SQL: Part 2

Simple Aggregation

- The count(*) counts all rows in the table *sp*
- count(expression) counts all non-null values in column

```
 name | item | price
------+------+-------
 S2   | P3   |   100
 S1   | P2   |    20
 S1   | P1   |    10
 S3   | P4   |  1000
 S2   | P1   |    11
 S4   | P1   |     9
      | P1   |
 S5   |      |
 S4   | P3   |
```

```
select count(*) as row_count, count(name) as name_count,
 count(item) as item_count, count(price) as price_count
from sp;
 row_count | name_count | item_count | price_count
-----------+------------+------------+-------------
         9 |          8 |          8 |           6
(1 row)
```

Aggregation

- The beginning of data analysis
- Extremely common (over 90% of queries in TPC-DS)
- SUM, COUNT, MIN, MAX
- AVG, STDDEV (samp/pop), VARIANCE, etc.
- COVAR, CORREL
- MEDIAN, PERCENTILE, etc.

GROUP BY

- FOR EACH unique value of GROUP BY columns
- *Virtually* segment the table into groups for each unique value
- Calculate aggregates over those segments

```
 item | price
------+-------
 P1   |     9
 P1   |    10
 P1   |     ?
 P1   |    11
 P2   |    20
 P3   |     ?
 P3   |   100
 P4   |  1000
 ?    |     ?
(9 rows)
```

```
select item, avg(price)
from sp
group by item;
 item |   avg
------+-------
      |
 P2   |   20.0
 P1   |   10.0
 P4   | 1000.0
 P3   |  100.0
(5 rows)
```

GROUP BY

- one or more columns (or expressions)
  - many systems would let you specify ordinal reference
- GROUP BY => aggregation
- Only columns (and expressions based on them) from GROUP BY and aggregate functions can be present in the SELECT list

```
select 'Part ' || item,
        avg(price)
from sp
group by 1;

 ?column? |  avg
----------+-------
          |
 Part P1  |   10.0
 Part P2  |   20.0
 Part P3  |  100.0
 Part P4  | 1000.0
(5 rows)
```

GROUP BY & DISTINCT

```
select name, item
from sp
group by name, item;
 name | item
------+------
 S3   | P4
 S1   | P2
      | P1
 S1   | P1
 S2   | P3
 S2   | P1
 S4   | P1
 S5   |
 S4   | P3
(9 rows)
```

```
select distinct name, item
from sp;

 name | item
------+------
 S3   | P4
 S1   | P2
      | P1
 S1   | P1
 S2   | P3
 S2   | P1
 S4   | P1
 S5   |
 S4   | P3
(9 rows)
```

- SELECT DISTINCT can be done using GROUP BY
- GROUP BY => aggregation, even without functions

GROUP BY & HAVING

- HAVING clause is a way to specify “WHERE” to results of aggregation
- Find items *where* average price is > 20
- HAVING requires a GROUP BY clause
- HAVING on GROUP BY columns?

```
select item, avg(price)
from sp
group by item;
 item |    avg
------+--------
      |
 P2   |   20.0
 P1   |   10.0
 P4   | 1000.0
 P3   |  100.0
(5 rows)
```

```
select item, avg(price)
from sp
group by item
having avg(price) > 20;
 item |    avg
------+--------
 P4   | 1000.0
 P3   |  100.0
(2 rows)
```

GROUP BY & HAVING

- HAVING can have subqueries
- Find items *where* avg price is > avg price of all items

```
select item, avg(price)
from sp
group by item
having avg(price) > (select avg(price)
                     from sp)
;
 item |          avg
------+-----------------------
 P4   | 1000.0000000000000000
(1 row)
```

Rows with Aggregate Property

- Want to determine item with Minimum Price (not for each item, find the minimum price)
- Find the row(s), thus item(s), where price = minimum price
- Is there another way?

```
 item | price
------+-------
 P1   |     9
 P1   |    10
 P1   |     ?
 P1   |    11
 P2   |    20
 P3   |     ?
 P3   |   100
 P4   |  1000
 ?    |     ?
(9 rows)
```

```
select item, min(price)
from sp;
ERROR:  column "sp.item" must appear
  in the GROUP BY clause
  or be used in an aggregate function
LINE 1: select item, min(price) from sp;
```

```
select item, price
from sp
where price = (select min(price)
               from sp);
```

```
 item | price
------+-------
 P1   |     9
(1 row)
```

Rows with Aggregate Property

- Find the row(s), thus items(s), where price = minimum price - not using the subquery
- Two are equivalent

```
select item, price
from sp, (select min(price)
          from sp) mp(min_price)
where price = min_price;
 item | price
------+-------
 P1   |     9
(1 row)
```

- One more way - using Window functions

```
select item, price
from (select item, price,
             rank() over (order by price) as prank
      from sp) ranked_price
where prank = 1;
 item | price
------+-------
 P1   |     9
(1 row)
```

Aggregates with DISTINCT

- Single argument aggregates (COUNT, SUM, STDDEV_POP) etc. can take a DISTINCT option
- COUNT(DISTINCT …) is the biggest use - to get an idea for number of unique values in columns
- Conceptually, first compute the DISTINCT values and then do the aggregate

```
 name | item | price
------+------+-------
 S2   | P3   |   100
 S1   | P2   |    20
 S1   | P1   |    10
 S3   | P4   |  1000
 S2   | P1   |    11
 S4   | P1   |     9
      | P1   |
 S5   |      |
 S4   | P3   |
```

```
select count(distinct name),
       count(distinct item)
from sp;
 count | count
-------+-------
     5 |     4
(1 row)
```

Available Functions

- Single Argument (Algebraic)
  - SUM, COUNT, MIN, MAX, AVG, STDDEV_(POP/SAMP), VAR_(POP/SAMP), etc.
- Two Argument
  - CORREL, COVAR_(POP/SAMP), REGR_ family
- Single Argument (Holistic)
  - PERCENTILE_(CONT/DISC), MEDIAN, QUARTILE
- ANY(boolean expression); EVERY (boolean expression)
  - T if True for any row in group; T if True for every row in group

Review

```
SELECT [options] column_expression_list
FROM table_expression_list
WHERE condition
GROUP BY groupby_list
HAVING condition
```

- Conceptually, we first evaluate the cross-products, joins and WHERE conditions
- Then we aggregate the rows according to the GROUP BY expression computing functions mentioned in SELECT list and HAVING condition
- Then we apply the HAVING condition to the resulting rows
- Finally we evaluate the SELECT list column expressions

Review: A Few Points

- The SELECT list can only contain references to GROUP BY list of column_expressions and aggregate functions
- Nothing else makes sense
  - Remember: for each [group by column] compute aggregate
- No nesting AVG(COUNT(…)) etc. as it doesn't make sense - use derived tables (nested structures)

Derived Tables

- In the FROM clause any query expression can be considered a table
- Syntax:
  - (<query expression>) AS <dtname> [(column_name_list)]
- Requirements:
  - All columns must have unique name
    - via "AS" renaming in SELECT of <query expression>, or
    - by listing out column names in the derived table name

Derived Tables - Example 1

- Find suppliers who offer a price lower or equal to avg price for item

```
select name, sp.item, price, avg_price
from sp, (select item, avg(price)
          from sp
          group by item) item_avg_price(item, avg_price)
where price <= avg_price and sp.item = item_avg_price.item
order by 1,2,3;
 name | item | price | avg_price
------+------+-------+-----------
 S1   | P1   |    10 |   10.0
 S1   | P2   |    20 |   20.0
 S2   | P3   |   100 |  100.0
 S3   | P4   |  1000 | 1000.0
 S4   | P1   |     9 |   10.0
(5 rows)
```

Derived Tables - Example 2

- Find the average supply price for items i.e. avg of avg

```
select avg(avg_price)
from (select item, avg(price)
      from sp
      group by item) item_avg_price(item, avg_price)
;
 name | item | price | avg_price
------+------+-------+-----------
 S1   | P1   |    10 |   10.0
 S1   | P2   |    20 |   20.0
 S2   | P3   |   100 |  100.0
 S3   | P4   |  1000 | 1000.0
 S4   | P1   |     9 |   10.0
(5 rows)
```

ROLLUP Aggregations

- Many times we like to “rollup” our aggregates over a hierarchy: e.g. “all items”, “item”, “item, supplier”
- or country, region, state, city
- NULLs used in place when group by column is aggregated over: ALL in example

```
select item, name, avg(price)
from spo
group by rollup(item, name)
order by item nulls last,
         name nulls last;
 item | name |  avg
------+------+---------
 P1   | S1   |   10.0
 P1   | S2   |   11.0
 P1   | S4   |    9.0
 P1   | ALL  |   10.0
 P2   | S1   |   20.0
 P2   | ALL  |   20.0
 P3   | S2   |  100.0
 P3   | ALL  |  100.0
 P4   | S3   | 1000.0
 P4   | ALL  | 1000.0
 ALL  | ALL  |  191.7
(11 rows)
```

NULLs and GROUPING(…)

- How can one distinguish between NULLs in data and NULLs introduced by ROLLUP?
  - GROUPING(…)
- also for giving meaning to NULL, 'ALL' works for character columns

```
select item, name, avg(price)
from sp
group by rollup(item, name)
order by item nulls last,
         name nulls last;
 item | name |  avg
------+------+--------
 P1   | S1   |   10.0
 P1   | S2   |   11.0
 P1   | S4   |    9.0
 P1   |      |
 P1   |      |   10.0
 P2   | S1   |   20.0
 P2   |      |   20.0
 P3   | S2   |  100.0
 P3   | S4   |
 P3   |      |  100.0
 P4   | S3   | 1000.0
 P4   |      | 1000.0
      | S5   |
      |      |
      |      |  191.7
(15 rows)
```

NULLs and GROUPING(…)

- How can one distinguish between NULLs in data and NULLs introduced by ROLLUP?
  - GROUPING(…)
- also for giving meaning to NULL, ‘ALL’ works for character columns

```
select case when grouping(item) = 1 then 'ALL'
            else item end as item1,
       case when grouping(name) = 1 then 'ALL'
            else name end as name1,
       avg(price)
from sp
group by rollup(item, name)
order by item, grouping(item),
         name, grouping(name) ;
 item1 | name1 |  avg
-------+-------+--------
 P1    | S1    |   10.0
 P1    | S2    |   11.0
 P1    | S4    |    9.0
 P1    |       |
 P1    | ALL   |   10.0
 P2    | S1    |   20.0
 P2    | ALL   |   20.0
 P3    | S2    |  100.0
 P3    | S4    |
 P3    | ALL   |  100.0
 P4    | S3    | 1000.0
 P4    | ALL   | 1000.0
       | S5    |
       | ALL   |
 ALL   | ALL   |  191.7
(15 rows)
```

Extended Grouping

- ROLLUP most common, but we may not want all levels in report
- GROUPING SETS lets us specify the exact groups we want
- On other extreme, CUBE generates every combination of grouping columns …
- HAVING works on top

```
select case when grouping(item) = 1 then 'ALL'
            else item end as item1,
       case when grouping(name) = 1 then 'ALL'
            else name end as name1,
       avg(price)
from sp
group by grouping sets((), (item, name))
order by item, grouping(item),
         name, grouping(name) ;
 item1 | name1 |  avg
-------+-------+--------
 P1    | S1    |   10.0
 P1    | S2    |   11.0
 P1    | S4    |    9.0
 P1    |       |
 P2    | S1    |   20.0
 P3    | S2    |  100.0
 P3    | S4    |
 P4    | S3    | 1000.0
       | S5    |
 ALL   | ALL   |  191.7
(10 rows)
```

NULLs Revisited

- In Tables: unknown or inapplicable values
- Operations with NULLs result in NULL
- Introduced by processing
  - Outer Joins
  - Extended Grouping

```
select a, b, a+b as aplusb,
(a > b) as agtb, (a = b) as aeqb
from t1;
 a  | b  | aplusb | agtb | aeqb
----+----+--------+------+------
 10 |NULL|   NULL | NULL | NULL
NULL|NULL|   NULL | NULL | NULL
NULL| 20 |   NULL | NULL | NULL
 20 | 30 |     50 | f    | f
 30 | 20 |     50 | t    | f
(5 rows)
```

Operations on NULLs

- Mostly operations with NULLs result in NULL values
- IS NULL and IS NOT NULL check for NULL values
- ZEROIFNULL(), COALESCE(), etc convert NULL values to something else.
  - CASE statement is the general case

NULLs and Aggregates

- Aggregations ignore nulls for most part, except count(*)
- Still all NULL values will result in NULL, except COUNT = 0
- All NULL group by values are grouped together (as if they are equal)!
- What if you really wanted to consider two null values equal?

```
select a, sum(b), count(b), count(*)
from t1
group by a;
 a  | sum | count | count
----+-----+-------+-------
    |  20 |     1 |     2
 30 |  20 |     1 |     1
 20 |  30 |     1 |     1
 10 |     |     0 |     1
(4 rows)
```

NULLs

```
select a, b, (a = b) as aeqb,
             (a = b) OR ((a is null) AND (b is NULL)) as aeqb_incl_null
from t1;
 a  | b  | aeqb | aeqb_incl_null
----+----+------+----------------
 10 |    |      |
    |    |      | t
    | 20 |      |
 20 | 30 | f    | f
 30 | 20 | f    | f
(5 rows)
```

- We can equate nulls if needed
- Sometimes systems may do that internally when needed

Ordered-Analytics

- Ranking - Find Top 10 rows
- Time Series
  - Find moving average of sales
  - Find cumulative sales for the month for each store
  - Compare values to a month-ago, year-ago
- Find ratio to report (monthly sales to annual sales)

A Simple Rank Query

- Ranking is one of the common analytic operations.
- Also used to determine top N rows
- Uses “olympic” ranking

```
select name, item, price,
       rank() over (order by price)
from sp1;
 name | item | price | rank
------+------+-------+------
 S4   | P1   |     9 |    1
 S1   | P1   |    10 |    2
 S2   | P1   |    11 |    3
 S2   | P2   |    20 |    4
 S1   | P2   |    20 |    4
 S2   | P3   |   100 |    6
 S3   | P4   |  1000 |    7
      | P1   |       |    8
 S5   |      |       |    8
 S4   | P3   |       |    8
(10 rows)
```

Ranking “for each” partition

- We may want to rank values within partitions (“groups” in aggregation)

```
select item, name, price,
       rank() over (partition by item
                    order by price)
from sp1;
 item | name | price | rank
------+------+-------+------
 P1   | S4   |     9 |    1
 P1   | S1   |    10 |    2
 P1   | S2   |    11 |    3
 P1   |      |       |    4
 P2   | S2   |    20 |    1
 P2   | S1   |    20 |    1
 P3   | S2   |   100 |    1
 P3   | S4   |       |    2
 P4   | S3   |  1000 |    1
      | S5   |       |    1
(10 rows)
```

Comparison to Totals

```
select item, name, price,
       avg(price) over
                (partition by item)
from sp1;
 item | name | price |  avg
------+------+-------+--------
 P1   | S2   |    11 |   10.0
 P1   | S4   |     9 |   10.0
 P1   | S1   |    10 |   10.0
 P1   |      |       |   10.0
 P2   | S2   |    20 |   20.0
 P2   | S1   |    20 |   20.0
 P3   | S4   |       |  100.0
 P3   | S2   |   100 |  100.0
 P4   | S3   |  1000 | 1000.0
      | S5   |       |
(10 rows)
```

- Can also report the totals for the “group” or partition across all rows (ratio-to-report, comparison to totals queries)
- No ORDER BY and No ROWS specification

Moving Sum and Avg

- Moving Sum / Avg are very common time-series operations
- Moving Computations
  - ROWS x PRECEDING
  - ROWS BETWEEN x PRECEDING and Y FOLLOWING
  - Basically any bounded "window"

```
select dept, cid, semid, grades,
       avg(grades) over (order by semid
                         rows 3 preceding) as
from e3avg;
 dept | cid | semid | grades |   mavg
------+-----+-------+--------+----------
 CS   | 564 |     1 |  3.000 | 3.00000
 CS   | 564 |     2 |  3.000 | 3.00000
 CS   | 564 |     3 |  4.000 | 3.33333
 CS   | 564 |     4 |  4.000 | 3.50000
 CS   | 564 |     5 |  2.500 | 3.37500
 CS   | 564 |     6 |  3.000 | 3.37500
 CS   | 564 |     7 |  2.667 | 3.04175
 CS   | 564 |     8 |  1.000 | 2.29175
 CS   | 564 |    10 |  3.000 | 2.41675
(9 rows)
```

Cumulative Sum and Avg

- Cumulative
  - ROWS UNBOUNDED PRECEDING
  - Basically any unbounded “window”

```
select dept, cid, semid, grades,
       avg(grades) over (order by semid
                    rows unbounded preceding) as cumavg
from e3avg;
 dept | cid | semid | grades |       cumavg
------+-----+-------+--------+--------------------
 CS   | 564 |     1 |  3.000 | 3.000000
 CS   | 564 |     2 |  3.000 | 3.000000
 CS   | 564 |     3 |  4.000 | 3.333333
 CS   | 564 |     4 |  4.000 | 3.500000
 CS   | 564 |     5 |  2.500 | 3.300000
 CS   | 564 |     6 |  3.000 | 3.250000
 CS   | 564 |     7 |  2.667 | 3.166714
 CS   | 564 |     8 |  1.000 | 2.895875
 CS   | 564 |    10 |  3.000 | 2.907444
(9 rows)
```

Window Functions

- Very verbose

-
  ```
  function (<arg>) OVER (
          [PARTITION BY …]
          [ORDER BY …]
          <window specification>)
  ```
- 3 kinds - Ranking, Aggregate, Reference
  - Ranking - Rank, Row_number, Percent_rank
  - Aggregate - all/most of the aggregate functions
  - Reference - Lag/Lead (to refer to a particular row in the order)

Window Specification

- MOVING
  - ROWS BETWEEN 3 PRECEDING AND 1 PRECEDING
  - ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING
  - ROWS 3 PRECEDING = BETWEEN 3 PRECEDING AND CURRENT ROW
- CUMULATIVE
  - ROWS UNBOUNDED PRECEDING
  - ROWS BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING
- “TOTAL”
  - none = ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

What about HAVING?

- Unfortunately, the SQL standard didn’t provide for HAVING equivalent

```
select item, name, price, myrank
from (select item, name, price,
           rank() over (partition by item
                        order by price)
                                 as myrank
      from sp1) as d1
where myrank = 1;
 item | name | price | myrank
------+------+-------+--------
 P1   | S4   |     9 |      1
 P2   | S2   |    20 |      1
 P2   | S1   |    20 |      1
 P3   | S2   |   100 |      1
 P4   | S3   |  1000 |      1
      | S5   |       |      1
(6 rows)
```

Nesting Aggregates

- Window Functions can have nested aggregates inside
- Equivalent to the semantics on the right

```
select item,
       cast(avg(price) as decimal(5,1)),
       rank() over (order by avg(price))
                 as myrank
from sp1
dgroup by item;
 item |  avg   | myrank
------+--------+--------
 P1   |   10.0 |      1
 P2   |   20.0 |      2
 P3   |  100.0 |      3
 P4   | 1000.0 |      4
      |        |      5
```

```
select item, avg_price,
      rank() over (order by avg_price)
                             as myrank
from (select item,
             avg(price) as avg_price
      from sp1
      group by item) item_avg_price;
 item |       avg_price       | myrank
------+-----------------------+--------
 P1   |   10.0000000000000000 |      1
 P2   |   20.0000000000000000 |      2
 P3   |  100.0000000000000000 |      3
 P4   | 1000.0000000000000000 |      4
      |                       |      5
(5 rows)
```