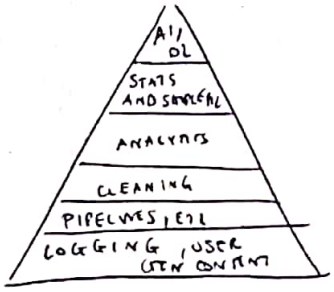
DATA ENGINEERING NANO DEGREE (UDACITY)

data engineering (def) → take raw data from database, do some work on it, deliver clean dataset (for end-users or analytics)



LEARN/OPTIMIZE
AGGREGATE/LABEL
EXPLORE/TRANSFORM
MOVE/STORE
} DATA ENGINEER
COLLECT

the pyramid of data science

Common data engineer activities:
- ingest data from a data source
- build and maintain a DW
- create a data pipeline
- create an analytics table for a specific use case
- migrate data to the cloud
usw.

data model: abstraction that organizes elements of data and how they will relate to each other

data modeling → the process of creating data models for an information system
a database is the end-state

- gather requirements (use cases)
- conceptual data modeling (think of the relationships) ER
- logical data modeling (tables, schemas)
- physical data modeling (use the DDL to code the database)
(data definition language)

it's an iterative process

relational databases

based on this relational model (def by Codd)

this model organizes data into one or more table (= relations) of columns and rows, with a unique key identifying each row.
generally each table represents one "entity type"

RDBMS: relational database management system
SQL: Structured Query Language used in rel. databases

example s
- Oracle
- MySql
- PostgreSQL
- Sqlite

database/schema = collection of tables

advantages of relational databases
- easy to use - SQL
- do JOINS
- do aggregation and analytics (Groupby, orderby, etc)
- smaller data volumes (not big data)
- easy to change business-requirements
- flexibility for queries.
usw.
- secondary index (used as addition to the 'primary key')
- ACID transactions - data integrity, means that data is always correct

ACID Transactions : properties of database transactions intended to guarantee validity even in the event of errors, power failures ...

Atomicity — the whole transaction is processed or nothing is
Consistency — only transactions that abide by constraints and rules are written into the db, otherwise the db keeps the previous state
Isolation — transactions are processed independently and securely, order does not matter
Durability — completed transactions are saved to database even in cases of system failure (once they have been committed)

disadvantages of relational databases (when not to use)

- With large amounts of data (that are not distributed)
- when need to store different data type formats
- when need high throughput -- fast reads
- need a flexible scheme
- need high availability (rel. databases have single point of failure since not distributed)
- need horizontal scalability (ability to add more nodes to increase system performance or space)

these are also the reasons why NoSQL databases were created

Postgre SQL — open source, builds on SQL but has its own syntax
allows duplicates

non-relational databases (NoSQL) — simpler design, horizontal scaling, finer control of availability
data structures are different ... and make operations faster

some examples:

- Apache Cassandra (Partition Row store)
- MongoDB (Document store)
- DynamoDB (Key-value store)
- Apache HBase (Wide Column store)
- Neo4J (Graph database)

basics of Apache Cassandra.

Keyspace: collection of tables (basically the database)
table: group of partitions
rows: single item

partition: fundamental unit of access, collection of rows
primary key: made up of partition key + clustering columns
columns: clustering and data, labeled element

Apache Cassandra — provides scalability and high availability without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it the perfect platform for mission-critical data.

has its own query language CQL (Cassandra Query Language)

ex. Uber and Netflix use it

useful for:

- transaction logging
- IOT
- time series data
- any workload that is heavy on writes

relational and non-relational databases do not replace each other for all tasks (the best model is dependant on the use cases)

NoSQL → no ACID (except MongoDB, HackLogic), no JOINS, no aggregation

RELATIONAL DATA MODELS

database (def): a set of related data and the way it is organized

database management system: the computer systems that allows users to interact with the database. sometimes the terms db and dbms are used interchangeably

invented by Codd at IBM in 1969, he proposed 3 rules

Rule 1: All information in a relational database is represented explicitly at the logical level and in exactly one way - by values in tables.

relational importance:
- standardization of data model
- flexibility in adding / altering tables
- data integrity
- Standard Query Language (SQL)
- simplicity
- intuitive organization

OLAP VS OLTP

Online Analytical Processing (OLAP): allow for complex analytical and ad hoc queries (optimized for reads)

Online Transactional Processing (OLTP): allow for less complex queries in large volume (types of queries are read/insert/update/delete)

normalization (Codd): reduce data redundancy (no copies) and increase data integrity (correctness)

denormalization: with heavy workloads to increase performance

objectives of normal form
1. free database from unwanted insertions, updates, and deletion dependencies
2. reduce the need for refactoring the db as new types of data are introduced
3. to make the relational model more informative
4. make db neutral to query statistics (every query will work, it's the opposite of what happens in NoSql)

Normal Forms

First Normal Form (1NF)
↓
Second Normal Form (2NF)
↓
Third Normal Form (3NF)

1NF: atomic values in each cell (no lists/sets)
add data without altering table
separate different relations into different tables
Keep relationships between tables using FOREIGN KEY

2NF: 1NF (reached first)
all columns in the table rely on primary key (it means that I only need the key to access one field)

3NF: 2NF
no transitive dependencies (a column values automatically determines the value in another column → this means the info could be potentially duplicated. I need to split into another table.

denormalization: process of trying to improve the read performance at the expense of losing some write performance by adding redundant copies of data

Keep the data consistent → in presence of more copies of the same data, it is important to update/delete them at the same time

example: with denormalization we want to think about the queries we are running and how we can reduce our number of JOINS even if that means duplicating data!
because JOINS can be slow

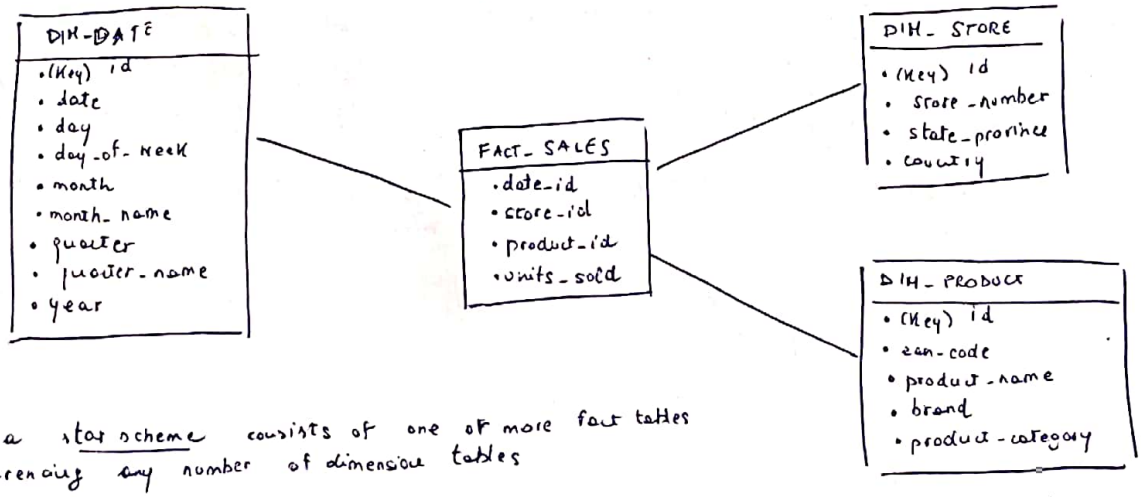
FACT TABLES: consist of the measurements, metrics or facts of a business process

DIMENSION TABLES: a structure that categorizes facts and measures in order to enable users to answer business questions. dimensions are people, products, place and time.

Schemas for DW:
1. Star schema
2. Snowflake schema

Star / Snowflake refers to the shape you have in ERDs (Entity-Relationship Diagrams)

example : star schema (fact table in the middle) dimension tables surrounding referenced by foreign key

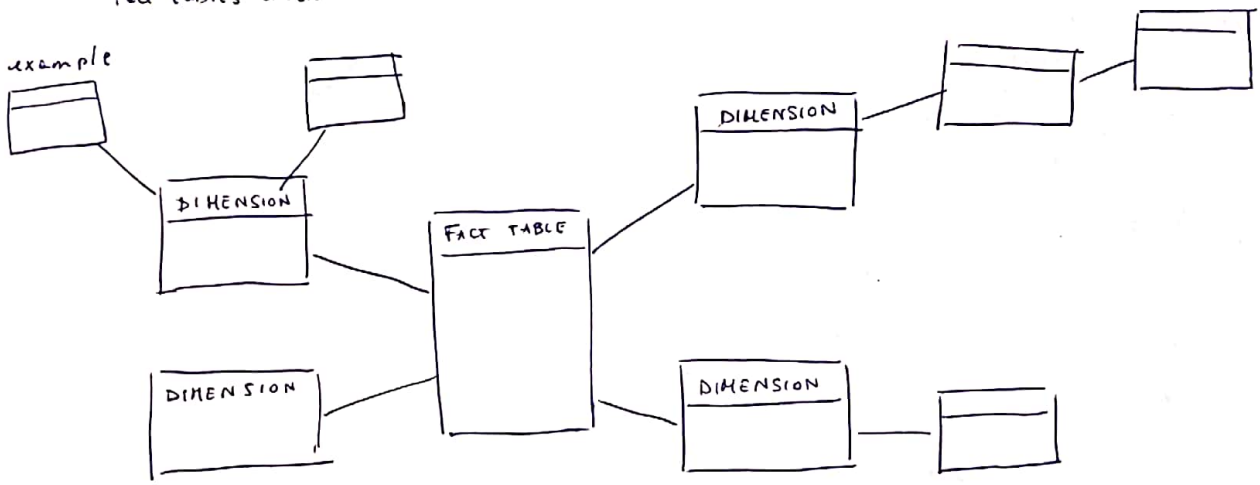


to a star scheme consists of one or more fact tables referencing any number of dimension tables

benefits :
- denormalized
- simplify queries
- fast aggregations

drawbacks :
- issues that come with denormalization (also data integrity)
- many to many relationships

Snowflake schema : logical arrangement of tables in a multidimensial database represented by centralized fact tables which are connected to multiple dimensions



to the star schema is a special case of snowflake schema with no one-to-many relationships, and less normalized. (snowflake only in 1NF or 2NF anyway)

Some useful CONSTRAINTS :

- NOT NULL . indicate that the column cannot contain a null value
- UNIQUE . used to specify that the data across all the rows in one column are unique within the table
- PRIMARY KEY / COMPOSITE KEY
  - always UNIQUE AND NOT NULL
  - when a group of columns are defined as a primary key

upsert : updating / inserting (in postgreSQL INSERT + ON CONFLICT)
DO NOTHING
UPDATE
See tutorials

WRAP UP :
- what is a relational db
- OLAP vs OLTP
- Normalization / denormalization
- fact / dimension tables
- star / snowflake schemas

ETL : Extract, Transform, Load
- Extract : read from files
- Transform : convert to db format
- Load : write to db

NoSQL DATA MODELS

"Not Only SQL"

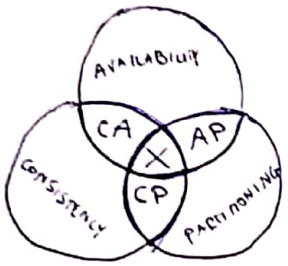
Apache Cassandra (open source) created at Facebook

- masterless architecture
- linear scalability, low latency
- high availability

distributed database -> to have high availability we need to have copies of data, if a node fails, we can use the remaining
↓ scaled horizontally between many machines

eventual consistency: a consistency model used in distributed computing to achieve high availability that informally guarantees data, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value
(this inconsistency could last milliseconds and there are measures implemented to prevent that inconsistent data "go astray".

CAP theorem: it is impossible for a distributed data system to simultaneously provide more than two out of the following three guarantees of consistency, availability, partition tolerance.



consistency: every read gets the latest and correct piece of data
availability: every request is received and a response is given - without a guarantee that the data is the latest update
partition tolerance: the system continues to work regardless of losing network connectivity between nodes

Apache Cassandra is an AP system, so it sacrifices consistency during network failures

denormalization is key —> think of the queries first! because there are no JOINS, we query only one table at a time
- need denormalization for fast reads
- Apache Cassandra is optimized for fast writes, so we don't care if we have many copies to update

to have 1 table for 1 query is a very acceptable practice

CQL: Cassandra Query Language (very similar to SQL in syntax)

Primary Key: how each row can be uniquely identified. the first element of the primary key is the partition key (which will determine the distribution), then in addition there can be clustering columns.
the partition key value is hashed and stored into the system node

primary keys must be unique, hashing of primary keys values result in placement on a particular node in the system, can be simple or composite

choose a primary (partition?) key that evenly distributes the data.

clustering columns: they determine the sort order within a partition (ascending order!)

```
CREATE TABLE music_library (year int, artist_name text, album_name text,
                            PRIMARY KEY ((year), artist_name, album_name))
```

using the () for the partition keys we can specify how many columns as we want for it! / partition key / clustering columns
otherwise the partition key is just the first column only (it always goes first)

WHERE clause: Apache Cassandra is focused on the WHERE clause
the partition key must be excluded in the query and any clustering columns can be used in the order they appear in the primary key

ALLOW FILTERING: keyword used to indicate to Cassandra to perform filtering when returning results from a SELECT * WHERE clause

Avoid using ALLOW FILTERING (better to redesign the tables) because it could lead to very slow performances

SELECT * IS highly discouraged

clustering columns must be used in order in a query! and other filtering columns must go after the 5

DATA WAREHOUSES

business → (made of) more business processes

- operational processes — "make it work"
- analytical processes — "what is going on?" e.g. assess performance

but operational databases are not good to support analytical processes (because of the complexity, too many joins...)
so we create a data warehouse: a system (including processes, technologies & data representation) that enables us to support analytical processes (OLTP → OLAP)
they are also called "Online Analytical Processing System".

under a technical perspective

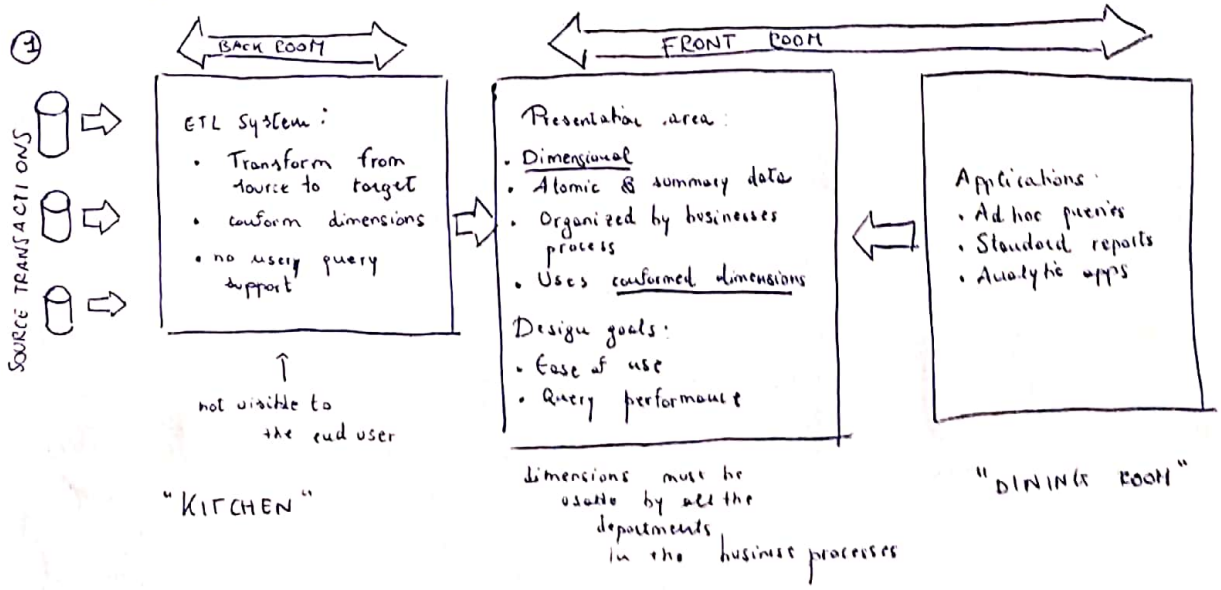
def 1: a DWH is a copy of transaction data specifically structured for query and analysis
def 2: a DWH is a subject-oriented, integrated, nonvolatile, and time-variant collection of data in support of management's decisions. (integrated: info comes from many sources; time-variant: depends on time)
def 3: a DWH is a system that retrieves and consolidates data periodically from the source systems into a dimensional or normalized data store. It usually keeps years of history and is queried for business intelligence or other analytical activities. It is typically updated in batches, not every time a transaction happens in the source system.

1. ETL pipeline
2. dimensional model (using optimized technologies)
3. business intelligence applications (e.g. visualization)

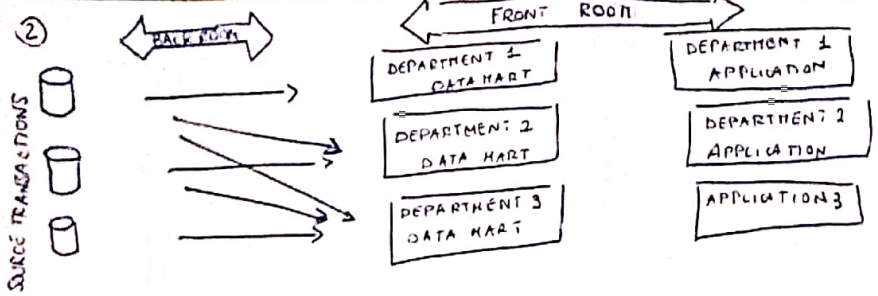
Dimensional modeling (recap)
- star schema
- easy to understand for BI
- fast read
- good for OLAP but not for OLTP

fact tables: columns contain events recorded in quantifiable metrics (quantity, duration, ratings...)
dimension tables: columns contain attributes (store at which the item is purchased, customer who made the sale etc.)
↳ always numeric and additive

DWH architectures
1. Kimball's Bus
2. Independent Data Marts
3. Inmon's Corporate Information Factory (CIF)
4. Hybrid Bus & CIF



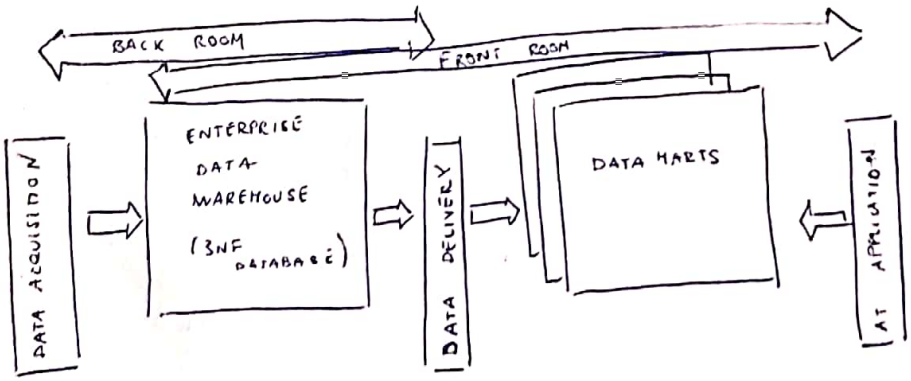
Bus Matrix: gives indication to what info is available to which business process

② 

- separate ETL processes for each department
- different dimensional data model
- problems: uncoordinated and inconsistent views
- advantages: every dept has autonomy, no need of overhead for communication across departments.

usually this architecture is discouraged!

③ (Inmon's CIF)



(with denormalized models) (the enterprise dwh)
- more data marts but a common normalized db
- the BI application can get info both from back room and front room

④ Hybrid BUS & CIF
- still expose the back room to the end-users
- no independent data marts, but an enterprise DW bus architecture

OLAP cube(s): aggregation of a fact metric on a number of dimensions (3D result)
↓ easy to communicate to business users

OLAP operations

rollup: summarizing the data along a dimension (by using an aggregate function) reducing n. of rows/columns
drilldown: navigate to the most detailed kind of data. increasing n. of rows/columns

the OLAP cube should store the finest grain of data (atomic level)

slicing: reducing an N-dimensional cube to N-1 dimensions by fixing one dimension
dice: computing a sub-cube by restricting some values of the dimensions (but the number of dimensions stays the same)

- group by cube (col1, col2, col3) → aggregate all possible combinations of groupings, of length 0, 1, 2, 3 ...
- GROUPING SETS → group by grouping sets ((), col1, (col1, col2), ... (col1, ... col2)) etc.

IMPORTANT: drop all the NONES before doing a cube! otherwise one gets confused when seeing the results (not knowing if a None was there already or not...)

OLAP cubes technology:

approach 1: pre-aggregate the OLAP cubes and saves them on a special purpose non-relational database (HOLAP)

approach 2: (more popular right now) compute the OLAP cubes on the fly from the existing relational databases where the dimensional model resides (ROLAP)

columnar format in ROLAP faster than row format

cstore_fdw extension for postgres (python-driver)

~~Sql-IO-Sql ETL~~

CLOUD COMPUTING

using a network of remote servers hosted on the internet to store, manage, and process data, rather than a local server or a personal computer

advantages:
- no need to invest in costly hw upfront
- rapidly provision resources
- provide efficient global access

AWS : Amazon Web Services
140 services: compute, storage, databases, networking, dev tools etc.

these services can be accessed via:
- AWS management console
- CLI
- SDKs - APIs built through different languages and platforms

For the instructions on how to use AWS see the lessons

DATA WAREHOUSE ON THE CLOUD

choices — cloud (low barrier to entry, scalability and elasticity out of the box)
— on-premise (problems: heterogeneity, scalability, elasticity of the tools, cost of ownership)

cloud — cloud-managed (Amazon RDS, Amazon DynamoDB, Amazon S3) → reuse of expertise (stuff already done in Amazon) but we lose control, things are already configured
— self-managed (EC2 + Postgresql, EC2 + Cassandra, EC2 + Unix FS) → most customizable configurations

We're going to use an approach cloud-managed + SQL + Amazon Redshift (SQL Columnar Massively Parallel)

Amazon Redshift

- column-oriented storage
- best-suited for storing OLAP workloads, summing over a long history
- internally, it's a modified postgresql adapted for columnar storage

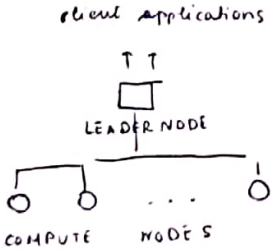
Most relational databases execute multiple queries in parallel when in presence of multiple cores/servers however, every query is always executed on a single CPU of a single machine.

instead, in Massively Parallel Processing (MPP) databases parallelize the execution of one query on multiple CPUs/machines.

This is done by partitioning a table and then processing these partitions in parallel across different CPUs

Redshift architecture

the cluster



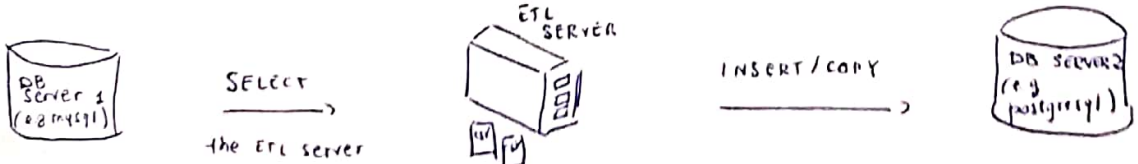
leader node: interacts with user applications. For example using protocols such as JDBC it coordinates the work of the compute nodes

each compute node is logically divided into a number of slices, we can consider a slice as a CPU with its disks

n slices = n partitions that can be parallelized
so the slice is the unit of parallelism

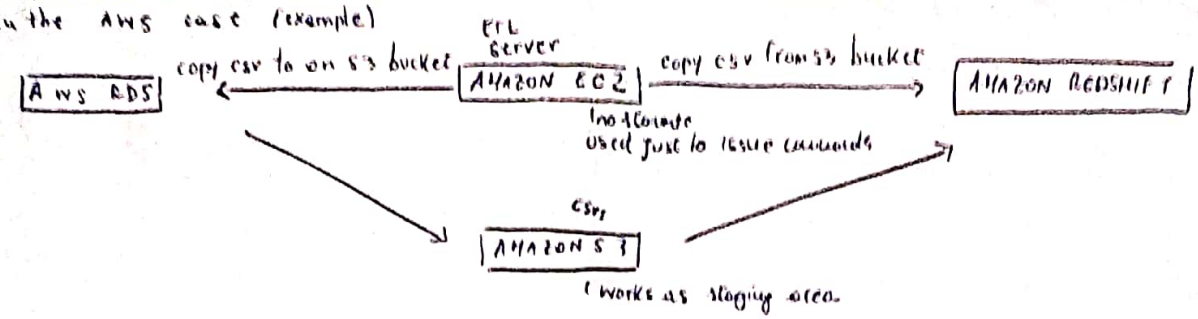
1 node = 1 AWS EC2 instance

SQL-TO-SQL ETL

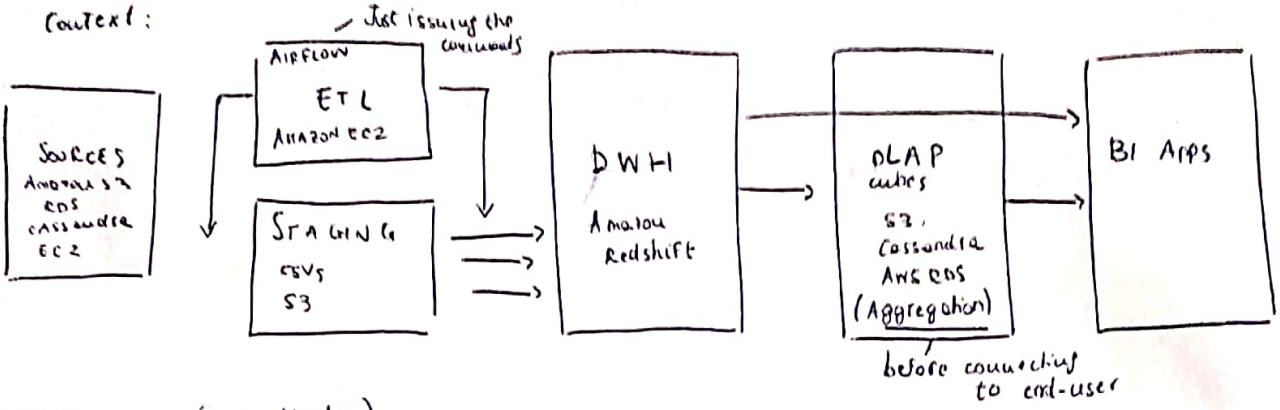


- if the two servers use the same RDBMS could be possible to do a SELECT INTO without having an ETL Server in the middle, but with two different RDBMs this is the architecture

In the AWS case (example)



In Context:



Redshift ETL (in particular)

COPY command: better than using INSERT, that is slow
transfer data from S3 staging area to Redshift
- if the file is large it's better to break it up to multiple files and then ingest in parallel (either using a common prefix or a manifest file)
- better to ingest from the same AWS region

example. common prefix:
part-00000.csv.gz
part-00001.csv.gz
:
part-00005.csv.gz

example. manifest file: you explicitely list the files used in a manifest file

column compression for optimization: there's a different strategy for each column type and redshift gives user control over the compression of each column
also the COPY command does

it is also possible to ingest directly using SSH from EC2 machines
- S3 needs to be used as a staging area
- (usually) an EC2 ETL worker needs to run the ingestion jobs orchestrated by a dataflow product like Airflow, Luigi etc.

Infrastructure-as-Code (IaC):
being in the cloud, we have the ability to create infrastructure i.e. machines, users
IaC lets you automate, maintain, deploy, replicate and share complex infrastructures as easily as you maintain code (this was impossible with on-premise deployment)
↓ point of intersection with DevOps

Python AWS SDK called boto3 - see demos

optimize table design: having an idea about the frequent access patterns to a table, we can create partitions from it in a way to speed it up

2 strategies:
- distribution style
- sorting key

Distribution styles: (sending keys all over the CPUs (problem with shuffling))

1) EVEN distribution -> round robin over all slices to achieve load balancing, good if table won't be joined
2) ALL " -> broadcasting: small tables could be replicated on all slices to speed up joins (frequent with dimension tables)
3) AUTO " -> leave decision to Redshift, small tables use ALL, and large tables use EVEN strategy
4) KEY " -> rows having similar values are placed in the same slice
useful when a dimension table is too big to be distributed with ALL strategy.

this allows Joins without shuffling of keys, Keyword 'distKey'. so it's useful to add it to the columns we're going to use as keys for joins

Sorting Key :
- one can define its columns as sorting key
- upon loading, rows are stored before distribution to slices
- minimizes the query time since each node already has contiguous ranges of rows based on the sorting key
- useful for columns that are used frequently in sorting like the date dimension and its corresponding foreign key in the fact table.

e.g. a column that is ordered by a lot is a good sorting key

the data lake is the evolution of data warehouse for big data.
(as architecture)

APACHE SPARK

one of the most popular tools for big data analytics (generally faster than Hadoop, although this one is still used by some companies)

"Numbers everyone should know" →

CPU operation	0.4 ns
memory reference	100 ns
random read from SSD	16 µs
round trip for data from EU to US	150 ms

CPU

Twitter → 6000 tweets / second, 200 bytes / tweet

CPU 2.5 GHz → 2.5 billion operations per second

the CPU is the brain of the computer directs other components and performs mathematical operations, it can memorize small amounts of data in the registers

distributed system / cluster, made up of nodes

Memory

Storage

Network : moving data across the network is the bottleneck when working with Big Data

Small data : works fine on local machine, memory can load all the data
big data : the memory is too small to contain all the data, so it must unload some onto the storage, making things slower to restart

medium data (size) : if a dataset is larger than the size of your RAM, it may still be possible to analyze the dataset on a laptop (e.g. pandas can split the dataset in smaller chunks)

distributed computing : in a network of CPUs, each node has its own memory, and they communicate with each other through messages

parallel computing : every node has access to the same memory to exchange information
(can be thought of as a particular type of distributed computing that is tightly coupled)

The Hadoop Ecosystem for big data storage and data analysis

- Hadoop Distributed File System (data storage): splits data into chunks and stores the chunks across a cluster of computers
- Map Reduce : a system for processing and analyzing large data sets in parallel
- YARN - resource manager : schedules jobs across a cluster
- Hadoop common - utilities

other projects

Apache PIG : a SQL-like language that runs on top of Hadoop Map Reduce
Apache HIVE : like PIG
Apache SPARK : another big data framework, contains libraries for data analysis, machine learning, graph analysis, and streaming live data. generally faster than Hadoop since Hadoop writes intermediate results to disk whereas Spark tries to keep intermediate results in memory whenever possible. (Spark does not include a file storage system, but can use S3 for example)
" STORM (streaming data) :
" FLINK (streaming data) :

data streaming : store and analyze data in real-time such as Facebook posts or Twitter tweets.

Map Reduce (in general) is a programming technique for manipulating large data sets. It works by first dividing a large dataset and then distributing the data across a cluster. In the map step, each data is analyzed and converted into a (key, value) pair. Then these key-value pairs are shuffled across the cluster so that all keys are on the same machine. In the reduce step, the values with the same keys are combined together.

DATA WRANGLING WITH SPARK

Spark is written in Scala (a functional programming language)
PySpark : Python API for Spark, written with Data principles in mind
Functional programming is useful in a distributed setting
Leslie Lamport "You know you are in a distributed system when your computer crashes because someone that you don't even know made a mistake"

Functional programming can help minimize this errors
↳ functions as intended in maths (pure functions) not as in imperative languages

pure functions : no side-effects (alter each one) on variables outside their scope,
no contamination of the input (no mutification)

each Spark function makes a copy of its input data and never alters the original one (input is immutable)
lazy evaluation: build first which functions which data will use, in a graph.
Directed Acyclical Graph (DAG) : once it is built Spark waits until the last possible moment to get the data.
stages : multi-step combos
maps : function that copy the original data and apply to it whatever function we specified

- collect() used to force evaluation (also show() can be used)

lambda : anonymous inline functions in Python (map. (lambda x : x.lower())) for writing functional style programs
"RDD" in the output refers to resilient distributed dataset (fault-tolerant datasets distributed across a cluster)

data formats : most common CSV, JSON, HTML, XML

Spark Program

- SparkContext (entry point) can be created via cmd line using a SparkConf

two ways for data wrangling : imperative programming (Pandas ~~Spark~~ DataFrames + Python) or declarative programming (Spark (SQL))
"HOW?" "What?"
Spark data frames → imperative perspective useful functions

- select()
- filter()
- where()
- groupBy()
- sort()
- dropDuplicates(): returns a new DataFrame with unique rows based on all or just a subset of columns
- WithColumn(): returns a new DataFrame by adding a columns or replacing the existing column that has the same name.

Spark SQL aggregate functions
User defined functions (UDF)
Window functions : combine the values of ranges of rows in a DataFrame.
SparkSQL -> declarative perspective (very similar to normal SQL)
createOrReplaceTempView() -> create a view from a table (or update it if already exists)
↳ write SQL queries directly on this view
toPandas() -> convert Spark DFs to Pandas DFs

You might prefer SparkSQL over data frames since syntax is clearer (and maybe we has more experience with SQL). The advantage of Spark data frames is that they can give more control (eg if it is possible to break down queries and make debugging easier

both methods are optimized (by Catalyst), so there is no difference in performance
the code generated by the API execution plan operates on a lower level data abstraction called RDD ("Resilient Distributed Data Set")

it's possible to use RDD API directly in imperative programming. It can give more flexibility but the code is harder to read and write

SPARK DEBUGGING AND OPTIMIZATION

3 cluster managers:
- Standalone mode
- MESOS
- YARN

We use AWS for the clusters
S3, EC2 (Spark cluster)

Amazon EHR: Service that provides EC2 instances already equipped with big data technologies, such as Hadoop and Spark

chmod og-rwx MyKey-cluster.pem → avoid rwx rights for all other users and groups

since the cluster is accessible only within the cluster, it is necessary to setup a proxy to access it from the outside (AWS recommends FoxyProxy from the Mozilla Foundation)
then the EMR cluster becomes accessible via command line

TODO FIX SSH CONNECTION TO SPARK CLUSTER

S3: Simple Storage Service
↳ We can read data from S3 and load them to a Spark cluster

an alternative to using S3 as storage with Spark (that requires moving files across the network), is to use HDFS installed onto the Spark cluster (but HDFS must be maintained) so sometimes it is just easier to use S3.

we can use scp to copy files to HDFS onto the Apache Spark cluster, log into the cluster then,

```
$ hdfs dfs -mkdir <folder_name>
$ hdfs dfs -copyFromLocal <file_to_copy> <folder_dest>   # this will effectively copy the file into the fs!
```

(we can inspect files from the web ui console accessible through the master node public DNS)
then it can be read using hdfs:// as protocol for the file address URI (we would use s3n for files stored into S3)

debugging: syntax (typos) errors, code errors, data errors (missing values, nan, data not conforming to the schema)

accumulators: work as global variables for the entire cluster, can be used for debugging (keyword global)

Spark WebUI: monitor the cluster independently from its workers
it provides the current cluster configuration, the DAG with its stages and tasks

Spark ports:
- 7077: port used by the master node to communicate with the workers (not intended for users)
- 8888: Jupyter notebooks
- 4040: shows active Spark jobs
- 8080: Spark webUI

cohort analysis: analyzing activities in groups = cohorts (of users)

data skew: non uniform distribution in a dataset

DATA LAKES

evolution of data warehouse (that in some domains it is still the best choice)

things that drove the evolution:
- abundance of unstructured data (text, xml, json, images...)
- unprecedented data volumes
- rise of Big Data technologies (HDFS, Spark...) → much lower cost per TB, big data tools allow to process data at scale on the same hardware used for storage
- new types of data analysis methods → extract value from data, freedom to play with datasets, dwh tables are too strict

we can still keep unstructured data in a DWH, e.g. deep json structures or blobs

so the data lake is a new warehouse that evolved to cope with:
- variety of data formats
- agile data-exploration activities needed by data scientists
- data transformation needed by advanced analytics such as machine learning

How Big Data technologies affected DWHs?

1. ETL offloading → use same hw for storage and processing
   dimensional modeling for high/known-value data
   storing (cheap) of low/unknown value data

2. Schema-on-Read : the schema for a table is either inferred, or specified, but the data is not inserted, upon read the data is checked against the specified schema.

→ basically Big Data tools made it easy to work with a file as it was working with a db.

the schema-on-read function could cause errors in reading the data types from file, so there are ways to handle these errors or to enforce the schema that we want through a structure

3. (Un-/Semi-) Structured support

- Spark can write/read to many formats
- read/write from a variety of file systems (local, HDFS, S3)
- read/write from a variety of databases (SQL through JDBC, NoSQL like MongoDB, Cassandra, Neo4j..)
- All exposed in the dataframe abstraction.

TODO DOWNLOAD ALL THE NOTEBOOKS AT THE END

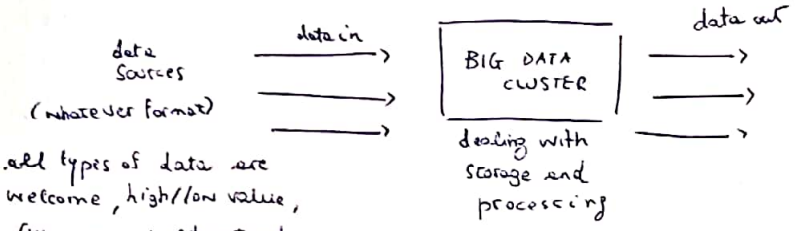
Schema-on-Read Demo

Advanced Analytics : NLP Demo using spark-nlp

query subfields using dot notations on JSON structs

part-of-speech tagging → which word correspond to which part of speech

DATA lake architecture



all types of data are welcome, high/low value, (un-, semi-) structured

data is stored as-is : so we talk about ELT Extract, Load, Transform (differently from DWHs)

data is processed later using schema on read

other features

- massive parallelism
- scalability
- columnar storage (parquet)
- MLib, Graphx for advanced analytics

DATA WAREHOUSE VS DATA LAKE

	DATA WAREHOUSE	DATA LAKE
data form	tabular format	all formats
data value	high only	high-value, medium-value and to-be discovered
INGESTION	ETL	ELT
DATA MODEL	star & snowflake with conformed dimensions or data-marts and OLAP cubes	star, snowflakes and OLAP still possible but also other ad-hoc representations
SCHEMA	Known before ingestion (schema-on-write)	on-the-fly at the time of analysis (schema-on-read)
TECHNOLOGY	expensive MPP databases with expensive disks and connectivity	commodity hw with parallelism as first-principle
DATA QUALITY	high with effort for consistency and clear rules for accessibility	mixed, some data remain in raw format, some data is transformed to higher quality
USERS	business analysts	data scientists, business analysts & ML engineers
ANALYTICS	reports and bi visualizations	ML, graph analytics and data exploration.

DWH → like bottled water, in particular size and shape

data lake → many streams flow into a water lake and it's up to the user to get the water how they want it.

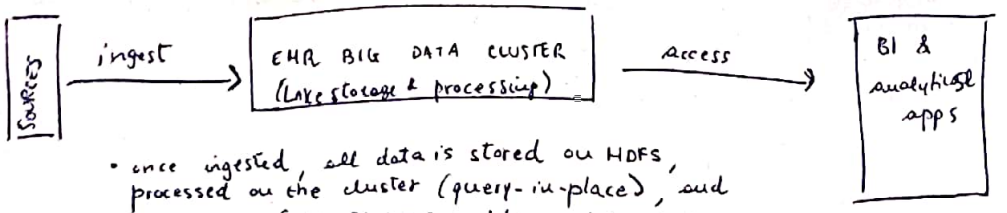
Data lakes options on AWS

two choices

storage	processing	AWS-Managed solution	Vendor-Managed (e.g. Cloudera, Databricks)
HDFS	Spark (plus Hive)	AWS EMR (HDFS + Spark)	EC2 + Vendor solution (1)
S3	Spark	AWS EMR (Spark only)	EC2 + Vendor solution (2)
S3	Serverless	AWS Athena	Serverless + Vendor solution (3)

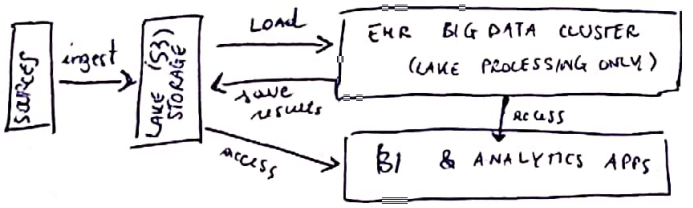
EMR : Elastic Map Reduce

1)



- once ingested, all data is stored on HDFS, processed on the cluster (query-in-place), and accessed from BI apps or Advanced Analytics Apps
- the EMR cluster is created once, but it can grow and is not supposed to be shutdown

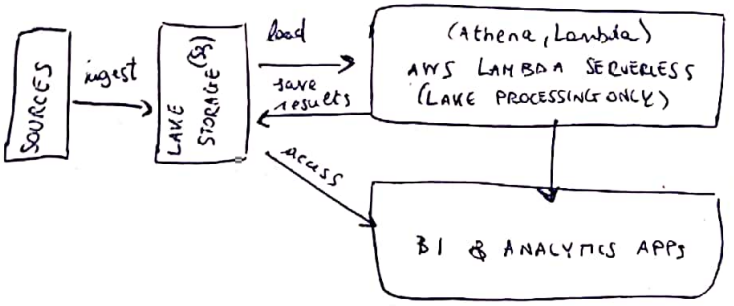
2)



no HDFS

- all data stored in S3
- data loaded to EMR for processing
- query/processing results saved back to S3
- EMR cluster spun on demand, shutdown otherwise

3)



abstraction from machines

- all data is stored in S3
- Athena is a service that can load & process data on Serverless Lambda resources (pay by execution time, not by machine up-time)

Access S3 from Spark — Demo

hadoop-aws — library to connect to S3

Data Lake on EMR — Demo

Data Lake on Athena — Demo — Amazon Glue: crawler, takes data, infers the schema and makes the data available to query.

Data lake issues:

- prone to be a data garbage dump
- difficult to implement data governance, given the wide accessibility of cross-department data
- not clear sometimes whether it should replace, offload or work in parallel.

DATA PIPELINES

a series of steps in which data is processed, steps could be also parallel and usually execution is scheduled
it usually uses ETL or ELT

T is a variant of ETL where the extracted data is first loaded into the target system. transformations are performed after the data is loaded into the DWH.

(Apache) Kafka: an open source stream-processing software platform developed by LinkedIn and donated to the Apache Software Foundation, written in Scala and Java.
It provides a unified, high throughput, low-latency platform for handling real-time data feeds.

data validation: ensuring that data is present, correct & meaningful (it's part of the pipeline)

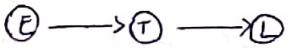
data pipelines provide logical guidelines to help with the process

DAGs are the model used to represent data pipelines

in general, a graph is a model that describes entities (nodes) and relationships (edges) between them.

in DAGs edges have a direction and there are no loops/cycles

in ETL, each step of the process typically depends on the last.



each step is a node and the dependencies on prior steps are directed edges

Apache Airflow's open-source tool which schedules data pipelines as DAGs
Created by Airbnb to simplify data pipelines creation
it can run data analysis itself, or trigger other frameworks.
Web UI => to visualize and interact with data pipelines

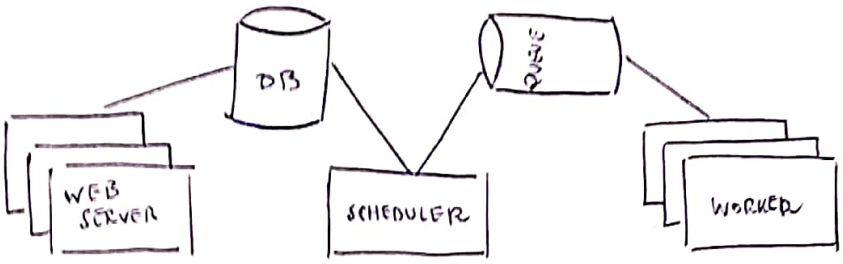
```
dag = DAG (lesson1.demo1,
           start_date = datetime.datetime.now()
)
```

logging => to see the output in Airflow logs

Workspace instructions => after updating the DAG, run /opt/airflow/start.sh to start the server (can take up to 10 minutes)

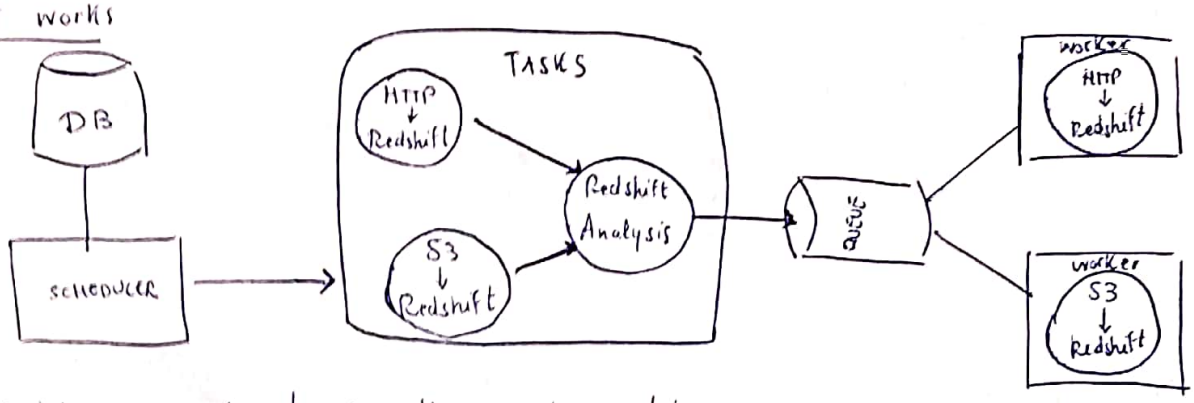
```
def my_function():
    logging.info("Ciao mondo!")

greet_task = PythonOperator(
    task_id = "ciao_task",
    python_callable = my_function,
    dag = dag
)
```

Airflow components



Scheduler: orchestrates the execution of jobs on a trigger or schedule
queue: used by the scheduler to decide which tasks must be delivered to the workers
worker: executes operations defined in the DAGs pulling tasks from the queue (FIFO)
database: saves credentials, connections, history, and configuration. Also stores the state of all tasks in the system.
web interface: provides a control dashboard for users and maintainers (built using Flask)

how it works



- Scheduler starts DAGs based on time or external triggers
- then it looks at the steps to be processed within the DAG
- it places these steps in the queue
- workers pick up tasks from the queue and run them
- once the worker has finished running the step, the final status of the task is recorded and additional tasks are placed by the scheduler until all tasks are complete
- once all tasks have been completed, the DAG is complete.

DAG definition:
start-time can be even in the past!
operators define the atomic steps of work that make up a DAG
where instantiated operators are referred to as task

schedules can be defined with cron strings or Airflow presets

many Operators are available : PythonOperator, PostgresOperator, etc.

task dependencies can be specified programmatically ① using >> and <<

e.g. a >> b task A comes before B (A) ⟶ (B) in the graph

example :

```
hello_world_task = PythonOperator( task_id = '...')
goodbye_world_task = PythonOperator( task_id ...)
hello_world_task >> goodbye_world_task
```

or another option ② use set_downstream, set_upstream.

example. hello_world_task.set_downstream(goodbye_world_task) // hello_world comes first

hooks : realize connections and can be written in code, they're reusable if for external systems

e.g. db_hook = PostgresHook('demo')

many hooks : HttpHook, MySqlHook, SlackHook...

```
def list_keys()
    hook = S3Hook( aws_conn_id = 'aws_credentials')
    bucket = Variable.get('s3_bucket')
    prefix = Variable.get('s3_prefix')
    logging.info(f"Listing Keys from {bucket}/{prefix}")
    keys = hook.list_keys(bucket, prefix=prefix)
    for key in keys:
        logging.info(f"- s3://{bucket}/{key}")
```

connections and variables are created through the Airflow Web UI

context variables can be included as kwargs (they work as runtime vars)

e.g. = PythonOperator(

provide_context = True; // opens access to runtime variables

)

(check the docs for all the variables available in Airflow

> logging.info(f"My execution date is {kwargs['ds']}

DAG with connection Airflow - Redshift Demo

(WebUI) Admin > Connections > Create >

conn id = 'Redshift'
Conn type = Postgres (then we'll use a PostgresHook)
host = <redshift aws endpoint>
schema = udacity
login = <access key ID>
pass = <secret access key ID>
port = 5439

```
create_table = PostgresOperator(
        task_id =
```

DAG > "Code View" (to see the code)

DATA LINEAGE

the d.l. of a dataset describes the discrete step involved in the creation, movement and calculation of that dataset.

Why it is important:

1. to build confidence in data consumers, being able to describe the data lineage ensures them that the data pipeline is creating meaningful results.
2. defining metrics: allows everyone to agree on the definition of how a particular metric is calculated
3. debugging: helps to track down errors

in general it has important implications for a business, since it allows to track data flowing across the different depts.

ex. using Airflow DAGs we can easily see which task has failed, DAGs are a natural representation for the movement and transformation of data.

useful tools for data lineage in Airflow:
- rendered code tab for a task
- graph view for a DAG
- historical runs under the tree view

pipelines are driven by schedules → that determine which data should be analyzed and when

e.g.
```
dag = DAG (
        "lesson1.demo2",
        start_date = ..,
        schedule_interval = "@monthly".        // like cron jobs
      )
```

schedules allow to make assumptions about the scope of the data → the scope can be defined as the time of the current execution until the end of the last execution.

the scope is based on
- size of data
- frequency of data
- related datasets

the schedules can help improving data quality, by limiting our analysis to relevant data in a time period.

start date: (always required) the date Airflow starts running the pipelines. a start date can also be set in the past, in that case all the schedules included between that date and now will be run. this is a very useful feature to analyze data retroactively.

end date: (not required) the date Airflow uses to stop a pipeline. also useful to update/redesign a pipeline: e.g update the old pipeline with an end date and then have the new pipeline start on the end date of the old pipeline.

interval schedule is also not required

max_active_runs → how many runs of the DAG can be active

data partioning: process of isolating data to be analyzed by one or more attributes, such as time, logical type, or data size.
it often leads to faster and more reliable pipelines.

logical partitioning breaks conceptually related data into discrete groups for processing
size partitioning: separates data for processing based on desired or required storage limits.

in general partitioning makes debugging and rerunning failed tasks much simpler.

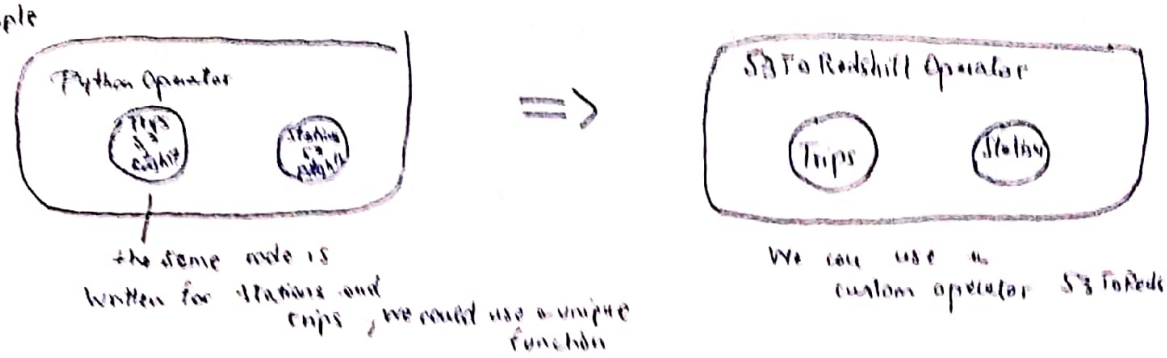
data quality: is the measure of how well a dataset satisfies its intended use
first of all, adherence to requirements (examples)
- data must be a certain size
- data must be accurate to some margin of error
- data must arrive within a given timeframe from the start of execution
- pipelines must run on a particular schedule
- data must not contain any sensitive information

PRODUCTION DATA PIPELINES

Airflow can be extended using plugins, typically custom operators and hooks

custom operators are typically used to capture frequently used operations into a reusable form

example



the same code is written for stations and trips, we could use a unique function

We can use the custom operator S3ToRedshift

custom hooks allow Airflow to integrate with non-standard data stores and systems

custom stuff is available in Airflow contrib (github) e.g. hooks for Spark, Cassandra, AWS etc.

Custom Operator

init()

execute()

template_fields = ("s3_key",)

this param must be templated

i.e. s3 -> templatized

__init__.py -> we need to define the plugins with the custom operators here, to make Airflow recognize them.

```
from airflow.plugins_manager import AirflowPlugin
import operators
class MyPlugin(AirflowPlugin):
    name = "<name>"
    operators = [
        operators.MyFirstOperator,
        operators.MySecondOperator
    ]
```

tasks should be defined such that they are

- atomic, single purpose (this helps reusability, maintainability and speed) easier to debug also using the UI
- maximize parallelism (speed up DAGs execution)
- make failure states obvious

SubDAGs -> can be created out of repeated series of tasks within DAGs (for reusable components)

there are some downsides:

- limited visibility in the UI (because single tasks are not immediately visible)
- harder to understand if not properly scoped
- encourages premature optimization (that can be bad because it obscures the code and makes difference to debug)

SubDAGs Demo SubDag Operator

monitoring: Airflow can surface metrics and emails to help stay on top of pipeline issues

Airflow can be configured with SLA (Service Level Agreement), which is defined as a time by which a DAG must complete

emails and alerts: on successes, failures, reruns (this is useful for mission-critical pipelines)

Airflow can be configured to publish metrics to statsd (that can be used with Grafana - to visualize metrics)

metrics aggregator