EECS E6893 Big Data Analytics
HW2: Steaming Big Data Analytics & Data Analytics Pipeline

TUTORIAL 2

Ajinkeya Chitrey
ac5166@columbia.edu

Agenda

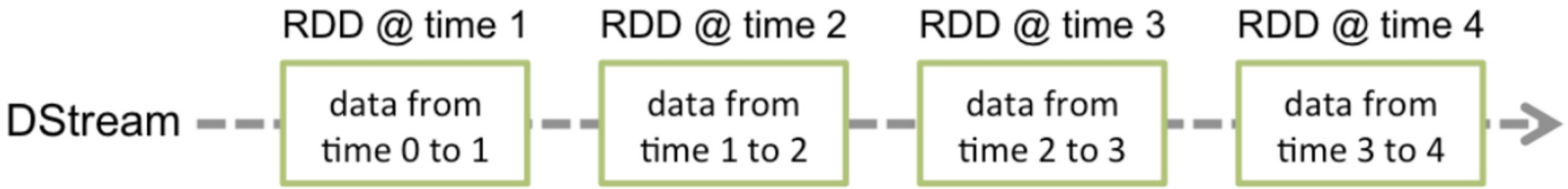
- Streaming Analytics
- Data pipeline orchestration using Apache Airflow

Streaming Analytics

Spark Streaming

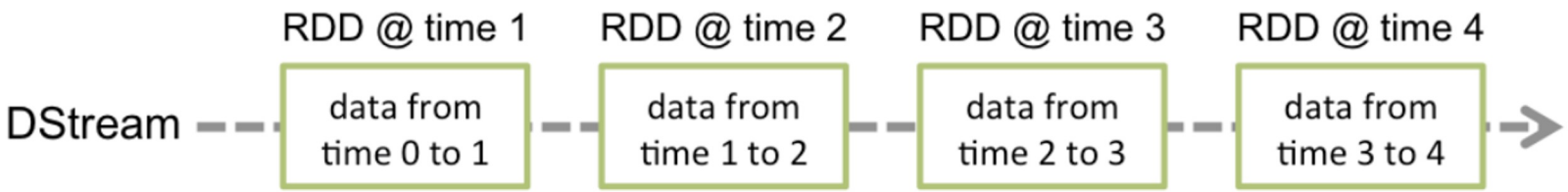
https://spark.apache.org/docs/latest/streaming-programming-guide.html

Dstream



- A basic abstraction provided by Spark Streaming
- Represents a continuous stream of data
- Contains a continuous series of RDDs at different time

Dstream Example



First, we import StreamingContext, which is the main entry point for all streaming functionality. We create a local StreamingContext with two execution threads, and batch interval of 1 second.

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

# Create a local StreamingContext with two working thread and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)
```

Using this context, we can create a DStream that represents streaming data from a TCP source, specified as hostname (e.g. `localhost`) and port (e.g. `9999`).

```
# Create a DStream that will connect to hostname:port, like localhost:9999
lines = ssc.socketTextStream("localhost", 9999)
```

This `lines` DStream represents the stream of data that will be received from the data server. Each record in this DStream is a line of text. Next, we want to split the lines by space into words.

```
# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))
```

This `lines` DStream represents the stream of data that will be received from the data server. Each record in this DStream is a line of text. Next, we want to split the lines by space into words.

```
# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))
```

`flatMap` is a one-to-many DStream operation that creates a new DStream by generating multiple new records from each record in the source DStream. In this case, each line will be split into multiple words and the stream of words is represented as the `words` DStream. Next, we want to count these words.

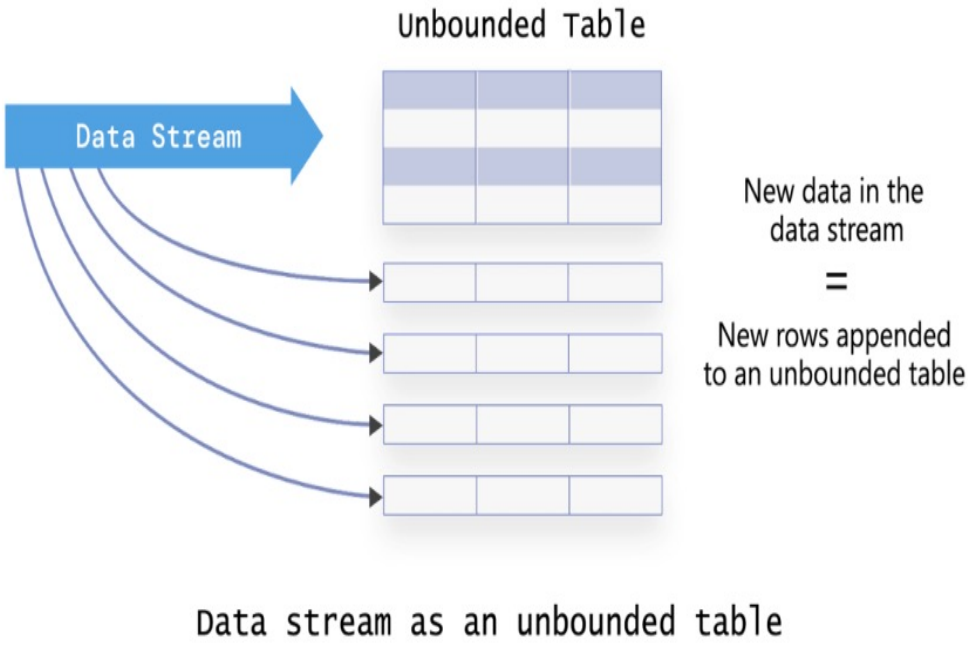
```
# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)

# Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.pprint()
```

The `words` DStream is further mapped (one-to-one transformation) to a DStream of `(word, 1)` pairs, which is then reduced to get the frequency of words in each batch of data. Finally, `wordCounts.pprint()` will print a few of the counts generated every second.

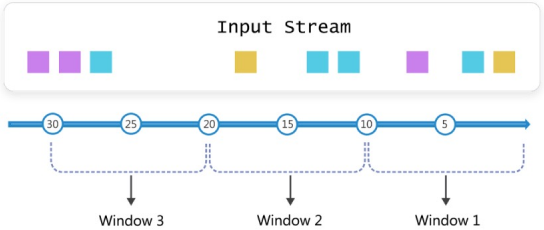
Note that when these lines are executed, Spark Streaming only sets up the computation it will perform when it is started, and no real processing has started yet. To start the processing after all the transformations have been setup, we finally call

```
ssc.start()             # Start the computation
ssc.awaitTermination()  # Wait for the computation to terminate
```

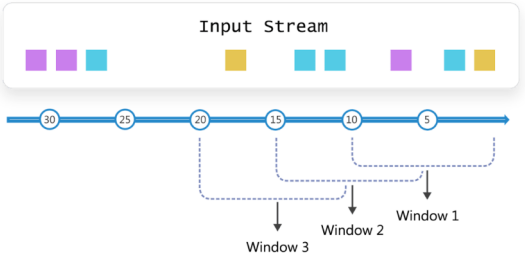
Spark Structured Streaming



Tumbling Window

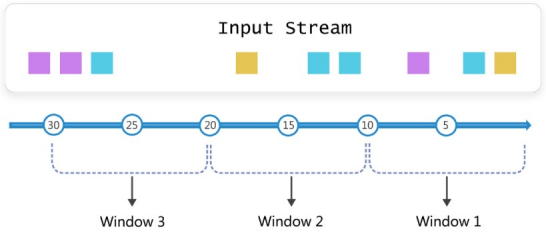


Sliding Window



Spark Structured Streaming

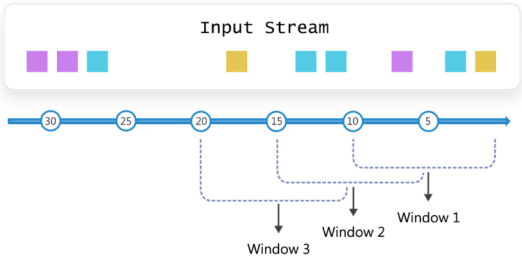
Tumbling Window



The following snippet calculates the visitors' count monthly. It is a windowed aggregation with a tumbling window of 30 days and counts all the records.

```
dfPatients \
 .groupBy(window(col("VisitDate"), "30 days")) \
 .agg(count("PID").alias("aggregate_count")) \
 .select("window.start", "window.end", "aggregate_count")
```

Sliding Window



The following snippet calculates the monthly visitor count and emits the results every week. We have created a window on the visitDate for every 30 days and a sliding interval of a week.

```
dfPatients \
 .groupBy(F.window(F.col("VisitDate"), "30 days", "1 week")) \
 .agg(F.count("PID").alias("aggregate_count")) \
 .select("window.start", "window.end", "aggregate_count")
```

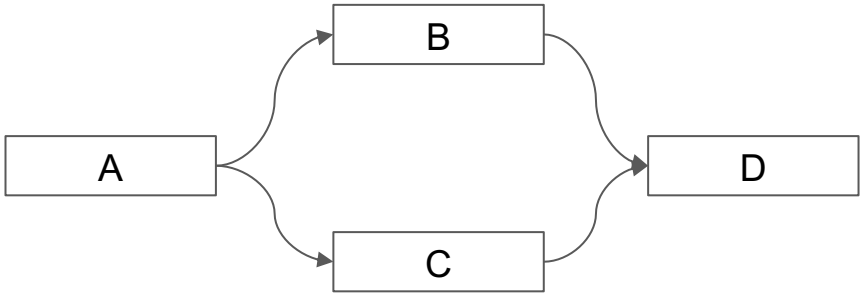
Airflow

Workflow

- A sequence of tasks involved in moving from the beginning to the end of a working process
- Started on a schedule or triggered by an event

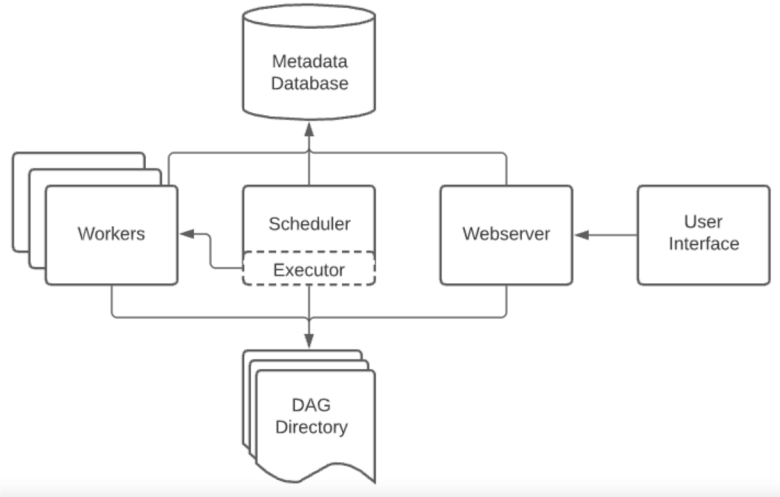
DAG (Directed Acyclic Graph)

- In Airflow, workflows are created using DAGs
- A DAG is a collection of tasks that you want to schedule and run, organized in a way that reflects their relationships and dependencies
- The tasks describe what to do, e.g., fetching data, running analysis, triggering other systems, or more
- A DAG ensures that each task is executed at the right time, in the right order, or with the right issue handling
- A DAG is written in Python



Airflow architecture

- **Scheduler:** handles both triggering scheduled workflows, and submitting tasks to the executor to run.
- **Executor**: handles running tasks.
- **Webserver**: a handy user interface to inspect, trigger and debug the behavior of DAGs and tasks.
- **A folder of DAG files**: read by the scheduler and executor
- **A metadata database**: used by the scheduler, executor and webserver to store state



Executors

- **SequencialExecutor:** The Sequential Executor in Apache Airflow executes tasks one after the other in the order they are scheduled. While simple and easy to set up, it lacks parallelism and doesn't support concurrent task execution, making it suitable only for very basic, non-concurrent workflows and small-scale development setups.

- **LocalExecutor:** The Local Executor, unlike the Sequential Executor, allows limited parallelism by running tasks in separate processes on the same machine.

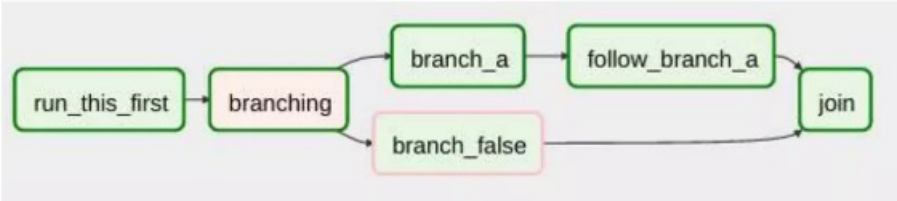
- **CeleryExecutor:** The Celery Executor in Apache Airflow leverages Celery, a distributed task queue system, to enable concurrent and parallel task execution across multiple worker nodes.

- **KubernetesExecutor:** The Kubernetes Executor in Airflow leverages Kubernetes clusters to distribute task execution across containers, allowing seamless scaling of resources based on demand.

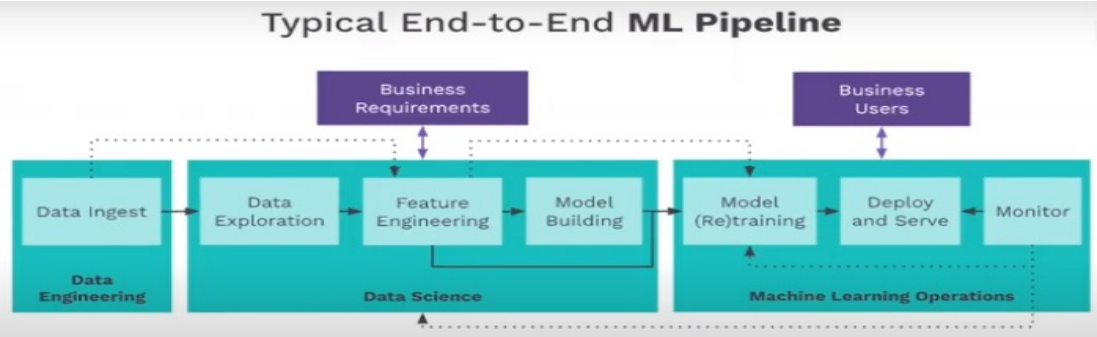
Airflow vs Streaming solutions

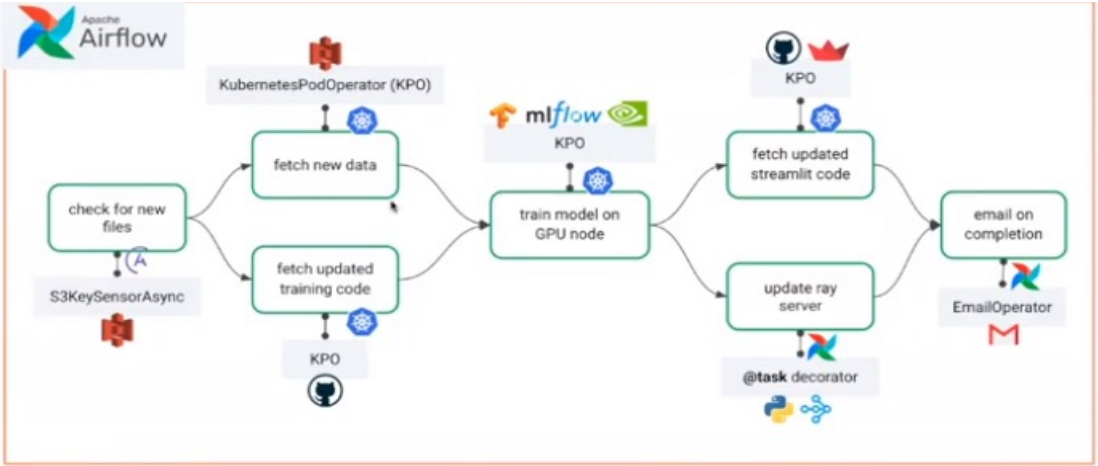
- Airflow is not a streaming solution. Tasks do not move data from one to another in a dynamic fashion as we do in streaming solutions.

- Workflows are expected to be slightly more static or slowly changing. This allows for clarity around a unit of work and consistent orchestration cycles.



Sample Use Case using Airflow for ML Pipelines



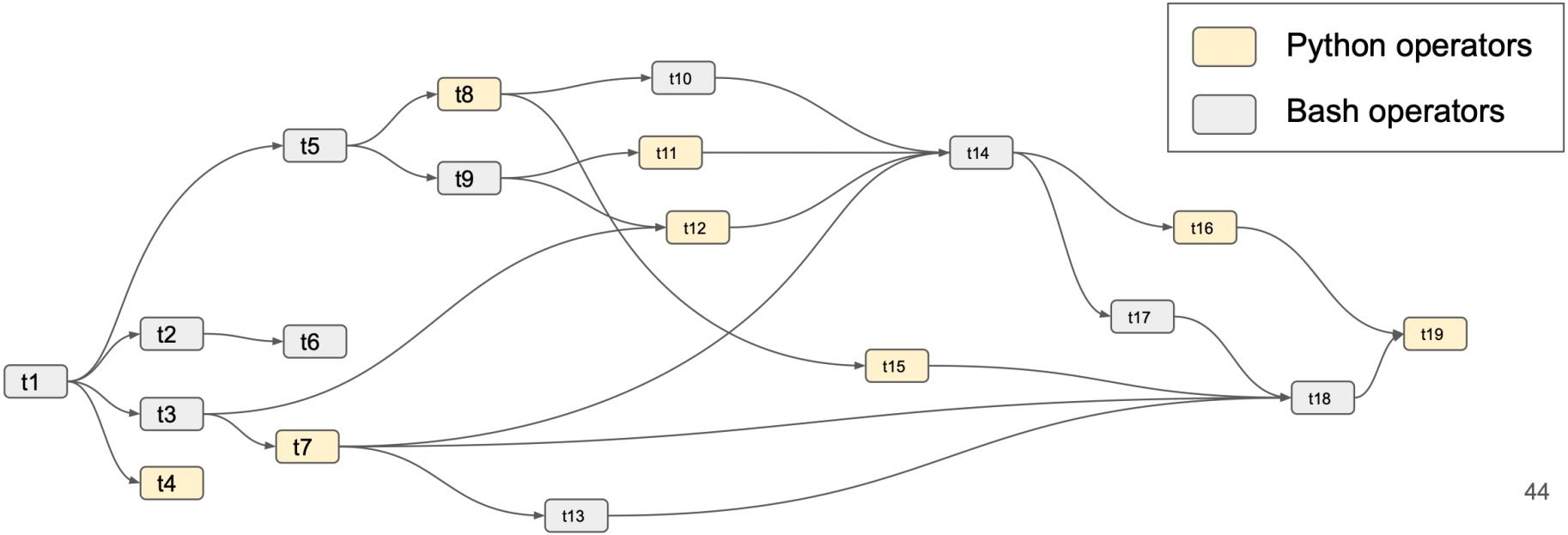


HW CLARIFICATIONS:

Task 2 Build workflows

Q2.1 Implement this DAG (25 pts)

- Tasks and dependencies (10 pts)
- Manually trigger it (10 pts)
- Schedule the first run immediately and running the program every 30 minutes (5 pts)



Tasks and Scheduler

```python
from datetime import datetime, timedelta
from textwrap import dedent
import time

# The DAG object; we'll need this to instantiate a DAG
from airflow import DAG

# Operators; we need this to operate!
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
# These args will get passed on to each operator
# You can override them on a per-task basis during operator initialization
default_args = {
    'owner': 'cong',
    'depends_on_past': False,
    'email': ['ch3212@columbia.edu'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(seconds=30),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
    # 'wait_for_downstream': False,
    # 'dag': dag,
    # 'sla': timedelta(hours=2),
    # 'execution_timeout': timedelta(seconds=300),
    # 'on_failure_callback': some_function,
    # 'on_success_callback': some_other_function,
    # 'on_retry_callback': another_function,
    # 'sla_miss_callback': yet_another_function,
    # 'trigger_rule': 'all_success'
}
```

```python
with DAG(
    'helloworld',
    default_args=default_args,
    description='A simple toy DAG',
    schedule_interval=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
    catchup=False,
    tags=['example'],
) as dag:
```

```python
def correct_sleeping_function():
    """This is a function that will run within the DAG execution"""
    time.sleep(2)
```

```python
with DAG(
    'helloworld',
    default_args=default_args,
    description='A simple toy DAG',
    schedule_interval=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
    catchup=False,
    tags=['example'],
) as dag:

    # t* examples of tasks created by instantiating operators
    t1 = PythonOperator(
        task_id='t1',
        python_callable=correct_sleeping_function,
    )

    t2_1 = PythonOperator(
        task_id='t2_1',
        python_callable=correct_sleeping_function,
    )

    t2_2 = PythonOperator(
        task_id='t2_2',
        python_callable=correct_sleeping_function,
        retries=3,
    )

    t2_3 = PythonOperator(
        task_id='t2_3',
        python_callable=correct_sleeping_function,
    )

    t3_1 = PythonOperator(
        task_id='t3_1',
        python_callable=correct_sleeping_function,
    )

    t3_2 = PythonOperator(
        task_id='t3_2',
        python_callable=correct_sleeping_function,
    )

    t4_1 = BashOperator(
        task_id='t4_1',
        bash_command='sleep 2',
        retries=3,
    )
```

Operators

1. **PythonOperator**
2. **BashOperator**
3. branch_operator
4. email_operator
5. mysql_operator
6. DataprocOperator

…

...

PythonOperator:

```
def function():
    print(123)

task = PythonOperator(
    task_id='task_id',
    python_callable=function,
  )
```

BashOperator:

```
task = BashOperator(
    task_id='task_id',
    bash_command='sleep 2',
  )
# other examples
bash_command='python python_code.py'
bash_command='bash bash_code.sh '
```

(must have a space to satisfy Jinja template !!)

Operators – Suggested Examples

Python Operators

```
count = 0

def correct_sleeping_function():
    """This is a function that will run within the DAG execution"""
    time.sleep(2)

def count_function():
    global count
    count += 1
    print('output of count_increase: {}'.format(count))
    time.sleep(2)

def print_function():
    print('This represents a python operator')
    time.sleep(2)
```

```
t7 = PythonOperator(
    task_id='t7',
    python_callable=correct_sleeping_function,
)

t8 = PythonOperator(
    task_id='t8',
    python_callable=print_function,
)
```

Bash Operators

```
t5 = BashOperator(
    task_id='t5',
    bash_command='sleep 1',
)

t6 = BashOperator(
    task_id='t6',
    bash_command='sleep 2',
)
```