Data Engineering 0602

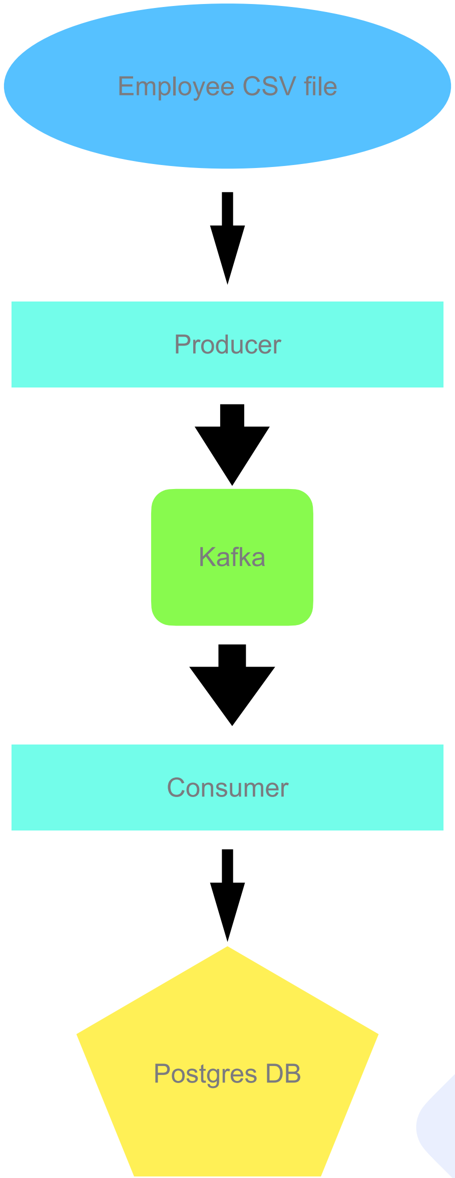
Kafka Project 1

Data Engineering 0602

Kafka Project 1: ETL pipeline

- This project will emphasize on creating a data pipeline in which CSV files are being generated on a regular basis, and then data is being cleaned, transformed and then later persisted into some storage for data analytics.

- To calculate the total salary of the departments.



Producer will ingest the CSV file, and send data to Kafka

Consumer will persist the data to our Postgres DB

Data Producer

DataProducer Responsibilities - (Extract + Transform)

- Ingest Employee_Salaries.csv file (will be in resources folder)

- Perform these transformations -
  - Ingest only these Departments -
    - ECC
    - CIT
    - EMS
  - Round off the Salary to lower number
  - Employees hired after 2010 (>= 2010)

- Send this data to Kafka

Data Consumer

DataConsumer Responsibilities - (Load)

- Ingest the data into the **Department_Employee** Table which will have this schema -
  - department_division: varchar
  - position_title: varchar
  - hire_date: Date
  - salary: int32
- With every message, also update the total salary given by each department. Schema for the department table -
  - department: varchar
  - total_salary: int64

Table Schema

```
CREATE TABLE department_employee(

department VARCHAR(200),

department_division   VARCHAR(200),

position_title VARCHAR(200),

hire_date DATE,

salary decimal

);
```

```
CREATE TABLE
public.department_employee_salary (

department varchar(200) NOT NULL,

total_salary int4 NULL,

CONSTRAINT department_employee_salary_pk PRIMARY KEY (department)

);
```

helper code

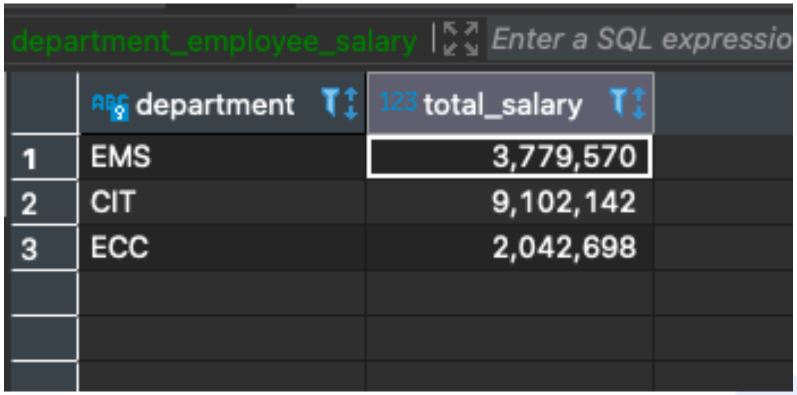
```
# update statement to update the table

cur.execute(
     "insert into  department_employee_salary  (department,total_salary)
  values (%s,%s) on conflict(department) do update set total_salary =
  department_employee_salary.total_salary + %s    ",
    (e.department, int(float(e.salary)), int(float(e.salary))))
```

Success Criteria

You should get the same results.

It’s OK if you don’t get exactly the same results.



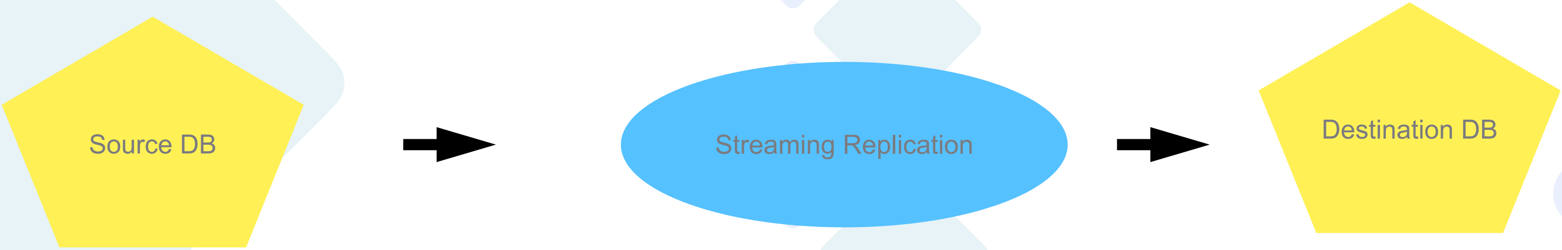
	department	total_salary
1	EMS	3,779,570
2	CIT	9,102,142
3	ECC	2,042,698

Data Engineering 0602

Kafka Project 2

Kafka Project 2: Changing Data Capture

- For this project, we will create a data tunnel to **maintain a sync** between 2 tables in two different databases.
- The stream processing would be responsible for:
  - snapshot processing => Reading all data from db1 => send asynchronously to db2
  - stream processing => Once a snapshot has been synchronized, stream processing will keep looking for the new
- Insert/Delete/Update => any changes on Emp_A should be reflected in Emp_B **< 1 sec**.



Two approaches

Polling on the tables -

With this approach we can create a python program which will start scanning the data from the head of the table till the last row, with every row being scanned will be sent to a streaming pipeline. This program would need to keep the track of the last offset/row which has been consumed, otherwise on the program crash this program would need to start all over again. The only problem with this approach is - this will take care of inserts, but not updates/deletes.

Use this approach!!!!

SQL Triggering (Postgres Triggers) -

Triggers + Functions

- Using Postgres triggers we can call an SQL function on every row insert/update/delete.
- The SQL function in this case should insert all the affected rows plus actions (insert, update or delete) into a new table. This new table will act as a CDC table.
- Producer will start scanning the data from the head of the CDC table till the last row, with every row being scanned will be sent to a streaming pipeline (Kafka). This program would need to keep the track of the last offset/row which has been consumed.
  - Otherwise we will perform full scan every time, and on the program crash this program would need to start all over again, which is very inefficient.
- Now this action should be passed as well as with the other information to the Kafka, and the consumers consuming this data will update all the syncs.

Steps

- 1, Modify the Docker compose file to have two databases, both port-forward to different ports.
- 2, Use the same schema for employee table and add this table in both databases
    - CREATE TABLE employees(emp_id SERIAL, first_name VARCHAR(100), last_name VARCHAR(100), dob DATE, city VARCHAR(100), salary INT);
- 3, Add PSQL functions and triggers on the employee_A table, which will insert the rows to the new cdc table.
    - CREATE TABLE emp_cdc(emp_id SERIAL, first_name VARCHAR(100), last_name VARCHAR(100), dob DATE, city VARCHAR(100), salary INT, action VARCHAR(100));
- 4, Modify the producer and consumer code (depending on your won configuration) to scan the employee_cdc table, and send the records to the topic; and to consume the data and update the employee_B table based on the action.

Some good practices

- There is no best solution to the problem
- Try implementing everything in an OOP manner to practice
- Always try and catch all exception you might have, especially in a production environment.
- Maintain a good coding style (ifUsingCamelToeNotation_then_do_not_switch_to_another_style)
- Some optional things you can try (absolutely no bonus)
    - Use a git repo for version control
    - Scale up/down your instances
    - Create a DLQ
    - Use Offset Explorer
    - Can use AirFlow to schedule producer.

Optional: DLQ

- Set up a DLQ topic in your docker-compose.yml file, or through admin.py (check confluent official documentations)
- You can implement your custom logic, i.e.
  - all employees should be born after 2007
  - salary should be greater than 10000
  - no negative emp id
  - …
- See if you can intentionally create some bad actions, and if you are able to get everything to work.