“ CDC project ”

This Spark session enables:

- Extract employee data from MySQL
- Apply **CDC logic** (insert / update / delete)
- Load transformed data back to MySQL
- Run everything locally from **Jupyter Notebook**

CDC Employee Data Pipeline – Project Explanation

Aim of the Project

The aim of this project is to keep employee data up to date by identifying and applying only the changes—such as new employees, updates, or removals—into another table, without reprocessing or reloading all the existing data every time.

Main Motive of the Project

The main motive of this project is to keep employee information updated in a smart and efficient way without repeatedly processing all the data. Instead of refreshing the entire employee list every time a change occurs, the system focuses only on what has changed.

When new employees join, existing employee details change, or employees leave the organization, only those specific changes are identified and applied. This saves time and effort while ensuring accuracy.

The updated employee information is then stored in a separate table that always shows the latest and correct data. At the same time, the original employee data remains untouched and safe.

This approach makes the overall system faster, more reliable, and easier to scale as the amount of data grows, which is important for real-world business operations.

Technologies Used

- **PySpark** – For distributed data processing and CDC transformations
- **Apache Spark SQL** – For DataFrame-based operations
- **MySQL** – Source and target relational database
- **JDBC Connector** – To connect Spark with MySQL
- **CSV** – Used as CDC change input (Insert / Update / Delete records)
- **Jupyter Notebook** – Development and testing environment

CDC Data Processing Workflow

1 Data Extraction

- Employee data is read from a **MySQL source table** into Spark using JDBC.
- This data acts as the **current baseline snapshot**.

2 CDC Change Ingestion

- A CDC CSV file is read into Spark.
- This file contains incremental changes marked as:
  - INSERT
  - UPDATE
  - DELETE

3 CDC Segregation

- The CDC data is split into three DataFrames:
  - Updates

    - Inserts
    - Deletes
- This allows independent and accurate handling of each change type.

4 UPDATE Processing

- Existing employee records are updated using a **left join**.
- The `coalesce()` function ensures:
    - Only changed fields are updated
    - Unchanged fields remain intact

5 DELETE Processing

- Employees marked for deletion are removed using a **left anti-join**.
- This efficiently excludes deleted records from the dataset.

6 INSERT Processing

- New employee records are created using CDC insert data.
- Default values are assigned where full details are not available.

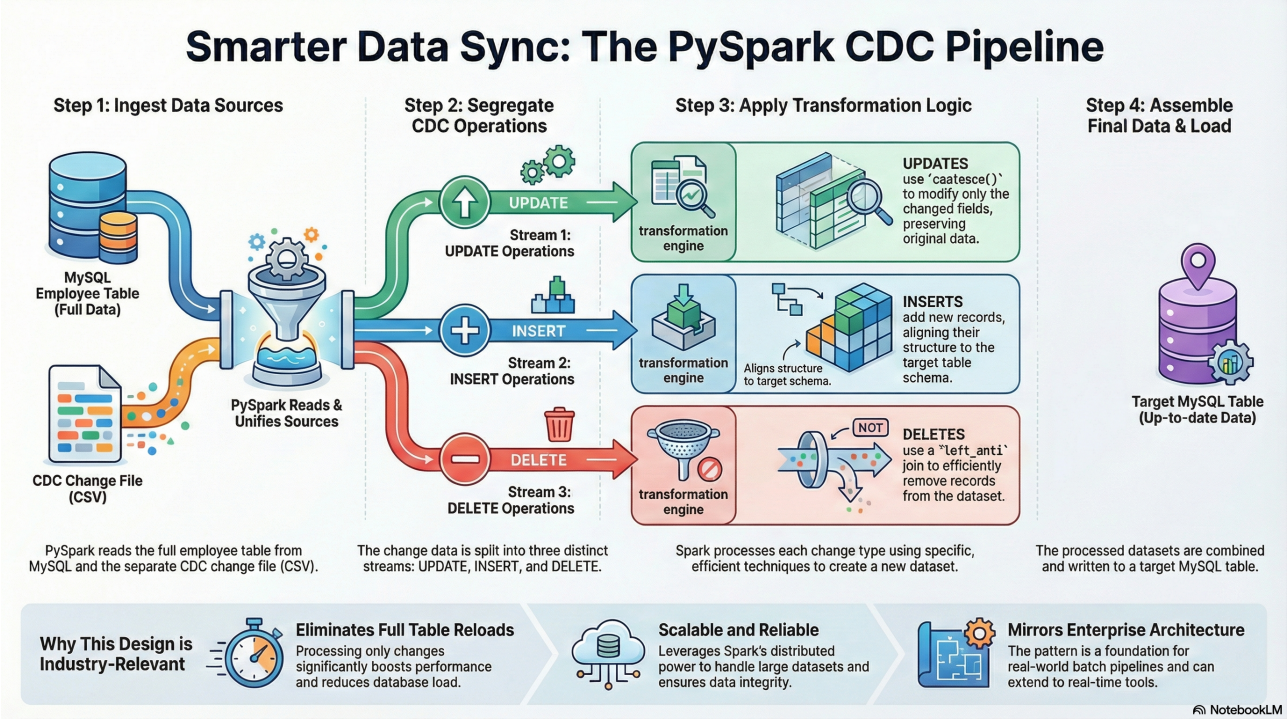
7 Final Assembly

- Updated and cleaned records are combined with new records using `unionByName`.
- This produces the **final CDC-applied dataset**.

8 Load to Target Table

- The final dataset is written to a **target MySQL table**.
- The target table always reflects the **latest state of data**.

Project architecture



What this architecture represents

This architecture shows a **batch Change Data Capture (CDC) pipeline** where PySpark processes incremental employee data changes and synchronizes them into a MySQL target table.

Left Side: Source Systems

1 MySQL Source Database

- Contains the **existing employee data**
- Acts as the **baseline snapshot**
- Data is read using **JDBC**
- This table is **not modified directly**

Purpose:

To provide the **current state of employee records**

2 CDC Input File (Side Input)

- Contains **incremental changes**
- Each record is tagged as:
  - INSERT
  - UPDATE
  - DELETE
- Simulates real-world CDC tools

Purpose:

To tell the system **what has changed since the last run**

Middle: PySpark CDC Processing Engine

This is the **core of the architecture**.

What PySpark does here:

- Reads source data from MySQL
- Reads CDC change data from CSV
- Separates CDC operations:
  - Updates
  - Inserts
  - Deletes

CDC Logic Inside Spark:

- **UPDATE**

  Uses joins and `coalesce()` to update only changed fields

- **DELETE**

  Uses left anti-join to remove records

- **INSERT**

  Creates new records and aligns schema

Purpose:

To apply CDC rules in a **scalable and controlled manner**

Right Side: Target System

3 MySQL Target Database

- Stores the **final CDC-applied data**
- Always reflects the **latest state**
- Separate from the source table

Purpose:

To serve as the **clean, updated dataset** for reporting and analytics

End-to-End Data Flow

“Employee data is read from the MySQL source database, while incremental changes are read from a CDC file. PySpark applies update, delete, and insert logic to these datasets and produces a final, consistent employee dataset, which is then written back to a MySQL target table.”

Code & Explanation :

```
from pyspark.sql import SparkSession


spark = SparkSession.builder \
    .appName("CDC_Employee_Project") \
    .config(
        "spark.jars",
        "/path/pyspark/jdbc/mysql-connector-j-8.4.0/mysql-connector-j-8.4.0.jar"
    ) \
    .getOrCreate()
```

1 `from pyspark.sql import SparkSession`

- Imports **SparkSession**, the **entry point of PySpark**
- Without this, you **cannot create DataFrames, run SQL, or connect to JDBC**
- Think of it as the **engine starter** for Spark

2 `SparkSession.builder`

- Uses the **builder pattern** to configure Spark
- Allows you to set:
  - App name
  - External JARs
  - Memory, cores, configs, etc.

3 `.appName("CDC_Employee_Project")`

- Assigns a **logical name** to your Spark job
- Useful for:
  - Spark UI
  - Logs
  - Debugging
- In production, this helps identify **which pipeline is running**

4 `.config("spark.jars", "...mysql-connector...jar")`

- **Most important line for JDBC**

- Tells Spark:

  “Load this MySQL JDBC driver so I can talk to MySQL”

Why this is needed:

- Spark **does not ship with MySQL drivers**
- JDBC communication **fails without this JAR**
- Required for:
  - `spark.read.format("jdbc")`
  - `df.write.format("jdbc")`

* Your path means:

- Local MySQL connector JAR
- Used for **local database connectivity** (not cloud)

`5.getOrCreate()`

- Creates a **new SparkSession** if one doesn’t exist
- Or **reuses an existing one**
- Prevents multiple Spark contexts from crashing your job

This code **starts Spark**, names your job, and **loads the MySQL driver** so PySpark can **read/write data to a local MySQL database using JDBC**.

```
jdbc_url = "jdbc:mysql://127.0.0.1:3306/company_db"
properties = {
    "user": "root",
    "password": “Your_Password”,
    "driver": "com.mysql.cj.jdbc.Driver"
}
employee_df = spark.read.jdbc(
    url=jdbc_url,
    table="employee_data",
    properties=properties
)
employee_df.show()
```

This code **connects PySpark to a MySQL database**, **reads the employee table**, and **loads it into a Spark DataFrame** so that **CDC (Create, Update, Delete) operations can be applied**.

1 `jdbc_url = "jdbc:mysql://127.0.0.1:3306/company_db"`

- Defines the **JDBC connection URL**
- Components:
  - `jdbc:mysql://` → JDBC protocol for MySQL
  - `127.0.0.1` → Local MySQL server
  - `3306` → Default MySQL port
  - `company_db` → Database name

This tells Spark **where the source database lives**.

2 `properties = { ... }`

```
properties = {
    "user": "root",
    "password": "*****",
    "driver": "com.mysql.cj.jdbc.Driver"
}
```

- Contains **authentication and driver details**
- `user` → MySQL username
- `password` → MySQL password
- `driver` → MySQL JDBC driver class

These properties allow Spark to **authenticate and establish a JDBC connection**.

3 `spark.read.jdbc(...)`

```
employee_df = spark.read.jdbc(
    url=jdbc_url,
    table="employee_data",
    properties=properties
)
```

- Reads the **employee_data table** from MySQL
- Loads it into a **Spark DataFrame**
- This is the **EXTRACT phase** of your data pipeline

At this stage:

- Data is **immutable**
- No changes are applied yet
- Spark holds a **snapshot of source data**

4 `employee_df.show()`

- Displays sample records from the DataFrame
- Used for:
  - Validation
  - Debugging
  - Confirming successful extraction

Simple CDC Flow Mapping

```
MySQL (employee_data)
        ↓
spark.read.jdbc
        ↓
Spark DataFrame (employee_df)
        ↓
CDC Transformations (Insert / Update / Delete)
        ↓
Target MySQL Table
```

```
cdc_df = spark.read \
    .option("header", "true") \
    .csv("/your_path/pyspark/employees_cdc.csv")
cdc_df.show()
```

This code **reads the CDC input file** (CSV) into Spark.

The file contains **change records** that tell Spark **which employee records to INSERT, UPDATE, or DELETE**.

1 spark.read

- Starts the **DataFrame reader**
- Used for reading external data sources like CSV, JSON, Parquet, etc.

In CDC terms:

This is the **change feed source**, not the main table.

2 .option("header", "true")

- Tells Spark that the **first row contains column names**
- Prevents Spark from treating headers as data

Important for CDC:

- Ensures correct **column mapping**
- Avoids schema mismatch during joins and updates

3 .csv("employees_cdc.csv")

- Reads the CDC file into Spark
- This file typically contains:
    - Employee ID
    - Employee details
    - Operation type (Insert / Update / Delete)

This file represents **incremental changes**, not full data.

4 cdc_df.show()

- Displays the CDC records
- Used to verify:
    - Data is loaded correctly
    - Operation flags are present

- No schema issues

* Role of `employee_df`

- Holds the **current snapshot** from MySQL
- Represents **existing records**

* Role of `cdc_df` (this code)

- Holds **new changes**
- Represents **what has changed since last load**

CDC Logic Flow (Conceptual)

```
employee_df  (existing data)
        +
cdc_df       (change data)
        ↓
Apply CDC rules:
- INSERT → New employee_id not present
- UPDATE → employee_id present with updated values
- DELETE → employee_id marked for deletion
        ↓
final_df
```

```
# Separate CDC Operations (UPDATE / INSERT / DELETE)
from pyspark.sql.functions import col, coalesce, lit
updates_df = cdc_df.filter(col("operation") == "UPDATE")
inserts_df = cdc_df.filter(col("operation") == "INSERT")
deletes_df = cdc_df.filter(col("operation") == “DELETE")
```

Heart of CDC logic

This code **splits the CDC change data** into **three separate DataFrames** based on the **type of change**:

- **UPDATE**

- **INSERT**
- **DELETE**

This makes CDC processing **clear, controlled, and scalable**.

```
1 from pyspark.sql.functions import col,
coalesce, lit
```

- `col()` → Refers to a DataFrame column
- `coalesce()` → Used later to update values safely
- `lit()` → Used to add constant values (flags, timestamps, etc.)

In this block:

- Only `col()` is used
- `coalesce` and `lit` are imported for **later CDC transformations**

```
2 updates_df = cdc_df.filter(col("operation")
== "UPDATE")
```

- Filters CDC records where the operation is **UPDATE**
- These records represent:
  - Existing employees
  - With one or more changed fields

CDC meaning:

- Modify existing rows
- Match using **primary key (employee_id)**

```
3 inserts_df = cdc_df.filter(col("operation")
== "INSERT")
```

- Filters records marked as **INSERT**
- Represents **new employees**
- These records do **not exist** in the target table yet

CDC meaning:

- Append new rows to target table

4 deletes_df = cdc_df.filter(col("operation") == "DELETE")

- Filters records marked as **DELETE**
- Represents employees that must be **removed**
- Usually only the **primary key** is required

CDC meaning:

- Remove rows from target
- Or mark them as inactive (soft delete)

Step-by-Step CDC Pipeline

```
employee_df   → Current data (from MySQL)
cdc_df        → Change data (CSV)
cdc_df
 ├── updates_df  → update existing rows
 ├── inserts_df  → add new rows
 └── deletes_df  → remove rows
Apply logic → final_df → write back to MySQL
```

used in real projects

- Improves **readability**
- Easier **debugging**
- Allows independent handling of:
  - Update joins
  - Insert unions
  - Delete filters
- Mirrors **enterprise CDC design patterns**

```
# Apply UPDATE Logic
# Update salary and city only if present in CDC file.

updated_employee_df = employee_df.alias("emp") \
    .join(updates_df.alias("cdc"), "emp_id", "left") \
    .select(
        col("emp_id"),
        col("emp.emp_name"),
        col("emp.dept"),
        coalesce(col("cdc.city"), col("emp.city")).alias("city"),#take new value if available, else keep old value
        coalesce(col("cdc.salary"), col("emp.salary")).alias("salary")
    )
```

This is **the most important CDC block**.

This code **applies UPDATE changes** from the CDC file to the existing employee data.

Only the columns present in the CDC update file are modified
All other values remain unchanged

This is **exactly how CDC UPDATE works in real projects**.

`1 updated_employee_df = employee_df.alias("emp")`

- Creates an alias **emp** for the main employee table
- Helps avoid column ambiguity during joins

`employee_df` = current snapshot (from MySQL)

2 .join(updates_df.alias("cdc"), "emp_id", "left")

- Performs a **LEFT JOIN** on `emp_id`
- Meaning:
  - Keep **all existing employees**
  - Bring update values **only where emp_id matches**

Why LEFT JOIN?

- If an employee has **no update**, their row is still retained
- Missing CDC values become `NULL`

3 .select(col("emp_id"), col("emp.emp_name"), col("emp.dept"), ...)

- Explicitly selects required columns
- Ensures **schema control**
- Avoids duplicate or unwanted columns

4 coalesce(col("cdc.city"), col("emp.city")).alias("city")

This is the core UPDATE logic

`coalesce()` returns the **first non-null value**

Scenario	Result
CDC has new city	CDC city used
CDC city is NULL	Old city retained

Translation:

“Update city **only if a new value is provided** in CDC.”

5 `coalesce(col("cdc.salary"), col("emp.salary")).alias("salary")`

- Same logic applied to salary
- Updates salary **only when CDC contains a new value**
- Prevents overwriting with NULLs

CDC UPDATE Flow (Conceptual)

```
employee_df (existing)
      ⬅ LEFT JOIN ➜ updates_df (CDC updates)
      ⬇
coalesce(new_value, old_value)
      ⬇
updated_employee_df
```

Why `coalesce` is critical in CDC

- Prevents **data loss**

- Handles **partial updates**
- Mimics **real-world CDC tools**
- Ensures idempotent updates

```
# Apply DELETE Logic
# Remove employees whose emp_id exists in DELETE records.
after_delete_df = updated_employee_df.join(
    deletes_df.select("emp_id"),
    on="emp_id",
    how="left_anti"
)
# left_anti = Give me rows from left side that do NOT exist in delete list
```

This is the **CDC DELETE logic**.

This code **removes employees** from the dataset **only if they appear in the CDC DELETE file**.

If an `emp_id` is marked as DELETE → it is **excluded**
All other employees remain untouched

This is how **DELETE is implemented in Spark CDC pipelines**.

1 `updated_employee_df.join(...)`

- Starts with the **already updated data**
- This ensures:
    - UPDATE is applied first
    - DELETE is applied on top of updated records

CDC order matters:

UPDATE → DELETE → INSERT

2 `deletes_df.select("emp_id")`

- Selects **only the primary key**
- DELETE operation does **not need full row data**
- Keeps the join efficient

CDC principle:

Delete decisions are always **key-based**

3 `on="emp_id"`

- Uses `emp_id` as the **join key**
- Identifies which records must be deleted

4 `how="left_anti"`

This is the key part

`left_anti` join means:

- Keep rows from **left DataFrame**
- **Exclude** rows that have a match in the right DataFrame

In simple terms:

“Keep employees **except** those listed in delete file.”

Example (Easy to Understand)

updated_employee_df

emp_id	name	city
101	A	Hyd
102	B	Blr
103	C	Pune

deletes_df

emp_id
102

after_delete_df (Result)

emp_id	name	city
101	A	Hyd
103	C	Pune

How this fits in your CDC pipeline

```
employee_df
   ↓
Apply UPDATE logic
   ↓
updated_employee_df
   ↓
Apply DELETE logic (left_anti join)
   ↓
after_delete_df
```

Why `left_anti` is used in real projects

- Very efficient for deletes
- Cleaner than filters or subqueries
- Commonly used in:
    - Data Lakes
    - Spark CDC pipelines
    - Slowly Changing Dimensions

```
# Apply INSERT Logic

# Create new employee records.

new_employees_df = inserts_df.select(

    col("emp_id"),

    lit("New Employee").alias("emp_name"),

    lit("NA").alias("dept"),

    col("city"),

    col("salary")

)
```

This is the **CDC INSERT logic**.

This code **creates new employee records** for employees marked as **INSERT** in the CDC file.

These employees **do not exist** in the current employee table

Spark builds **new rows** with default and CDC-provided values

1 new_employees_df = inserts_df.select(...)

- Uses only the **INSERT CDC records**
- Prepares them in the **same schema** as the target employee table
- Required before merging with existing data

CDC rule:

Insert records must match the target table structure.

2 col("emp_id")

- Takes the employee ID from the CDC file
- Acts as the **primary key**
- Ensures uniqueness when merged

3 lit("New Employee").alias("emp_name")

- Assigns a **default name**
- Used when CDC does not provide full employee details

Real-world use:

- Placeholder values
- Later enrichment from another source

4 lit("NA").alias("dept")

- Assigns a default department
- Ensures **non-null consistency**
- Avoids schema mismatch

5 `col("city")`

- Takes city directly from CDC file
- Represents **newly introduced data**

6 `col("salary")`

- Takes salary from CDC file
- Completes the required fields

How this fits into your CDC pipeline

```
inserts_df (CDC new records)
        ↓
Build new rows with defaults
        ↓
new_employees_df
        ↓
Union with existing data
```

```
# Final CDC Result

# Combine updated + inserted records.

final_df = after_delete_df.unionByName(new_employees_df)

final_df.show(30)
```

This is the **final CDC assembly step**.

This code **combines all CDC-processed data** into one final dataset:

- **Updated records** (UPDATE applied)
- **Deleted records removed**
- **New records inserted**

The result represents the **latest, correct state** of the employee table.

`1 final_df = after_delete_df.unionByName(new_employees_df)`

Why `unionByName`?

- Combines two DataFrames **row-wise**
- Matches columns **by name**, not position
- Prevents schema mismatch errors

CDC reasoning:

- `after_delete_df` → existing employees after UPDATE + DELETE
- `new_employees_df` → brand-new employees
- Union = **final state**

2 Why INSERT is applied last

CDC best practice order:

```
UPDATE  → modify existing rows
DELETE  → remove unwanted rows
INSERT  → add new rows
```

Applying INSERT last avoids:

- Duplicate keys
- Overwriting updated data
- Inconsistent results

3 `final_df.show(30)`

- Displays the final CDC result
- Used to:
  - Validate CDC logic
  - Verify row counts
  - Confirm updates, deletes, inserts

End-to-End CDC Flow (Your Project)

```
MySQL employee_data
        ↓
spark.read.jdbc
        ↓
employee_df
        ↓
Apply UPDATE (coalesce)
        ↓
```

```
updated_employee_df
        ↓
Apply DELETE (left_anti)
        ↓
after_delete_df
        ↓
Apply INSERT (unionByName)
        ↓
final_df
```

```
final_df.write.jdbc(
   url=jdbc_url,
   table="employee_target",
   mode="overwrite",
   properties=properties
)
```

This is the **LOAD phase** of your CDC pipeline.

This code **writes the final CDC-processed data** from Spark **back into MySQL**, storing the **latest employee state** in a target table.

This completes the **ETL + CDC lifecycle**.

1 `final_df.write.jdbc(...)`

- Uses Spark's **JDBC writer**

- Sends data from a Spark DataFrame to a relational database
- Acts as the **LOAD step** in your CDC pipeline

2 `url=jdbc_url`

- Specifies the **database connection**
- Same JDBC URL used during extraction
- Ensures consistency between source and target systems

3 `table="employee_target"`

- Target table where CDC results are stored
- Keeps **source table untouched**
- Follows best practice:
    - Source → Read-only
    - Target → CDC-applied data

This allows:

- Auditing
- Rollback
- Easy comparison

4 `mode="overwrite"`

- Drops existing data in the target table
- Replaces it with the **latest full snapshot**

CDC perspective:

- Suitable for **batch CDC**
- Guarantees data consistency
- Avoids partial loads

In real-time CDC, this might be `append` or `merge`, but for batch CDC, `overwrite` is common.

`5 properties=properties`

- Uses database credentials and JDBC driver
- Enables authentication and communication

How this fits into your CDC project

```
Source MySQL (employee_data)
        ↓
Spark (CDC logic)
        ↓
final_df
        ↓
Target MySQL (employee_target)
```

- `employee_target` always reflects the **current truth**
- Can be used for:
    - Reporting
    - Analytics
    - Downstream pipelines

Conclusion :

In conclusion, this CDC project provides a robust mechanism for maintaining data consistency between source systems and the SQL database by efficiently handling incremental data changes. The initial dataset is loaded into the SQL database, after which all insert, update, and delete operations are processed in Spark using well-defined transformations and business logic. Instead of reloading the entire dataset, only the changed records are identified and applied, which significantly improves performance and scalability. After processing, the latest and accurate state of the data is written back to the SQL database. This ensures data integrity, reduces processing time, and enables near real-time availability of updated data for downstream analytics and reporting use cases.

CDC in Real-World Environments:

- Used in **HR systems** to track employee joins, updates, and exits
- Used in **banking and finance** to keep customer data synchronized
- Used in **retail and e-commerce** for order and customer updates
- Used in **healthcare systems** to maintain up-to-date patient records
- Used in **data warehouses and reporting systems** for fresh analytics data
- Used during **cloud migration** to sync data between old and new systems
- Used in **enterprise applications** where data changes frequently
- Helps avoid **full data reloads**, improving performance and efficiency