# Udacity : Data Engineering

## 1. Welcome

03/18/2021

Project 1: Data Modeling
- Postgres, NoSQL - Apache Cassandra

Project 2: Cloud Data Warehousing
- ELT pipeline
- S3, Redshift

Project 3: Data Lakes w Apache Spark
- ELT from S3
- process w Spark

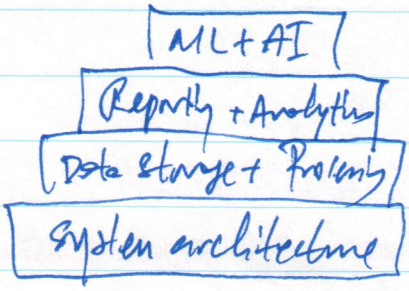
Project 4: Data Pipelines w Apache Airflow
- scheduling pipelines + triggers

Project 5: Data Engineering Capstone

## 2. Data Modeling

### 2.1. Introduction to Data Engineering

Data engineering comprises of all engineering + operational tasks required to make data available for end-use for analytics, model building, app development, ... etc.

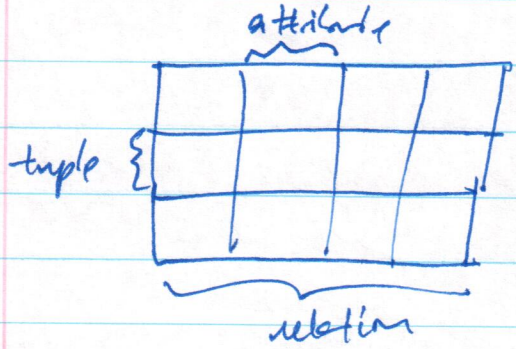


Common DE duties
- ingest data from source
- build + maintain data warehouse
- create pipeline
- create analytics table
- migrate data to cloud
- schedule + automate pipeline
- backfill data
- optimize queries
- design database

## 2.1 - Data Modeling

Data Modeling - organize data and how they relate to one another
- conceptual data modeling
  ~~relationship between~~ create tables
- logical data modeling - convert table DDL
- physical data modeling (data definition language)
  actual coding / implementation

Relational Model




SQL -
- Oracle
- Teradata
- MySQL
- PostgreSQL
- Sqlite

Database / Schema - collection of tables
Table / Relation - groups of rows (tuples, attributes)

Advantages of Relational DataBases
- ease of SQL
- JOINS
- aggregation + analytics
- smaller data volumes
- easier to change based on new requirements
- flexibility for queries
- modeling data, not modeling queries
- secondary indexes available
- ACID transaction - data integrity even in
  - guarantee validity even of errors, power failure...
  - Atomicity - all or nothing
  - Consistency - have to abide by rules/constraints
  - Isolation - processed independently + securely, order does not matter
  - Durability - completed transaction are saved, even if committed power failure.

When not to use Relational DataBases
- large data - RDMS can only scale vertically
- different data types
- high throughput - fast read
- flexible schema
- high availability - little downtime
- scale horizontally - add more servers to increase performance

PostgreSQL
- open source, SQL

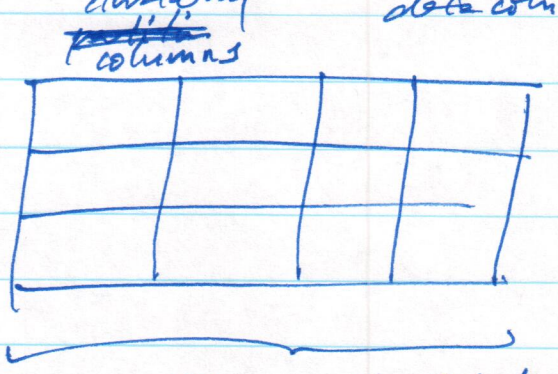
```
connection = psycopg2.connect("host=127.0.0.1 dbname=studentdb
                                user=kay")
  cursor = connection.cursor()
connection.set_session(autocommit=true)
 cursor.execute("create database udacity)
cursor.execute("Create Table If Not Exists music_library
                (album_name varchar, artist_name varchar,
                 year int);")
cursor.execute("select count(*) from music_library")
print(cur.fetchall())
cursor.execute("Insert Into music_library (album_name, artist_name,
                                           year)
                Values (%s, %s, %s)", ("    ", "    ", 1970))
cursor.execute("Select * From music_library;")
```

NoSQL

NoSQL = Not Only SQL
horizontal scaling

- Apache Cassandra (Partition Row store)
- MongoDB (Document store)
- DynamoDB (key-value store)
- Apache HBase (Wide column store)
- Neo4J (Graph Database)

Apache Cassandra
- keyspace - collection of tables
- table - group of partitions
- row
- does not allow duplicates
- scalable
- high availability / fault-tolerance
- CQL query language
- good for - transaction logging
  - IoT
  - time series data
  - any workload that writes to database
- can't do ad-hoc queries, but can add clustering columns + create new tables




When to use NoSQL
- large data
- horizontal scalability
- high throughput - fast reads
- flexible schema
- high availability
- store different data type
- user are distributed - low latency

When not to use NoSQL
- need ACID transactions
- need to do JOINS
- need to do aggregations + analytics
- changing business requirements
- queries not available, need flexibility
- small dataset

## 2.2. Relational Data Models

Database Management System - DMS
- all information represented by tables

Relational importance
- standardize of data model
- flexibility of adding + altering tables
- data integrity
- SQL
- simplicity
- intuitive organization

Online Analytical Processing (OLAP)
- optimized for reads, allow for complex, ad hoc queries
- heavy on aggregate

Online Transactional Processing (OLTP)
- allow for less complex queries in large volumes
- optimized for read, insert, update, delete
- little aggregate

Structuring Database
- Normalization - reduce data redundancy
  - increase data integrity
- Denormalization - done in read heavy workloads to increase performance

Objectives of Normal Form
- free database from unwanted insertion, update + deletion dependencies
- reduce need for refactoring database as new types of data are introduced

- make relational model more informative to users
- make database neutral to query statistics

First normal form (1NF)
- atomic values – each cell contains unique and single value
- able to add data without altering tables
- separate different relations into different tables
- keep relationships between tables together w/ foreign keys

Second normal form (2NF)
- reach 1NF
- all columns in table must rely on Primary key

Third normal form (3NF)
- reach 2NF
- no transitive dependencies

When update data, just do in 1 place.

e.g.

| Album ID | Album Name | Artist Name | Year | List of Songs |
|---|---|---|---|---|
| 1 | Rubber Soul | Beatles | 1965 | [ , , ] |
| 2 | Let it Be | Beatles | 1970 | [ , , ] |

1NF:

| Album_ID | Album_Name | Artist Name | Year | SongName |
|---|---|---|---|---|
| 1 | Rubber Soul | Beatles | 1965 | ' |
| 1 | " | " | " | ' |
| 2 | Let It Be | ' | 1970 | ' |
| 2 | " | ' | " | ' |

2NF:

| Album ID | Album Name | Artist Name | Year |
|---|---|---|---|
| 1 | Rubber Soul | Beatles | 1965 |
| 2 | Let It Be | " | 1970 |

| Song ID | Album Name | Song Name |
|---|---|---|
| 1 | 1 | - |
| 2 | 1 | - |
| 3 | 2 | - |
| 4 | 2 | - |

3NF:

Song library

| Song ID | ~~Album~~ Album ID | Song Name |
|---|---|---|
| 1 | 1 | - |
| 2 | 1 | - |
| 3 | 2 | - |
| 4 | 2 | - |

Album library

| Album ID | Album Name | Artist ID | Year |
|---|---|---|---|
| 1 | Rubber Soul | 1 | 1965 |
| 2 | Let it Be | 1 | 1970 |

Artist library

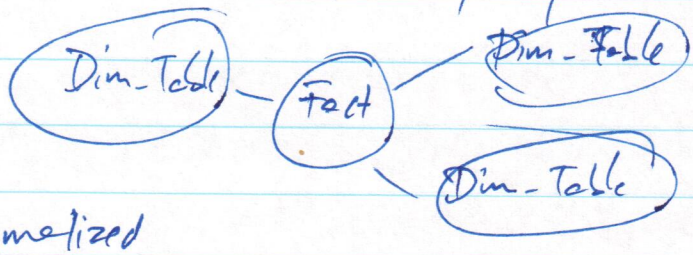
| Artist_ID | Artist Name |
|---|---|
| 1 | Beatles |

Denormalizati - JOINS are slow
- improve read performance of a database at expense of losing write performance by adding redundant copies of data
- undo after normalizati
- need more space due to copies
- write slower + on multiple tables

Fact Tables - measurements, metrics & facts
Dimension Table - people, products, place + time

Star Schema
Snowflake Schema
} Entity Relationship Diagrams (ERD)

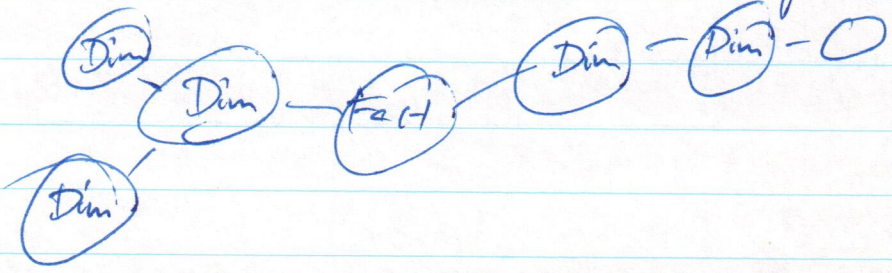
Star Schema
- > 1 fact tables referencing any dimension tables




- denormalized
- Simpler queries
- fast aggregations

Draw backs
- issues from denormalized
- data integrity
- decreased query flexibility
- many-to-many relationships

Snowflake Schema
- centralized fact table connected to multiple dimension tables.



- does not allow for 1 to many relationships, while star does
- more normalized than Star, but only in 1NF or 2NF

upsert

## 2.3 Project 1 : Data Modeling with Postgres

- fact + dimension tables for star schema
- ETL pipeline to transfer data from files in 2 local directories into tables in Postgres

- Song dataset - JSON - metadata
  - num_songs
  - artist_id
  - artist_latitude, longitude, location, name
  - song_id, title, duration, year

log dataset - JSON - year/month/events.json
- artist
- auth
- firstName, gender, lastName
- itemInSession, length, level, location, method, page, registration
- sessionId, song, status, ts, userAgent, userId

Fact table - songplays - songplay_id, start_time, user_id, level, song_id, artist_id, session_id, location, user_agent

Dimensions tables - users - user_id, first_name, last_name, gender, level
- songs
- artists
- time

## 2.4 NoSQL Data Models

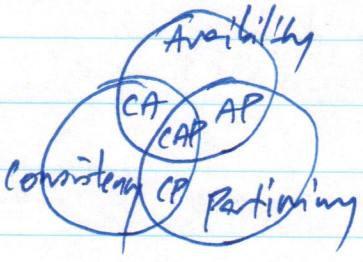
Not Only SQL

Apache Cassandra - open source NoSQL
- high available
- linear scalability
- created Facebook
- masterless architecture
- (node = server, system, ...) each node connected to another node.
- uses - Netflix, Apple, FB, Twitter

Distributed Databases - for high availability, need copies of data.

Eventual consistency - if no new updates, all data in nodes are same.
but for few seconds of latency, maybe inconsistent.

CAP theorem - impossible for distributed ~~system~~ data store to simultaneously provide > 2 out of 3 guarantees
- consistency, availability, partition tolerance
  - every read give correct data
  - always on
  - )work regardless of loss of network




C in CAP ≠ not ACID

'every read has correct data.

transaction must abide by constraints + rules.

mostly C and P are guaranteed.

Denormalize in Apache Cassandra
- no JOINs.
- consider queries first / before designing tables.
- ~~optimize~~ design for first needs.
- 1 table per query is good strategy, allows for redundancy.
- must be denormalized

CQL - similar to SQL, but JOINS, GROUP BY

Primary key - uniquely identified, no duplicates
- 1st element is Partition key or plus ~~Partition~~ clustering columns
  - Partition key is hashed into numeric
  - data distributed by this partition key
  - simple or composite
  - ?, 1 clustering columns

Primary key (a) - partition key = a
" " (a,b) - " " " is a, clustering key = b
" " ((a,b)) - composite partition = (a,b)
" " (a,b,c) - part key is a, composite clustering key (b,c)
" " ((a,b),c) - composite partition is (a,b), clustering key = c
" " ((a,b),c,d) - " " " (a,b), " " = (c,d)

# 3. Cloud Data Warehouses

## 3.1 Introduction to Data Warehouses

- Operational process - "make it work"
- Analytical processes - "what's going on"

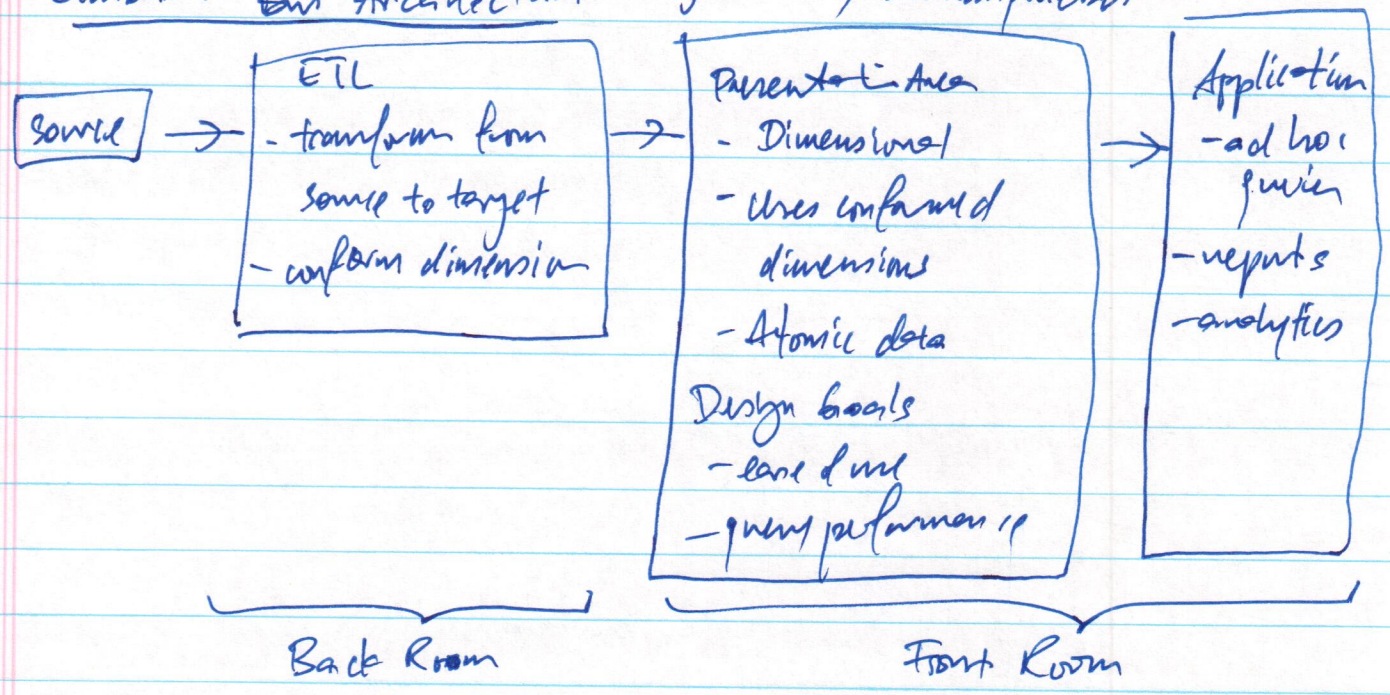

Dimensional Modeling (Recap)

- star schema - good for OLAP, not OLTP
- Normalized Form (3NF)
- Facts, Dimensional tables - dates + time, locations, values
  (facts are numeric + additive (usually)

```
% load_ext sql

% sql select count(*) from table;        ⇒ single line

%%sql
select parameter1, sum(parameter2) as x from table
group by (parameter 1, parameter 2, ...)
order by x desc
limit N;
```

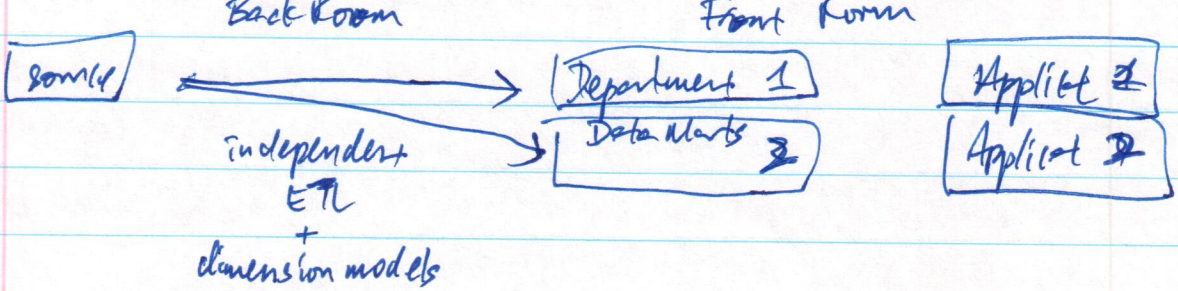
Entity Relational Diagram

## Kimball's Bus Architecture - organized by business process



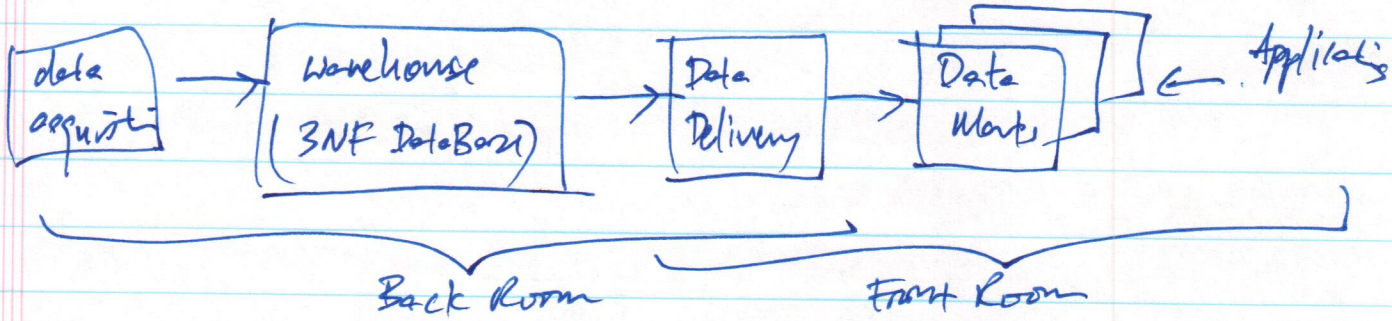

ETL
- extract - get data from sources
- transforming - integrate sources together
  - cleansing
  - diagnostic metadata
- loading - structuring + loading data into dimensional data model

## Independent Data Marts



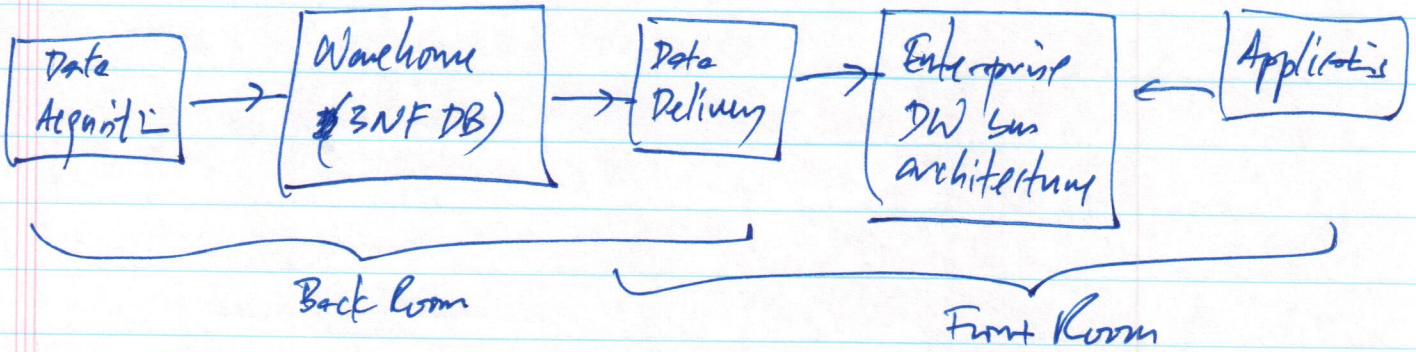

- no conformed dimensions
- uncoordinated efforts → inconsistent views

## Inmon's Coorporate Information Factory (CIF)




- 2 ETL Process
  - sample → 3NF DB
  - 3NF DB → data marts
- 3NF DB enterprise wide data store
  - single source
- Data marts dimensionally modelled (~~similar~~ dissimilar to Kimball's), one aggregated.

## Hybrid Kimball Bus + Inmon CIF



OLAP Cubes - aggregation of a fact metric on a number of dimensions.

Operations:

- Roll-up - sum up
- Drill-down - decompose in smaller composites down to atomic data.
- Slice - fix one dimension to single value.
- Dice - restrict some values of dimensions, same dimension cube output.

SQL - GROUP BY CUBE (x, y, z)

- one pass through fact table, aggregating all combinations of groupings.
  - by x , by x, y , by x, y, z
    by y , by y, x
    by z , by x, z

OLAP cube Technology

1. Pre-aggregate OLAP cubes, save on MOLAP (non relational db)

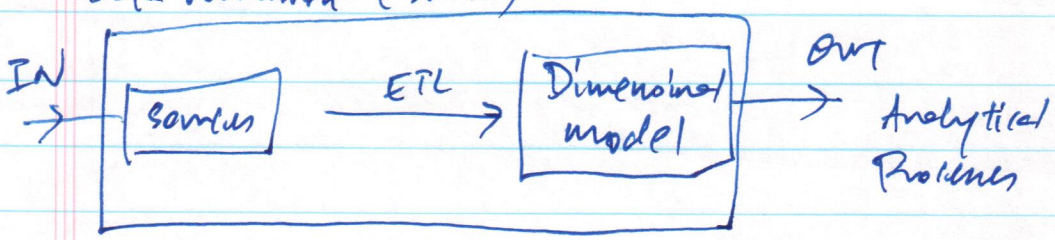
2. compute OLAP cubes on the fly from existing db, (ROLAP)

Postgres - columnar storage extension (cstore_fdw)

- faster than row

## 3.3 Implementing Data Warehouses on AWS

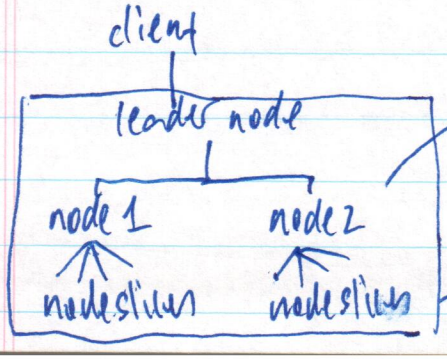
Data Warehouse (DWH)




DWH
- cloud
  - rent hardware
  - scalability + elasticity
  - operation cost on IT
  - Cloud-Managed (AWS)
  - Self-Managed (EC2 + PostgreSQL, Cassandra)
- on-premise
  - scalability, elasticity
  - IT skills
  - cost of hardware

AWS - cloud managed
- SQL → AWS RDS
- NoSQL
- Files
- SQL Columnar → RedShift

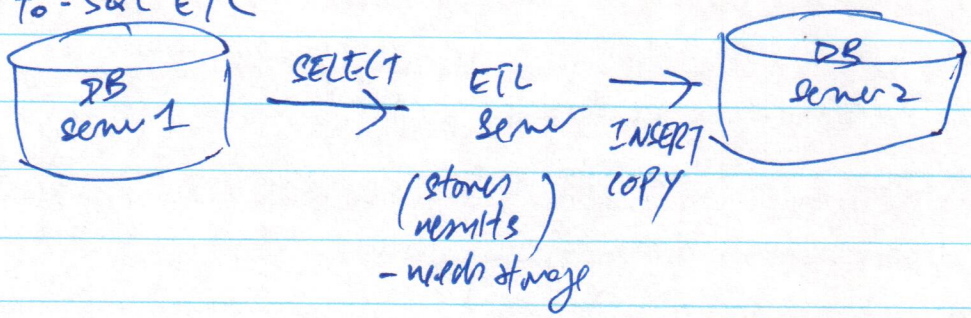
Amazon RedShift
- column-oriented storage
- best for OLAP - summing over long history
- postgres + extension for columnar
- run multiple queries in parallel, 1 query on n CPUs (many)
- massively parallel processing (MPP)

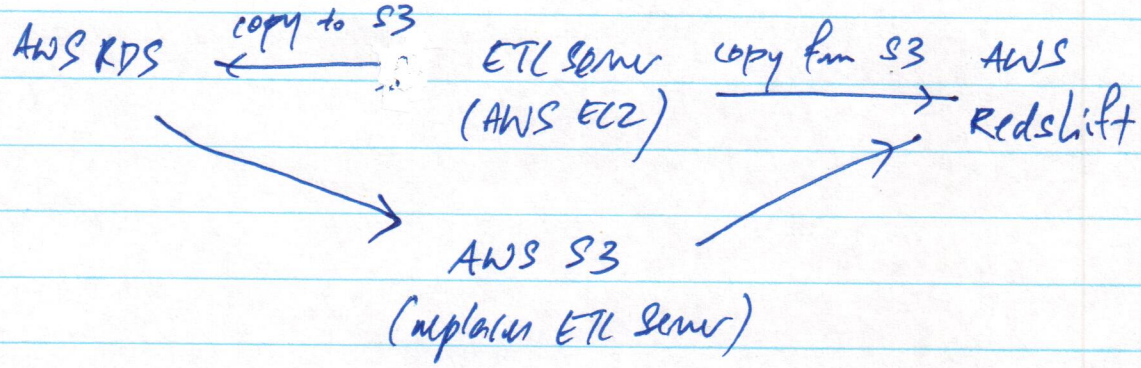


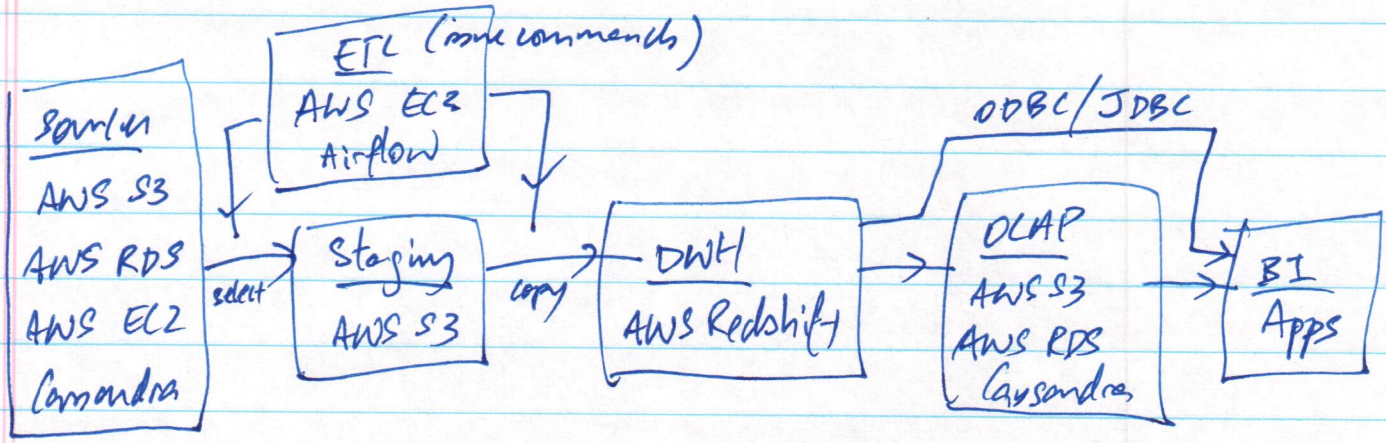

each node - CPU, memory - an instance
scale up - more powerful nodes
scale out - more nodes of same power

n slices can process n partitions of a table simultaneously

SQL-to-SQL ETL




- Transfer data from S3 staging to Redshift with COPY
- INSERT = slow
- If file is large, split in multiple files, ingest in parallel (common prefix or use manifest file)
- compress CSV

```
COPY table FROM 's3://udacity-labs/tpcd'
CREDENTIALS 'aws_iam_role= ...
gzip DELIMITER ';' REGION 'us-west-2';
```

COPY table
FROM 's3://...'
IAM_ROLE 'arn:aws:iam:...'
manifest;

manifest - {
"entries": [
{"url": "s3//....", "mandatory": true}
]
}

ETL out of Redshift
UNLOAD ('select * from table limit 10')
to 's3://...'
iam_role 'arn:aws:iam:....';

---

AWS
- Redshift - IAM - 'myRedshiftRole'
- EC2 service - ~~redshift~~ security group - 'redshift_security_group'
- Redshift - create cluster

- Udacity Data

IAM
- user - airflow_redshift_user - redshiftFullAccess
  - credentials - .csv S3FullAccess

AWS SDK = boto3

Infrastructure As Code

dwh.cfg

[AWS] (IAM user)

KEY =

SECRET =

[DWH] (create cluster)

DWH_CLUSTER_TYPE= multi_node

DWH_NUM_NODE= 4

DWH_NODE_TYPE= dc2.large

DWH_IAM_ROLE_NAME =dwhRole ← DWH_CLUSTER_IDENTIFIER= dwhCluster

DWH_DB = dwh

DWH_DB_USER= dwhuser

DWH_DB_PASSWORD= Passw0rd

DWH_PORT= 5439

- load dwh.cfg
- create clients for EC2, S3, IAM, Redshift using boto3
- create IAM Role - iam.create_role
  Attach Policy - iam.attach_role_policy
- create Redshift cluster - redshift.create_cluster
  - redshift.describe_cluster
- open incoming TCP port to access cluster endpoint
- load_ext sql
  conn_string = `postgresql://{}:{}@{}:{}/{}`.format(DWH_DB_USER, DWH_DB_PASSWORD, DWH_ENDPOINT, DWH_PORT, DWH_DB)
  (↳ faster loading partitioned data (split))
  %sql $conn_string
- delete cluster - redshift.delete_cluster(clusterIdentifier=DWH_CLUSTER_IDENTIFIER, SkipFinalClusterSnapShot=True)

- delete role_policy - iam.detach_role_policy(RoleName = DWH_IAM_ROLE_NAME,
PolicyArn = )

delete role - iam.delete_role(RoleName = DWH_IAM_ROLE_NAME)

Optimize Table Design (over multiple CPUs)

- Distribution Styles - distkey
  - Even
  - all
  - auto
  - key

- Even - load-balancing, round-robin
  - good if a table won't be joined
  - high cost to join

- All - small tables replicated on all slices to speed up joins
  - use frequently for dimension tables
  - AKA 'broadcasting'
  - split only Fact table (even distribute)

- Auto - leave decision to Redshift
  - if small enough, distributed by All
  - if large, distributed by Even

- Key - rows having similar values placed in same slice.
  - skewed distribution if some values more frequent than others
  - useful if ~~fact~~ dimension table is too large for All
  - eliminates shuffling

Sorting key - especially date.
- sortkey

## 4. Data Lakes with Spark

### 4.1 Power of Spark

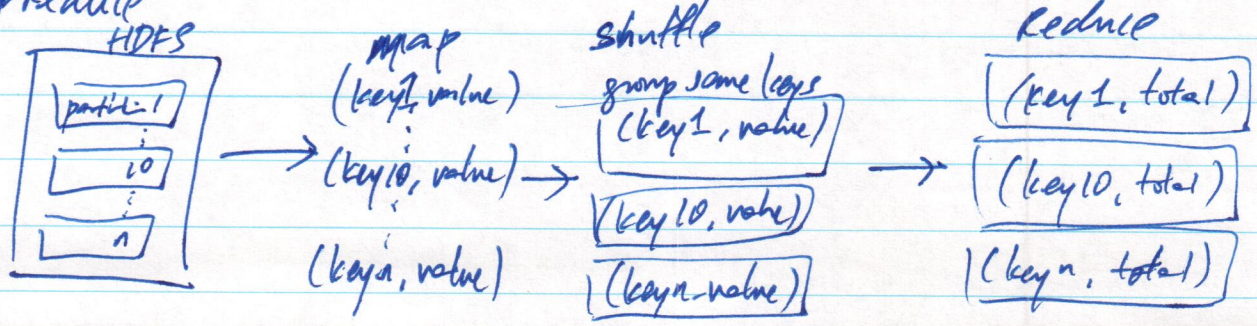
speed decreasing  CPU → memory → SSD → network

distributed cluster - groups of nodes consist of a CPU + memory.
parallel computing - bunch of CPUs share some memory.
Pandas need in all from disk into memory.
└ can read into chunks

Hadoop Framework
- HDFS - data storage
- Map reduce - data processing
- Yarn - resource manager
- hadoop utilities

Apache Spark
Apache Storm - streaming data } in real time, low latency than
Apache Flink - streaming data } Spark streaming.

Map Reduce



Spark tries to do all calculations in memory, avoiding moving data back forth to disk.

Hadoop writes immediate calculations to disk, less efficient, slower.

pip install mrjob

Spark modes – local mode
– cluster manager – standalone
– YARN
\ Mesos
Driver (master to drive)

Don't always need Spark, if using small data that can fit into local computer, use
- AWK
- R
- Python PyData – pandas, matplotlib, numpy, sikit-learn
\ need data in chunks.

If data in relational database, use SQL to extract, filter, aggregate data. e.g SQLAlchemy.

Spark is NOT a data storage system.

## Data Wrangling with Spark

Scala – written, functional language
API – Python, JAVA, R → PySpark

Functional programming - perfect for distributed computing
Spark does not change input data, makes a copy, ie immutable.
Lazy evaluation - build step-by-step direction of what functions + data needed

→ Directed Acyclical Graph (DAG)
- pragmatic running function to get data till ~~every~~ end.
for ~~using~~ processing using 'collect()'

```
import findspark
findspark.init('spark-2.3.2-bin-hadoop2.7')

import pyspark
sc = pyspark.SparkContext(appName='testing')
log_of_songs = []
distributed_log_songs = sc.parallelize(log_of_songs)
distributed_log_songs.map(lambda song: song.lower()).collect()
```

Data Formats
- csv, JSON, HTML, XML,

HDFS - Hadoop Distributed File System/Storage
- splits into 64 or 128 MB blocks
- replicates blocks into clusters
- fault tolerance + digestible chunks
- AWS S3 - store raw data

```
from pyspark import SparkContext, SparkConf
configure = SparkConf().setAppName('name').setMaster('IP Address' ('local'))
sc = SparkContext(conf = configure)
```

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('appName') \
                    .config('config option', 'config value') \
                    .getOrCreate()
spark.SparkContext.getConf().getAll()

user_log = spark.read.json(path)
           spark.read.csv(path, header=true)
```

Imperative Programming — Spark dataframes + Python — how?

vs

Declarative Programming — SQL — what? — abstraction layer above imperative programming.

Spark dataframe
- select()
- filter()
- where()
- groupBy()
- sort()
- withColumn()
- dropDuplicates()

Spark SQL
- user defined functions
- window functions — partitionBy, rangeBetween, rowsBetween

```
user_log.createOrReplaceTempView('user_log_table')
spark.sql('SELECT * FROM user_log_table').show()
spark.udf.register('funct', lambda x: x.lower())
```

Spark query optimizer — catalyst — translate to DAGs from either pyspark or SQL

DAG execution plan on RDD (resilient distributed data set)

Spark Cluster Managers
- Standalone
- MESOS } sharing a common theme
- YARN }

S3 ⟶ EC2 ⟶ local PC
(Spark cluster)

Install Spark individually to each machine, manual, painful, prone to user error.
- AWS EMR (Elastic Map Reduce) does all this automatically

HDFS uses MapReduce as a resource manager for distribution.
Spark lacks this; but is faster.
* Install Spark on HDFS.

Spark Cluster
1. EC2 - key pair → .pem
2. EMR - create cluster
emr-5.20.0 + Spark 2.4.0 + Hadoop 2.8.5
m5.xlarge

spark-submit --master yarn ./lower_songs.py

AWS S3 - object storage system, stores all kinds of format.
HDFS - distributed file system, guarantee fault tolerance via duplication, only format avro and parquet

## 4.4 Debugging + Optimization

_corrupt_record - populates if input data is bad.

```
log_data.where(log_data['_corrupt_record'].isNotNull()).collect()
```

Accumulators - for debugging.

```
incorrect_records = SparkContext.accumulator(0, 0)
incorrect_records.value
def add_incorrect_record():
    global incorrect_records
    incorrect_records += 1
```

Spark WebUI - DAG
Stages
tasks

SSH - port 22
HTML - port 80
Jupyter notebook - port 8888
Spark Jobs - port 4040
WebUI - port 8080

Transformation functions - select, filter → do not show up in tasks
Action functions - write, head → shows up in tasks

```
SparkContext.setLogLevel("ERROR")
                        ("INFO")
```

Data skew - Pareto Principle
- identify by sampling + summarizing (min, max identify bounds roughly)
- change workload division by timestamp or user or ...
- partition data
  - ↓ df.repartition(number of rows)
  - ↘ make composite keys

Out of memory errors, increase number of partitions
Inefficient queries.

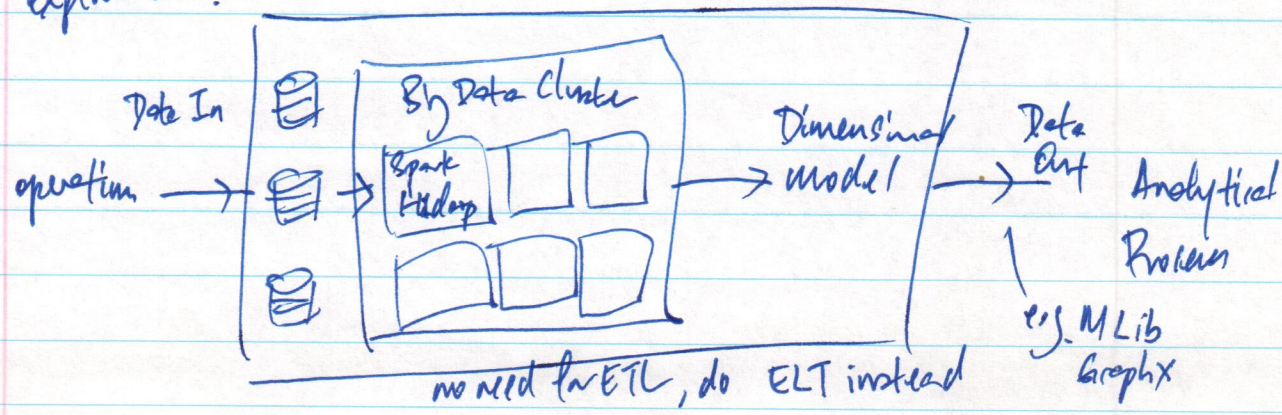
## 4.5 Data Lakes

- unstructured data (text, xml, json, ...)
- volume (IoT, social)
- by data technologies (HDFS, Spark)
- new data analysis (deep learning, ...)

→ harder to define into facts/dimensional tables/json schema-on-read - no creation of tables

→ cheaper with HDFS than MPP

→ freedom to expand beyond existing tables/schema. ad-hoc data exploration.



Schema-On-Read
- skip database with hadoop / Spark, no insertions
- read file immediately

```
spark.read.csv("          .csv",
               inferSchema = True,
               header = True,
               sep = ";",
               mode = "DROPMALFORMED")
```

Un-/Semi-/Structured support
- spark.read.text("... .gz") } text based
  spark.read.csv("..... csv") }
- binary - Avro, Parquet
- compressed

read
-/write to local, HDFS, S3

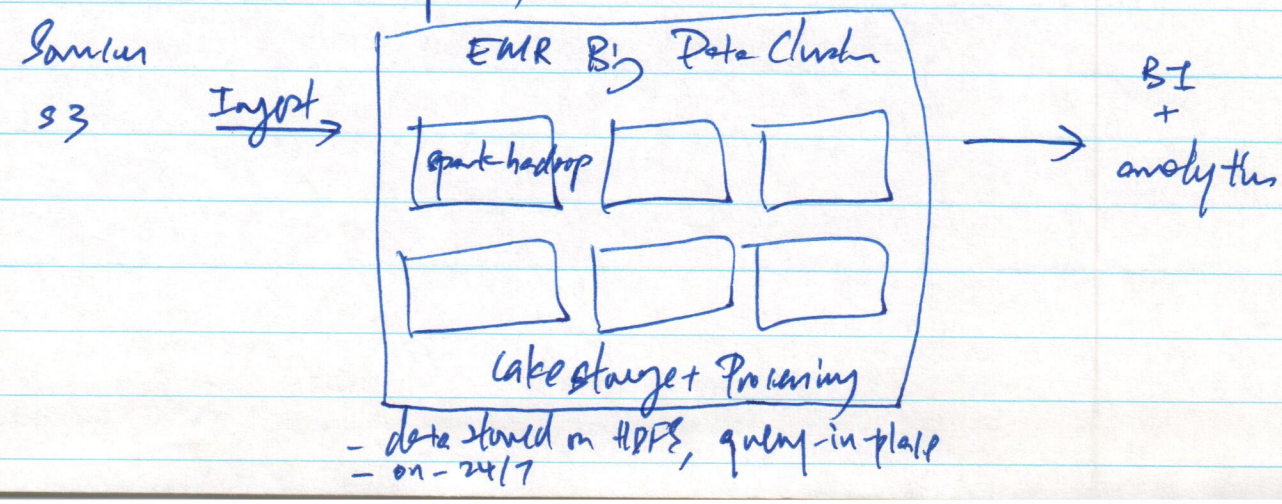
- read/write to databases
  - SQL through JDBC
  - NoSQL - MongoDB, Cassandra,

AWS

| Storage | Processing | AWS-Managed | Vendor-Managed |
|---|---|---|---|
| HDFS | Spark | EMR (HDFS + Spark) | EC2 + Vendor solution |
| S3 | Spark | EMR (Spark only) | EC2 + " |
| S3 | Serverless | Athena | Serverless + " |

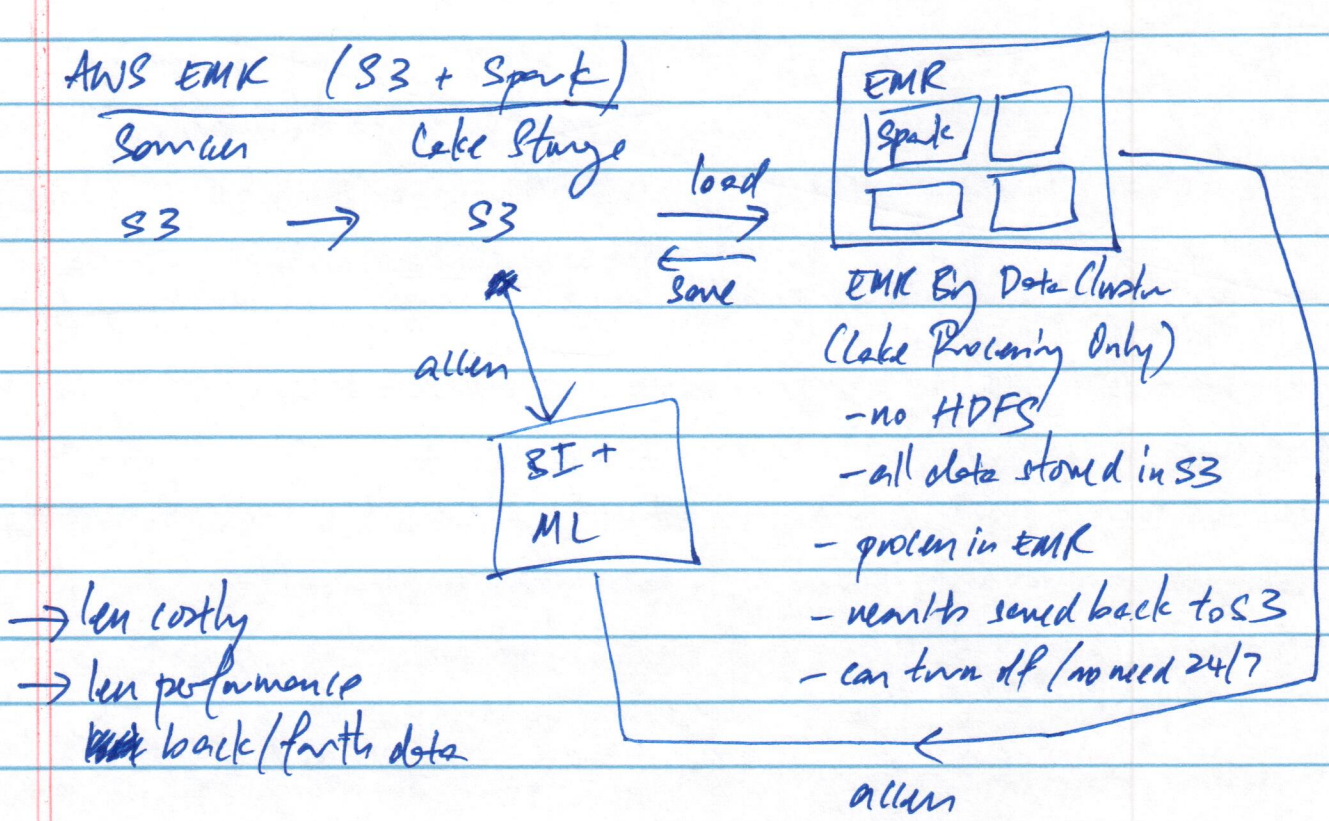
| | Data Warehouse | Data Lake |
|---|---|---|
| Data Form | tabular | all formats |
| Data value | High value | high, medium |
| Ingestion | ETL | ELT |
| Data model | Star, snowflake schema w conformed dimensions & data-marts and OLAP cubes | Star, snowflake, OLTP and-hoc representation |
| Schema | schema-on-write | schema-on-read |
| Technology | Expensive MPP database | Commodity hardware with parallelism |
| Data Quality | High with effort for consistency + clear rules for auditability | Mixed - raw data, some transformed to higher quality |
| Users | Business analyst | Data scientist, business analyst, ML engineer |
| Analytics | Report + BI | Machine learning, graph analytics data exploration. |

AWS EMR (HDFS + Spark)

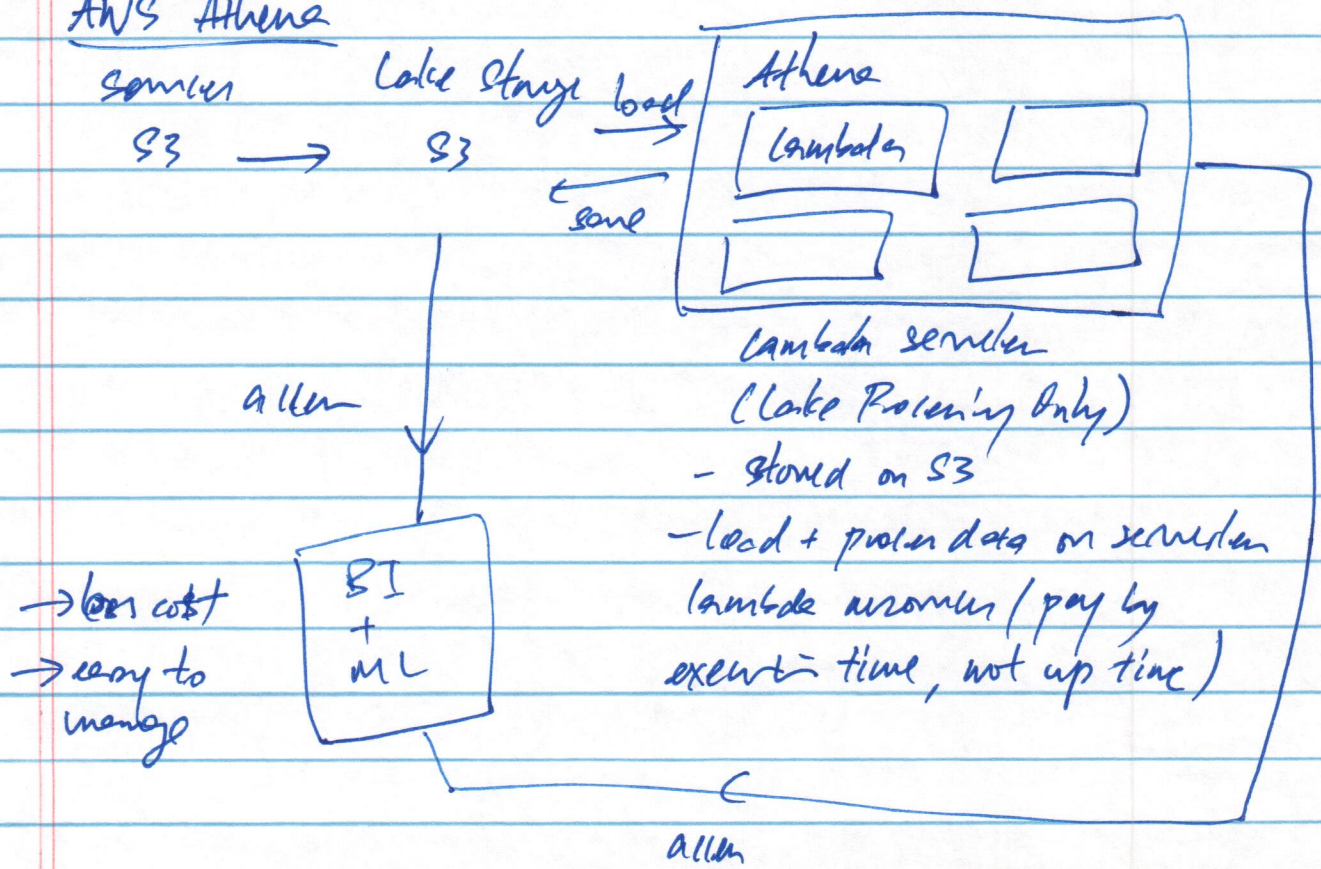



- data stored on HDFS, query-in-place
- on-24/7

## AWS EMR (S3 + Spark)




## AWS Athena



```
import pyspark.sql.functions as F
df.withColumn('payment_date', F.to_timestamp('payment_date'))

from pyspark.sql.types import StructType as R
                              StructField as Fld
                              DoubleType as Dbl
                              StringType as Str
                              IntegerType as Int
                              DateType as Date
paymentSchema = R([Fld('payment_date', Int()),
                   ''
                   ])
spark.read.csv("s3://...", sep=";", schema=paymentSchema,
               header=True)
```

AWS Glue crawler - scheduled ETL
- point to S3 raw data
- schema-on-read

AWS EMR + notebook

Spark cluster on AWS EC2

5. Data Pipelines w/ Airflow

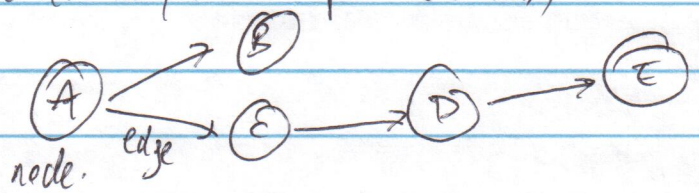
5.1 Data Pipeline
- series of steps in which data is processed.

3V Kafka - open source stream processing software platform for LinkedIn.

Data validation - process of ensuring data is present, correct + meaningful.

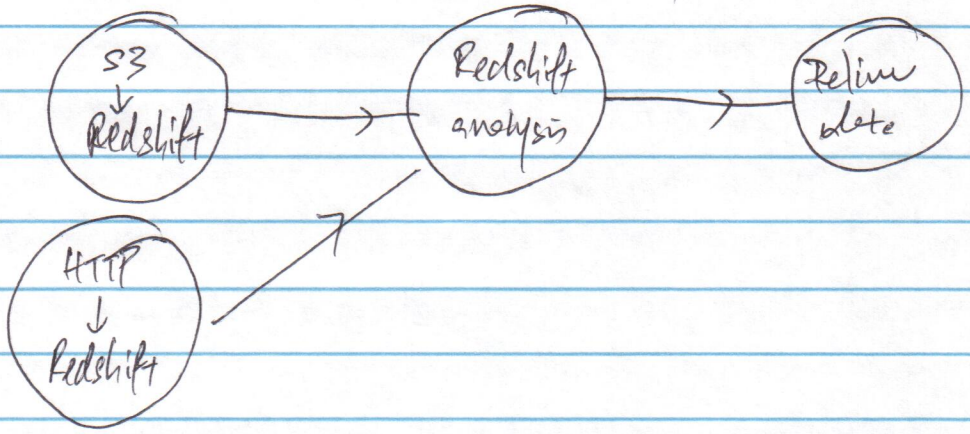
- automated checks.
- valid number of rows in Redshift equals in S3

Directed Acyclic Graphs (DAGs)




- do not contain cycles
- have direction

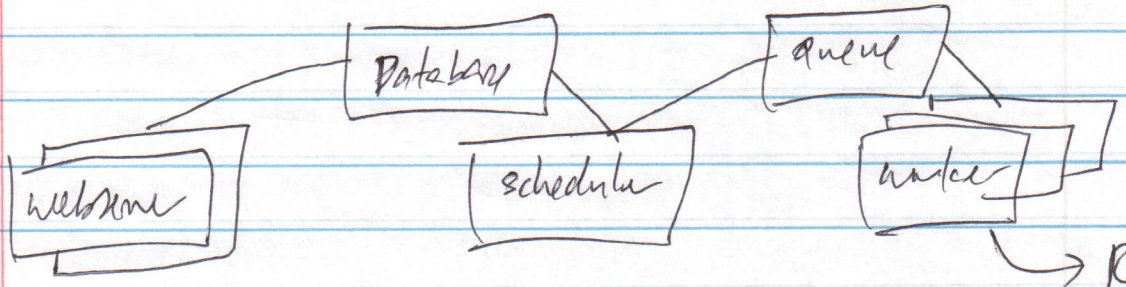

ETL - each step is node
- dependent on prior steps are directed edges




Airflow - open source tool to structure data pipeline as DAGs
- Airbnb in 2015
- DAGs written in Python, run on schedule or from trigger
- integrate with external tools Redshift, Spark, Hadoop etc easily.
- scheduler executes tasks in array of workers

- web-base UI to visualize + interact with data pipelines

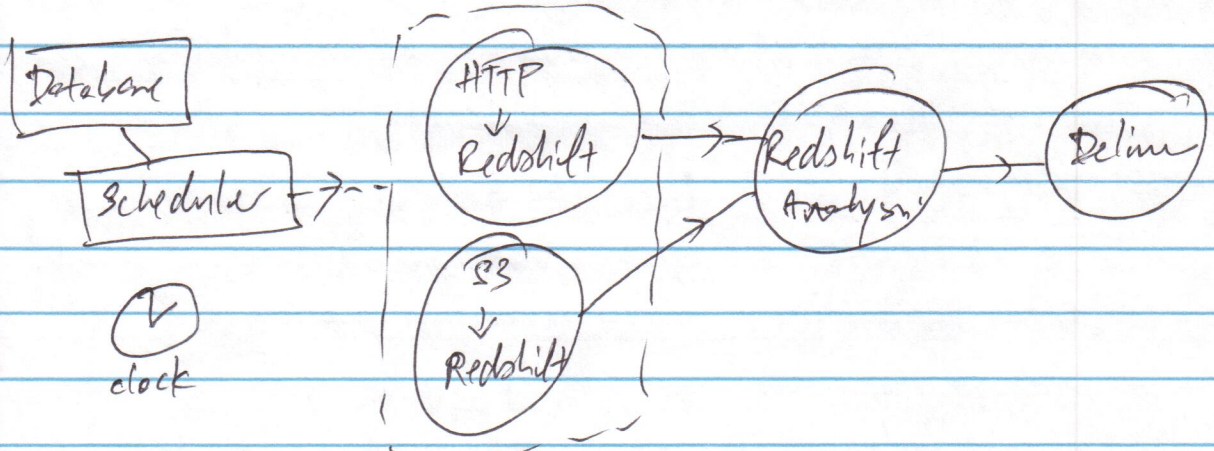
```
import logging
from airflow import DAG
from airflow.operators.python_operator import PythonOperator

def helloworld():
    logging.info("helloworld")

dag = DAG('lesion1.exercise1',
          start_date=datetime.datetime.now())
greet_task = PythonOperator(task_id="hellow_world_task",
                            python_callable=helloworld,
                            dag=dag)
```

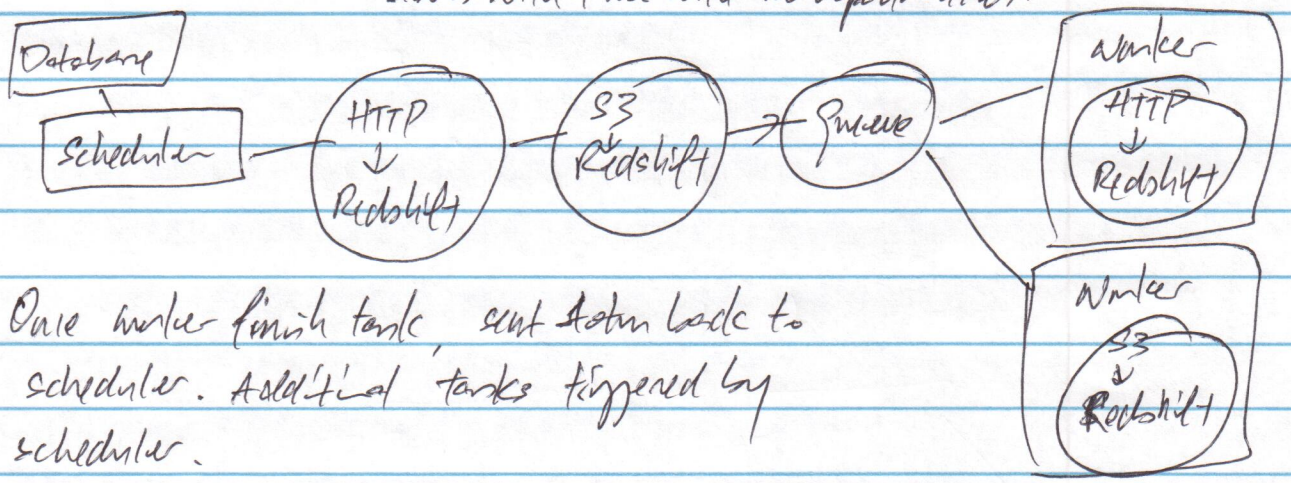



- database - credentials, connections, history
- webserver - control dashboard

- coordinate movement of data between data storage and data processing tools.
- don't pass data in memory, not data processing framework.
- limit to processes in 1 machine.

Airflow scheduler starts DAGs based on time & external triggers.



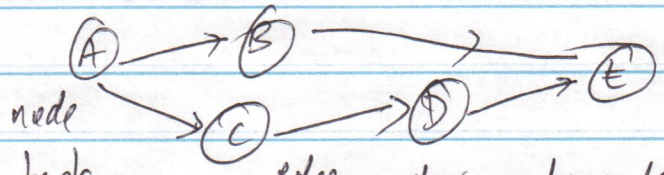

Once worker finish task, sent status back to scheduler. Additional tasks triggered by scheduler.

Creating a DAG

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

DAG → divvy_dag = DAG ('divvy',
                       description = "analyze data ",
                       start_date = datetime(2019, 2, 4),
                       schedule_interval = '@daily')
Operator → task = PythonOperator (task_id = 'hello_world',
                                  python_callable = helloworld,
                                  dag = divvy_dag)
```

Schedules are optional - @once
@hourly
@daily
@weekly
@monthly
@yearly
None
cron strings

Airflow Operators - Python Operator
- Postgres "
- Redshift To S3 "
- S3 To Redshift "
-
- Bash Operator
- Simple Http "
- Sensor
-
-




node
- task

edge - ordering + dependencies between tasks

a >> b  a before b
a << b  a after b

```
hello_world_task = PythonOperator(....)
goodbye_world_task = PythonOperator(....)
hello_world_task >> goodbye_world_task
or " " .set_downstream( " " )
```

Connections via hooks - reusable interface to external systems and databases.

```
eg. db_hook = PostgresHook('demo')
    df = db_hook.get_pandas_df('SELECT * FROM rides')
```

HttpHook — SlackHook
PostgresHook (Redshift) — PrestoHook
MySqlHook

```
hook = S3Hook(aws_conn_id='aws_credentials')
bucket = Variable.get('s3_bucket')
prefix = Variable.get('s3_prefix')

def hello_date(*args, **kwargs):
    print(f"Hello {kwargs['execution_date']}")

divvy_dag = DAG(...)
task = PythonOperator(task_id='hello_date',
                      python_callable=hello_date,
                      provide_context=True,
                      dag=divvy_dag)

def load_data_to_redshift(    ):
    aws_hook = AwsHook('aws_credentials')
    credentials = aws_hook.get_credentials()
    redshift_hook = PostgresHook('redshift')
    redshift_hook.run(sql_statements.COPY_ALL_TRIPS_SQL.
                      format(credentials.access_key, credentials.
                             secret_key))

create_table = PostgresOperator(task_id='create_table',
                                dag=dag, postgres_conn_id='redshift',
                                sql=sql_statements.CREATE
```

```
copy_task = PythonOperator(task_id='load_fact_S3_redshift',
                           dag=dag,
                           python_callable=load_data_to_redshift)
create_tables >> copy_task
```

5.2 Data Quality

Data lineage - discrete steps involved in creation, movement, calculation of dataset.
- instilled confidence
- define metrics
- helps in debugging
- Airflow DAGs are natural representation for transformation + movement of data

e.g. create_table → load_to_redshift → calculati

schedules - based on size of data
- frequency of data
- related datasets

Backfill - retrospectively analyze.

End_dates - end of life

DAG - max_active_runs

Data partitioning - process of isolating data to be analyzed by one or more attributes (time, logical type, data size)
- for speed, more reliable pipelines.

plugins/operators/__init__.py
/s3_to_redshift.py

DAG tasks - atomics, have single purpose.
- maximize parallelism.
- make failure states obvious.

SubDAGs - commonly repeated series of ~~task~~'s within DAGs
- reduce code, debug, easier to understand.
- limits visibility in Airflow UI, ~~task~~ obfuscate

SLA - email alerts
- publish metrics to statsd