

Real-World Use Case of Airflow

Session-03

SESSION OVERVIEW

01

Data Architecture Explanation

02

Demonstration of Batch ETL Orchestration

03

More Advanced Topics in Airflow

04

Best Practices in Airflow

05

Advantages and Limitations of Airflow

PROBLEM DESCRIPTION

- Enable efficient OLAP on the trips and bookings data for a ride hailing company
- Design and schedule a data pipeline to generate the following insights:
 - Get the **car types involved with the highest number of trips** for each city
 - Get the **throughput of the trips** (no. of trips / no. of bookings) for each city

TABLES

```
mysql> show tables;
+-----+
| Tables_in_events |
+-----+
| booking          |
| trip             |
+-----+
2 rows in set (0.00 sec)
```

BOOKING TABLE DESCRIPTION

```
mysql> desc booking;
```

Field	Type	Null	Key	Default	Extra
booking_id	int(11)	YES		NULL	
city	varchar(50)	YES		NULL	
booking_ts	timestamp	YES		NULL	
car_type	varchar(10)	YES		NULL	

```
4 rows in set (0.01 sec)
```

BOOKING TABLE SAMPLE RECORDS

```
mysql> select * from booking LIMIT 10;
```

booking_id	city	booking_ts	car_type
1	mumbai	2020-09-16 10:00:10	economy
2	bangalore	2020-09-16 14:10:09	sedan
3	chennai	2020-09-16 18:09:15	economy
4	chennai	2020-09-16 11:11:19	sedan
5	bangalore	2020-09-16 12:16:11	sedan
6	mumbai	2020-09-16 19:27:58	sedan
7	mumbai	2020-09-16 19:27:58	NULL
8	mumbai	2020-09-16 10:00:10	economy
9	mumbai	2020-09-16 10:00:12	sedan
10	bangalore	2020-09-16 04:10:15	sedan

```
10 rows in set (0.00 sec)
```

TRIP TABLE DESCRIPTION

```
mysql> desc trip;
```

Field	Type	Null	Key	Default	Extra
trip_id	int(11)	YES		NULL	
booking_id	int(11)	YES		NULL	
pickup_location_id	int(11)	YES		NULL	
dropoff_location_id	int(11)	YES		NULL	
trip_start_ts	timestamp	YES		NULL	
trip_end_ts	timestamp	YES		NULL	
trip_distance	int(11)	YES		NULL	

7 rows in set (0.00 sec)

TRIP TABLE SAMPLE RECORDS

```
mysql> mysql> select * from trip LIMIT 10;
```

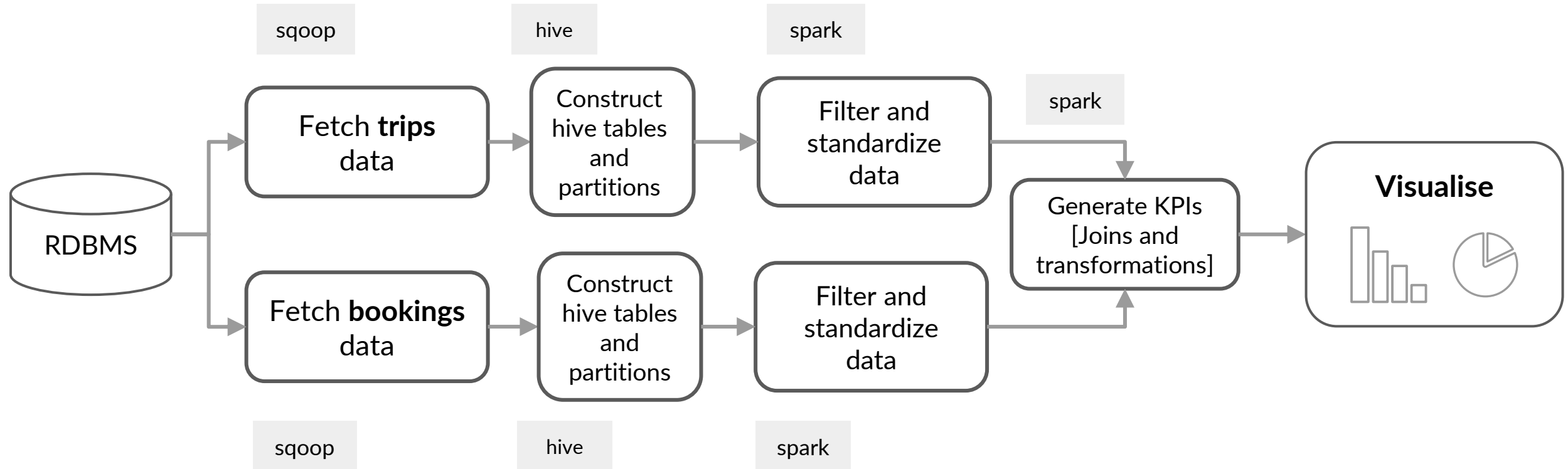
trip_id	booking_id	pickup_location_id	dropoff_location_id	trip_start_ts	trip_end_ts	trip_distance
1	1	101	102	2020-09-16 10:05:10	2020-09-16 10:15:10	5
3	3	113	111	2020-09-16 18:19:15	2020-09-16 18:59:15	15
4	4	111	114	2020-09-16 11:14:19	2020-09-16 11:34:19	4
5	5	108	106	2020-09-16 12:15:11	2020-09-16 12:19:11	2
6	6	101	101	2020-09-16 19:47:58	2020-09-16 20:27:58	35
8	8	105	101	2020-09-16 10:04:10	2020-09-16 10:12:10	5
9	9	101	104	2020-09-16 10:01:12	2020-09-16 10:10:12	5
10	10	106	108	2020-09-16 04:15:15	2020-09-16 04:20:15	4
11	11	106	108	2020-09-16 17:09:00	2020-09-16 17:28:00	20
12	12	108	109	2020-09-16 16:13:09	2020-09-16 16:19:09	5

```
10 rows in set (0.00 sec)
```


STEPS INVOLVED

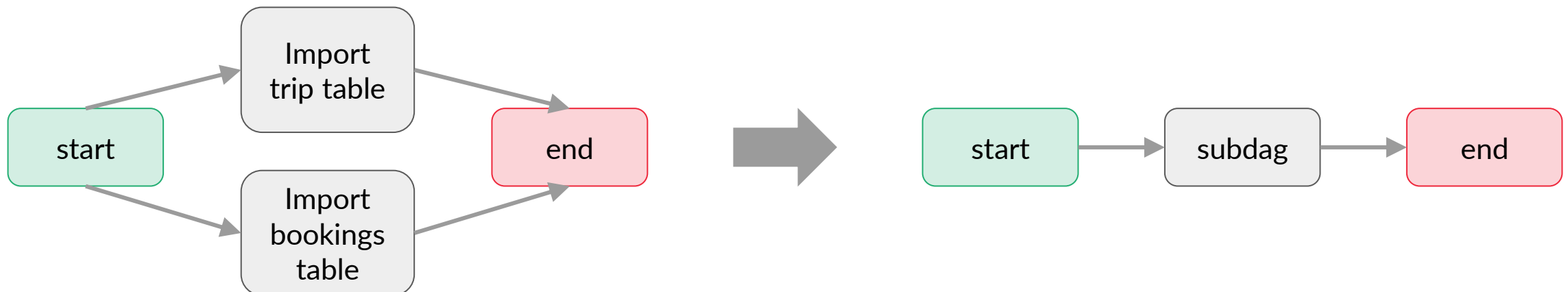
- Bring data from MySQL to HDFS via Sqoop
- Create necessary directories in HDFS
- Create Hive tables on the data imported
- Construct partitions in the Hive table
- Filter invalid records using Spark
- Run analysis to generate aggregated result using Spark
- View the result

DAG ARCHITECTURE



SubDAGs


- SubDAGs are widely used to **group logically similar or parallel tasks** in a DAG.
- SubDAG is **defined as a function** that returns a DAG object.
- In terms of a graph, each SubDAG behaves like a **vertex (a single node)** in a graph.
- For example:



SubDAG SAMPLE CODE


```
def gen_subdag(parent_dag_name, child_dag_name, args):  
    extraction_subdag = DAG(  
        dag_id='%s.%s' % (parent_dag_name, child_dag_name),  
        default_args=args,  
        schedule_interval="@daily",  
    )  
  
    import_trips_table = SqoopOperator(...,  
                                       dag=extraction_subdag)  
    import_bookings_table = SqoopOperator(...,  
                                           dag=extraction_subdag)  
  
    return extraction_subdag
```

SubDAG function:
The SubDAG and its operators are defined and returned in this function.



```
extraction_subdag_operator = SubDagOperator(  
    task_id='extract',  
    subdag = gen_subdag(DAG_NAME, 'extract', args),  
    dag=dag,  
)
```

SubDAG task: This can be used as a single task in the parent DAG.



BACKFILLING

- Backfilling is used to run the DAGs for **older schedules**.
- For example, let's say we have a DAG that runs daily and we wish to re-run the DAG from some a **start_date** to an **end_date**, we can backfill the DAG to reprocess the data
- It is useful to **reprocess the data** if application code has been updated or the data in the source has been corrected/refreshed
- Backfilling using CLI :
 - `airflow backfill -s START_DATE -e END_DATE dag_id`
- Airflow supports **automatic backfilling** of data according to start time, current time and the schedule
- To disable it, set **catchup=False** in the DAG object

AIRFLOW VARIABLES

- Variables are **key-value stores** in Airflow's metadata database.
- They are a generic way to store and retrieve arbitrary content or settings within Airflow.
- Airflow will mask all sensitive values in these variables (e.g., passwords).
- They can be JSON (deserialized as a Python dictionary), a string or of None type.
- Apart from variables, we can also make use of jinja templating, by the use of reference parameters like `{{ ds }}` (for execution date), `{{run_id }}` (the id of the current run) etc. and macros

AIRFLOW VARIABLES

- They are accessible through the Airflow UI(Admin -> Variables) , CLI and can be referred inside the DAG code
- We can import and export a set of variables together using CLI.
- Sample use of variable:

```
from airflow.models import Variable  
foo = Variable.get("foo")
```

XComs

- Tasks in Airflow use XComs or 'cross-communications' to communicate with each other.
- A task can push and also pull XCom(s) generated by other tasks
- You can create XComs through the UI or code.
- Sample use of XComs :

```
#Pushing (Generating) XCom
```

```
context['task_instance'].xcom_push(<key>, <value>)
```

```
#Pulling (Accessing) XCom
```

```
context['task_instance'].xcom_pull(<key>,  
task_ids=<ids of tasks generating XCom>)
```


TRIGGER RULES

- In general, a task is triggered when all directly upstream tasks have succeeded but Airflow allows for more complex dependency settings.
- All operators have a **trigger_rule** argument that decides when the task will get triggered.
- By default, trigger_rule = all_success.
- Some common trigger_rule values are as follows:
 - all_failed
 - all_done
 - one_success
 - none_failed
 - none_failed_or_skipped

MONITORING AND ALERTING

- Monitoring:
 - Using the Airflow UI, we can monitor the task statuses(running, success, failed, skipped etc.)
 - We can monitor the **time taken to execute dependency tasks**
 - We can **monitor any delays in DAG execution** with **SLA** duration
 - **Health checks** of workers and scheduler can be configured
 - The **StatsD** package helps in collecting Airflow statistics.

ALERTING:

- Email operator can be used to **send automated emails** whenever a task fails or is completed at the DAG level.
- Airflow can also send notifications to popular online services such as **Slack and HipChat**

TIPS AND BEST PRACTICES

01

Do not use Airflow to debug your application code.

02

The start date and the actual execution of a DAG differ by one schedule interval. Consider this while creating/scheduling your DAG.

03

Never store any config file in the local file system. If possible, use XCOM to communicate small messages.

04

By default, the DAGs are turned off as a cautionary measure. Make sure that you turn them ON.

05

Sensors are mostly idle processes; So, do not use unnecessarily.

TIPS AND BEST PRACTICES

06

Use static `start_date` for the DAGs.

07

Ensure that DAGs are as independent as possible.

08

Make DAGs idempotent; running multiple times is the same as running once.

09

Renaming a DAG will introduce a new DAG.

10

Run different components on different machines.

ADVANTAGES AND LIMITATIONS OF AIRFLOW

Advantages

Best suited for batch ETL process orchestration

Extensive UI and connectors

Plenty of retry options, visual DAGs for troubleshooting in the UI

Supports complex workflows

Limitations

Not for scheduling streaming jobs; tools such as Nifi/StreamSets can be used for stream process scheduling or Supervisor for monitoring

No native support for Windows

There is no standby process for the scheduler.

SESSION SUMMARY

01

Demonstration of Batch Process Orchestration

02

Demonstration of Hive Table Creation Orchestration

03

Demonstration of ETL Orchestration

04

Best Practices in Airflow

05

Advantages and Limitations of Airflow

THANK
YOU