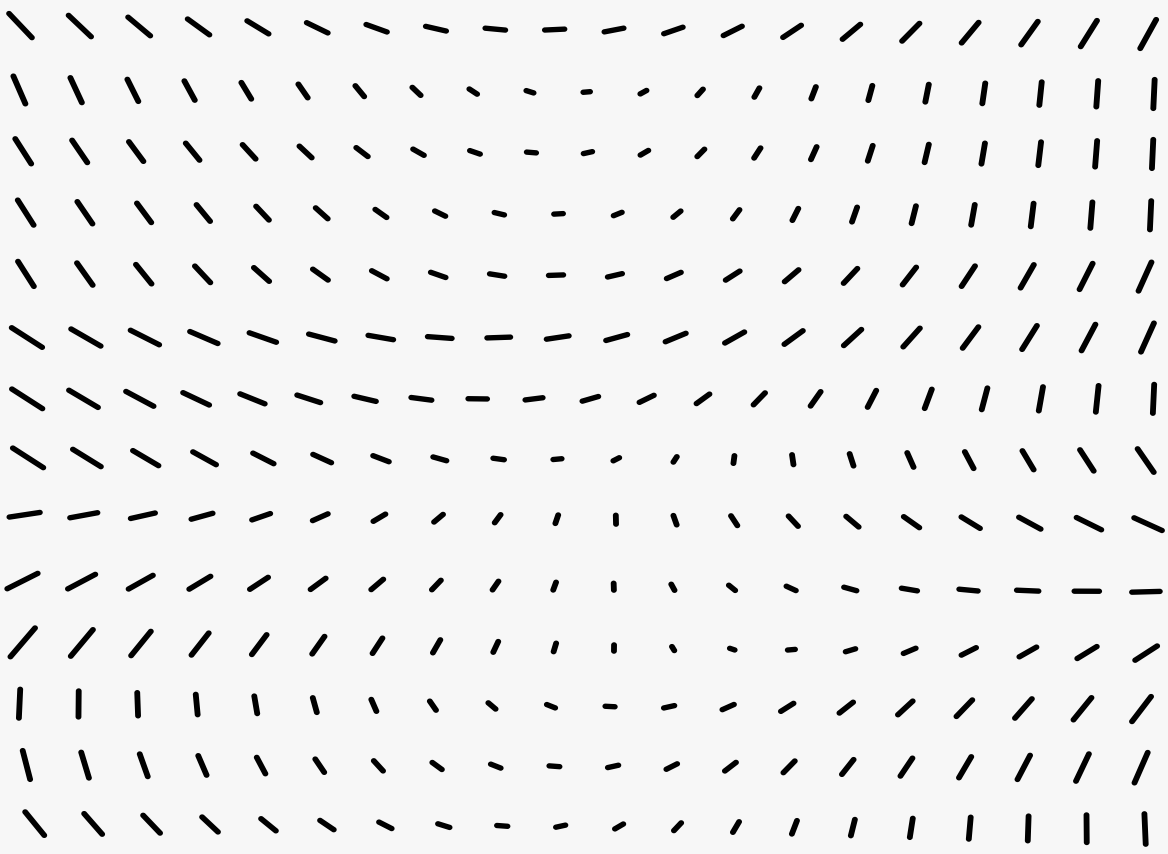


Intro to Data Engineering

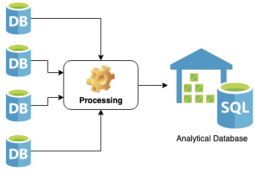
WHAT DOES A DATA ENGINEER DO?

WHEN...

- data is scattered
- not optimized for analyses
- legacy code is causing corrupt data

DATA ENGINEER TO THE RESCUE!

- gather data from ∄ sources
- optimize database for analyses
- remove corrupt data



HAPPY DATA SCIENTIST!

DATA ENGINEER	DATA SCIENTIST
develop scalable data architecture	mining data for patterns
streamline data acquisition	statistical modeling
set up processes to bring together data	predictive models using ML
clean corrupt data	monitor business processes
well versed in cloud technology	clean outliers in data

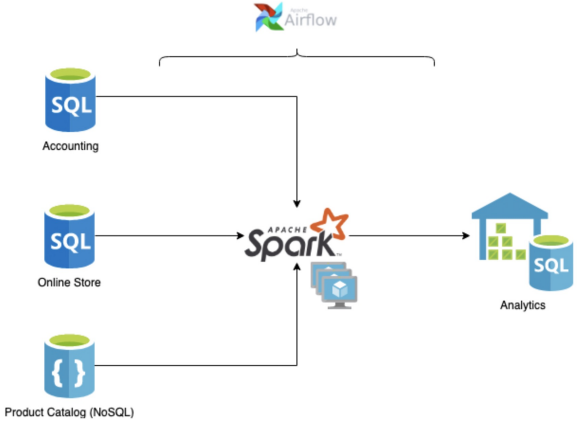
TOOLS OF THE DATA ENGINEER

databases

processing

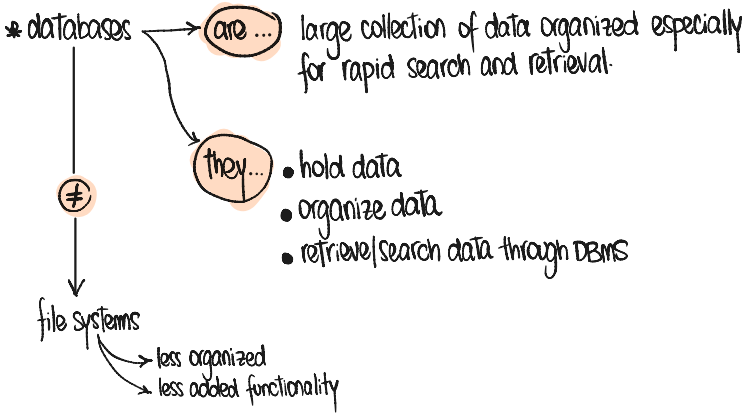
Scheduling

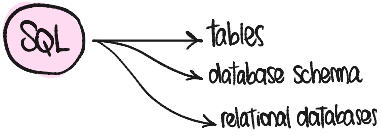
A data pipeline...

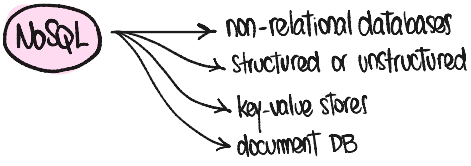


Services	Computation	databases
* AWS S3	* AWS EC2	* AWS RDS
* Azure Blob Storage	* Azure Virtual Machines	* Azure SQL Database
* Google Cloud Storage	* Google Compute Engine	* Google Cloud SQL

DATABASES

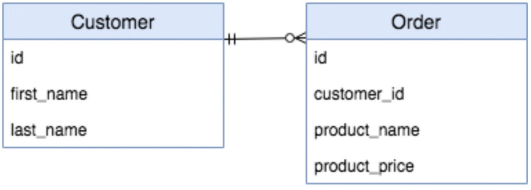






The database schema...

```sql
-- Create Customer Table
CREATE TABLE "Customer" (
  "id" SERIAL NOT NULL,
  "first_name" varchar,
  "last_name" varchar,
  PRIMARY KEY ("id")
);

-- Create Order Table
CREATE TABLE "Order" (
  "id" SERIAL NOT NULL,
  "customer_id" integer REFERENCES "Customer",
  "product_name" varchar,
  "product_price" integer,
  PRIMARY KEY ("id")
);
```




```sql
-- Join both tables on foreign key
SELECT * FROM "Customer"
INNER JOIN "Order"
ON "customer_id" = "Customer"."id";
```

```
id | first_name | ... | product_price
 1 | Vincent    | ... |            10
```

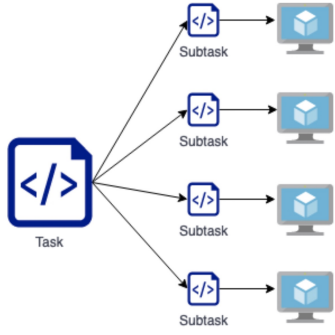
```python
# Complete the SELECT statement
data = pd.read_sql("""
SELECT first_name, last_name FROM "Customer"
ORDER BY last_name, first_name
""", db_engine)

# Show the first 3 rows of the DataFrame
print(data.head(3))

# Show the info of the DataFrame
print(data.info())

```

Python snippet to retrieve data from SQL

PARALLEL COMPUTING

* idea behind parallel computing → memory, processing power

- Split task into substacks
- distribute substacks over several computers
- work together to finish task



* risk of parallel computing → bottlenecks when the processing requirements are not substantial, or if one has too little processing units.

```python
from multiprocessing import Pool

def take_mean_age(year_and_group):
    year, group = year_and_group
    return pd.DataFrame({"Age": group["Age"].mean()}, index=[year])


with Pool(4) as p:
    results = p.map(take_mean_age, athlete_events.groupby("Year"))


result_df = pd.concat(results)
```

multiprocessing.Poll

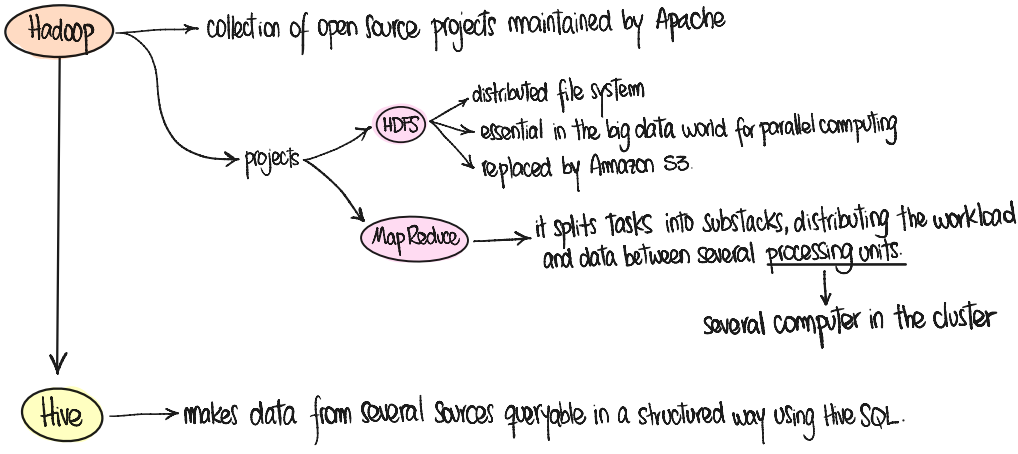
→ low level API to distribute work over several cores on the same machine

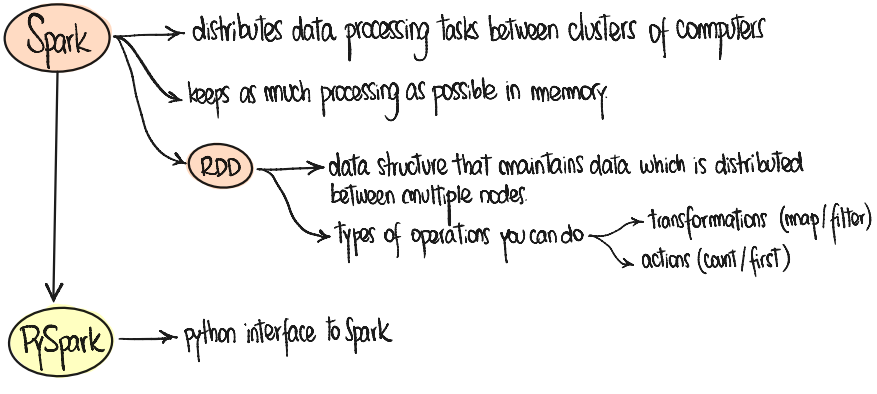
```python
# Function to apply a function over multiple cores
@print_timing
def parallel_apply(apply_func, groups, nb_cores):
    with Pool(nb_cores) as p:
        results = p.map(apply_func, groups)
    return pd.concat(results)

# Parallel apply using 1 core
parallel_apply(take_mean_age, athlete_events.groupby('Year'), 1)

# Parallel apply using 2 cores
parallel_apply(take_mean_age, athlete_events.groupby('Year'), 2)

# Parallel apply using 4 cores
parallel_apply(take_mean_age, athlete_events.groupby('Year'), 4)

```

dask

→ it offers a level of abstraction to avoid having to write such low-level code

PARALLEL COMPUTATION FRAMEWORKS






```python
# Print the type of athlete_events_spark
print(type(athlete_events_spark))

# Print the schema of athlete_events_spark
print(athlete_events_spark.printSchema())

# Group by the Year, and find the mean Age
print(athlete_events_spark.groupBy('Year').mean('Age'))

# Group by the Year, and find the mean Age
print(athlete_events_spark.groupBy('Year').mean('Age').show())
```

WORKFLOW SCHEDULING FRAMEWORKS

DAGs → Directed Acyclic Graph

→ set of nodes that are connected by directed edges

→ there are no cycles in the graph, which means that no path sees a specific node more than once

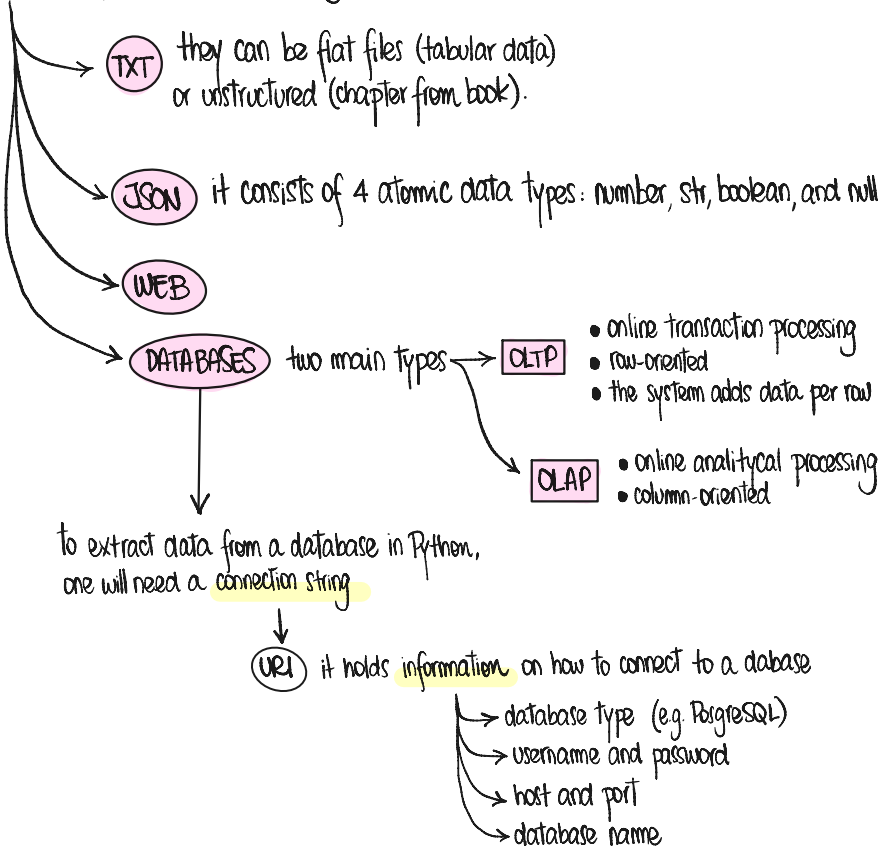
Apache Airflow → built by Airbnb


```python
# Create the DAG object
dag = DAG(dag_id="example_dag", ..., schedule_interval="0 * * * *")

# Define operations
start_cluster = StartClusterOperator(task_id="start_cluster", dag=dag)
ingest_customer_data = SparkJobOperator(task_id="ingest_customer_data", dag=dag)
ingest_product_data = SparkJobOperator(task_id="ingest_product_data", dag=dag)
enrich_customer_data = PythonOperator(task_id="enrich_customer_data", ..., dag = dag)

# Set up dependency flow
start_cluster.set_downstream(ingest_customer_data) 1
ingest_customer_data.set_downstream(enrich_customer_data) 2
ingest_product_data.set_downstream(enrich_customer_data) 3
```

Extract Transform Load

EXTRACT

* extract data from persistent storage → Amazon S3 / SQL Database



TRANSFORM

* typically done using parallel computing
* kind of transformations:
  - selection of attribute
  - translation of code values
  - data validation
  - splitting columns into multiple columns
  - joining from multiple sources

customer_id	email	username	domain
1	jane.doe@theweb.com	jane.doe	theweb.com

```
customer_df # Pandas DataFrame with customer data

# Split email column into 2 columns on the '@' symbol
split_email = customer_df.email.str.split("@", expand=True)
# At this point, split_email will have 2 columns, a first
# one with everything before @, and a second one with
# everything after @

# Create 2 new columns using the resulting DataFrame.
customer_df = customer_df.assign(
  username=split_email[0],
  domain=split_email[1],
)
```

```
import pyspark.sql

spark = pyspark.sql.SparkSession.builder.getOrCreate()
                                  HELPS SPARK CONNECT TO (2D)
spark.read.jdbc("jdbc:postgresql://localhost:5432/pagila",
                "customer",
                properties={"user":"repl","password":"password"})
```

LOADING

* once we've extracted and transformed the data, we have to load it and make it ready for analytics.
  - we can optimize for: OLAP / OLTP
* MPP Databases
  - column-oriented databases optimized for analytics
  - queries are split into subtasks and distributed among several nodes
  - examples:
    - Amazon Redshift
    - Azure SQL Data Warehouse
    - Google BigQuery

Load data to Amazon Redshift

```
# Pandas .to_parquet() method
df.to_parquet("./s3://path/to/bucket/customer.parquet")
# PySpark .write.parquet() method
df.write.parquet("./s3://path/to/bucket/customer.parquet")
```

```
COPY customer
FROM 's3://path/to/bucket/customer.parquet'
FORMAT as parquet
...
```

Load data to PostgreSQL

```
# Transformation on data
recommendations = transform_find_recommendatins(ratings_df)

# Load into PostgreSQL database
recommendations.to_sql("recommendations",
                       db_engine,
                       schema="store",
                       if_exists="replace")
```

PUTTING IT ALL TOGETHER

the ETL function

```
def extract_table_to_df(tablename, db_engine):
  return pd.read_sql("SELECT * FROM {}".format(tablename), db_engine)

def split_columns_transform(df, column, pat, suffixes):
  # Converts column into str and splits it on pat...

def load_df_into_dwh(film_df, tablename, schema, db_engine):
  return pd.to_sql(tablename, db_engine, schema=schema, if_exists="replace")

db_engines = { ... } # Needs to be configured
def etl():
  # Extract
  film_df = extract_table_to_df("film", db_engines["store"])
  # Transform
  film_df = split_columns_transform(film_df, "rental_rate", ".", ["_dollar", "_cents"])
  # Load
  load_df_into_dwh(film_df, "film", "store", db_engines["dwh"])
```

✱ Scheduling with DAGs in Airflow

```
from airflow.models import DAG
from airflow.operators.python_operator import PythonOperator

dag = DAG(dag_id="etl_pipeline",
          schedule_interval="0 0 * * *")

etl_task = PythonOperator(task_id="etl_task",
                          python_callable=etl,
                          dag=dag)

etl_task.set_upstream(wait_for_this_task)
```