/\*Analytic Functions

After the database server has completed all of the steps necessary to

evaluate a query, including joining, filtering, grouping, and sorting,

the result set is complete and ready to be returned to the caller. If

your result set contains sales data, perhaps you might want to generate

rankings for salespeople or regions, or calculate percentage

differences between one time period and another. If you are generating

results for a financial report, perhaps you would like to calculate

subtotals for each report section, and a grand total for the final

section. Using analytic functions, you can do all of these things and

more.\*/

/\*Data Windows - Write a query that generates monthly sales totals for a may to august

0f 2005 period.\*/

SELECT

QUARTER(payment\_date) quarter,

MONTHNAME(payment\_date) month\_name,

SUM(amount) monthly\_sales

FROM

payment

WHERE

YEAR(payment\_date) = 2005

GROUP BY QUARTER(payment\_date), MONTHNAME(payment\_date);

SELECT

QUARTER(payment\_date) quarter,

MONTHNAME(payment\_date) month\_name,

SUM(amount) monthly\_sales

FROM

payment

WHERE

YEAR(payment\_date) = 2005

GROUP BY 1, 2;

SELECT

quarter(payment\_date) AS quarter,

monthname(payment\_date) AS month\_name,

sum(amount) AS monthly\_sales,

max(sum(amount)) OVER () max\_overall\_sales,

max(sum(amount)) OVER (PARTITION BY quarter(payment\_date)) AS max\_qrtr\_sales

FROM

payment

WHERE

YEAR(payment\_date) = 2005

GROUP BY quarter(payment\_date), monthname(payment\_date);

/\*To accommodate this type of analysis, analytic functions include

the ability to group rows into windows, which effectively partition

the data for use by the analytic function without changing the overall

result set. Windows are defined using the over clause combined with

an optional partition by subclause.

In the previous query, both analytic functions include an over clause,

but the first one is empty, indicating that the window should include

the entire result set, whereas the second one specifies that the

window should include only rows within the same quarter.\*/

/\*Localized Sorting\*/

SELECT

quarter(payment\_date) AS quarter,

monthname(payment\_date) AS month\_nm,

sum(amount) AS monthly\_sales,

RANK() OVER (ORDER BY sum(amount) DESC) AS sales\_rank

FROM

payment

WHERE

YEAR(payment\_date) = 2005

GROUP BY quarter(payment\_date), monthname(payment\_date)

ORDER BY 1, MONTH(payment\_date);

SELECT

quarter(payment\_date) AS quarter,

monthname(payment\_date) AS month\_nm,

sum(amount) AS monthly\_sales,

RANK() OVER (PARTITION BY quarter(payment\_date) ORDER BY sum(amount) DESC) AS qtr\_sales\_rank

FROM

payment

WHERE

YEAR(payment\_date) = 2005

GROUP BY quarter(payment\_date), monthname(payment\_date)

ORDER BY 1, MONTH(payment\_date);

/\*Ranking

There are multiple ranking functions available in the SQL standard,

with each one taking a different approach to how ties are handled:

row\_number -

Returns a unique number for each row, with rankings arbitrarily

assigned in case of a tie

rank -

Returns the same ranking in case of a tie, with gaps in the rankings

dense\_rank -

Returns the same ranking in case of a tie, with no gaps in the rankings\*/

SELECT

customer\_id,

COUNT(\*) AS number\_of\_rentals

FROM

rental

GROUP BY customer\_id

ORDER BY 2 DESC;

SELECT

customer\_id,

COUNT(\*) AS num\_rentals,

ROW\_NUMBER() OVER (ORDER BY COUNT(\*) DESC) AS row\_number\_rnk,

RANK() OVER (ORDER BY COUNT(\*) DESC) AS rank\_rnk,

DENSE\_RANK() OVER (ORDER BY COUNT(\*) DESC) AS dense\_rank\_rnk

FROM

rental

GROUP BY customer\_id

ORDER BY 2 DESC;

/\*Generating Multiple Rankings\*/

SELECT

customer\_id,

MONTHNAME(rental\_date) AS rental\_month,

COUNT(\*) AS num\_rentals

FROM

rental

GROUP BY customer\_id, MONTHNAME(rental\_date)

ORDER BY 2, 3 DESC;

SELECT

customer\_id,

MONTHNAME(rental\_date) AS rental\_month,

COUNT(\*) AS num\_rentals,

RANK() OVER (PARTITION BY MONTHNAME(rental\_date) ORDER BY COUNT(\*) DESC) AS rank\_rnk

FROM

rental

GROUP BY customer\_id, MONTHNAME(rental\_date)

ORDER BY 2, 3 DESC;

SELECT

customer\_id,

rental\_month,

num\_rentals,

rank\_rnk AS ranking

FROM

(SELECT

customer\_id,

MONTHNAME(rental\_date) AS rental\_month,

COUNT(\*) AS num\_rentals,

RANK() OVER (PARTITION BY MONTHNAME(rental\_date) ORDER BY COUNT(\*) DESC) AS rank\_rnk

FROM

rental

GROUP BY customer\_id, MONTHNAME(rental\_date)) cust\_rankings

WHERE rank\_rnk <= 5

ORDER BY rental\_month, num\_rentals DESC, rank\_rnk;

/\*Since analytic functions can be used only in the SELECT clause, you

will often need to nest queries if you need to do any filtering or

grouping based on the results from the analytic function.\*/

/\*Reporting Functions

Query that generates monthly and grand totals for all payments of $10

or higher\*/

SELECT

MONTHNAME(payment\_date) AS payment\_month,

amount,

SUM(amount) OVER (PARTITION BY MONTHNAME(payment\_date)) AS monthly\_total,

SUM(amount) OVER () AS grand\_total

FROM

payment

WHERE

amount >= 10

ORDER BY 1;

/\*Calculatoion using grand\_total column\*/

SELECT

MONTHNAME(payment\_date) AS payment\_month,

SUM(amount) AS month\_total,

ROUND(SUM(amount) / SUM(SUM(amount)) OVER () \* 100, 2) AS pct\_of\_total

FROM

payment

GROUP BY MONTHNAME(payment\_date);

/\*Window Frames

Data windows for analytic functions are defined using the partition

by clause, which allows you to group rows by common values.\*/

SELECT

SUM(amount)

FROM

payment;

SELECT

YEARWEEK(payment\_date) AS payment\_week,

SUM(amount) AS week\_total,

SUM(SUM(amount)) OVER (ORDER BY YEARWEEK(payment\_date) ROWS UNBOUNDED PRECEDING) AS rolling\_sum

FROM

payment

GROUP BY YEARWEEK(payment\_date)

ORDER BY 1;

/\*rolling avg\*/

SELECT

YEARWEEK(payment\_date) AS payment\_week,

SUM(amount) AS week\_total,

AVG(SUM(amount)) OVER (ORDER BY YEARWEEK(payment\_date) ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS rolling\_3wk\_avg

FROM

payment

GROUP BY YEARWEEK(payment\_date)

ORDER BY 1;

/\*The rolling\_3wk\_avg column defines a data window consisting of the

current row, the prior row, and the next row. The data window will

therefore consist of three rows, except for the first and last rows,

which will have a data window consisting of just two rows (since there

is no prior row for the first row and no next row for the last row).\*/

SELECT

DATE(payment\_date),

SUM(amount),

AVG(SUM(amount)) OVER (ORDER BY DATE(payment\_date) RANGE BETWEEN INTERVAL 3 DAY PRECEDING AND INTERVAL 3 DAY FOLLOWING) AS 7\_day\_avg

FROM

payment

WHERE

payment\_date BETWEEN '2005-07-01' AND '2005-09-01'

GROUP BY DATE(payment\_date)

ORDER BY 1;

/\*Lag and Lead

Along with computing sums and averages over a data window, another

common reporting task involves comparing values from one row to

another.

For example, if you are generating monthly sales totals, you may be

asked to create a column showing the percentage difference from the

prior month, which will require a way to retrieve the monthly sales

total from the previous row. This can be accomplished using the lag

function, which will retrieve a column value from a prior row in the

result set, or the lead function, which will retrieve a column value

from a following row.\*/

SELECT

YEARWEEK(payment\_date) AS payment\_week,

SUM(amount) AS week\_total,

LAG(SUM(amount), 1) OVER (ORDER BY YEARWEEK(payment\_date)) AS prev\_wk\_tot,

LEAD(SUM(amount), 1) OVER (ORDER BY YEARWEEK(payment\_date)) AS next\_wk\_tot

FROM

payment

GROUP BY YEARWEEK(payment\_date)

ORDER BY 1;

/\*percentage difference using lag function\*/

SELECT

YEARWEEK(payment\_date) AS payment\_week,

SUM(amount) AS week\_total,

ROUND((SUM(amount) - LAG(SUM(amount), 1) OVER (ORDER BY YEARWEEK(payment\_date))) / LAG(SUM(amount), 1) OVER (ORDER BY YEARWEEK(payment\_date)) \* 100, 1) AS pct\_diff

FROM

payment

GROUP BY YEARWEEK(payment\_date)

ORDER BY 1;

/\*Column Value Concatenation

The group\_concat function is used to pivot a set of column values into

a single delimited

string, which is a handy way to denormalize your result set for

generating XML or

JSON documents.\*/

SELECT

f.title,

GROUP\_CONCAT(a.last\_name ORDER BY a.last\_name SEPARATOR ', ') AS actors

FROM

actor a

INNER JOIN film\_actor fa ON a.actor\_id = fa.actor\_id

INNER JOIN film f ON fa.film\_id = f.film\_id

GROUP BY f.title

HAVING COUNT(\*) = 3;

/\*Exercise - For all exercises in this section, use the following data

set from the Sales\_Fact table: Sales\_Fact\*/

/\*Exercise - 1

Write a query that retrieves every row from Sales\_Fact, and add a

column to generate a ranking based on the tot\_sales column values.

The highest value should receive a ranking of 1, and the lowest a

ranking of 24.\*/

SELECT

year\_no,

month\_no,

tot\_sales,

RANK() OVER (ORDER BY tot\_sales DESC) AS sales\_rank

FROM

sales\_fact;

/\*Exercise 2

Modify the query from the previous exercise to generate two sets of

rankings from 1 to 12, one for 2019 data and one for 2020.\*/

SELECT

year\_no,

month\_no,

tot\_sales,

RANK() OVER (PARTITION BY year\_no ORDER BY tot\_sales DESC) AS sales\_rank

FROM

sales\_fact;

/\*Exercise - 3

Write a query that retrieves all 2020 data, and include a column that

will contain the tot\_sales value from the previous month.\*/

SELECT

year\_no,

month\_no,

tot\_sales,

LAG(tot\_sales) OVER (ORDER BY month\_no) AS prev\_month\_sales

FROM

sales\_fact

WHERE

year\_no = 2020;