SQL for Data Analysis Cheat Sheet

SQL

SQL, or *Structured Query Language*, is a language for talking to databases. It lets you select specific data and build complex reports. Today, SQL is a universal language of data, used in practically all technologies that process data.

SELECT

Fetch the id and name columns from the `product` table:

```
SELECT id, name
FROM product;
```

Concatenate the name and the description to fetch the full description of the products:

```
SELECT name || ' - ' || description
FROM product;
```

Fetch names of products with prices above 15:

```
SELECT name
FROM product
WHERE price > 15;
```

Fetch names of products with prices between 50 and 150:

```
SELECT name
FROM product
WHERE price BETWEEN 50 AND 150;
```

Fetch names of products that are not watches:

```
SELECT name
FROM product
WHERE name != 'watch';
```

Fetch names of products that start with a `'P'` or end with an `'s'`:

```
SELECT name
FROM product
WHERE name LIKE 'P%' OR name LIKE '%s';
```

Fetch names of products that start with any letter followed by `'rain'` (like `'train'` or `'grain'`):

```
SELECT name
FROM product
WHERE name LIKE '_rain';
```

Fetch names of products with non-null prices:

```
SELECT name
FROM product
WHERE price IS NOT NULL;
```

GROUP BY

PRODUCT	
name	**category**
Knife	Kitchen
Pot	Kitchen
Mixer	Kitchen
Jeans	Clothing
Sneakers	Clothing
Leggings	Clothing
Smart TV	Electronics
Laptop	Electronics

→

category	count
Kitchen	3
Clothing	3
Electronics	2

AGGREGATE FUNCTIONS

Count the number of products:

```
SELECT COUNT(*)
FROM product;
```

Count the number of products with non-null prices:

```
SELECT COUNT(price)
FROM product;
```

Count the number of unique category values:

```
SELECT COUNT(DISTINCT category_id)
FROM product;
```

Get the lowest and the highest product price:

```
SELECT MIN(price), MAX(price)
FROM product;
```

Find the total price of products for each category:

```
SELECT category_id, SUM(price)
FROM product
GROUP BY category_id;
```

Find the average price of products for each category whose average is above 3.0:

```
SELECT category_id, AVG(price)
FROM product
GROUP BY category_id
HAVING AVG(price) > 3.0;
```

ORDER BY

Fetch product names sorted by the `price` column in the default ASCending order:

```
SELECT name
FROM product
ORDER BY price [ASC];
```

Fetch product names sorted by the `price` column in DESCending order:

```
SELECT name
FROM product
ORDER BY price DESC;
```

COMPUTATIONS

Use +, -, *, / to do basic math. To get the number of seconds in a week:

```
SELECT 60 * 60 * 24 * 7;
-- result: 604800
```

ROUNDING NUMBERS

Round a number to its nearest integer:

```
SELECT ROUND(1234.56789);
-- result: 1235
```

Round a number to two decimal places:

```
SELECT ROUND(AVG(price), 2)
FROM product
WHERE category_id = 21;
-- result: 124.56
```

TROUBLESHOOTING

INTEGER DIVISION

In PostgreSQL and SQL Server, the / operator performs integer division for integer arguments. If you do not see the number of decimal places you expect, it is because you are dividing between two integers. Cast one to decimal:

```
123 / 2 -- result: 61
CAST(123 AS decimal) / 2 -- result: 61.5
```

DIVISION BY 0

To avoid this error, make sure the denominator is not `0`. You may use the `NULLIF()` function to replace `0` with a `NULL`, which results in a `NULL` for the entire expression:

```
count / NULLIF(count_all, 0)
```

JOIN

JOIN is used to fetch data from multiple tables. To get the names of products purchased in each order, use:

```
SELECT
 orders.order_date,
 product.name AS product,
 amount
FROM orders
JOIN product
  ON product.id = orders.product_id;
```

Learn more about JOINs in our interactive SQL JOINs course.

INSERT

To insert data into a table, use the `INSERT` command:

```
INSERT INTO category
VALUES
(1, 'Home and Kitchen'),
(2, 'Clothing and Apparel');
```

You may specify the columns to which the data is added. The remaining columns are filled with predefined default values or NULLs.

```
INSERT INTO category (name)
VALUES ('Electronics');
```

UPDATE

To update the data in a table, use the `UPDATE` command:

```
UPDATE category
SET
 is_active = true,
 name = 'Office'
WHERE name = 'Ofice';
```

DELETE

To delete data from a table, use the `DELETE` command:

```
DELETE FROM category
WHERE name IS NULL;
```

SQL for Data Analysis Cheat Sheet

DATE AND TIME

There are 3 main time-related types: `date`, `time`, and `timestamp`. Time is expressed using a 24-hour clock, and it can be as vague as just hour and minutes (e.g., `15:30` – 3:30 p.m.) or as precise as microseconds and time zone (as shown below):

```
2021-12-31 14:39:53.662522-05
|  date   | |      time      |
|          timestamp         |
YYYY-mm-dd HH:MM:SS.ssssss±TZ
```

`14:39:53.662522-05` is almost 2:40 p.m. CDT (e.g., in Chicago; in UTC it'd be 7:40 p.m.). The letters in the above example represent:

In the date part:

- YYYY – the 4-digit year.
- mm – the zero-padded month (01—January through 12—December).
- dd – the zero-padded day.

In the time part:

- HH – the zero-padded hour in a 24-hour clock.
- MM – the minutes.
- SS – the seconds. *Omissible.*
- ssssss – the smaller parts of a second – they can be expressed using 1 to 6 digits. *Omissible.*
- ±TZ – the timezone. It must start with either + or −, and use two digits relative to UTC. *Omissible.*

CURRENT DATE AND TIME

Find out what time it is:

```sql
SELECT CURRENT_TIME;
```

Get today's date:

```sql
SELECT CURRENT_DATE;
```

In SQL Server:

```sql
SELECT GETDATE();
```

Get the timestamp with the current date and time:

```sql
SELECT CURRENT_TIMESTAMP;
```

CREATING DATE AND TIME VALUES

To create a date, time, or timestamp, write the value as a string and cast it to the proper type.

```sql
SELECT CAST('2021-12-31' AS date);
SELECT CAST('15:31' AS time);
SELECT CAST('2021-12-31 23:59:29+02'
       AS timestamp);
SELECT CAST('15:31.124769' AS time);
```

Be careful with the last example – it is interpreted as 15 minutes 31 seconds and 124769 microseconds! It is always a good idea to write 00 for hours explicitly: `'00:15:31.124769'`.

SORTING CHRONOLOGICALLY

Using `ORDER BY` on date and time columns sorts rows chronologically from the oldest to the most recent:

```sql
SELECT order_date, product, quantity
FROM sales
ORDER BY order_date;
```

order_date	product	quantity
2023-07-22	Laptop	2
2023-07-23	Mouse	3
2023-07-24	Sneakers	10
2023-07-24	Jeans	3
2023-07-25	Mixer	2

Use the DESCending order to sort from the most recent to the oldest:

```sql
SELECT order_date, product, quantity
FROM sales
ORDER BY order_date DESC;
```

COMPARING DATE AND TIME VALUES

You may use the comparison operators <, <=, >, >=, and = to compare date and time values. Earlier dates are less than later ones. For example, `2023-07-05` is "less" than `2023-08-05`.

Find sales made in July 2023:

```sql
SELECT order_date, product_name, quantity
FROM sales
WHERE order_date >= '2023-07-01'
  AND order_date <  '2023-08-01';
```

Find customers who registered in July 2023:

```sql
SELECT registration_timestamp, email
FROM customer
WHERE registration_timestamp >= '2023-07-01'
  AND registration_timestamp <  '2023-08-01';
```

Note: Pay attention to the end date in the query. The upper bound `'2023-08-01'` is not included in the range. The timestamp `'2023-08-01'` is actually the timestamp `'2023-08-01 00:00:00.0'`. The comparison operator < is used to ensure the selection is made for all timestamps less than `'2023-08-01 00:00:00.0'`, that is, all timestamps in July 2023, even those close to the midnight of August 1, 2023.

INTERVALS

An interval measures the difference between two points in time. For example, the interval between `2023-07-04` and `2023-07-06` is 2 days.

To define an interval in SQL, use this syntax:

```sql
INTERVAL '1' DAY
```

The syntax consists of three elements: the INTERVAL keyword, a quoted value, and a time part keyword. You may use the following time parts: `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, and `SECOND`.

Adding intervals to date and time values

You may use + or – to add or subtract an interval to date or timestamp values.

Subtract one year from `2023-07-05`:

```sql
SELECT CAST('2023-07-05' AS TIMESTAMP)
         - INTERVAL '1' year;
-- result: 2022-07-05 00:00:00
```

Find customers who placed the first order within a month from the registration date:

```sql
SELECT id
FROM customers
WHERE first_order_date >
  registration_date + INTERVAL '1' month;
```

Filtering events to those in the last 7 days

To find the deliveries scheduled for the last 7 days, use:

```sql
SELECT delivery_date, address
FROM sales
WHERE delivery_date <= CURRENT_DATE
  AND delivery_date >= CURRENT_DATE
      - INTERVAL '7' DAY;
```

Note: In SQL Server, intervals are not implemented – use the DATEADD() and DATEDIFF() functions.

Filtering events to those in the last 7 days in SQL Server

To find the sales made within the last 7 days, use:

```sql
SELECT delivery_date, address
FROM sales
WHERE delivery_date <= GETDATE()
  AND delivery_date >=
    DATEADD(DAY, -7, GETDATE());
```

EXTRACTING PARTS OF DATES

The standard SQL syntax to get a part of a date is

```sql
SELECT EXTRACT(YEAR FROM order_date)
FROM sales;
```

You may extract the following fields:
`YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, and `SECOND`.

The standard syntax does not work In SQL Server.
Use the DATEPART(part, date) function instead.

```sql
SELECT DATEPART(YEAR, order_date)
FROM sales;
```

GROUPING BY YEAR AND MONTH

Find the count of sales by month:

```sql
SELECT
 EXTRACT(YEAR FROM order_date) AS year,
 EXTRACT(MONTH FROM order_date) AS month,
 COUNT(*) AS count
FROM sales
GROUP BY
 year,
 month
ORDER BY
 year
 month;
```

year	month	count
2022	8	51
2022	9	58
2022	10	62
2022	11	76
2022	12	85
2023	1	71
2023	2	69

Note that you must group by both the year and the month. `EXTRACT(MONTH FROM order_date)` only extracts the month number (1, 2, ..., 12). To distinguish between months from different years, you must also group by year.

SQL for Data Analysis Cheat Sheet

CASE WHEN

`CASE WHEN` lets you pass conditions (as in the WHERE clause), evaluates them in order, then returns the value for the first condition met.

```
SELECT
 name,
 CASE
   WHEN price > 150 THEN 'Premium'
   WHEN price > 100 THEN 'Mid-range'
   ELSE 'Standard'
 END AS price_category
FROM product;
```

Here, all products with prices above 150 get the *Premium* label, those with prices above 100 (and below 150) get the *Mid-range* label, and the rest receives the *Standard* label.

CASE WHEN and GROUP BY

You may combine `CASE WHEN` and `GROUP BY` to compute object statistics in the categories you define.

```
SELECT
 CASE
   WHEN price > 150 THEN 'Premium'
   WHEN price > 100 THEN 'Mid-range'
   ELSE 'Standard'
 END AS price_category,
 COUNT(*) AS products
FROM product
GROUP BY price_category;
```

Count the number of large orders for each customer using `CASE WHEN` and `SUM()`:

```
SELECT
  customer_id,
  SUM(
    CASE WHEN quantity > 10
    THEN 1 ELSE 0 END
  ) AS large_orders
FROM sales
GROUP BY customer_id;
```

... or using `CASE WHEN` and `COUNT()`:

```
SELECT
  customer_id,
  COUNT(
    CASE WHEN quantity > 10
    THEN order_id END
  ) AS large_orders
FROM sales
GROUP BY customer_id;
```

GROUP BY EXTENSIONS

GROUPING SETS

`GROUPING SETS` lets you specify multiple sets of columns to group by in one query.

```
SELECT region, product, COUNT(order_id)
FROM sales
GROUP BY
  GROUPING SETS ((region, product), ());
```

region	product	count	
USA	Laptop	10	
USA	Mouse	5	GROUP BY (region, product)
UK	Laptop	6	
NULL	NULL	21	GROUP BY () – all rows

CUBE

`CUBE` generates groupings for all possible subsets of the GROUP BY columns.

```
SELECT region, product, COUNT(order_id)
FROM sales
GROUP BY CUBE (region, product);
```

region	product	count	
USA	Laptop	10	
USA	Mouse	5	GROUP BY region, product
UK	Laptop	6	
USA	NULL	15	GROUP BY region
UK	NULL	6	
NULL	Laptop	16	GROUP BY product
NULL	Mouse	5	
NULL	NULL	21	GROUP BY () – all rows

ROLLUP

`ROLLUP` adds new levels of grouping for subtotals and grand totals.

```
SELECT region, product, COUNT(order_id)
FROM sales
GROUP BY ROLLUP (region, product);
```

region	product	count	
USA	Laptop	10	
USA	Mouse	5	GROUP BY region, product
UK	Laptop	6	
USA	NULL	15	GROUP BY region
UK	NULL	6	
NULL	NULL	21	GROUP BY () – all rows

COALESCE

`COALESCE` replaces the first `NULL` argument with a given value. It is often used to display labels with GROUP BY extensions.

```
SELECT region,
 COALESCE(product, 'All'),
 COUNT(order_id)
FROM sales
GROUP BY ROLLUP (region, product);
```

region	product	count
USA	Laptop	10
USA	Mouse	5
USA	All	15
UK	Laptop	6
UK	All	6
All	All	21

COMMON TABLE EXPRESSIONS

A common table expression (CTE) is a named temporary result set that can be referenced within a larger query. They are especially useful for complex aggregations and for breaking down large queries into more manageable parts.

```
WITH total_product_sales AS (
 SELECT product, SUM(profit) AS total_profit
 FROM sales
 GROUP BY product
)

SELECT AVG(total_profit)
FROM total_product_sales;
```

Check out our hands-on courses on Common Table Expressions and GROUP BY Extensions.

WINDOW FUNCTIONS

Window functions compute their results based on a sliding window frame, a set of rows related to the current row. Unlike aggregate functions, window functions do not collapse rows.

COMPUTING THE PERCENT OF TOTAL WITHIN A GROUP

```
SELECT product, brand, profit,
  (100.0 * profit /
    SUM(profit) OVER(PARTITION BY brand)
  ) AS perc
FROM sales;
```

product	brand	profit	perc
Knife	Culina	1000	25
Pot	Culina	3000	75
Doll	Toyze	2000	40
Car	Toyze	3000	60

RANKING

Rank products by price:

```
SELECT RANK() OVER(ORDER BY price), name
FROM product;
```

RANKING FUNCTIONS

RANK – gives the same rank for tied values, leaves gaps.
DENSE_RANK – gives the same rank for tied values without gaps.
ROW_NUMBER – gives consecutive numbers without gaps.

name	rank	dense_rank	row_number
Jeans	1	1	1
Leggings	2	2	2
Leggings	2	2	3
Sneakers	4	3	4
Sneakers	4	3	5
Sneakers	4	3	6
T-Shirt	7	4	7

RUNNING TOTAL

A running total is the cumulative sum of a given value and all preceding values in a column.

```
SELECT date, amount,
  SUM(amount) OVER(ORDER BY date)
    AS running_total
FROM sales;
```

MOVING AVERAGE

A moving average (*a.k.a.* rolling average, running average) is a technique for analyzing trends in time series data. It is the average of the current value and a specified number of preceding values.

```
SELECT date, price,
  AVG(price) OVER(
    ORDER BY date
    ROWS BETWEEN 2 PRECEDING
      AND CURRENT ROW
  ) AS moving_averge
FROM stock_prices;
```

DIFFERENCE BETWEEN TWO ROWS (DELTA)

```
SELECT year, revenue,
  LAG(revenue) OVER(ORDER BY year)
    AS revenue_prev_year,
  revenue -
    LAG(revenue) OVER(ORDER BY year)
    AS yoy_difference
FROM yearly_metrics;
```