Introduction to Big Data

Pooya Jamshidi

pooya.jamshidi@ut.ac.ir

Ilam University

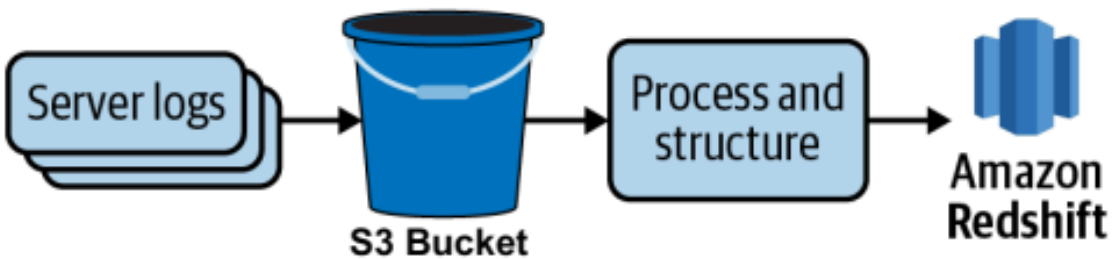
School of Engineering,
Computer Group

April 27, 2025

Data Pipelines

Data Processing Pipelines

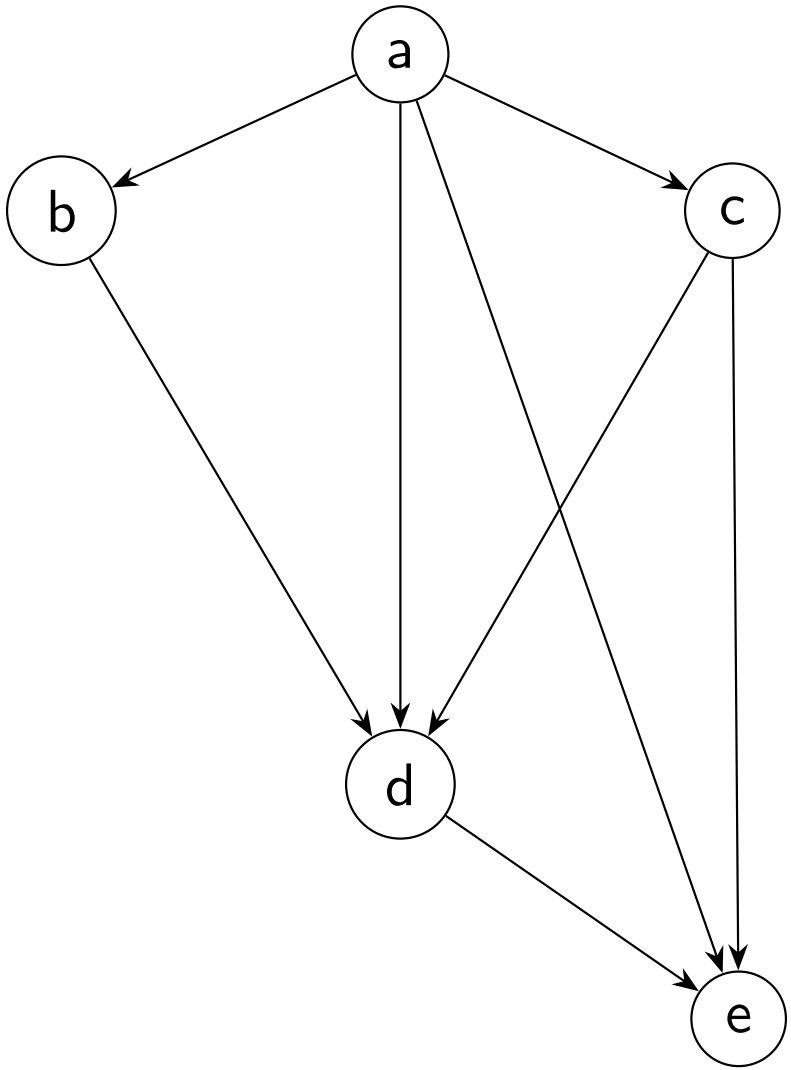
- A pipeline is a series of data operations with a specific order of execution.
- Some operations can be executed in parallel, i.e., it is a *Directed Acyclic Graph* (DAG) of operations that typically:
  - Read one or more sources of data
  - Perform transformations (selecting, filtering, joining, math, etc.)
  - Write the output to one or more sinks of data
- General considered hard, or at least complicated, especially in a Big Data setting.



Directed Acyclic Graph (DAG)

- DAG is a directed graph without any cycle.
- It is used to model a workflow system, which consists of tasks and their dependency.
- Suppose a DAG has n nodes, each of which takes $d(n)$ to finish and m machines exists.
  - How would you *schedule* this graph such that dependencies are satisfied?
  - The problem is actually more complex in practice. There are various types of resources and each node may request different types of resources.
- DAG scheduling is NP-hard.

Directed Acyclic Graph (DAG)



Unix Cron Task Scheduling

- Cron daemon – controls periodic processes in the Linux system
- Reads one or more configuration files containing lists of command lines and times they are to run.
- `crontab` also known as "cron table" – cron configuration file
- Cron wakes and sleeps *every minute* to check all configuration files, reloads any files that have changed, and executes any that are scheduled.

Cron locations

- Each user in the system can store their own cron file in `/var/spool/cron`
- System maintenance files located `/etc/cron.d` and `/etc/crontab`
  - Generally `/etc/crontab` is the file sys admins change by hand.
  - `/etc/cron.d` is the location software packages can install crontab entries.

cron Task Scheduling

- Runs ping command every five minutes and redirects stdout and stderr to `/dev/null`.

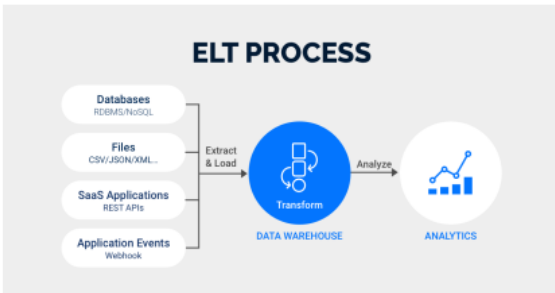
```
$ crontab -e
#  ┌──────────── minute (0 - 59)
#  │ ┌────────── hour (0 - 23)
#  │ │ ┌──────── day of the month (1 - 31)
#  │ │ │ ┌────── month (1 - 12)
#  │ │ │ │ ┌──── day of the week (0 - 6) (Sunday to Saturday;
#  │ │ │ │ │                                7 is also Sunday on some systems)
#  │ │ │ │ │
#  │ │ │ │ │
# * * * * * <command to execute>
  */5 * * * * ping -c 2 google.com >/dev/null 2>&1
$ crontab --help
usage:  crontab [-u user] file
        crontab [ -u user ] [ -i ] { -e | -l | -r }
                (default operation is replace, per 1003.2)
        -e      (edit user's crontab)
        -l      (list user's crontab)
        -r      (delete user's crontab)
        -i      (prompt before deleting user's crontab)
```

Try it easily here:
`https://crontab.guru`

Symbol	Meaning
*	any value
,	value list separator
-	range of values
/	step values
@yearly	(non-standard)
@annually	(non-standard)
@monthly	(non-standard)
@weekly	(non-standard)
@daily	(non-standard)
@hourly	(non-standard)
@reboot	(non-standard)

tl;dr this is all just (distributed) ETL/ELT

- A data pipeline is just ETL or ELT
  - Extract == Read
  - Transform == (ahem) Transform
  - Load == Write
- Many pipelines can be composed together as part of a larger ETL/ELT system



ETL vs. ELT

- **Definition**
  - ETL: Extract from source, transform externally, then load.
  - ELT: Extract from source, load first, then transform inside destination.
- **Extract**
  - ETL: Raw data extracted via API connectors.
  - ELT: Raw data extracted via API connectors.
- **Transform**
  - ETL: Transformation happens on a processing server.
  - ELT: Transformation happens inside the target system.
- **Load**
  - ETL: Load transformed data into destination.
  - ELT: Load raw data directly into target.
- **Speed**
  - ETL: Slower; data is transformed before loading.
  - ELT: Faster; load first, transform in parallel.

ETL vs. ELT (cont'd)

- **Code-Based Transformations**
  - ETL: Performed on a secondary server.
  - ELT: Performed inside the database; faster and more efficient.
- **Maturity**
  - ETL: 20+ years of development; well-known protocols.
  - ELT: Newer; less documentation and experience.

ETL vs. ELT (cont'd)

- **Privacy**
  - ETL: Pre-load transformation can remove *personal identification information (PII)*.
  - ELT: Requires stronger privacy safeguards.
- **Maintenance**
  - ETL: Secondary server increases maintenance burden.
  - ELT: Fewer systems mean less maintenance.
- **Costs**
  - ETL: Separate servers can add to cost.
  - ELT: Simplified stack costs less.

ETL vs. ELT (cont'd)

- **Requeries**
  - ETL: Raw data cannot be reprocessed after transformation.
  - ELT: Raw data can be reprocessed endlessly.
- **Data Output**
  - ETL: Typically structured data.
  - ELT: Structured, semi-structured, or unstructured data.
- **Data Volume**
  - ETL: Best for small, complex datasets.
  - ELT: Best for large datasets needing speed and efficiency.
- **Source:** `rivery.io/blog/etl-vs-elt/`

The Big Data Problem

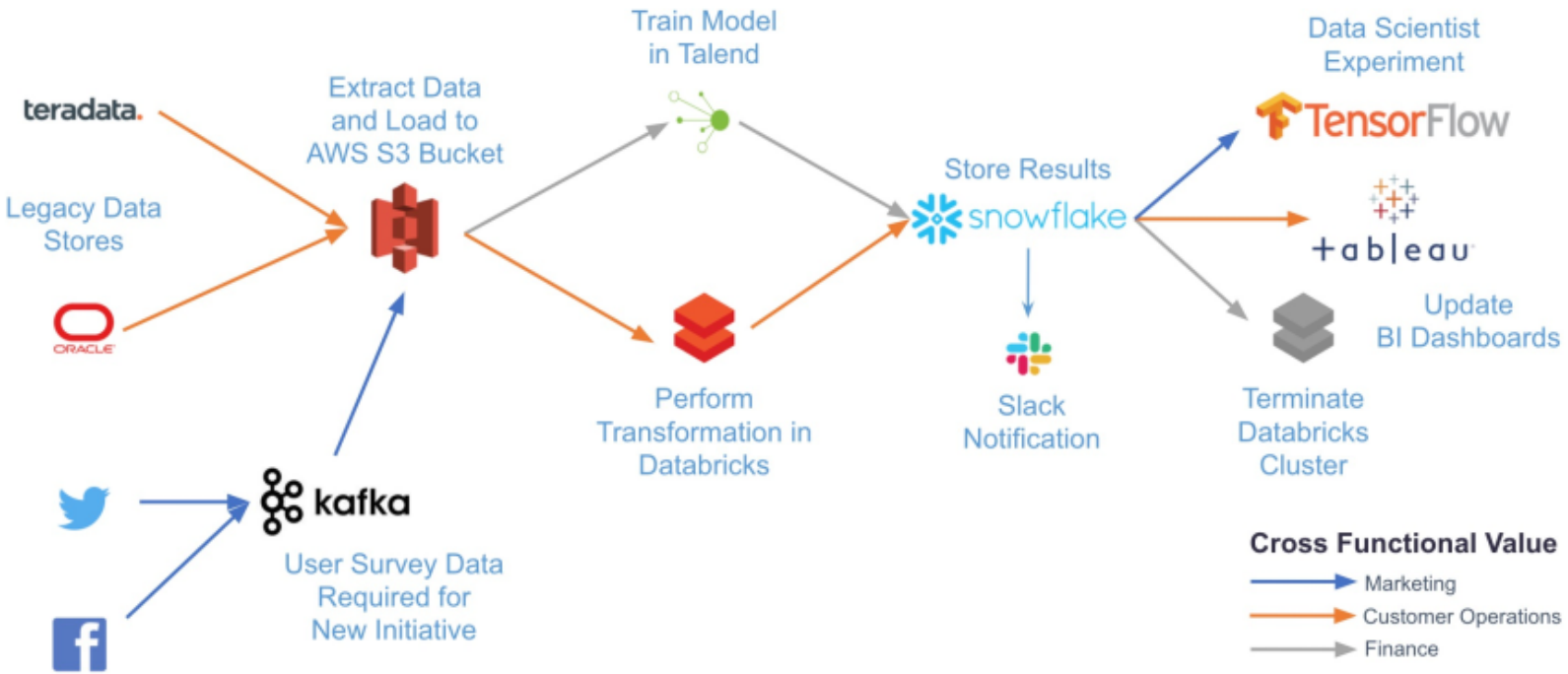
- When data is on more than one machine, simple scripts no longer work.
- We need to write complex distributed transformation programs.
- What was once easy is no longer:
  - Difficult to test, debug, and optimize.
  - Difficult to get right in terms of semantics.
  - Very costly to build, even harder to modify in-flight.
  - You may need to have your queries formally reviewed.

Apache Airflow

Introduction

- Airflow is a platform to programmatically author, schedule, and monitor workflows, A.K.A. DAGs.
- Short history:
  - 2014: Started at Airbnb by Maxime Beauchemin to manage the company's increasingly complex workflows.
  - April 2016: Open sourced and joined the Apache incubator.
  - January 2019: Became an Apache top-level project.
- Key benefits:
  - **Dynamic**: Anything you can do in Python, you can do in Airflow.
  - **Extensible**: Has many available plugins for interacting with most common external systems.
  - **Scalable**: Teams use Airflow to run thousands of different tasks per day.

An Exemplary Pipeline



A Simple Airflow Script

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta


default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2015, 12, 1),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    'schedule_interval': '@hourly',
}

dag = DAG('tutorial', catchup=False, default_args=default_args)
```

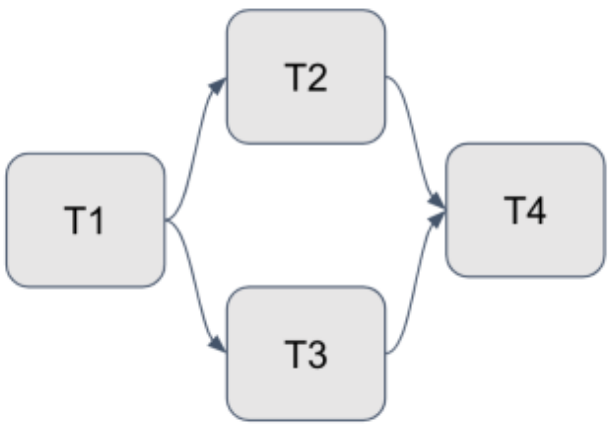
Scheduling a Pipeline

- `schedule_interval`: Takes a cron expression as a `str`, or a `datetime.timedelta` object. Alternatively, you can also use one of these cron "preset":

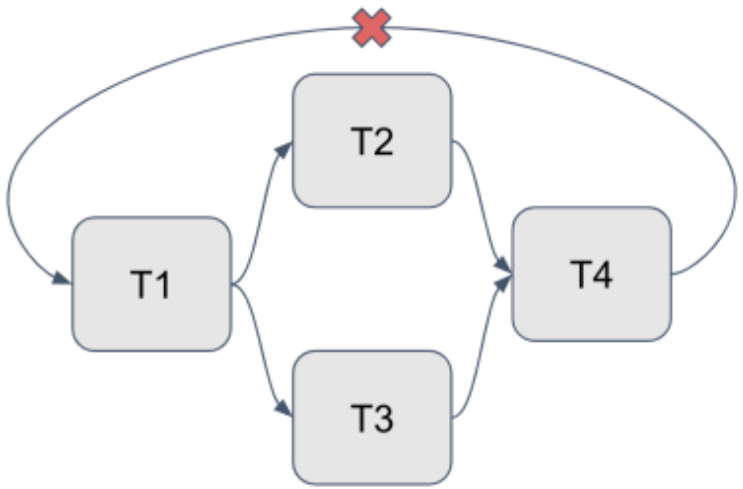
Preset	**Meaning**	**cron**
None	Don't schedule, use for exclusively "externally triggered" DAGs	
`@once`	Schedule once and only once	
`@hourly`	Run once an hour at the beginning of the hour	0 * * * *
`@daily`	Run once a day at midnight	0 0 * * *
`@weekly`	Run once a week at midnight on Sunday morning	0 0 * * 0
`@monthly`	Run once a month at midnight of the first day of the month	0 0 1 * *
`@yearly`	Run once a year at midnight of January 1	0 0 1 1 *

Core Concepts: DAG

We talked about it in detail before...



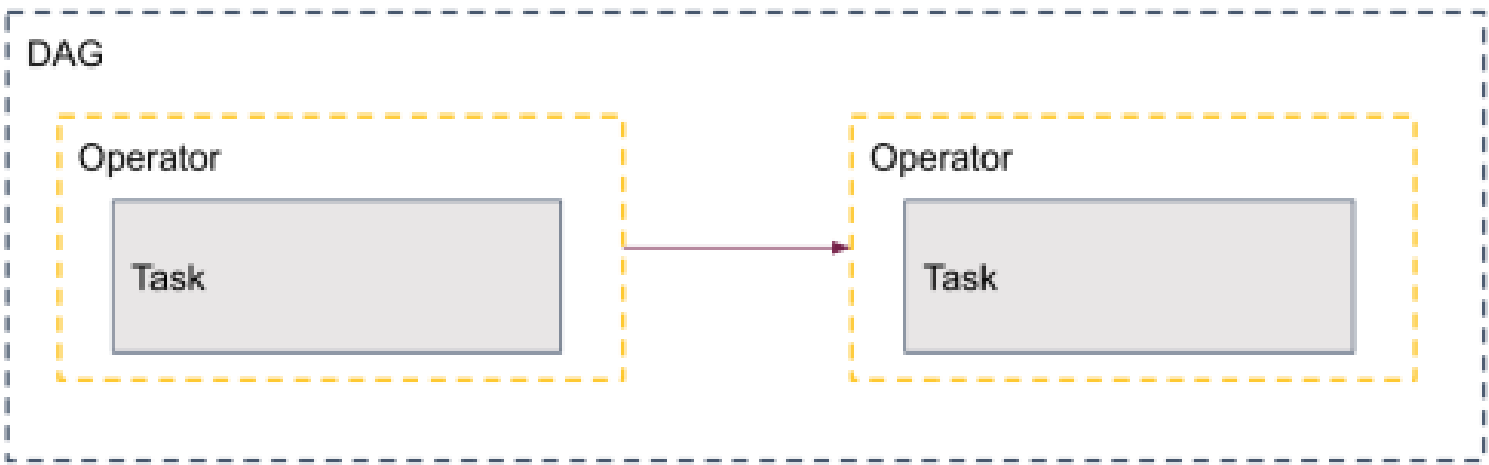
Valid



Invalid

Core Concepts: Task

- The basic unit of execution in Airflow.
- Represents each node of a defined DAG.
- Tasks are arranged into DAGs, and then have upstream and downstream *dependencies* set between them to express the order they should run in.
- **A special task:**
  - **A TaskFlow-decorated** `@task`: A custom Python function packaged up as a Task.



Core Concepts: Operators

- A template for a predefined Task, that you can just define declaratively inside your DAG:

```
with DAG("my-dag") as dag:
    ping = SimpleHttpOperator(endpoint="http://example.com/update/")
    email = EmailOperator(to="admin@example.com", subject="Update complete")

    ping >> email
```

- DAGs make sure that operators get scheduled and run in a certain order, while operators define the work that must be done at each step of the process.
- Examples of built-in operators:
  - `BashOperator`: executes a bash command
  - `PythonOperator`: calls an arbitrary Python function
  - `EmailOperator`: sends an email

Core Concepts: Operators (cont'd)

- Operator types:
  - **Action operators:** Predefined task templates that you can string together quickly to build most parts of your DAGs.
  - **Transfer operators:** Move data from a source to a destination.
  - **Sensor operators:** A special subclass of Operators which are entirely about waiting for an external event to happen.

Core Concepts: Plugins

- Many operators can be installed via plugins such as:
  - `SimpleHttpOperator`
  - `MySqlOperator`
  - `PostgresOperator`
  - `MsSqlOperator`
  - `OracleOperator`
  - `JdbcOperator`
  - `DockerOperator`
  - `HiveOperator`
  - `S3FileTransformOperator`
  - `PrestoToMySqlOperator`
  - `SlackAPIOperator`
- Other packages:
  `airflow.apache.org/docs/apache-airflow/stable/extra-pac`

Core Concepts: Dependencies

```
t1.set_downstream(t2)

t2.set_upstream(t1)

t1 >> t2

t2 << t1

t1 >> t2 >> t3

# The following three are equivalent
t1.set_downstream([t2, t3])
t1 >> [t2, t3]
[t2, t3] << t1
```

Core Concepts: Templating with Jinja

- Allows pipeline author with a set of built-in parameters and macros.

```
# The start of the data interval as YYYY-MM-DD
date = "{{ ds }}"
t = BashOperator(
    task_id="test_env",
    bash_command="/tmp/test.sh ",
    dag=dag,
    env={"DATA_INTERVAL_START": date},
)
```

- {{ ds }} is a templated variable.
- Templates reference:
  airflow.apache.org/docs/apache-airflow/stable/templates

Quiz