# Advanced SQL Programming

# Advanced SQL Statements

**UNION**

- The purpose of the SQL UNION query is to combine the results of two queries together.
- UNION is somewhat similar to JOIN in that they are both used to related information from multiple tables.
- One restriction of UNION is that all corresponding columns need to be of the same data type.
- Also, when using UNION, only distinct values are selected (similar to SELECT DISTINCT).

Eg.

[SQL Statement 1]
UNION
[SQL Statement 2]

# Union

| store_name | Sales | Date |
|---|---|---|
| Los Angeles | $1500 | Jan-05-1999 |
| San Diego | $250 | Jan-07-1999 |
| Los Angeles | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

Table ***Store_Information***

| Date | Sales |
|---|---|
| Jan-07-1999 | $250 |
| Jan-10-1999 | $535 |
| Jan-11-1999 | $320 |
| Jan-12-1999 | $750 |

Table ***Internet_Sales***

```
SELECT Date FROM Store_Information
UNION
SELECT Date FROM Internet_Sales
```

| Date |
|---|
| Jan-05-1999 |
| Jan-07-1999 |
| Jan-08-1999 |
| Jan-10-1999 |
| Jan-11-1999 |
| Jan-12-1999 |

# Union All

- The purpose of the SQL UNION ALL command is also to combine the results of two queries together.
- The difference between UNION ALL and UNION is that, while UNION only selects distinct values, UNION ALL selects all values.

[SQL Statement 1]
UNION ALL
[SQL Statement 2]

```
SELECT Date FROM Store_Information
UNION ALL
SELECT Date FROM Internet_Sales
```

| Date |
| --- |
| Jan-05-1999 |
| Jan-07-1999 |
| Jan-08-1999 |
| Jan-08-1999 |
| Jan-07-1999 |
| Jan-10-1999 |
| Jan-11-1999 |
| Jan-12-1999 |

# Intersect

- Similar to the UNION command, INTERSECT also operates on two SQL statements.
- The difference is that, while UNION essentially acts as an OR operator (value is selected if it appears in either the first or the second statement), the INTERSECT command acts as an AND operator (value is selected only if it appears in both statements).

[SQL Statement 1]
INTERSECT
[SQL Statement 2]

```
SELECT Date FROM Store_Information
INTERSECT
SELECT Date FROM Internet_Sales
```

| Date |
| --- |
| Jan-07-1999 |

# Minus

- The MINUS operates on two SQL statements.
- It takes all the results from the first SQL statement, and then subtract out the ones that are present in the second SQL statement to get the final answer.
- If the second SQL statement includes results not present in the first SQL statement, such results are ignored.

SQL Statement 1]
MINUS
[SQL Statement 2]

```
SELECT Date FROM Store_Information
MINUS
SELECT Date FROM Internet_Sales
```

| Date |
| --- |
| Jan-05-1999 |
| Jan-08-1999 |

# Limit

- Sometimes we may not want to retrieve all the records that satisfy the criteria specified in WHERE or HAVING clauses.

SQL Statement 1]
LIMIT [N]

```
SELECT store_name, Sales, Date
FROM Store_Information
ORDER BY Sales DESC
LIMIT 2;
```

| store_name | Sales | Date |
|---|---|---|
| Los Angeles | $1500 | Jan-05-1999 |
| Boston | $700 | Jan-08-1999 |

# Top

- In the previous section, we saw how LIMIT can be used to retrieve a subset of records in MySQL.
- In Microsoft SQL Server, this is accomplished using the TOP keyword. The syntax for TOP is as follows:
  SELECT TOP [TOP argument] "column_name"
  FROM "table_name"
  where [TOP argument] can be one of two possible types:
  1. [N]: The first N records are returned.
  2. [N'] PERCENT: The number of records corresponding to N'% of all qualifying records are returned

**SELECT TOP 2 store_name, Sales, Date**
**FROM Store_Information ORDER BY Sales DESC;**

| store_name | Sales | Date |
|---|---|---|
| Los Angeles | $1500 | Jan-05-1999 |
| Boston | $700 | Jan-08-1999 |

**SELECT TOP 25 PERCENT store_name, Sales, Date**
**FROM Store_Information ORDER BY Sales DESC;**

# Subquery

- It is possible to embed a SQL statement within another.
- When this is done on the WHERE or the HAVING statements, we have a subquery construct.

```
SELECT "column_name1"
FROM "table_name1"
WHERE "column_name2" [Comparison Operator]
(SELECT "column_name3"
FROM "table_name2"
WHERE [Condition])
```

[Comparison Operator] could be equality operators such as =, >, <, >=, <=. It can also be a text operator such as "LIKE". The portion in red is considered as the "inner query", while the portion in green is considered as the "outer query".

```
SELECT SUM(Sales) FROM Store_Information
WHERE Store_name IN
(SELECT store_name FROM Geography
WHERE region_name = 'West')
```

# Exists

- In the previous section, we used IN to link the inner query and the outer query in a subquery statement.
- IN is not the only way to do so -- one can use many operators such as >, <, or =.
- EXISTS is a special operator that we will discuss in this section.
- EXISTS simply tests whether the inner query returns any row.
- If it does, then the outer query proceeds.
- If not, the outer query does not execute, and the entire SQL statement returns nothing.

```
SELECT "column_name1"
FROM "table_name1"
WHERE EXISTS
(SELECT *
FROM "table_name2"
WHERE [Condition])
```

```
SELECT SUM(Sales) FROM Store_Information
WHERE EXISTS
(SELECT * FROM Geography
WHERE region_name = 'West')
```

# Case

- CASE is used to provide if-then-else type of logic to SQL.

syntax is:

```
SELECT CASE ("column_name")
  WHEN "condition1" THEN "result1"
  WHEN "condition2" THEN "result2"
  ...
  [ELSE "resultN"]
  END
FROM "table_name"
```

```
SELECT store_name, CASE store_name
  WHEN 'Los Angeles' THEN Sales * 2
  WHEN 'San Diego' THEN Sales * 1.5
  ELSE Sales
  END
"New Sales",
Date
FROM Store_Information
```

"condition" can be a static value or an expression.

The ELSE clause is optional.

| store_name | New Sales | Date |
|---|---|---|
| Los Angeles | $3000 | Jan-05-1999 |
| San Diego | $375 | Jan-07-1999 |
| San Francisco | $300 | Jan-08-1999 |
| Boston | $700 | Jan-08-1999 |

# Null

- In SQL, NULL means that data does not exist.
- NULL does not equal to 0 or an empty string.
- Both 0 and empty string represent a value, while NULL has no value.
  Any mathematical operations performed on NULL will result in NULL.
  10 + NULL = NULL
  Aggregate functions such as SUM, COUNT, AVG, MAX, and MIN exclude NULL values.
- This is not likely to cause any issues for SUM, MAX, and MIN.
- This can lead to confusion with AVG and COUNT.

# ISNULL Function

- The ISNULL function is available in both SQL Server and MySQL. However, their uses are different:

  In SQL Server, the ISNULL() function is used to replace NULL value with another value.

  Table Sales_Data
  store_name  Sales
  Store A  300
  Store B  NULL

- SELECT SUM(ISNULL(Sales,100)) FROM Sales_Data;

returns 400. This is because NULL has been replaced by 100 via the ISNULL function

# IFNULL Function

- The IFNULL() function is available in MySQL, and not in SQL Server or Oracle.
- This function takes two arguments.
- The first argument is not NULL, the function returns the first argument.
- The second argument is returned.
- This function is commonly used to replace NULL value with another value.
- It is similar to the NVL function in Oracle and the ISNULL Function in SQL Server

**SELECT SUM(IFNULL(Sales,100)) FROM Sales_Data;**
returns 400. This is because NULL has been replaced by 100 via the ISNULL function.

# NVL Function

- The NVL() function is available in Oracle, and not in MySQL or SQL Server.
- This function is used to replace NULL value with another value.
- It is similar to the IFNULL Function in MySQL and the ISNULL Function in SQL Server.

Table: Sales_Data

| store_name | Sales |
|---|---|
| Store A | 300 |
| Store B | NULL |
| Store C | 150 |

**SELECT SUM(NVL(Sales,100)) FROM Sales_Data;** returns 550. This is because NULL has been replaced by 100 via the ISNULL function, hence the sum of the 3 rows is 300 + 100 + 150 = 550.

# Coalesce Function

- The COALESCE function in SQL returns the first non-NULL expression among its arguments.
- It is the same as the following CASE statement:

```
SELECT CASE ("column_name")
  WHEN "expression 1 is not NULL" THEN "expression 1"
  WHEN "expression 2 is not NULL" THEN "expression 2"
  ...
  [ELSE "NULL"]
  END
```

| Name | Business_Phone | Cell_Phone | Home_Phone |
|---|---|---|---|
| Jeff | 531-2531 | 622-7813 | 565-9901 |
| Laura | NULL | 772-5588 | 312-4088 |
| Peter | NULL | NULL | 594-7477 |

1. If a person has a business phone, use the business phone number.
2. If a person does not have a business phone and has a cell phone, use the cell phone number.
3. If a person does not have a business phone, does not have a cell phone, and has a home phone, use the home phone number.

We can use the **COALESCE** function to achieve our goal:

```
SELECT Name, COALESCE(Business_Phone, Cell_Phone, Home_Phone) Contact_Phone
FROM Contact_Info;
```

# NULLIF Function

- The NULLIF function takes two arguments.
- If the two arguments are equal, then NULL is returned. Otherwise, the first argument is returned.
  It is the same as the following CASE statement:

```
SELECT CASE ("column_name")
  WHEN "expression 1 = expression 2 " THEN "NULL"
  [ELSE "expression 1"]
  END
FROM "table_name"
```

| Store_name | Actual | Goal |
|---|---|---|
| Store A | 50 | 50 |
| Store B | 40 | 50 |
| Store C | 25 | 30 |

We want to show NULL if actual sales is equal to sales goal, and show actual sales if the two are different. To do this, we issue the following SQL statement:
**SELECT Store_name, NULLIF(Actual,Goal) FROM Sales_Data;**

# Rank

- Displaying the rank associated with each row is a common request, and there is no straightforward way to do so in SQL.
- To display rank in SQL, the idea is to do a self-join, list out the results in order, and do a count on the number of records that's listed ahead of (and including) the record of interest.
- Let's use an example to illustrate the following table

Table Total_Sales

| Name | Sales |
|---|---|
| John | 10 |
| Jennifer | 15 |
| Stella | 20 |
| Sophia | 40 |
| Greg | 50 |
| Jeff | 20 |

```
SELECT a1.Name, a1.Sales, COUNT(a2.sales) Sales_Rank
FROM Total_Sales a1, Total_Sales a2
WHERE a1.Sales <= a2.Sales or (a1.Sales=a2.Sales and
a1.Name = a2.Name)
GROUP BY a1.Name, a1.Sales
ORDER BY a1.Sales DESC, a1.Name DESC;
```

# Median

- To get the median, we need to be able to accomplish the following:

  Sort the rows in order and find the rank for each row.

  Determine what is the "middle" rank. For example, if there are 9 rows, the middle rank would be 5.

  Obtain the value for the middle-ranked row.

Table Total_Sales

| Name | Sales |
|---|---|
| John | 10 |
| Jennifer | 15 |
| Stella | 20 |
| Sophia | 40 |
| Greg | 50 |
| Jeff | 20 |

```
SELECT Sales Median FROM
(SELECT a1.Name, a1.Sales, COUNT(a1.Sales) Rank
FROM Total_Sales a1, Total_Sales a2
WHERE a1.Sales < a2.Sales OR (a1.Sales=a2.Sales AND
a1.Name <= a2.Name)
group by a1.Name, a1.Sales
order by a1.Sales desc) a3
WHERE Rank = (SELECT (COUNT(*)+1) /2 FROM
Total_Sales);
```

# Running Totals

- Displaying running totals is a common request, and there is no straightforward way to do so in SQL.
- The idea for using SQL to display running totals similar to that for displaying rank: first do a self-join, then, list out the results in order.
- Where as finding the rank requires doing a count on the number of records that's listed ahead of (and including) the record of interest, finding the running total requires summing the values for the records that's listed ahead of (and including) the record of interest.

| Name | Sales |
|---|---|
| John | 10 |
| Jennifer | 15 |
| Stella | 20 |
| Sophia | 40 |
| Greg | 50 |
| Jeff | 20 |

```
SELECT a1.Name, a1.Sales, SUM(a2.Sales) Running_Total
FROM Total_Sales a1, Total_Sales a2
WHERE a1.Sales <= a2.sales or (a1.Sales=a2.Sales and
a1.Name = a2.Name)
GROUP BY a1.Name, a1.Sales
ORDER BY a1.Sales DESC, a1.Name DESC
```

# Percent To Total

- To display percent to total in SQL, we want to leverage the ideas we used for rank/running total plus subquery.
- Different from what we saw in the SQL Subquery section, here we want to use the subquery as part of the SELECT.
- Let's use an example to illustrate. Say we have the following table

Table Total_Sales

| Name | Sales |
|---|---|
| John | 10 |
| Jennifer | 15 |
| Stella | 20 |
| Sophia | 40 |
| Greg | 50 |
| Jeff | 20 |

```
SELECT a1.Name, a1.Sales, a1.Sales/(SELECT SUM(Sales)
FROM Total_Sales) Pct_To_Total
FROM Total_Sales a1, Total_Sales a2
WHERE a1.Sales <= a2.sales or (a1.Sales=a2.Sales and
a1.Name = a2.Name)
GROUP BY a1.Name, a1.Sales
ORDER BY a1.Sales DESC, a1.Name DESC;
```