ISIT312 Big Data Management

Session 2, 2022

Exercise-5

SQL for Data Warehousing and still more of Hive

In this exercise, you will learn how to use Hive HQL extensions for Data Warehousing. A laboratory includes application of `SELECT` statement with GROUP BY clause, advanced features of `GROUP BY` clause (`ROLLUP`, `CUBE`, `GROUPING SETS`), windowing and analytics functions.

Be careful when copying the Linux commands in this document to your working Terminal, because it is error-prone. Maybe you should type those commands by yourself.

Prologue
Login to your system and start VirtualBox.

When ready start a virtual machine `ISIT312-BigDataVM-07-SEP-2020`.

(1) How to start Hadoop, Hive Server 2, beeline or SQL Developer ?

Open a new Terminal window and start Hadoop in the following way.

```
$HADOOP_HOME/sbin/.start-all.sh
```

When ready minimize the Terminal window used to start Hadoop. We shall refer to this window as to "Hadoop window" and we shall use it later on.

When ready navigate to a folder where you plan to keep your HQL scripts from this lab (you may have to create such folder now) and start Hive Server 2 in the following way.
To start Hive's metastore service, open ***a Terminal window***, type:

```
$HIVE_HOME/bin/hive --service metastore
```

A message shows that metastore is up and running:

```
SLF4J: Actual binding is of type
[org.apache.logging.slf4j.Log4jLoggerFactory]
```

To start hiveserver2, open ***another Terminal window*** and type:

```
$HIVE_HOME/bin/hiveserver2
```

The same message shows that hiveserver2 is up and running.

[IMPORTANT] Don't use Zeppelin to run the above two commands, because the %sh interpreter has a timeout threshold.

You can use Hive's own interface to interact with Hive. Open another new Terminal window and process the following command.

```
$HIVE_HOME/bin/beeline
```

Process the following command in front of `beeline>` prompt.

```
!connect jdbc:hive2://localhost:10000
```

The statement connects you through JDBC interface to Hive 2 server running on a `localhost` and listening to a port `10000`.

Press Enter when prompted about user name. Press Enter when prompted about password. The system should reply with a prompt

```
0: jdbc:hive2://localhost:10000>
```

To find what databases are available process the following command.

```
show databases;
```

At the moment only `default` database is available. We shall create new databases in the future. To find what tables have been created so far process a statement

```
show tables;
```

If you plan to use SQL Developer then leftclick at SQL Developer Icon at the top of task bar. Then rightclick at hive connection icon, pick "Connection" option from a menu, enter any user name and password and leftclick at OK button.

We shall use a default database for Hive tables created in this laboratory exercise.

(2) How to create and how to load data into an internal table ?

We shall use default database for Hive table created, loaded, and used in this laboratory exercise. Create the following internal table to store information about items.

```
create table ORDERS(
 part     char(7),
 customer varchar(30),
 amount   decimal(8,2),
 oyear    decimal(4),
 omonth   decimal(2),
 oday     decimal(2) )
   row format delimited fields terminated by ','
              stored as textfile;
```

The table represents a three-dimensional data cube. A fact entity `orders` is described by a measure `amount`. The dimensions include `part`, `customer`, and obviously

`time(oyear,omonth,oday)` dimension. There is a hierarchy over `time` dimension where years consist of months and months consist of days.

Next create a text file `items.txt` with sample data given below and save a file in a folder where you plan to keep HQL scripts from this lab (you already started Hive Server 2 from this folder).

```
bolt,James,200,2016,01,01
bolt,Peter,100,2017,01,30
bolt,Bob,300,2018,05,23
screw,James,20,2017,05,11
screw,Alice,55,2018,01,01
nut,Alice,23,2018,03,16
washer,James,45,2016,04,24
washer,Peter,100,2016,05,12
bolt,James,200,2018,01,05
bolt,Peter,100,2018,01,05
```

To load data into a table `orders` process the following `load` statement.

```
load data local inpath 'items.txt' into table orders;
```

To verify the contents of a table orders process the following select statement.

```
select * from orders;
```

(3) How to perform a simple aggregation with `group by` and `having` clauses ?

We start from implementation of a query that *finds the total number of orders per each part*, i.e. we perform aggregation along a dimension part. Create a process the following `select` statement.

```
select part, count(*)
from orders
group by part;
```

Next, we *find the total number of ordered parts summarized per each part*. It is another aggregation along a dimension `part`.

```
select part, sum(amount)
from orders
group by part;
```

Next, we *find the total number of orders per customer* and we list only the customers who submitted more than one order.

```
select customer, count(*)
from orders
group by customer
having count(*) > 1;
```

Now, assume that we would like *to find in one query the total number of orders per each part, per each customer and per both part and customer*.

```
select part, NULL, count(*)
from orders
group by part
union
select NULL, customer, count(*)
from orders
group by customer
union
select part, customer, count(*)
from orders
group by part, customer;
```

(4) How to perform aggregations with `rollup` operator ?

Assume that we would like to perform aggregation over two dimensions, then over one of the two dimensions used earlier and then aggregation over all orders. For example, *find the total number of parts ordered and summarized per part and per customer, then per part and then total number of all parts ordered*.

```
select part, customer, sum(amount)
from orders
group by part, customer
union
select part, null, sum(amount)
from orders
group by part
union
select null, null,sum(amount)
from orders;
```

The same query can be implemented as a single `select` statement with `rollup` operator.

```
select part, customer, sum(amount)
from orders
group by part, customer with rollup;
```

In the next example we use `rollup` operator to implement a query that *finds the total number of parts ordered and summarized per year and month, per year, and the total number of parts ordered.*

```
select oyear, omonth, sum(amount)
from orders
group by oyear,omonth with rollup;
```

(5) How to perform aggregation with `cube` operator ?

Assume that we would like to *find an average number of parts ordered and summarized per part, per customer, per both part and customer and an average number of parts per order.*

```
select part, customer, avg(amount)
from orders
group by part, customer with cube;
```

It is possible to verify some of the results with the following queries.

```
select avg(amount)
from orders;
```

and

```
select part,avg(amount)
from orders
where part='bolt'
group by part;
```

(6) How to perform aggregations with `grouping sets` operator ?

Assume that we would like to *find the total number of orders per part and per customer and both per year and customer*.

```
select part, customer, oyear, count(*)
from ORDERS
group by part, customer, oyear
grouping sets ((part),(customer), (oyear,customer));
```

In another example we *find the total number of parts ordered and summarized per year, month, day, per year and month , and per year, and the total number of parts ordered*.

```
SELECT oyear, omonth, oday, sum(amount)
from orders
group by oyear,omonth,oday grouping
sets((oyear,omonth,oday),(oyear,omonth), (oyear),() );
```

(7) How to perform window based aggregations ?

It is possible to use `group by` clause of select statement to find the total number of ordered parts summarized per each part.

```
select part, sum(amount)
from orders
group by part;
```

It is possible to get the similar result as from a query with `group by` clause with so called *windowing*.

```
select part, SUM(amount) over (partition by part)
from orders;
```

```
bolt        900.00
bolt        900.00
bolt        900.00
bolt        900.00
bolt        900.00
nut         23.00
```

```
screw       75.00
screw       75.00
washer      145.00
washer      145.00
```

To get the same results we have to use distinct keyword.

```
select distinct part, SUM(amount) over (partition by part)
from orders;
```

```
bolt        900.00
nut         23.00
screw       75.00
washer      145.00
```

Now we use windowing to implement a query that *finds for each part, for each customer, and for each amount ordered by customer the largest total number of parts ordered and aggregated per part*.

```
select part, customer, amount, MAX(amount) over (partition by part)
from orders;
```

```
bolt        Peter     100.00     300.00
bolt        James     200.00     300.00
bolt        Bob       300.00     300.00
bolt        Peter     100.00     300.00
bolt        James     200.00     300.00
nut         Alice     23.00      23.00
screw       Alice     55.00      55.00
screw       James     20.00      55.00
washer      Peter     100.00     100.00
washer      James     45.00      100.00
```

A table orders is partitioned (grouped by) the values in column part and for each part the largest amount is found and added to each output row that consists of `part`, `customer` and `amount`.

It is possible to use more than one aggregation. For example, we extend a query above with the *summarization of the amounts per each part*.

```
select part, customer, amount,
       MAX(amount) over (partition by part),
       SUM(amount) over (partition by part)
from orders;
```

(7) How to perform window aggregations and window ordering ?

We start from a query that *finds for each part and amount ordered the total number of parts ordered and summarized per each part*.

```
select part, amount, SUM(amount) over (partition by part)
from orders;
```

The results are the following.

```
bolt      100.00      900.00
bolt      200.00      900.00
bolt      300.00      900.00
bolt      100.00      900.00
bolt      200.00      900.00
nut       23.00       23.00
screw     55.00       75.00
screw     20.00       75.00
washer    100.00      145.00
washer    45.00       145.00
```

Now, we add a clause `order by` to *windowing*. Process the following statement.

```
select part, amount,
       SUM(amount) over (partition by part order by amount)
from orders;
```

```
bolt      100.00      200.00 |<-- 100+100
bolt      100.00      200.00 |<-- 100+100
bolt      200.00      600.00   |<-- 200+200
bolt      200.00      600.00   |<-- 200+200
bolt      300.00      900.00
nut       23.00       23.00
screw     20.00       20.00
screw     55.00       75.00
washer    45.00       45.00
washer    100.00      145.00
```

Addition of `order by` clause computes the increasing results of summarization over the amounts and orders the rows in each partition by the summarized amount. If two rows have the same values of `order by amount` then the rows are treated as one row with summarized `amount`. For example the first two rows have the same values of `order by amount` and because of that a value of `SUM(amount) = 100+100`. The same applies to the next two rows. If two or more rows have the same values of part and amount then summarization is performed in one step over all such rows. This problem (if it is really a problem ?) can be solved with more selective `order by` key. For example, the rows in each window can be ordered by `amount`, `oyear`, `omonth`, and `oday`.

```
select part,amount,
       SUM(amount) over (partition by part
                         order by amount, oyear, omonth, oday)
from orders;
```

In this case the rows in each window are ordered by `amount`, `oyear`, `omonth`, oday and summarization is performed in a row-by-row way.

```
bolt      100.00      100.00
bolt      100.00      200.00
bolt      200.00      400.00
bolt      200.00      600.00
bolt      300.00      900.00
nut       23.00       23.00
screw     20.00       20.00
screw     55.00       75.00
washer    45.00       45.00
```

```
washer     100.00     145.00
```

To *find how the ordered amounts of ordered parts changed year by year* process the following `select` statement.

```
select part, amount, oyear,
       SUM(amount) over (partition by part order by oyear)
from orders;
```

```
bolt        200.00     2016     200.00
bolt        100.00     2017     300.00
bolt        100.00     2018     900.00
bolt        200.00     2018     900.00
bolt        300.00     2018     900.00
nut         23.00      2018     23.00
screw       20.00      2017     20.00
screw       55.00      2018     75.00
washer      100.00     2016     145.00
washer      45.00      2016     145.0
```

Now, we change aggregation function to `AVG`.

```
select part, amount,
             AVG(amount) over (partition by part
                               order by amount, oyear, omonth, oday)
from orders;
```

The statement finds so called *walking average*.

```
bolt     100.00     100.000000 AVG(100)
bolt     100.00     100.000000 AVG(100+100)
bolt     200.00     133.333333 AVG(100+100+200)
bolt     200.00     150.000000 AVG(100+100+200+200)
bolt     300.00     180.000000 AVG(100+100+200+200+300)
nut      23.00      23.000000
screw    20.00      20.000000
screw    55.00      37.500000
washer   45.00      45.000000
washer   100.00     72.500000
```

(8) How to perform window aggregations and window framing ?

Next, implement a query that for each part and amount *finds an average of amount ordered by year, month and day.*

```
select part, amount,
             AVG(amount) over (partition by part
                               order by oyear, omonth, oday)
from orders;
```

```
bolt     200.00     200.000000 AVG(200)
bolt     100.00     150.000000 AVG(200+100)
bolt     100.00     150.000000 AVG(200+100)
bolt     200.00     150.000000 AVG(200+100+100+200)
```

```
bolt     300.00     180.000000 AVG(200+100+100+200+300)
nut      23.00      23.000000
screw    20.00      20.000000
screw    55.00      37.500000
washer   45.00      45.000000
washer   100.00     72.500000
```

Processing of aggregation (average) is performed over an expanding frame. At the beginning a *frame* includes the first row, next first row an second row, next first 3 rows, etc.

It is possible to make a frame fixed size and smaller than a window. Process the following statement.

```
select part, amount,
             AVG(amount) over (partition by part
                               order by oyear, omonth, oday
                               rows 1 preceding)
from orders;
```

The statement finds for each part and amount an average amount of the current and previous one amount when the amounts are sorted in time.

```
bolt     200.00     200.000000 AVG(200)
bolt     100.00     150.000000 AVG(200+100)
bolt     100.00     100.000000 AVG(100+100)
bolt     200.00     150.000000 AVG(100+200)
bolt     300.00     250.000000 AVG(200+300)
nut      23.00      23.000000
screw    20.00      20.000000
screw    55.00      37.500000
washer   45.00      45.000000
washer   100.00     72.500000
```

Processing of a statement

```
select part, amount,
             AVG(amount) over (partition by part
                               order by oyear, omonth, oday
                               rows unbounded preceding)
from orders;
```

returns the same results as without rows *frame* (the first select statement in this section).

The options of *window framing* are the following.

```
(ROWS | RANGE) BETWEEN (UNBOUNDED | [num]) PRECEDING AND ([num]
PRECEDING | CURRENT ROW | (UNBOUNDED | [num]) FOLLOWING)
```

For example:

```
ROWS BETWEEN 3 PRECEDING AND CURRENT ROW,
ROWS BETWEEN UNBOUNDED PRECEDING AND 2 FOLLOWING
```

```
(ROWS | RANGE) BETWEEN CURRENT ROW AND (CURRENT ROW | (UNBOUNDED |
[num]) FOLLOWING)
```

For example:

```
ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
```

```
(ROWS | RANGE) BETWEEN [num] FOLLOWING AND (UNBOUNDED | [num]) FOLLOWING
```

For example:

```
ROWS BETWEEN 2 FOLLOWING AND UNBOUNDED FOLLOWING
```

(9) How to use `window` clause ?

It is possible to simplify syntax a bit with `window` clause (definition). Implement the following query.

```
select part, SUM(amount) over w
from orders
window w as (partition by part);
```

(10) How to use `LEAD` and `LAG` functions ?

`LEAD` and `LAG` functions allow to access the previous and the next values in a column. For example, we would like to *find the current and the next amount for each `part` ordered by year, month, day*.

```
select part, amount,
             LEAD(amount) over (partition by part
                                order by oyear, omonth,oday)
from orders;
```

```
bolt     200.00      100.00
bolt     100.00      100.00
bolt     100.00      200.00
bolt     200.00      300.00
bolt     300.00
nut      23.00
screw    20.00       55.00
screw    55.00
washer   45.00       100.00
washer   100.00
```

Next, we would like to *find the current and the previous amount for each `part` ordered by year, month, day*.

```
select part, amount,
             LAG(amount) over (partition by part
                               order by oyear, omonth, oday)
from orders;
```

```
bolt       200.00
bolt       100.00    200.00
bolt       100.00    100.00
bolt       200.00    100.00
```

```
bolt        300.00     200.00
nut         23.00
screw       20.00
screw       55.00      20.00
washer      45.00
washer      100.00      45.00
```

Next we subtract the previous row value from the current row value.

```
select part, amount,
             amount-LAG(amount) over(partition by part
                                     order by oyear, omonth, oday)
from orders;
```

```
bolt        200.00
bolt        100.00     -100.00
bolt        100.00     0.00
bolt        200.00     100.00
bolt        300.00     100.00
nut         23.00
screw       20.00
screw       55.00      35.00
washer      45.00
washer      100.00      55.00
```

Empty places (NULLs) can be eliminated with a parameter `0` in `LAG` function.

```
select part, amount,
             amount+lag(amount,1,0) over(partition by part
                                       order by oyear, omonth, oday)
from orders;
```

```
bolt        200.00     200.00
bolt        100.00     300.00
bolt        100.00     200.00
bolt        200.00     300.00
bolt        300.00     500.00
nut         23.00      23.00
screw       20.00      20.00
screw       55.00      75.00
washer      45.00      45.00
washer      100.00     145.00
```

(11) How to use analytics functions ?

Finally, we implement windowing with the analytics functions `RANK()`, `DENSE_RANK()`, `CUM_DIST()`,

A function `RANK()` assigns a rank to row such that the rows with the same value of `amount` are ranked with the same number and rank is increased by the total number of rows with the same value.

```
select part, amount,
             RANK() over (partition by part order by amount)
from orders;
```

```
bolt        100.00     1 RANK=1
bolt        100.00     1 RANK=1
bolt        200.00     3 RANK=1+2
bolt        200.00     3 RANK=1+2
bolt        300.00     5 RANK=3+2
nut         23.00      1
screw       20.00      1
screw       55.00      2
washer      45.00      1
washer      100.00     2
```

A function `DENSE_RANK()` assigns a rank to row such that the rows with the same value of `amount` are ranked with the same number and rank is increased by 1 for each group of rows with the same value of `amount`.

```
select part, amount,
             DENSE_RANK() over (partition by part
                                order by amount)
from orders;
```

```
bolt        100.00     1
bolt        100.00     1
bolt        200.00     2
bolt        200.00     2
bolt        300.00     3
nut         23.00      1
screw       20.00      1
screw       55.00      2
washer      45.00      1
washer      100.00     2
```

A function `CUME_DIST()` computes the relative position of a specified value in a group of values. For a given row r, the `CUME_DIST()` the number of rows with values lower than or equal to the value of r, divided by the number of rows being evaluated, i.e. entire window.

```
select part, amount,
             CUME_DIST() over (partition by part
```

```
                                    order by amount)
from orders;
```

```
bolt        100.00    0.4 2 rows/5
bolt        100.00    0.4 2 rows/5
bolt        200.00    0.8 4 rows/5
bolt        200.00    0.8 4 rows/5
bolt        300.00    1.0 5 rows/5
nut         23.00     1.0
screw       20.00     0.5
screw       55.00     1.0
washer      45.00     0.5
washer      100.00    1.0
```

A function `PERCENT_RANK()` is similar to a function `CUME_DIST()`. For a row `r`, `PERCENT_RANK()` calculates the rank of `r` minus 1, divided by the number of rows being evaluated -1 , i.e. entire window-1.

```
select part, amount,
       PERCENT_RANK() over (partition by part
                            order by amount)
from orders;
```

```
bolt        100.00    0.0
bolt        100.00    0.0
bolt        200.00    0.5
bolt        200.00    0.5
bolt        300.00    1.0
nut         23.00     0.0
screw       20.00     0.0
screw       55.00     1.0
washer      45.00     0.0
washer      100.00    1.0
```

A function `NTILE(k)` divides a window into a number of buckets indicated by `k` and assigns the appropriate bucket number to each row. The buckets are numbered from `1` to `k`.

```
select part, amount,
       NTILE(2) over (partition by part
                                    order by amount)
from orders;
```

```
bolt        100.00    1
bolt        100.00    1
bolt        200.00    1
bolt        200.00    2
bolt        300.00    2
nut         23.00     1
screw       20.00     1
screw       55.00     2
washer      45.00     1
washer      100.00    2
```

```
select part, amount,
       NTILE(5) over (partition by part
                                 order by amount)
from orders;
```

```
bolt        100.00     1
bolt        100.00     2
bolt        200.00     3
bolt        200.00     4
bolt        300.00     5
nut         23.00      1
screw       20.00      1
screw       55.00      2
washer      45.00      1
washer      100.00     2
```

Finally, a function `ROW_NUMBER` does not need any explanations.

```
select part, amount,
       ROW_NUMBER() over (partition by part
                                  order by amount)
from orders;
```

```
bolt        100.00     1
bolt        100.00     2
bolt        200.00     3
bolt        200.00     4
bolt        300.00     5
nut         23.00      1
screw       20.00      1
screw       55.00      2
washer      45.00      1
washer      100.00     2
```

End of Exercise 5