



Week III

SQL

Intermediate Data Engineer in
Azure
Trainer: Balazs Balogh

SQL recap – SELECT / FROM

```
taxi_data_2024_08 = (  
    spark  
    .read  
    .format("parquet")  
    .load("../data/data_for_pyspark_tutorial/yellow_tripdata  
_2024-08.parquet")  
)
```

```
taxi_data_2024_08.createOrReplaceTempView("taxi_2024_08")
```

```
spark.sql("""  
  
    SELECT * FROM taxi_2024_08  
  
""").show()
```

The **SELECT** statement is used to **query** a database and retrieve data from one or more tables.

It allows you to specify the columns you want to retrieve, either by name or using wildcard (*) to select all columns. You can also apply functions, calculations, or transformations to the data in the SELECT clause.

The result of a SELECT query is typically a table of data that matches the specified criteria.

The **FROM** clause **specifies the table** or tables from which to retrieve the data. It is used in conjunction with the SELECT statement to define the source of the data.

You can join multiple tables using the JOIN clause in the FROM section to combine data from related tables.

The FROM clause can also include subqueries, allowing you to use the result of another query as a data source.

SQL recap - WHERE

```
spark.sql("""  
    SELECT  
        VendorID,  
        passenger_count  
    FROM  
        taxi_2024_08  
    WHERE  
        passenger_count > 4  
""").show(10)
```

The **WHERE** clause is used to filter records based on specified conditions.

It allows you to define conditions using comparison operators like =, !=, >, <, >=, <=.

You can combine multiple conditions using logical operators such as AND, OR.

The IN operator checks if a value matches any value in a list, while NOT IN excludes values from the specified list.

This enables you to filter data effectively according to specific requirements.

SQL recap – ORDER BY

```
spark.sql("""  
    SELECT  
        VendorID,  
        passenger_count  
    FROM  
        taxi_2024_08  
    ORDER BY  
        passenger_count DESC  
""").show(10)
```

The **ORDER BY** clause is used to sort the result set by one or more columns.

By default, the data is sorted in **ascending** order, but you can specify **DESC** for descending order.

Sorting helps in organizing data for easier interpretation, especially when dealing with large datasets.

You can order data by multiple columns, and the order of the ORDER BY clauses will dictate the priority of sorting.

SQL recap – GROUP BY / HAVING

```
spark.sql("""  
    SELECT  
        VendorID,  
        MAX(passenger_count) AS max_passenger_count,  
        AVG(total_amount) AS avg_total_amount  
    FROM  
        taxi_2024_08  
    GROUP BY  
        VendorID  
    HAVING  
        avg_total_amount > 28  
""").show(10)
```

The **GROUP BY** clause is used to group rows that share a common property, allowing aggregate functions like SUM, MAX, MIN, AVG, and COUNT to be applied to each group.

This is useful for calculating summary statistics, such as total sales or average age within a certain category.

The result is a dataset where each row represents a group, and the aggregate functions compute the desired metrics for each group.

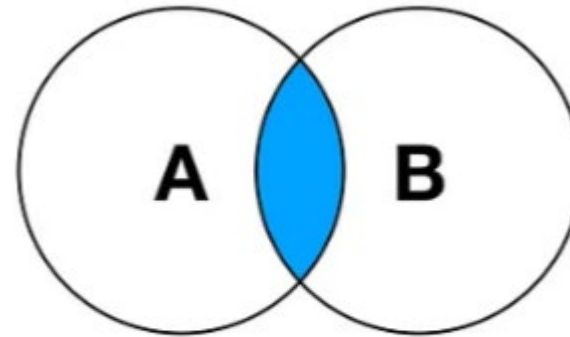
With **HAVING** we can filter groups post aggregation. WHERE won't work here, it can't be after the GROUP BY.

SQL recap – INNER JOIN

```
spark.sql("""  
  
    SELECT  
        e.EmployeeID,  
        e.Name,  
        e.Salary,  
        d.DepartmentName  
    FROM  
        employees e  
    INNER JOIN  
        departments d  
    ON  
        e.DepartmentID = d.DepartmentID  
  
""").show()
```

Joins are used to combine rows from two or more tables based on a related column.

The **INNER JOIN** returns only the rows where there is a **match in both** tables.



INNER JOIN

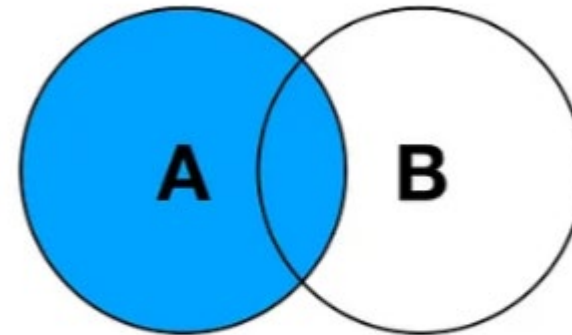
SQL recap – LEFT JOIN

```
spark.sql("""
```

```
    SELECT
        e.EmployeeID,
        e.Name,
        e.Salary,
        d.DepartmentName
    FROM
        employees e
    LEFT JOIN
        departments d
    ON
        e.DepartmentID = d.DepartmentID
```

```
""").show()
```

The LEFT JOIN includes **all rows from the left** table and the **matching rows from the right** table, filling in NULLs where there is no match.

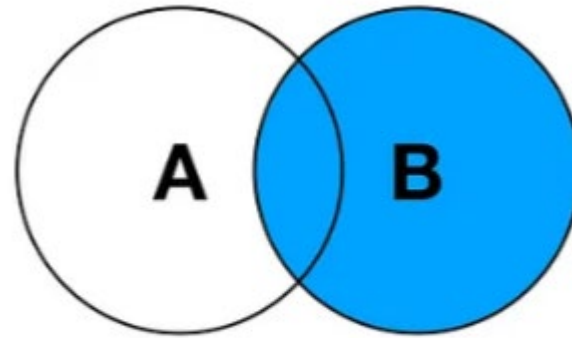


LEFT JOIN

SQL recap – RIGHT JOIN

```
spark.sql("""  
  
    SELECT  
        e.EmployeeID,  
        e.Name,  
        e.Salary,  
        d.DepartmentName  
    FROM  
        employees e  
    RIGHT JOIN  
        departments d  
    ON  
        e.DepartmentID = d.DepartmentID  
  
""").show()
```

The RIGHT JOIN is the **reverse** of the LEFT JOIN, including all rows from the right table.



RIGHT JOIN

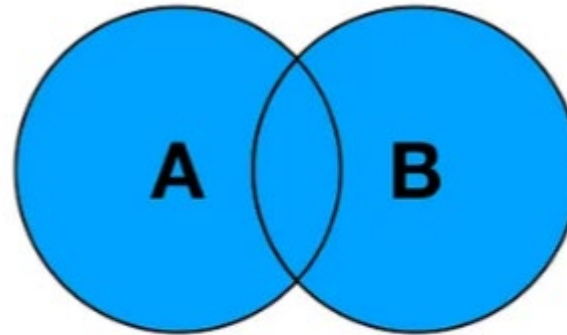
SQL recap – FULL OUTER JOIN

```
spark.sql("""
```

```
SELECT
    e.EmployeeID,
    e.Name,
    e.Salary,
    d.DepartmentName
FROM
    employees e
FULL OUTER JOIN
    departments d
ON
    e.DepartmentID = d.DepartmentID
```

```
""").show()
```

A FULL OUTER JOIN returns all rows when there is a match in either table, filling in NULLs where there are no matches.



FULL OUTER JOIN

SQL advanced – Handling dates

```
data = [  
    ("12/01/2023",),  
    ("11/15/2023",),  
    ("10/25/2023",),  
    ("09/17/2023",)  
]  
  
schema = StructType([StructField("date_string",  
    StringType(), True)])  
  
date_df = spark.createDataFrame(data, schema=schema)  
date_df.createOrReplaceTempView("dates")  
  
spark.sql("""  
  
    SELECT  
        date_string,  
        TO_DATE(date_string, 'MM/dd/yyyy') AS parsed_date  
    FROM  
        dates  
  
""").show()
```

The **TO_DATE** function converts a string into a DATE data type. This is useful when working with textual data containing date values.

The function requires the input string and the format in which the date is stored.

For example, `TO_DATE('2024-12-31', 'YYYY-MM-DD')` converts the string '2024-12-31' into a date. Always ensure the string matches the specified format to avoid parsing errors.

SQL advanced – Handling dates: without conversion

```
spark.sql("""
```

```
    SELECT
        YEAR(date_string) AS year
    FROM
        dates
```

```
""").show(10)
```

Without TO_DATE results will be NULL, because it can't handle this format (MM/mm/YYYY). Spark tries to automatically convert, but data has to be in the right format.

```
+-----+
|year|
+-----+
|NULL|
|NULL|
|NULL|
|NULL|
+-----+
```

SQL advanced – Handling dates: extracting parts

```
spark.sql("""
```

```
    SELECT
        TO_DATE(date_string, 'MM/dd/yyyy') AS parsed_date,
        YEAR(TO_DATE(date_string, 'MM/dd/yyyy')) AS year,
        MONTH(TO_DATE(date_string, 'MM/dd/yyyy')) AS month,
        DAY(TO_DATE(date_string, 'MM/dd/yyyy')) AS day,
        DATE_FORMAT(TO_DATE(date_string, 'MM/dd/yyyy'), 'E')
    AS weekday_short,
        DATE_FORMAT(TO_DATE(date_string, 'MM/dd/yyyy'),
        'EEEE') AS weekday
    FROM
        dates

""").show(10)
```

You can extract parts of the date, even the name of the day in shorter, or normal format (DATE_FORMAT()).

```
+-----+-----+-----+-----+-----+-----+
|parsed_date|year|month|day|weekday_short|  weekday|
+-----+-----+-----+-----+-----+-----+
| 2023-12-01|2023|  12|  1|        Fri|   Friday|
| 2023-11-15|2023|  11| 15|        Wed| Wednesday|
| 2023-10-25|2023|  10| 25|        Wed| Wednesday|
| 2023-09-17|2023|   9| 17|        Sun|   Sunday|
+-----+-----+-----+-----+-----+-----+
```

SQL advanced – Handling dates - Adding / removing

```
spark.sql("""
```

```
    SELECT
        TO_DATE(date_string, 'MM/dd/yyyy') AS parsed_date,
        DATE_ADD(TO_DATE(date_string, 'MM/dd/yyyy'), 10) AS
add_10_days,
        DATE_SUB(TO_DATE(date_string, 'MM/dd/yyyy'), 5) AS
subtract_5_days,
        ADD_MONTHS(TO_DATE(date_string, 'MM/dd/yyyy'), 2) AS
add_2_months,
        ADD_MONTHS(TO_DATE(date_string, 'MM/dd/yyyy'), -1)
AS subtract_1_month,
        ADD_MONTHS(TO_DATE(date_string, 'MM/dd/yyyy'), 12)
AS add_1_year
    FROM
        dates
```

```
""").show(10)
```

SQL supports adding or subtracting days, months, or years to/from a date using functions like DATE_ADD and DATE_SUB.

+-----+-----+-----+-----+-----+-----+					
parsed_date	add_10_days	subtract_5_days	add_2_months	subtract_1_month	add_1_year
+-----+-----+-----+-----+-----+-----+					
2023-12-01	2023-12-11	2023-11-26	2024-02-01	2023-11-01	2024-12-01
2023-11-15	2023-11-25	2023-11-10	2024-01-15	2023-10-15	2024-11-15
2023-10-25	2023-11-04	2023-10-20	2023-12-25	2023-09-25	2024-10-25
2023-09-17	2023-09-27	2023-09-12	2023-11-17	2023-08-17	2024-09-17
+-----+-----+-----+-----+-----+-----+					

SQL advanced – Handling dates – BETWEEN

```
spark.sql("""
```

```
    SELECT
        date_string,
        TO_DATE(date_string, 'MM/dd/yyyy') AS parsed_date
    FROM
        dates
    WHERE
        TO_DATE(date_string, 'MM/dd/yyyy') BETWEEN '2023-11-
01' AND '2023-12-31'

""").show()
```

The BETWEEN keyword is used to filter rows based on a date range.

```
+-----+-----+
|date_string|parsed_date|
+-----+-----+
| 12/01/2023| 2023-12-01|
| 11/15/2023| 2023-11-15|
+-----+-----+
```

SQL advanced – Handling dates: days between dates

```
spark.sql("""
```

```
    WITH date_diff_example AS (
        SELECT
            TO_DATE(date_string, 'MM/dd/yyyy') AS
parsed_date,
            CURRENT_DATE() AS today
        FROM
            dates
    )
    SELECT
        parsed_date,
        today,
        DATEDIFF(today, parsed_date) AS days_difference
    FROM
        date_diff_example
```

```
""").show()
```

The DATEDIFF function calculates the number of days between two dates.

parsed_date	today	days_difference
2023-12-01	2024-12-18	383
2023-11-15	2024-12-18	399
2023-10-25	2024-12-18	420
2023-09-17	2024-12-18	458

SQL advanced – TIMESTAMP type

```
spark.sql("""
```

```
    SELECT
        timestamp_column,
        TO_DATE(timestamp_column, 'yyyy/MM/dd') AS
parsed_date,
        HOUR(timestamp_column) AS hour,
        MINUTE(timestamp_column) AS minute,
        SECOND(timestamp_column) AS second,
        DATE_TRUNC('HOUR', timestamp_column) AS
truncated_hour
    FROM
        timestamps

""").show()
```

```
+-----+-----+-----+-----+-----+-----+
| timestamp_column | parsed_date | hour | minute | second | truncated_hour |
+-----+-----+-----+-----+-----+-----+
| 2024-12-31 23:59:59 | 2024-12-31 | 23 | 59 | 59 | 2024-12-31 23:00:00 |
| 2023-01-01 10:15:30 | 2023-01-01 | 10 | 15 | 30 | 2023-01-01 10:00:00 |
+-----+-----+-----+-----+-----+-----+
```

Sometimes you have to deal with **TIMESTAMP** columns which contains hours, minutes, seconds also.

Here's a selection of common functions, most of them are introduced with the DATE datatype.

SQL advanced – CASE WHEN

```
data = [  
    (1, "Alice", 25, "F"),  
    (2, "Bob", 17, "M"),  
    (3, "Catherine", 40, "F"),  
    (4, "David", 15, "M"),  
    (5, "Eva", 29, "F")  
]  
columns = ["id", "name", "age", "gender"]  
  
df = spark.createDataFrame(data, columns)  
df.createOrReplaceTempView("people")
```

id	name	age	gender
1	Alice	25	F
2	Bob	17	M
3	Catherine	40	F
4	David	15	M
5	Eva	29	F

CASE WHEN expression in Spark SQL is used for conditional logic, **similar to if-else** in programming. It allows you to evaluate conditions and return specific values based on those conditions.

Syntax: CASE WHEN THEN ELSE END AS.

```
spark.sql("""
```

```
    SELECT  
        id,  
        name,  
        age,  
        CASE  
            WHEN age >= 18 THEN 'Adult'  
            ELSE 'Minor'  
        END AS age_group  
    FROM  
        people  
""").show()
```

id	name	age	age_group
1	Alice	25	Adult
2	Bob	17	Minor
3	Catherine	40	Adult
4	David	15	Minor
5	Eva	29	Adult

SQL advanced – CASE WHEN: creating flags

```
spark.sql("""
```

```
SELECT
  id,
  name,
  gender,
  CASE
    WHEN gender = 'F' THEN 'Eligible'
    ELSE 'Not Eligible'
  END AS eligibility
FROM
  people
```

```
""").show()
```

id	name	gender	eligibility
1	Alice	F	Eligible
2	Bob	M	Not Eligible
3	Catherine	F	Eligible
4	David	M	Not Eligible
5	Eva	F	Eligible

CASE WHEN is often used when creating flags.

Multiple conditions can be used.

```
spark.sql("""
```

```
SELECT
  id,
  name,
  age,
  gender,
  CASE
    WHEN age >= 18 AND gender = 'F' THEN 'Adult Female'
    WHEN age >= 18 AND gender = 'M' THEN 'Adult Male'
    ELSE 'Minor'
  END AS category
FROM
  people
```

```
""").show()
```

id	name	age	gender	category
1	Alice	25	F	Adult Female
2	Bob	17	M	Minor
3	Catherine	40	F	Adult Female
4	David	15	M	Minor
5	Eva	29	F	Adult Female

SQL advanced – UNION / UNION ALL

Sample DataFrames:

DataFrame 1:

id	name	age	gender
1	Alice	25	F
2	Bob	17	M
3	Catherine	40	F

DataFrame 2:

3	Catherine	40	F
4	David	15	M
5	Eva	29	F

Queries:

```
spark.sql("""  
  
    SELECT * FROM df1  
    UNION  
    SELECT * FROM df2  
  
""").show()
```

```
spark.sql("""  
  
    SELECT * FROM df1  
    UNION ALL  
    SELECT * FROM df2  
  
""").show()
```

Results:

id	name	age	gender
1	Alice	25	F
2	Bob	17	M
3	Catherine	40	F
4	David	15	M
5	Eva	29	F

id	name	age	gender
1	Alice	25	F
2	Bob	17	M
3	Catherine	40	F
3	Catherine	40	F
4	David	15	M
5	Eva	29	F

Notice that UNION will apply a DISTINCT on the result set. If you need the possible duplications, use UNION ALL.

SQL advanced – CAST

```
data = [  
    ("2023-12-01", "100", "123.45"),  
    ("2024-01-15", "200", "678.90"),  
    ("2025-05-20", "300", "555.55"),  
    # ("2025-05-22", "abc", "555.55")  
]  
columns = ["date_string", "integer_string", "float_string"]  
  
df = spark.createDataFrame(data, columns)  
df.createOrReplaceTempView("data_casting")  
  
df.show()  
df.printSchema()
```

```
+-----+-----+-----+  
|date_string|integer_string|float_string|  
+-----+-----+-----+  
| 2023-12-01|         100|    123.45|  
| 2024-01-15|         200|    678.90|  
| 2025-05-20|         300|    555.55|  
+-----+-----+-----+  
  
root  
|-- date_string: string (nullable = true)  
|-- integer_string: string (nullable = true)  
|-- float_string: string (nullable = true)
```

Spark makes our life easier with the **implicit casting**, but here's how to explicitly cast values. First, here are some examples for implicit casting.

- Without casting, because of spark's implicit casting, sum can be calculated.
- (uncomment the fourth line and run again) Again without casting but with a non integer value ("abc") spark **still** can calculate, because behind the scenes the casting to integer fails, and it will be NULL, and NULLs are ignored in SUM, so the result will be 600 no matter if the fourth row is part of the DataFrame, or not.

Important: Traditional databases do not generally handle implicit casting as Spark does. Don't rely on aggregating on **string** columns, CAST explicitly them.

```
spark.sql("""  
  
    SELECT  
        SUM(integer_string) AS sum_int  
    FROM  
        data_casting  
  
""").show(10)
```

```
+-----+  
|sum_int|  
+-----+  
|   600.0|  
+-----+
```

SQL advanced – CAST

```
spark.sql("""
```

```
    SELECT
        SUM(CAST(integer_string AS INT)) AS sum_int
    FROM
        data_casting
```

```
""").show(10)
```

```
+-----+
|sum_int|
+-----+
|    600|
+-----+
```

CAST string to integer, and immediately can use the SUM. As seen, it can be calculated without explicit casting, but here's how to do it.

SQL advanced – COALESCE (working with NULLs)

```
data = [  
    (1, "Alice", 25, "F"),  
    (2, None, 17, "M"),  
    (3, "Catherine", None, "F"),  
    (4, "David", 15, "M"),  
    (5, "Eva", 29, "F")  
]  
columns = ["id", "name", "age", "gender"]  
  
df = spark.createDataFrame(data, columns)  
df.createOrReplaceTempView("people")  
  
df.show()
```

id	name	age	gender
1	Alice	25	F
2	NULL	17	M
3	Catherine	NULL	F
4	David	15	M
5	Eva	29	F

```
spark.sql("""  
  
    SELECT  
        id,  
        COALESCE(name, "unknown") AS name,  
        COALESCE(age, 0) AS age,  
        gender  
    FROM  
        people  
""").show(10)
```

id	name	age	gender
1	Alice	25	F
2	unknown	17	M
3	Catherine	0	F
4	David	15	M
5	Eva	29	F

COALESCE is used to substitute NULL values, like in the example NULL to “unknown”.

SQL advanced – Subqueries and CTE

A **subquery** is a **nested query** within another query. It is enclosed in parentheses and is typically used in the SELECT, FROM, or WHERE clause of the main query.

A **CTE** is a **temporary result set** defined within a **WITH** clause that can be referenced within the main query. It's similar to a subquery but is declared first and often improves readability. CTEs are designed to make queries easier to read and maintain by allowing complex queries to be broken into modular components.

Subqueries vs. CTEs

Feature	Subqueries	CTEs
Readability	Harder to read in complex queries	Easier to read and maintain
Reusability	Cannot reuse subquery results	Can reuse the CTE across the query
Performance	Similar in most cases	Similar, but CTEs might optimize better in Spark
Use case	Simple one-off calculations	Complex queries needing multiple steps

SQL advanced – Subqueries and CTE

Subqueries can be in SELECT / FROM / WHERE. Here's an example for being in the WHERE part.
Getting the trips with above average distance.

```
spark.sql("""  
  
    SELECT  
        COUNT(*) AS trips_with_above_avg_distance  
    FROM  
        taxi_2024_08  
    WHERE  
        trip_distance > (  
            SELECT AVG(trip_distance)  
            FROM taxi_2024_08  
        )  
    """).show(10)
```

With **CTE** the average calculation moved from the back of the query to the top.

Multiple WITH statements can be used in a query.

CTE is cleaner, easier to read. Often used to break complex queries to smaller logical parts.

```
spark.sql("""  
  
    WITH average_distance AS (  
        SELECT AVG(trip_distance) AS avg_distance  
        FROM taxi_2024_08  
    )  
    SELECT  
        count(*) AS trips_with_above_avg_distance  
    FROM  
        taxi_2024_08, average_distance  
    WHERE  
        trip_distance > avg_distance  
    """).show(10)
```

SQL advanced – Window functions

Window functions are powerful tools for performing operations across a set of rows related to the current row. They don't collapse rows into aggregates like GROUP BY does; instead, they allow you to calculate aggregates and rankings over a "window" of rows.

Syntax starts with an **action**, like SUM(), AVG(). The **OVER** clause defines the "window" (i.e., the subset of rows) on which the function operates. Mostly you will see **PARTITION BY** as the next keyword. This divides the dataset into groups (or partitions) for the window function to operate on. Without PARTITION BY, the window function operates on all rows as a single group.

We also have ROWS and RANGE but these are used much less than the PARTITION BY:

ROWS: Operates on a physical number of rows relative to the current row.

```
SUM(fare_amount) OVER (ORDER BY trip_distance ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
```

RANGE: Operates on a logical range of values (e.g., rows where a column value falls within a range).

```
SUM(fare_amount) OVER (ORDER BY trip_distance RANGE BETWEEN 10 PRECEDING AND 10 FOLLOWING)
```

- Common **aggregate** window functions: SUM(), AVG(), MIN(), MAX(), COUNT().
- Common **ranking** window functions: ROW_NUMBER(), RANK(), DENSE_RANK().
- Common **analytic** window functions: LEAD(), LAG().

SQL advanced – Window functions: aggregate

spark.sql("""

```
SELECT
    VendorID,
    fare_amount,
    COUNT(fare_amount) OVER (PARTITION BY VendorID) AS trip_count,
    MIN(fare_amount) OVER (PARTITION BY VendorID) AS min_fare,
    MAX(fare_amount) OVER (PARTITION BY VendorID) AS max_fare,
    AVG(fare_amount) OVER (PARTITION BY VendorID) AS avg_fare
FROM
    taxi_2024_08
WHERE
    VendorID = 1
""").show(10)
```

VendorID	fare_amount	trip_count	min_fare	max_fare	avg_fare
1	28.9	680242	0.0	700.0	19.268989918294295
1	52.0	680242	0.0	700.0	19.268989918294295
1	17.0	680242	0.0	700.0	19.268989918294295
1	5.1	680242	0.0	700.0	19.268989918294295
1	5.8	680242	0.0	700.0	19.268989918294295
1	45.0	680242	0.0	700.0	19.268989918294295
1	7.2	680242	0.0	700.0	19.268989918294295
1	10.7	680242	0.0	700.0	19.268989918294295
1	8.6	680242	0.0	700.0	19.268989918294295
1	44.3	680242	0.0	700.0	19.268989918294295

SQL advanced – Window functions: ROW_NUMBER

```
spark.sql("""
```

```
SELECT
    VendorID,
    PULocationID,
    DOLocationID,
    fare_amount,
    ROW_NUMBER() OVER (PARTITION BY VendorID ORDER BY fare_amount DESC) AS row_num
FROM
    taxi_2024_08
WHERE
    VendorID = 1
```

```
""").show(10)
```

Assign a unique number to each row within the partition of a result set. WHERE clause is to show different VendorIDs.

VendorID	PULocationID	DOLocationID	fare_amount	row_num
1	145	145	700.0	1
1	162	162	655.35	2
1	132	265	650.0	3
1	132	265	650.0	4
1	132	265	627.4	5
1	138	265	580.0	6
1	132	265	550.0	7
1	132	265	543.4	8
1	164	265	509.1	9
1	145	145	500.0	10

SQL advanced – Window functions: RANK / DENSE_RANK

```
spark.sql("""
```

```
SELECT
  VendorID,
  PULocationID,
  DOLocationID,
  fare_amount,
  RANK() OVER (PARTITION BY VendorID ORDER BY fare_amount DESC) AS rank,
  DENSE_RANK() OVER (PARTITION BY VendorID ORDER BY fare_amount DESC) AS dense_rank
FROM
  taxi_2024_08
WHERE
  VendorID = 1
```

```
""").show(10)
```

RANK: Assign a rank based on a specific column, and partition by another column. Order the fare_amount in descending by VendorID. WHERE clause is to show different VendorIDs.

DENSE_RANK: It is often a question in interviews, what's the difference between RANK and DENSE_RANK? It's similar to RANK, but it **doesn't skip the ranks** if there are duplicates.

In RANK the order was 1, 2, 3, 3, 5!, 6, while in DENSE_RANK it is 1, 2, 3, 3, 4, 5.

VendorID	PULocationID	DOLocationID	fare_amount	rank	dense_rank
1	145	145	700.0	1	1
1	162	162	655.35	2	2
1	132	265	650.0	3	3
1	132	265	650.0	3	3
1	132	265	627.4	5	4
1	138	265	580.0	6	5
1	132	265	550.0	7	6
1	132	265	543.4	8	7
1	164	265	509.1	9	8
1	145	145	500.0	10	9

SQL advanced – Window functions: LEAD / LAG

```
spark.sql("""  
    SELECT  
        VendorID,  
        fare_amount,  
        LAG(fare_amount) OVER (PARTITION BY VendorID ORDER BY tpep_pickup_datetime) AS prev_fare,  
        LEAD(fare_amount) OVER (PARTITION BY VendorID ORDER BY tpep_pickup_datetime) AS next_fare  
    FROM  
        taxi_2024_08  
    WHERE  
        VendorID = 1  
""").show(10)
```

Get the next and previous fare_amount by VendorID.

VendorID	fare_amount	prev_fare	next_fare
1	70.0	NULL	37.3
1	37.3	70.0	11.4
1	11.4	37.3	16.3
1	16.3	11.4	45.0
1	45.0	16.3	27.5
1	27.5	45.0	34.5
1	34.5	27.5	6.5
1	6.5	34.5	8.6
1	8.6	6.5	12.8
1	12.8	8.6	70.0

SQL advanced – SQLite

Download the IMDB title basics, and title ratings [here](#).

SQLite [home page](#).



SQL advanced – Indexes

What are the indexes?

Indexes in SQL are special data structures **used to improve the performance** of database queries. They work like an index in a book, allowing the database to locate rows more quickly without scanning the entire table.

Key Points about Indexes:

Purpose:

Speed up SELECT queries.

Improve JOIN, WHERE, ORDER BY, and GROUP BY operations.

Trade-Off:

Indexes require additional disk space.

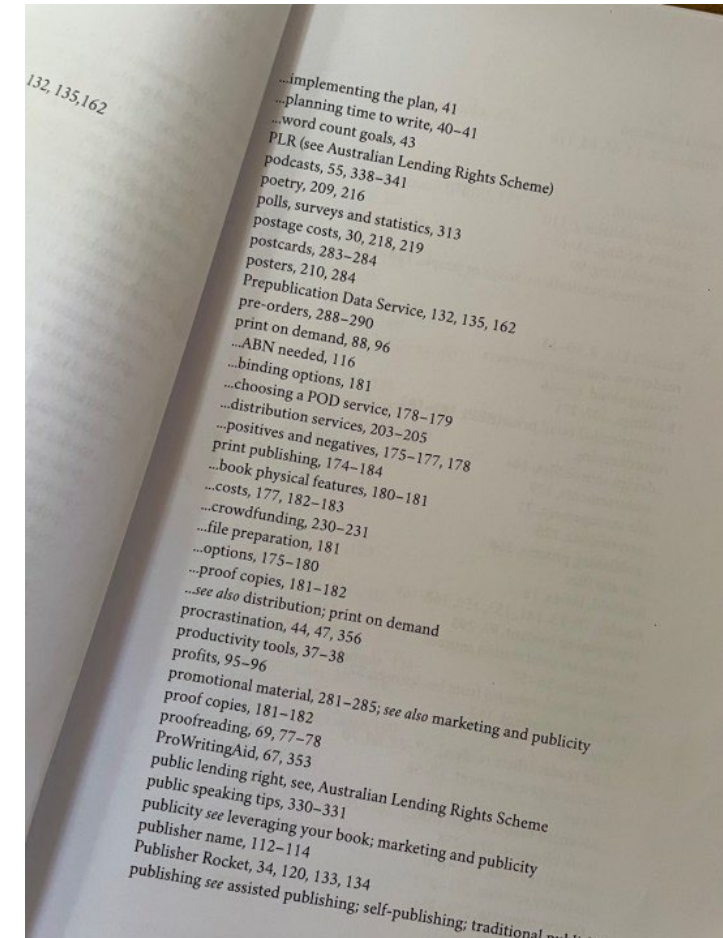
They slow down INSERT, UPDATE, and DELETE operations since the index also needs to be updated.

Underlying Mechanism:

Typically implemented as B-Trees or Hash Tables for efficient lookups.

When Should Indexes be Created?

- Indexes are particularly beneficial for large datasets where full table scans would be slow.
- A column contains a wide range of values.
- A column does not contain a large number of null values.
- One or more columns are frequently used together in a where clause or a join condition.



```
CREATE INDEX idx_average_rating ON title_ratings (averageRating DESC);
```

SQL advanced – Indexes

When not to use Indexes?

Small Tables: For very small tables, the overhead of maintaining indexes may not be worth the performance gains.

Frequent Inserts/Updates: If a table is frequently updated or inserted into, maintaining indexes can be costly because each modification requires updating the indexes.

Low-Cardinality Columns: Indexing columns with a small number of unique values (low cardinality) often doesn't provide significant benefits.

Types of indexes:

Clustered index:

A clustered index is where the data in the table is physically sorted according to the order of the index.

Each table can have only one clustered index because the rows of the table can only be sorted in one order.

The primary key constraint automatically creates a clustered index on the primary key column.

Non-clustered indexes:

A non-clustered index creates a separate object from the table. It contains pointers to the actual table data.

A table can have multiple non-clustered indexes.

Non-clustered indexes are useful when you frequently query specific columns, but these columns don't need to be part of the table's physical order.

SQL advanced – Views

A view is a **virtual table** in SQL that is based on the result of a SELECT query.

It **does not store data physically** but provides a way to simplify complex queries, enhance security by limiting access to certain data, and abstract data structure changes from end users.

Views can be queried like regular tables, and they update dynamically when the underlying data changes.

Views in SQL can help maintain data security by providing controlled access to data.

A view can **include only specific columns or rows** that a user or group is allowed to access, hiding sensitive data.

```
CREATE VIEW public_employee_info AS
SELECT
EmployeeID,
Name,
Department
FROM
Employees
WHERE
Role != 'Admin';
```

It can **filter rows based on specific conditions**, such as limiting access to data relevant to a user's department or role

```
CREATE VIEW department_data AS
SELECT
*
FROM
Employees
WHERE
Department = 'Sales';
```

```
CREATE VIEW topRated_movies AS
SELECT
tb.primaryTitle,
tb.startYear,
tr.averageRating,
tr.numVotes
FROM
title_basics tb
INNER JOIN
title_ratings tr
ON
tb.tconst = tr.tconst
WHERE
tb.titleType = 'movie'
AND tb.startYear BETWEEN 2010 AND 2020
AND tr.averageRating > 7
AND tr.numVotes > 1000;

SELECT * FROM topRated_movies;
```

SQL advanced – DML / DDL

DML and DDL are types of SQL commands:

DML stands for **Data Manipulation Language**, and contains commands like:

- **SELECT** - Retrieves data from one or more tables or views.
- **INSERT** - Adds new rows to a table.
- **UPDATE** - Modifies existing rows in a table.
- **DELETE** - Remove rows from a table.
- **MERGE** - Combines INSERT, UPDATE, and optionally DELETE operations in a single statement, often used for upserts. (More on the next slide.)

DDL is **Data Definition Language** with commands like:

- **CREATE** - Creates a new table, view, index, or other database object.
- **ALTER** - Modifies the structure of an existing database object.
- **DROP** - Removes an existing database object.
- **TRUNCATE** - Deletes all rows from a table but retains its structure.
- **RENAME** - Renames a database object.

```
SELECT * FROM customers WHERE age > 30;
INSERT INTO customers (id, name, age) VALUES (1, 'John Doe', 35);
UPDATE customers SET age = 36 WHERE id = 1;
DELETE FROM customers WHERE id = 1;
```

```
CREATE TABLE customers (
  id INT PRIMARY KEY,
  name TEXT,
  age INT
);
ALTER TABLE customers ADD COLUMN email TEXT;
DROP TABLE customers;
TRUNCATE TABLE customers;
ALTER TABLE customers RENAME TO clients;
```

SQL advanced – DML - Merge

The MERGE statement is a powerful SQL feature that combines **INSERT**, **UPDATE**, and optionally **DELETE** operations into a single command.

It is typically used for **upserts**, where rows are inserted if they don't exist or updated if they already do.

Not all SQL engines support MERGE. Microsoft SQL Server, Oracle, PostgreSQL, Snowflake, BigQuery do, SQLite, MySQL don't.

Key Features:

Source Table and Target Table:

- The MERGE statement compares a source table (or query) with a target table.
- Rows are matched based on a condition, often involving primary or unique keys.

Operations in MERGE:

- **INSERT**: Adds rows from the source table to the target table if they do not already exist.
- **UPDATE**: Updates rows in the target table that match rows in the source table.
- **DELETE** (Optional): Removes rows from the target table based on conditions.

Simplifies Complex Logic:

- Without MERGE, you would typically write separate INSERT, UPDATE, and possibly DELETE statements, often involving EXISTS or NOT EXISTS subqueries. MERGE consolidates this logic into one concise statement.
- Used a lot in SCD2 scenarios (more about SCD2 later).

MERGE INTO

```
customers AS target
USING
new_customers AS source
ON
target.id = source.id
WHEN MATCHED THEN
UPDATE SET
    target.name = source.name,
    target.age = source.age
WHEN NOT MATCHED THEN
INSERT (id, name, age)
VALUES (source.id, source.name, source.age)
WHEN NOT MATCHED BY SOURCE THEN
DELETE; -- Optional
```

- **WHEN MATCHED**: Defines what happens when rows in the source and target match.
- **WHEN NOT MATCHED**: Defines what happens when a source row has no match in the target.
- **WHEN NOT MATCHED BY SOURCE**: Defines what happens when a target row has no match in the source (optional, typically used for deletes).