PIM Training Program

SQL

Advanced Datanet Features and Mathematical Aggregations

Learning Objective

At the end of the module, the participant should be able to understand advanced features of data net as well as perform mathematical aggregations within SQL query.

Agenda

- RECAP
- SQL
  - Aggregate Queries - Pulling DISTINCT Records
  - Aggregate Functions
  - Aggregate Functions with DISTINCT
  - GROUP BY Clause
  - HAVING Clause
- ETL
  - Partitions
  - Comments
  - USE_HASH
  - Scheduling Jobs
  - Publishing to Folders
  - Using the Job Wildcards
- Lesson 3: Assignment

SQL RECAP

SQL Recap – Select From

SELECT/FROM

```
SELECT
fc.REGION_ID
, fc.WAREHOUSE_ID
FROM D_WAREHOUSES fc
;
```

So far, we've started with SELECT and FROM, which allowed us to get data out of tables in data clusters across Redshift

SQL Recap – Where

WHERE

```
SELECT
fc.REGION_ID
, fc.WAREHOUSE_ID
FROM D_WAREHOUSES fc
WHERE fc.REGION_ID = 2
ORDER BY REGION_ID
;
```

Then we added ORDER BY, to sort the results.

And last we added WHERE, to filter the results to only those we want.

Aggregate Queries - Pulling DISTINCT Records

Aggregate Queries - Pulling DISTINCT Records

```
SELECT
fc.REGION_ID
, fc.WAREHOUSE_ID
FROM D_WAREHOUSES fc
WHERE fc.REGION_ID = 2
ORDER BY REGION_ID
;
```

Let's look at some ways to aggregate the results, since we don't always want the full details from the table. Sometimes, we want a summary of the contents of a table.

Aggregate Queries - Pulling DISTINCT Records

```
SELECT DISTINCT
fc.REGION_ID
FROM D_WAREHOUSES fc
;
```

The simplest form of aggregation is to use the keyword DISTINCT at the start of your SELECT statement.

Using distinct clause in a single column output or within count clause is advisable. Recommend using GROUP BY to obtain distinct values.

Aggregate Queries - Pulling DISTINCT Records

```
SELECT DISTINCT
fc.REGION_ID
FROM D_WAREHOUSES fc
;
```

REGION_ID
1
3
4
2

It returns one row for each unique combination of the columns listed in the SELECT clause.

So we learn there are 4 distinct values in the REGION_ID column.

If there's just one column in the SELECT clause, it's a list of all the DISTINCT values in that column.

Aggregate Queries - Pulling DISTINCT Records

```
SELECT DISTINCT
fc.REGION_ID
, fc.IS_DROPSHIP
FROM D_WAREHOUSES fc
;
```

REGION_ID	IS_DROPSHIP
1	N
1	Y
2	N
2	Y
3	N
3	Y
4	Y

If we add a second column, the results will return one row for each unique combination of the two columns.

Aggregate Queries - Pulling DISTINCT Records

```
SELECT DISTINCT
fc.REGION_ID
, fc.IS_DROPSHIP
FROM D_WAREHOUSES fc
;
```

REGION_ID	IS_DROPSHIP
1	N
1	Y
2	N
2	Y
3	N
3	Y
4	Y

For example: here we get one row for Region 1, Is Dropship N, and another row for Region 1, Is Dropship Y.

Aggregate Queries - Pulling DISTINCT Records

```
SELECT DISTINCT
fc.REGION_ID
, fc.IS_DROPSHIP
FROM D_WAREHOUSES fc
;
```

REGION_ID	IS_DROPSHIP
1	N
1	Y
2	N
2	Y
3	N
3	Y
4	Y

If we add a second column, the results will return one row for each unique combination of the two columns.

Reiterating the example: here we get one row for Region 1, Is Dropship N, and another row for Region 1, Is Dropship Y.

This use of the DISTINCT keyword applies to the entire SELECT statement, so it goes at the start. We'll use DISTINCT in another way later.

Aggregate Functions

Aggregate Functions

```
SELECT
COUNT(fc.REGION_ID)
FROM
D_WAREHOUSES fc
;
```

More complex is when we use Aggregate functions to COUNT or SUM, or determine the MIN, MAX, or AVG of all values in a column.

If we want to count how many REGION_ID values are in the table, we can wrap the column REGION_ID with the COUNT() function.

COUNT(FC.REGION_ID)
5096

Notice we get 5096 as the COUNT of REGION_ID.

Aggregate Functions

```
SELECT
COUNT(fc.REGION_ID)
FROM D_WAREHOUSES fc
;
```

COUNT(FC.REGION_ID)
5096

"We get 5096 as the COUNT of REGION_ID because we're counting how many rows have a non-NULL value in the REGION_ID column. We're NOT counting how many unique values are in that column.

Aggregate Functions with Distinct

Aggregate Functions with Distinct

```
SELECT
COUNT(DISTINCT fc.REGION_ID)
FROM D_WAREHOUSES fc
;
```

COUNT(DISTINCTFC.REGION_ID)
4

By adding the DISTINCT keyword INSIDE the COUNT() function – right before the column name – the function will now COUNT how many unique values it finds in that column.

Here, as expected, it returns 4 as the COUNT of DISTINCT values in the REGION_ID column.

Aggregate Functions with Distinct

```
SELECT
COUNT(DISTINCT fc.REGION_ID)
, SUM(fc.IP_ADDRESS_LIST_ID)
, MIN(fc.WAREHOUSE_ID)
, MAX(fc.WAREHOUSE_ID)
FROM D_WAREHOUSES fc
;
```

As long as your SELECT statement only includes Aggregate functions, you can add as many as you want.

COUNT(DISTINCTFC.REGION_ID)	SUM(FC.IP_ADDRESS_LIST_ID)	MIN(FC.WAREHOUSE_ID)	MAX(FC.WAREHOUSE_ID)
4	72406	A00D	ZAPO

ORDER BY

```
SELECT
fc.REGION_ID
, COUNT(fc.WAREHOUSE_ID)
FROM D_WAREHOUSES fc
WHERE fc.IS_DROPSHIP = 'Y'
ORDER BY fc.REGION_ID
;
```

However, if you add a non-Aggregate element to your SELECT clause, such as a column from the table, you'll get an error when you run Explain Plan.

ORA-00937: not a single-group function

GROUP BY

GROUP BY

```
SELECT
fc.REGION_ID
, COUNT(fc.WAREHOUSE_ID)
FROM D_WAREHOUSES fc
WHERE fc.IS_DROPSHIP = 'Y'
GROUP BY
fc.REGION_ID
ORDER BY fc.REGION_ID
;
```

Any time you have Aggregate functions (e.g. SUM, COUNT, MIN, MAX, AVG, etc) in your SELECT clause and you have non-Aggregate elements in the SELECT clause, you need a GROUP BY clause.

The GROUP BY clause comes after the WHERE clause and before the ORDER BY clause.

The GROUP BY clause should contain ALL non-Aggregate elements from the SELECT clause.

GROUP BY

```
SELECT
fc.REGION_ID
, COUNT(fc.WAREHOUSE_ID)
FROM D_WAREHOUSES fc
WHERE fc.IS_DROPSHIP = 'Y'
GROUP BY
fc.REGION_ID
ORDER BY fc.REGION_ID
;
```

REGION_ID	COUNT(FC.WAREHOUSE_ID)
1	4756
2	166
3	82
4	1

It will return one row for each unique combination of non-Aggregate elements (in this case, REGION_ID), and then run the aggregate functions for all rows corresponding to those buckets.

It’s a lot like a Pivot Table in Excel.

GROUP BY

```
SELECT
fc.REGION_ID as REGION
, COUNT(fc.WAREHOUSE_ID)
FROM D_WAREHOUSES fc
WHERE fc.IS_DROPSHIP = 'Y'
GROUP BY
ORDER BY fc.REGION_ID
;
```

The simplest way to make sure your GROUP BY clause is correct is to copy your entire SELECT clause.

GROUP BY – Copy SELECT

```
SELECT
fc.REGION_ID as REGION
, COUNT(fc.WAREHOUSE_ID)
FROM D_WAREHOUSES fc
WHERE fc.IS_DROPSHIP = 'Y'
GROUP BY
fc.REGION_ID as REGION
ORDER BY fc.REGION_ID
;
```

… and paste it into the GROUP BY clause…

… then delete any aggregate functions…

GROUP BY – Copy SELECT – remove aliases

```
SELECT
fc.REGION_ID as REGION
, COUNT(fc.WAREHOUSE_ID)
FROM D_WAREHOUSES fc
WHERE fc.IS_DROPSHIP = 'Y'
GROUP BY
fc.REGION_ID
ORDER BY fc.REGION_ID
;
```

REGION	COUNT(FC.WAREHOUSE_ID)
1	4756
2	166
3	82
4	1

A simplest way to make sure your GROUP BY clause is correct is to copy your entire SELECT clause…

… and paste it into the GROUP BY clause…

… then delete any aggregate functions…

… and remove any column aliases.

Having

Having

```
SELECT
fc.REGION_ID as REGION
, COUNT(fc.WAREHOUSE_ID)
FROM D_WAREHOUSES fc
WHERE fc.IS_DROPSHIP = 'Y'
GROUP BY
fc.REGION_ID
ORDER BY fc.REGION_ID
;
```

Sometimes, we want to restrict results of a query that includes aggregate functions to only those results where a COUNT, SUM, etc. meets certain criteria.

Having

```
SELECT
fc.REGION_ID as REGION
, COUNT(fc.WAREHOUSE_ID)
FROM D_WAREHOUSES fc
WHERE fc.IS_DROPSHIP = 'Y'
GROUP BY
fc.REGION_ID
ORDER BY fc.REGION_ID
;
```

For example, we may want to limit the results to only those REGIONs that have 100 or more WAREHOUSE_IDs.

Having

```
SELECT
fc.REGION_ID as REGION
, COUNT(fc.WAREHOUSE_ID)
FROM D_WAREHOUSES fc
WHERE fc.IS_DROPSHIP = 'Y'
AND fc.WAREHOUSE_ID >= 100
GROUP BY
fc.REGION_ID
ORDER BY fc.REGION_ID
;
```

If we tried to do this with the WHERE clause, we'll run into trouble – because the WHERE clause applies to every row in the table.

Having

```
SELECT
fc.REGION_ID as REGION
, COUNT(fc.WAREHOUSE_ID)
FROM D_WAREHOUSES fc
WHERE fc.IS_DROPSHIP = 'Y'
GROUP BY
fc.REGION_ID
HAVING
COUNT(fc.WAREHOUSE_ID) >= 100
ORDER BY fc.REGION_ID
;
```

Instead, we want a condition that applies to the results of the COUNT() function.

The HAVING clause does just that. It waits until the GROUP BY is applied and the results of the aggregate function (e.g. COUNT) are determined, then applies it's condition.

REGION	COUNT(FC.WAREHOUSE_ID)
1	4756
2	166

Notice it comes after the GROUP BY and before the ORDER BY clauses.

ETL - Partitions

Partitions

How do you know which values you need for the partitions?

The good news is there are a limited number of fields that tend to be used for Range Partitions: REGION_ID, MARKETPLACE_ID, LEGAL_ENTITY_ID, and Date fields like ACTIVITY_DAY, SHIP_DAY, etc.

REGION_ID denotes the region of the world (NA, EU, FE, or AS)

MARKETPLACE_ID denotes the web site used during the transaction, such as 1 (www.amazon.com) or 2 (www.target.com)

LEGAL_ENTITY_ID denotes the legal owner associated with a transaction (loosely the country/web site), such as 101 (US/amazon.com).

All three have robust wiki documentation available at https://w.amazon.com/index.php/DWPartitions

Partitions

How does all this make queries more efficient?

Imagine the D_DAILY_ORDERS table has the following rows of data only.

REGION_ID	ACTIVITY_DAY	MERCHANT	QUANTITY
1	1/2/2012	1	1
2	1/2/2012	7	5
1	1/3/2012	1	30
1	1/3/2012	2	2
2	1/3/2012	8	6
3	1/3/2012	13	12
2	1/4/2012	7	5

Partitions

How does all this make queries more efficient?

If we include a WHERE clause condition on REGION_ID = 1, since that column is partitioned it will apply that condition without having to check each row – the partitioning already tells which rows are in that subset.

REGION_ID	ACTIVITY_DAY	MERCHANT	QUANTITY
1	1/2/2012	1	1
2	1/2/2012	7	5
1	1/3/2012	1	30
1	1/3/2012	2	2
2	1/3/2012	8	6
3	1/3/2012	13	12
2	1/4/2012	7	5

```
SELECT
*
FROM D_DAILY_ORDERS ddo
WHERE
ddo.REGION_ID = 1
;
```


Partitions

How does all this make queries more efficient?

It effectively ignores all the rows in the table that don't have REGION_ID = 1, as if the table looked like this:

REGION_ID	ACTIVITY_DAY	MERCHANT	QUANTITY
1	1/2/2012	1	1
1	1/3/2012	1	30
1	1/3/2012	2	2

```
SELECT
*
FROM D_DAILY_ORDERS
ddo
WHERE
ddo.REGION_ID = 1
;
```


Partitions

How does all this make queries more efficient?

Adding a second condition on the other Partitioned column, ACTIVITY_DAY, will have the same effect. If the table does not have any column it does not match our condition:

REGION_ID	ACTIVITY_DAY	MERCHANT	QUANTITY
1	1/3/2012	1	30
1	1/3/2012	2	2

```
SELECT
*
FROM D_DAILY_ORDERS ddo
WHERE
ddo.REGION_ID = 1
AND ddo.ACTIVITY_DAY =
TO_DATE('20120103','YYYYMMDD')
;
```


Partitions

How does all this make queries more efficient?

Only AFTER the WHERE clause conditions on Partitioned columns are applied does the query apply the other WHERE clause conditions – so it has fewer rows to check.

REGION_ID	ACTIVITY_DAY	MERCHANT	QUANTITY
1	1/3/2012	1	30
1	1/3/2012	2	2

```
SELECT
*
FROM D_DAILY_ORDERS ddo
WHERE
ddo.REGION_ID = 1
AND ddo.ACTIVITY_DAY =
TO_DATE('20120103','YYYYMMDD')
AND ddo.MERCHANT = 1
;
```

Partitions

Some points to remember are

- The first thing to look at is the Partition Type.
- If it says the table is RANGE partitioned, then you need to add WHERE clause conditions to make use of the Partitions.
- Just above Partition Type is a list of Columns that are partitioned. You should make sure to include a WHERE clause condition for each of these.
- Some tables also have Recommended Query Keys, which you can treat similarly.

Partitions

Whenever a table is range partitioned, the first thing to do (after adding the table name and table alias to the FROM clause) is add a WHERE clause with these columns while writing a query against the D_DAILY_ORDERS table,

```
SELECT

FROM D_DAILY_ORDERS ddo
WHERE
ddo.REGION_ID = 1
AND ddo.ACTIVITY_DAY = TO_DATE('20120213','YYYYMMDD')
AND ddo.MARKETPLACE_ID = 1
;
```

ETL - Comments

Comments

When writing SQL (or any programming language), it can be helpful to leave notes for yourself or others to explain why a query includes a certain condition, what a section of SQL is used for, etc.

In SQL, we enclose comments in between /* or */ or --

```
SELECT
dma.ASIN
, dma.ITEM_NAME   /* this is the book title */
FROM D_MP_ASINS_ESSENTIALS dma
WHERE dma.REGION_ID = 1
AND dma.MARKETPLACE_ID = 1
AND dma.GL_PRODUCT_GROUP = 14   /* this limits to books only */
;
```

- Scheduling Jobs,
- Publishing to Folders
- Using the Job Wildcards

Other ETL Topics

- Scheduling Jobs,
- Publishing to Folders
- Using the Job Wildcards

Reference Link: https://w.amazon.com/index.php/DanGSQLClass/IntroToSqlEtl/Lesson3#ETL_TOPICS

Lesson 3: Homework

Lesson 3: Assignment

1. Check out the table D_DISTRIBUTOR_ORDERS in the BI Metadata. How is it partitioned?
2. Create a query to count how many POs have been created for the vendor code PRBRC in the US, using the table D_DISTRIBUTOR_ORDERS. Remember to use BI Metadata to determine which columns are Partitioned, and make sure you include those in your WHERE clause. Run the query through an Explain Plan before running it.
3. Add elements to find the first ORDER_DAY and the last ORDER_DAY for PRBRC Pos.
4. Add a column to count how many distinct ORDER_DAY values there are.
5. Add a column to sum up the total SHIPPING_COST.
6. Add DISTRIBUTOR_ID as an element of your SELECT clause. You'll need to add a GROUP BY clause, since this is not an aggregated column.
7. Add HANDLER as an element in the SELECT clause. Who created the most POs for PRBRC? When was the last time danac created one?
8. Use the HAVING clause to limit the results to only handlers who created between 30 and 40 PRBRC POs.
9. Further limit the results to only handlers who created PRBRC POs on at least 10 different days.
10. Rerun the query, but this time, have it publish to the folder \\ant\dept\BMVDSA\Books\ETL_Practice\, rather than email you the results. Have the file name include both the Job Profile and Job Run Wildcards, with the appropriate (.txt) extension for a tab delimited text file.

END