 |
| 90 | 17000 |
| 90 | 17000 |
| 60 | 9000 |
| 60 | 6000 |
| 60 | 4200 |
| 50 | 5800 |
| 50 | 3500 |
| 50 | 3100 |
| 50 | 2600 |
| 50 | 2500 |
| ... | ... |

| MAX(SALARY) |
|---|
| 7000 |
| 24000 |
| 13000 |
| ... |

# GROUP BY Example

- But how can we tell which maximum salary belongs to which department?

| DEPT_ID | SALARY |
|---|---|
| 90 | 24000 |
| 90 | 17000 |
| 90 | 17000 |
| 60 | 9000 |
| 60 | 6000 |
| 60 | 4200 |
| 50 | 5800 |
| 50 | 3500 |
| 50 | 3100 |
| 50 | 2600 |
| 50 | 2500 |
| ... | ... |

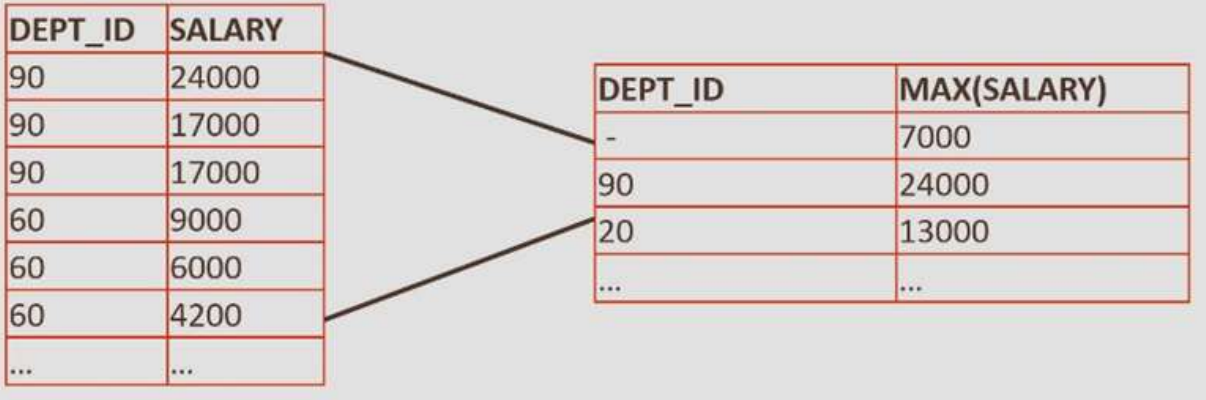


| MAX(SALARY) |
|---|
| 7000 |
| 24000 |
| 13000 |
| ... |

# GROUP BY in SELECT

- Usually we want to include the GROUP BY column in the SELECT list

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



| DEPT_ID | SALARY |
|---|---|
| 90 | 24000 |
| 90 | 17000 |
| 90 | 17000 |
| 60 | 9000 |
| 60 | 6000 |
| 60 | 4200 |
| ... | ... |

| DEPT_ID | MAX(SALARY) |
|---|---|
| - | 7000 |
| 90 | 24000 |
| 20 | 13000 |
| ... | ... |

# GROUP BY Clause

- Group functions require that any column listed in the SELECT clause that is not part of a group function must be listed in a GROUP BY clause
- What is wrong with this example?

```
SELECT job_id, last_name, AVG(salary)
FROM employees
GROUP BY job_id;
```

ORA-00979: not a GROUP BY expression

ORACLE Academy







Job_id is fine in the SELECT list, but last_name is not, because each unique group of job_id's produces only one row of output (as it is the GROUP BY column). However, there may be many different employees having that same job_id, for example there are three employees with a job_id of SA_REP.

# COUNT

- This example shows how many countries are in each region
- Remember that group functions ignore null values, so if any country does not have a country name, it will not be included in the COUNT

```
SELECT COUNT(country_name), region_id
FROM wf_countries
GROUP BY region_id
ORDER BY region_id;
```

| COUNT(COUNTRY_NAME) | REGION_ID |
|---|---|
| 15 | 5 |
| 28 | 9 |
| 21 | 11 |
| 8 | 13 |
| 7 | 14 |
| 8 | 15 |
| 5 | 17 |
| 17 | 18 |

# COUNT

- Of course this is unlikely, but when constructing SQL statements, we have to think about all of the possibilities
- It would be better to write the query using COUNT(*):

```
SELECT COUNT(*), region_id
FROM wf_countries
GROUP BY region_id
ORDER BY region_id;
```

- This would count all of the rows in each region group, without the need to check which columns contained NULL values

# WHERE Clause

- We can also use a WHERE clause to exclude rows before the remaining rows are formed into groups

```
SELECT department_id, MAX(salary)
FROM employees
WHERE last_name != 'King'
GROUP BY department_id;
```

| LAST_NAME | DEPT_ID | SALARY |
|---|---|---|
| King | 90 | 24000 |
| Kochhar | 90 | 17000 |
| De Haan | 90 | 17000 |
| Hunold | 60 | 9000 |
| Ernst | 60 | 6000 |
| Lorentz | 60 | 4200 |
| ... | ... | ... |

| DEPT_ID | MAX(SALARY) |
|---|---|
| - | 7000 |
| 90 | 17000 |
| 20 | 13000 |
| ... | ... |







As employee King is excluded by the WHERE clause, the MAX(salary) for department 90 is returned as 17000.

# More GROUP BY Examples

- Show the average population of all countries in each region
- Round the average to a whole number

```
SELECT region_id, ROUND(AVG(population)) AS population
FROM wf_countries
GROUP BY region_id
ORDER BY region_id;
```

- Count the number of spoken languages for all countries

```
SELECT country_id, COUNT(language_id) AS "Number of
languages"
FROM wf_spoken_languages
GROUP BY country_id;
```

# GROUP BY Guidelines

- Important guidelines to remember when using a GROUP BY clause are:
  - If you include a group function (AVG, SUM, COUNT, MAX, MIN, STDDEV, VARIANCE) in a SELECT clause along with any other individual columns, each individual column must also appear in the GROUP BY clause
  - You cannot use a column alias in the GROUP BY clause
  - The WHERE clause excludes rows before they are divided into groups

# Groups Within GROUPS

- Sometimes you need to divide groups into smaller groups
- For example, you may want to group all employees by department; then, within each department, group them by job
- This example shows how many employees are doing each job within each department

```
SELECT department_id, job_id,
count(*)
FROM employees
WHERE department_id > 40
GROUP BY department_id, job_id;
```

| DEPT_ID | JOB_ID | COUNT(*) |
|---|---|---|
| 110 | AC_ACCOUNT | 1 |
| 50 | ST_CLERK | 4 |
| 80 | SA_REP | 2 |
| 90 | AD_VP | 2 |
| 50 | ST_MAN | 1 |
| ... | ... | ... |

# Nesting Group Functions

- Group functions can be nested to a depth of two when GROUP BY is used

```
SELECT max(avg(salary))
FROM employees
GROUP by department_id;
```

- How many values will be returned by this query?
- The answer is one – the query will find the average salary for each department, and then from that list, select the single largest value

# HAVING

- Suppose we want to find the maximum salary in each department, but only for those departments which have more than one employee?
- What is wrong with this example?

```
SELECT department_id, MAX(salary)
FROM employees
WHERE COUNT(*) > 1
GROUP BY department_id;
```

ORA-00934: group function is not allowed here

ORACLE Academy







A WHERE clause can be used only to include/exclude individual rows, not groups of rows. Therefore we cannot use group functions in a WHERE clause.

# HAVING

- In the same way you used the WHERE clause to restrict the rows that you selected, you can use the HAVING clause to restrict groups
- In a query using a GROUP BY and HAVING clause, the rows are first grouped, group functions are applied, and then only those groups matching the HAVING clause are displayed

# HAVING

- The WHERE clause is used to restrict rows; the HAVING clause is used to restrict groups returned from a GROUP BY clause

```
SELECT department_id,MAX(salary)
FROM employees
GROUP BY department_id
HAVING COUNT(*)>1
ORDER BY department_id;
```

| DEPARTMENT_ID | MAX(SALARY) |
|---|---|
| 20 | 13000 |
| 50 | 5800 |
| 60 | 9000 |
| 80 | 11000 |
| 90 | 24000 |
| 110 | 12000 |







The query first finds the MAX salary for each department in the employees table. The HAVING clause then restricts the groups returned to those departments that have more than 1 employee.

# HAVING

- This query finds the average population of the countries in each region
- It then only returns the region groups with a lowest population greater than three hundred thousand

```
SELECT region_id,
  ROUND(AVG(population))
FROM wf_countries
GROUP BY region_id
HAVING MIN(population)>300000
ORDER BY region_id;
```

| REGION_ID | ROUND(AVG(POPULATION)) |
|---|---|
| 14 | 27037687 |
| 17 | 18729285 |
| 30 | 193332379 |
| 34 | 173268273 |
| 143 | 12023602 |
| 145 | 8522790 |
| 151 | 28343051 |







The HAVING and GROUP BY clauses can use different columns. The example on the slide GROUPs BY region_id, but the HAVING clause restricts groups based on population.

# HAVING

- Although the HAVING clause can precede the GROUP BY clause in a SELECT statement, it is recommended that you place each clause in the order shown
- The ORDER BY clause (if used) is always last!

```
SELECT column, group_function
FROM table
WHERE
GROUP BY
HAVING
ORDER BY
```

# Summary

- In this lesson, you should have learned how to:
  - Construct and execute a SQL query using GROUP BY
  - Construct and execute a SQL query using GROUP BY HAVING
  - Construct and execute a GROUP BY on more than one column
  - Nest group functions

# Manipulating Data

- Adding new rows in a table
  - `INSERT` statement
- Changing data in a table
  - `UPDATE` statement
- Removing rows from a table:
  - `DELETE` statement

# Data Manipulation Language

- A DML statement is executed when you:
  - Add new rows to a table
  - Modify existing rows in a table
  - Remove existing rows from a table
- A *transaction* consists of a collection of DML statements that form a logical unit of work.

# Adding a New Row to a Table

| | | | | |
|---|---|---|---|---|
| | 70 | Public Relations | 100 | 1700 |

New row

DEPARTMENTS

| | DEPARTMENT_ID | DEPARTMENT_NAME | MANAGER_ID | LOCATION_ID |
|---|---|---|---|---|
| 1 | 10 | Administration | 200 | 1700 |
| 2 | 20 | Marketing | 201 | 1800 |
| 3 | 50 | Shipping | 124 | 1500 |
| 4 | 60 | IT | 103 | 1400 |
| 5 | 80 | Sales | 149 | 2500 |
| 6 | 90 | Executive | 100 | 1700 |
| 7 | 110 | Accounting | 205 | 1700 |
| 8 | 190 | Contracting | (null) | 1700 |

Insert new row into the DEPARTMENTS table.

| | DEPARTMENT_ID | DEPARTMENT_NAME | MANAGER_ID | LOCATION_ID |
|---|---|---|---|---|
| 1 | 10 | Administration | 200 | 1700 |
| 2 | 20 | Marketing | 201 | 1800 |
| 3 | 50 | Shipping | 124 | 1500 |
| 4 | 60 | IT | 103 | 1400 |
| 5 | 80 | Sales | 149 | 2500 |
| 6 | 90 | Executive | 100 | 1700 |
| 7 | 110 | Accounting | 205 | 1700 |
| 8 | 190 | Contracting | (null) | 1700 |

| | | | | |
|---|---|---|---|---|
| 9 | 70 | Public Relations | 100 | 1700 |

# INSERT Statement Syntax

- Add new rows to a table by using the INSERT statement:

```
INSERT INTO  table [(column [, column...])]
VALUES       (value [, value...]);
```

- With this syntax, only one row is inserted at a time.

# Inserting New Rows

- Insert a new row containing values for each column.
- List values in the default order of the columns in the table.

```
INSERT INTO departments(department_id,
       department_name, manager_id, location_id)
VALUES (70, 'Public Relations' , 100, 1700);
```

- Enclose character and date values within single quotation marks.

# Inserting Rows with Null Values

- Implicit method: Omit the column from the column list.

```
INSERT INTO  departments (department_id,
                          department_name)
VALUES       (30, 'Purchasing');
```

- Explicit method: Specify the NULL keyword in the VALUES clause.

```
INSERT INTO  departments
VALUES       (100, 'Finance', NULL, NULL);
```

| Method | Description |
|---|---|
| Implicit | Omit the column from the column list. |
| Explicit | Specify the NULL keyword in the VALUES list; specify the empty string ('') in the VALUES list for character strings and dates |

# Inserting Special Values

- The `SYSDATE` function records the current date and time.

```
INSERT INTO employees (employee_id,
                       first_name, last_name,
                       email, phone_number,
                       hire_date, job_id, salary,
                       commission_pct, manager_id,
                       department_id)
VALUES                (113,
                       'Louis', 'Popp',
                       'LPOPP', '515.124.4567',
                       SYSDATE, 'AC_ACCOUNT', 6900,
                       NULL, 205, 110);
```

# Inserting Specific Date and Time Values

– Add a new employee.

```
INSERT INTO employees
VALUES       (114,
              'Den', 'Raphealy',
              'DRAPHEAL', '515.127.4561',
              TO_DATE('FEB 3, 1999', 'MON DD,
YYYY'),
              'SA_REP', 11000, 0.2, 100, 60);
```

| EMPLOYEE_ID | FIRST_NAME | LAST_NAME | EMAIL | PHONE_NUMBER | HIRE_DATE | JOB_ID | SALARY | COMMISSION_PCT |
|---|---|---|---|---|---|---|---|---|
| 114 | Den | Raphealy | DRAPHEAL | 515.127.4561 | 03-FEB-99 | SA_REP | 11000 | 0.2 |

# Copying Rows from Another Table

- Write your `INSERT` statement with a subquery:

```
INSERT INTO sales_reps(id, name, salary, commission_pct)
  SELECT employee_id, last_name, salary, commission_pct
  FROM   employees
  WHERE  job_id LIKE '%REP%';
```

- Do not use the `VALUES` clause.
- Match the number of columns in the `INSERT` clause to those in the subquery.
- Inserts all the rows returned by the subquery in the table, `sales_reps`.

# Changing Data in a Table

EMPLOYEES

| EMPLOYEE_ID | FIRST_NAME | LAST_NAME | SALARY | MANAGER_ID | COMMISSION_PCT | DEPARTMENT_ID |
|---|---|---|---|---|---|---|
| 100 | Steven | King | 24000 | (null) | (null) | 90 |
| 101 | Neena | Kochhar | 17000 | 100 | (null) | 90 |
| 102 | Lex | De Haan | 17000 | 100 | (null) | 90 |
| 103 | Alexander | Hunold | 9000 | 102 | (null) | 60 |
| 104 | Bruce | Ernst | 6000 | 103 | (null) | 60 |
| 107 | Diana | Lorentz | 4200 | 103 | (null) | 60 |
| 124 | Kevin | Mourgos | 5800 | 100 | (null) | 50 |

Update rows in the EMPLOYEES table:

| EMPLOYEE_ID | FIRST_NAME | LAST_NAME | SALARY | MANAGER_ID | COMMISSION_PCT | DEPARTMENT_ID |
|---|---|---|---|---|---|---|
| 100 | Steven | King | 24000 | (null) | (null) | 90 |
| 101 | Neena | Kochhar | 17000 | 100 | (null) | 90 |
| 102 | Lex | De Haan | 17000 | 100 | (null) | 90 |
| 103 | Alexander | Hunold | 9000 | 102 | (null) | 80 |
| 104 | Bruce | Ernst | 6000 | 103 | (null) | 80 |
| 107 | Diana | Lorentz | 4200 | 103 | (null) | 80 |
| 124 | Kevin | Mourgos | 5800 | 100 | (null) | 50 |

# UPDATE Statement Syntax

- Modify existing values in a table with the UPDATE statement:

```
UPDATE      table
SET         column = value [, column = value,
...]
[WHERE            condition];
```

- Update more than one row at a time (if required).

# Updating Rows in a Table

– Values for a specific row or rows are modified if you specify the `WHERE` clause:

```
UPDATE employees
SET    department_id = 50
WHERE  employee_id = 113;
```

– Values for all the rows in the table are modified if you omit the `WHERE` clause:

```
UPDATE       copy_emp
SET          department_id = 110;
```

– Specify `SET` *`column_name`*`= NULL` to update a column value to `NULL`.

# Updating Two Columns with a Subquery

- Update employee 113's job and salary to match those of employee 205.

```
UPDATE   employees
SET      job_id   = (SELECT   job_id
                     FROM     employees
                     WHERE    employee_id = 205),
         salary   = (SELECT   salary
                     FROM     employees
                     WHERE    employee_id = 205)
WHERE    employee_id    =  113;
```

# Updating Rows Based on Another Table

- Use the subqueries in the `UPDATE` statements to update row values in a table based on values from another table:

```
UPDATE   copy_emp
SET      department_id  =   (SELECT department_id
                            FROM employees
                            WHERE employee_id =
100)
WHERE    job_id         =   (SELECT job_id
                            FROM employees
                            WHERE employee_id =
200);
```

# Removing a Row from a Table

DEPARTMENTS

| | DEPARTMENT_ID | DEPARTMENT_NAME | MANAGER_ID | LOCATION_ID |
|---|---|---|---|---|
| 1 | 10 | Administration | 200 | 1700 |
| 2 | 20 | Marketing | 201 | 1800 |
| 3 | 50 | Shipping | 124 | 1500 |
| 4 | 60 | IT | 103 | 1400 |
| 5 | 80 | Sales | 149 | 2500 |
| 6 | 90 | Executive | 100 | 1700 |
| 7 | 110 | Accounting | 205 | 1700 |
| 8 | 190 | Contracting | (null) | 1700 |

Delete a row from the DEPARTMENTS table:

| | DEPARTMENT_ID | DEPARTMENT_NAME | MANAGER_ID | LOCATION_ID |
|---|---|---|---|---|
| 1 | 10 | Administration | 200 | 1700 |
| 2 | 20 | Marketing | 201 | 1800 |
| 3 | 50 | Shipping | 124 | 1500 |
| 4 | 60 | IT | 103 | 1400 |
| 5 | 80 | Sales | 149 | 2500 |
| 6 | 90 | Executive | 100 | 1700 |
| 7 | 110 | Accounting | 205 | 1700 |

# DELETE Statement

- You can remove existing rows from a table by using the DELETE statement:

```
DELETE [FROM]       table
[WHERE  condition];
```

# Deleting Rows from a Table

– Specific rows are deleted if you specify the

```
DELETE FROM departments
WHERE  department_name = 'Finance';
```

– All rows in the table are deleted if you omit the `WHERE` clause:

```
DELETE FROM  copy_emp;
```

# Deleting Rows Based on Another Table

- Use the subqueries in the `DELETE` statements to remove rows from a table based on values from another table:

```
DELETE FROM employees
WHERE  department_id =
                  (SELECT department_id
                   FROM   departments
                   WHERE  department_name
                          LIKE '%Public%');
```

Remove employees from departments that has the word ‘Public’ in their name

# Summary

- In this lesson, you should have learned how to use the following statements:

| Function | Description |
| --- | --- |
| `INSERT` | **Adds a new row to the table** |
| `UPDATE` | **Modifies existing rows in the table** |
| `DELETE` | **Removes existing rows from the table** |