Data Engineering with AWS

1 Data Modeling

1.1 Introduction to data modeling

Data Modeling: Conceptual DM → Logical DM → Physical DM
Relational and non-relational models/databases

Relational model

- Database = schema: collection of tables
    - Schema = table name + column names + data types
    - Schema: stable over time
- Table = relation: group of rows sharing the same labeled elements or columns
- Row = tuple: single item
    - Unique key which identifies each row (primary key, surrogate key)
    - State: dynamic, represents aspects that change over time
- Column = attribute: labeled element
    - Each column has a data type
    - Degree of the table = number of columns
- Relational database → RDBMS
- Language SQL
- Constraints

ACID transactions

- Atomicity
    - The whole transaction is processed or nothing is processed
- Consistency
    - Only transactions that abide by constraints and rules are written in the DB otherwise DB keeps previous stage
- Isolation
    - Transactions are processed independently and securely, order does not matter
- Durability
    - Completed transactions are saved to the DB even in cases of system failure

When to use a relational database

- Flexibility (and use of use) for writing in SQL queries

- Modeling the data not modeling queries
- Ability to do JOINS
- Ability to do aggregations and analytics
- Secondary indexes available
- Smaller data volumes
- ACID transactions
- Easier to change to business requirements

When not to use a relational database

- Large amounts of data
    - RDB are not distributed databases → They can only scale vertically
- Need to be able to store different data type formats
    - RDB not designed to handle unstructured data
- Need high throughput / fast reads
    - ACID transactions slow down the process of reading and writing data
- Need a flexible schema
- Need high availability
    - RDB not distributed or coordinator/worker architecture → single point of failure
    - High availability describe a DB where there is very little downtime of the system, it is always on and functioning
- Need horizontal scalability
    - Ability to add servers/machines/nodes to the system to increase performance and space for data
- Users are distributed

NoSQL databases

- Created to handle the limitations that exist with relational databases
- Build for Big Data / very low latency
- Some examples and implementations of NoSQL DB:
    - Apache Cassandra (partition row store)
    - MongoDB (document store)
    - DynamoDB (key-value store)
    - Apache HBase (side columns store)
    - Neo4j (graph database)

When not to use a NoSQL database

- Have small dataset
- Need ACID transactions
- Need ability to do JOINS

    - Not allowed in NoSQL DB as this will result in full table scans
- Ability to do aggregations and analytics
- Have changing business requirements
- Queries are not available in advance and need to have flexibility

NoSQL DB and RDB do not replace each other for all tasks → should be utilized for the use cases they fit well.

1.2 Relational data models

Lesson outline

- Normalization / denormalization
- Fact / dimension tables
- Different schema models

The importance of relational databases

Codd's 12 rules (Wikipedia)

Rule 1: the information rule

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

- Standardization of data model
  - Once your data is transformed into the rows and columns format, your data is standardized and you can query it with SQL
- Flexibility in adding and altering tables
- Data integrity
- Structured Query Language (SQL)
- Simplicity
- Intuitive organization

OLAP vs OLTP

- **Online Analytical Processing (OLAP)**
  - Databases optimized for these workloads allow for complex analytical and ad hoc queries, including aggregations.
  - Optimized for reads.
  - Example: total stock of shoes a particular store sold.
- **Online Transactional Processing (OLTP)**
  - Databases optimized for these workloads allow for less complex queries in large volumes.
  - Types of queries for these databases are read, insert, update, and delete.
  - Example: get the price of a particular shoe.

Structuring the database

- Normalization

    - The process of structuring a relational database in accordance with a series of normal forms in order to reduce data redundancy and increase data integrity.
    - It organizes the columns and tables in a database to ensure that their dependencies are properly enforced by database integrity constraints.
    - We don't want or need extra copies of our data (data redundancy).
    - We want to be able to update data in one place and have that be the source of truth (data integrity).
- Denormalization
    - Must be done in read heavy workloads to increase performance.

Objectives of Normal Form

- To free the database from unwanted insertions, updates and deletion dependencies.
- To reduce the need for refactoring the database as new types of data are introduced.
- To make the relational model more informative to users.
- To make the database neutral to the query statistics, where these statistics are liable to change as time goes by.

Normal Forms

The process of normalization is a step by step process:

1. How to reach **First Normal Form (1NF)**
    a. Atomic values: each cell contains unique and single values.
    b. Be able to add data without altering tables (without adding columns).
    c. Separate different relations into different tables.
    d. Keep relationships between tables together with foreign keys.

2. How to reach **Second Normal Form (2NF)**
    a. Have reached 1NF.
    b. All columns in the table must rely on the Primary Key.
        i. I shouldn't need two elements (composed key) to get a third.
        ii. If the column depends on multiple columns it may need its own table.

3. How to reach **Third Normal Form (3NF)**
    a. Have reached 2NF.
    b. No transitive dependencies.
        i. To get from A-> C, avoid going through B.
        ii. When you want to update data, we want to be able to do it in just 1 place.

Denormalization

Process of trying to improve the read performance of a database at the expense of losing some write performance by adding redundant copies of data.

- JOINS on the database allow for outstanding flexibility but are extremely slow.
- If you are dealing with heavy reads on your database, you may want to think about denormalizing your tables.
- You get your data into normalized form, and then you proceed with denormalization. So, denormalization comes after normalization.
- Requires more space in the system (as there will be more copies of the data).

Logical design change:

- The designer is in charge of keeping data consistent.
- Reads will be faster (SELECT).
- Writes will be slower (INSERT, UPDATE, DELETE).

Normalization vs denormalization

- Normalization
    - Try to **increase data integrity** by reducing the number of copies of the data.
    - Data that needs to be added or updated will be done in as few places as possible.
- Denormalization
    - Try to **increase performance** by reducing the number of joins between tables (as joins can be slow).
    - Data integrity will take a bit of a potential hit, as there will be more copies of the data (to reduce JOINS).

Fact and dimension tables

- They work together to create an organized data model.
- They differ in concept, not in the way we create them.
- They'll help us understand star and snowflake schemas.
- **Fact tables** (Wikipedia)
    - Consist of measurements, metrics or facts / events of a business process.
    - Normally have ints or numbers.
    - A fact table typically has two types of columns: those that contain facts and those that are a foreign key to dimension tables.
- **Dimensions** (Wikipedia)
    - Structures that categorize facts and measures in order to enable users to answer business questions.
    - People, products, place, time.
    - Normally, textual information or numbers that are not used for analysis, like year or date.

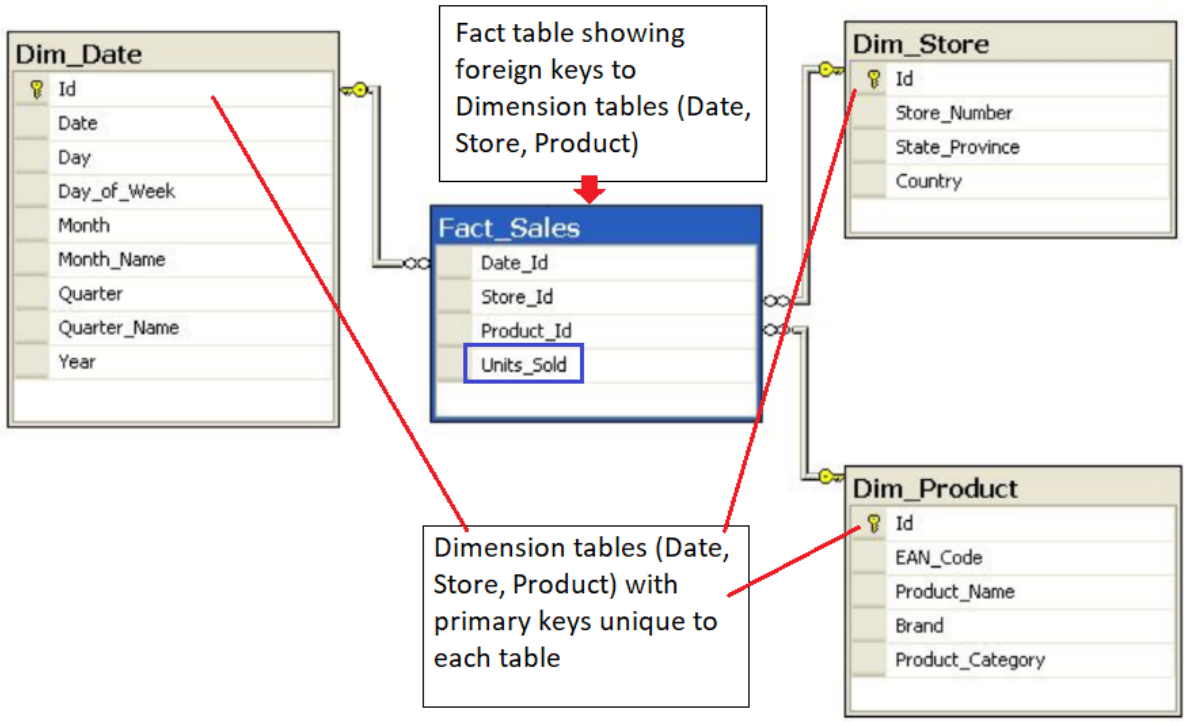
- Each dimension table will have one or more fact tables joined together with a foreign key.

In the example, it helps to think about the **dimension tables** providing the following information:

- **When** was the product bought? (Dim_Date table).
- **Where** was the product bought? (Dim_Store table).
- **What** product was bought? (Dim_Product table).

The **fact table** provides the **metric of the business process** (here Sales).

- **How many units** of products were bought? (Fact_Sales table).



Entity Relationhip Diagrams (ERD) (Wikipedia).

Implementing different schemas

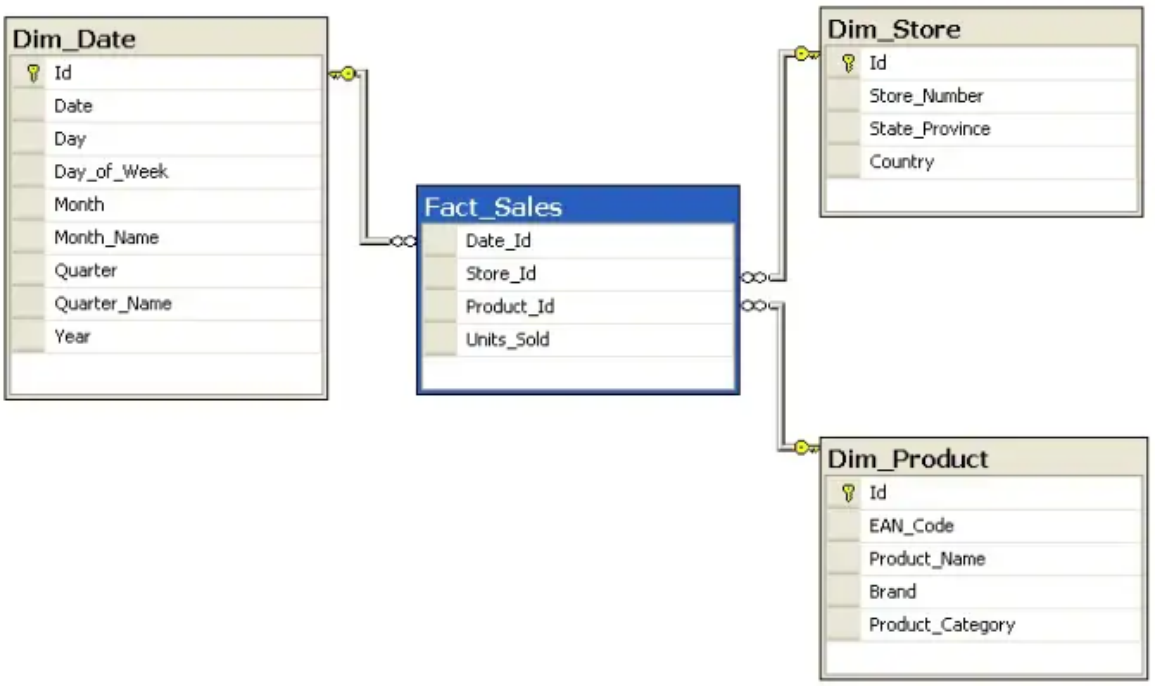
Two of the most popular data mart schema for data warehouses are:

- Star schema (Wikipedia)
- Snowflake schema (Wikipedia)

Interesting article on Medium.

Star schema

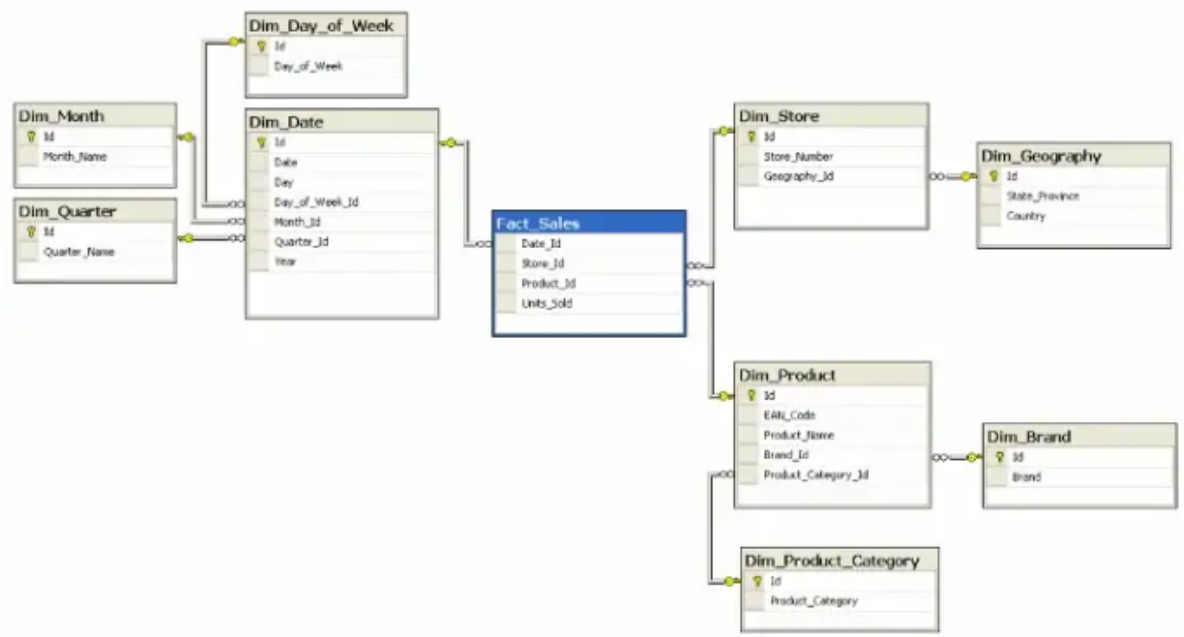
- Simplest style of data mart schema.
- Consists of one or more fact tables referencing any number of dimension tables.
- Why “star”?
    - Gets its name from the physical model resembling a star shape.
    - A fact table is at its center.
    - Dimension table surrounds the fact table representing the star points.
- **Benefits** of star schema:
    - Denormalized tables.
    - Simplifies queries:
        - Getting a table into 3NF is a lot of hard work, JOINs can be complex even on simple data.
        - Star schema allows for the relaxation of these rules and makes queries easier with simple JOINS.
    - Fast aggregations:
        - Aggregations perform calculations and clustering of our data so that we do not have to do that work in our application.
        - Examples : COUNT, GROUP BY, etc
- **Drawbacks** of star schema:
    - Issues that come with denormalization:
        - Data integrity.
        - Decrease query flexibility.
    - Many to many relationship - simplified.



Snowflake schema

- Logical arrangement of tables in a multidimensional database represented by centralized fact tables which are connected to multiple dimensions.

- Star Schema is a special, simplified case of the snowflake schema.
- Snowflake allows for one to many relationships.
- Snowflake is more normalized than star schema, but only in 1NF or 2NF.



1.3 NoSQL data models

Lesson outline

Fundamentals of data modeling for NoSQL databases.

- Basics of NoSQL database design.
- Denormalization.
- Primary keys.
- Clustering columns.
- The WHERE clause.

Non relational databases

NoSQL / Non relational = interchangeable terms.
NoSQL = Not Only SQL

When to use NoSQL:

- Need high availability in the data.
- Have large amounts of data.
- Need linear scalability.
- Need low latency.
- Need fast reads and writes.

Apache Cassandra

- Open source NoSQL DB.
- Masterless architecture.
- High availability.
- Linearly scalable.
- Used by Uber, Netflix, Hulu, Twitter, Facebook, etc.
- In Apache Cassandra every node is connected to every node -- it's peer to peer database architecture (Cassandra architecture).
    - All the nodes in a cluster play the same role. Each node is independent and at the same time interconnected to other nodes.
    - Each node in a cluster can accept read and write requests, regardless of where the data is actually located in the cluster.
    - When a node goes down, read/write requests can be served from other nodes in the network.

Basics of NoSQL database design

Distributed database

- Database that has been scaled out horizontally. It's not a single system, but a DB made up of multiple machines.
- High availability: my system is always up and there is no or very little downtime

- In a distributed DB, in order to have high availability, you will need copies of your data (because nodes will go down).
- Since the data is copied throughout the system, the data may not be up to date in all locations → eventual consistency.

Eventual consistency (Wikipedia)
A consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.
Over time (if no new changes are made) each copy of the data will be the same, but if there are new changes, the data may be different in different locations. The data may be inconsistent for only milliseconds. There are workarounds in place to prevent getting stale data.

CAP theorem

A theorem in Computer Science (Wikipedia) that states it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees of consistency, availability and partition tolerance.

- **Consistency**
  - Every read from the database gets the latest (and correct) piece of data or an error.
- **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.

If the system is running fine (no network failures) you can also have availability and consistency.
But when the system has network failures, then you may only have consistency OR availability.
Apache Cassandra and other NoSQL databases choose to be **highly available at potential cost of consistency** → AP (Availability and Partition tolerance) database.

The CAP Theorem

Consistency
Every read from the database gets the latest (and correct) piece of data or an error

Availability
Every requests is received and a response is given -- without a guarantee that the data is the latest update

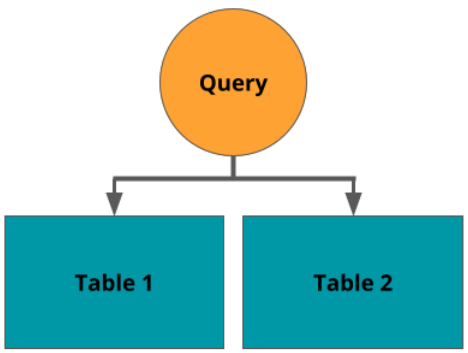
Partition Tolerance
The system continues to work regardless of losing network connectivity between nodes.

Data modeling in Apache Cassandra

- Denormalization in Apache Cassandra is a must.
- Denormalization must be done for fast reads.
- Apache Cassandra has been optimized for fast writes.
- Apache Cassandra does not allow for JOINs between tables.
- ALWAYS think before data modeling what queries will be performed on your data.
    - Model your data to your queries.
    - If your business needs calls for quickly changing requirements, you need to create a new table to process the data.
- Great strategy: one table per query.
    - This will lead to data replication (redundant data).
    - But the performance benefits and high availability of the system far outweigh any additional storage costs.

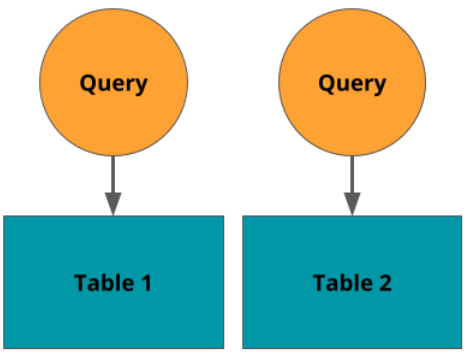
[DataStax Apache Cassandra data modeling concepts]

Relational Databases



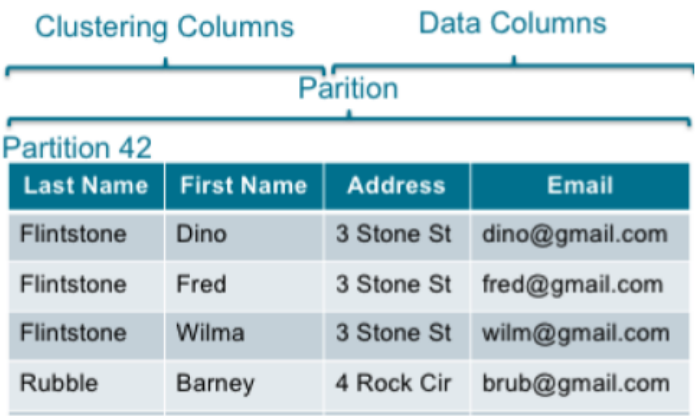
In a relational database, one query can access and join data from multiple tables

NoSQL Databases



In Apache Cassandra, you cannot join data, queries can only access data from one table

- Partition
  - Fundamental unit of access
  - Collection of row(s)
  - How data is distributed
- Primary Key
  - Primary key is made up of a partition key and clustering columns
- Columns
  - Clustering and Data
  - Labeled element



Last Name	First Name	Address	Email
Flintstone	Dino	3 Stone St	dino@gmail.com
Flintstone	Fred	3 Stone St	fred@gmail.com
Flintstone	Wilma	3 Stone St	wilm@gmail.com
Rubble	Barney	4 Rock Cir	brub@gmail.com

Cassandra Query Language (CQL)

- Very similar to SQL.
- The following are NOT supported by CQL:
  - JOIN.
  - GROUP BY.
  - Subqueries.

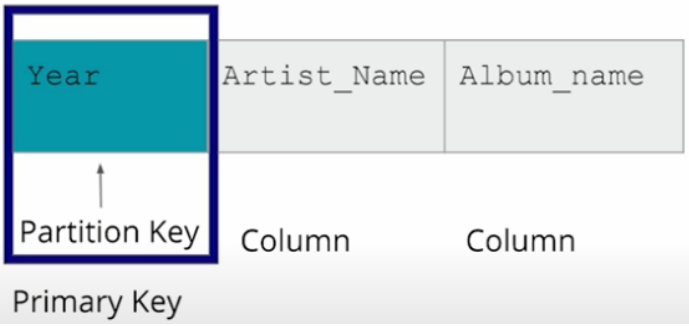
Primary Key

- How each row is uniquely identified and how the data is distributed across the nodes or servers in our system.
- Must be unique to each row.
- Is made up of either just the PARTITION KEY or may also include additional CLUSTERING COLUMNS.
  - A Simple PRIMARY KEY is just one column that is also the PARTITION KEY.
  - A Composite PRIMARY KEY is made up of more than one column and will assist in creating a unique value and in your retrieval queries.
- The PARTITION KEY will determine the distribution of data across the system.
  - The partition key row value will be hashed (turned into a number) and stored on the node in the system that holds that range of values.

[DataStax - Primary Keys]

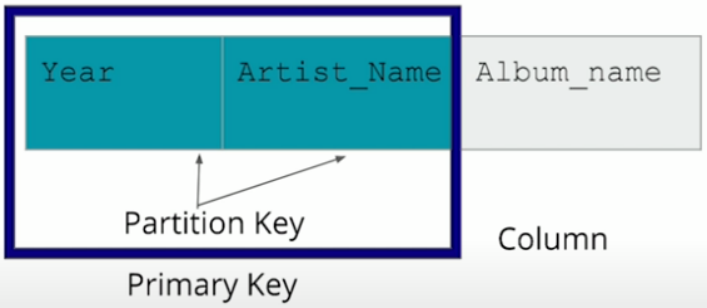
Primary Key Simple

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



Primary Key Composite

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



Clustering columns

- Will determine the sort order within a partition (sorted ascending order, e.g., alphabetical order).
- More than one clustering column can be added, or none.
- From there the clustering columns will sort in order of how they were added to the primary key.
- You cannot use the clustering columns out of order in the SELECT statement. You may choose to omit using a clustering column in your SELECT statement. That's OK. Just remember to use them in order when you are using the SELECT statement.

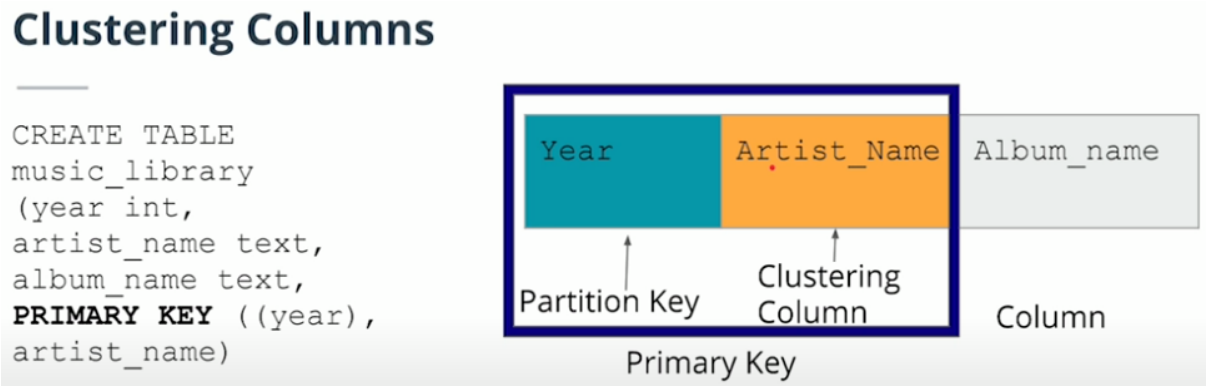
[DataStax - Compound Primary Key]
[DataStax - Clustering columns]

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

```
cqlsh:udacity> select * from music_library;

 year | artist_name | album_name
------+-------------+-----------------
 1965 |       Elvis |      Blue Hawaii
 1965 | The Beatles |      Rubber Soul
 1965 | The Beatles |    Showing order
 1965 | The Monkees | Meet the Monkees

(4 rows)
cqlsh:udacity>
```



WHERE clause

- Data Modeling in Apache Cassandra is query focused, and that focus needs to be on the WHERE clause.
- The PARTITION KEY must be included in our query, and any CLUSTERING COLUMNS can be used in the order they appear in our PRIMARY KEY.
- Failure to include a WHERE clause will result in an error.
- It is recommended that one partition be queried at a time for performance implications.
- It is possible to do a "SELECT * FROM table" if we add a configuration ALLOW FILTERING to our query. This is risky, but available if absolutely necessary (DataStax - ALLOW FILTERING).

Why do we need to use a WHERE statement since we are not concerned about analytics? Is it only for debugging purposes?
The WHERE statement is allowing us to do the fast reads. With Apache Cassandra, we are talking about big data -- think terabytes of data -- so we are making it fast for read purposes. Data is spread across all the nodes. By using the WHERE statement, we know which node to go to, from which node to get that data and serve it back. For example, imagine we have 10 years of data on 10 nodes or servers. So 1 year's data is on a separate node. By using the WHERE year = 1 statement we know which node to visit fast to pull the data from.

Other useful links about Apache Cassandra

Cassandra best practices
Reasons to use Cassandra

2 Cloud Data Warehouses

2.1 Course outline

Data Warehouses

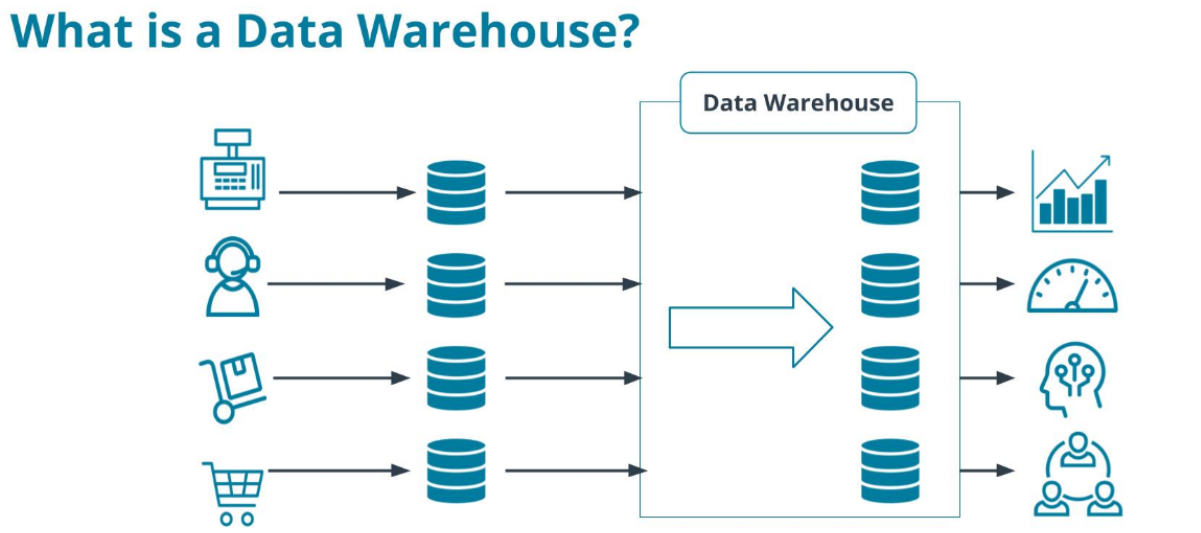
- The business case for data warehouses.
- Data warehouse architecture.
- Dimensional modeling for data warehouses.
- SQL to SQL Extract, Transform, and Load (ETL) for data warehouses
- OLAP Cubes

Cloud data warehouses

- Extract, Transform, and Load (ETL) vs Extract, Load and Transform (ELT).
- Managed database services in the cloud.
- Cloud storage and ETL pipeline services.
- Cloud data warehouse solutions.

Amazon Web Services (AWS) data services

- Amazon S3 (Simple Storage Service).
- AWS Redshift.
- Python scripts that interact with AWS data tools using the AWS BOTO3 SDK.



Data Warehouse is a system (including processes, technologies & data representations) that enables us to support analytical processes

2.2 Introduction to Data Warehouses

Lesson outline

- Business needs for Data Analytics:
    - Business perspective.
    - Technical perspective.
    - OLTP.
    - OLAP.
- Data Warehouse design:
    - Dimensional modeling.
    - ETL processes for data warehousing.
- Data Warehouse architecture:
    - Kimball's architecture.
    - DW components.
    - Processes of each DW component.
    - Storage architecture components for optimization.
- OLAP cubes:
    - Data dimensions and analysis.
    - OLAP cube operations.

Operational vs Analytical Business Processes

Operational processes: Make it work.

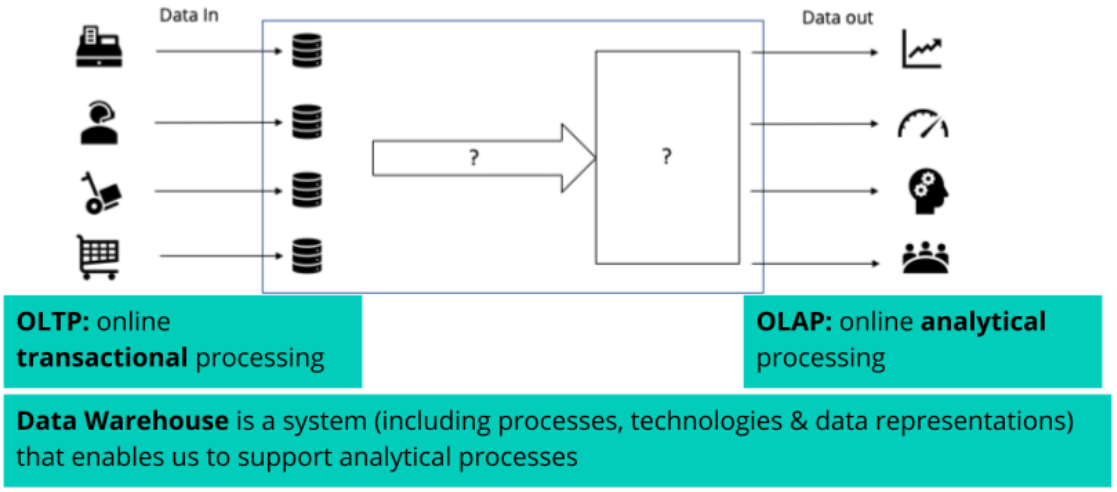
- Find goods & make orders (for customers).
- Stock and find goods (for inventory staff).
- Pick up & deliver goods (for delivery staff).

Analytical processes: What is going on?

- Assess the performance of sales staff (for HR).
- See the effect of different sales channels (for marketing).
- Monitor sales growth (for management).

OLTP	**OLAP**
Scale	Flexibility
3rd Normal Form	Understandable
Performance	Aggregated
Relational	Dimensional

OLTP and OLAP



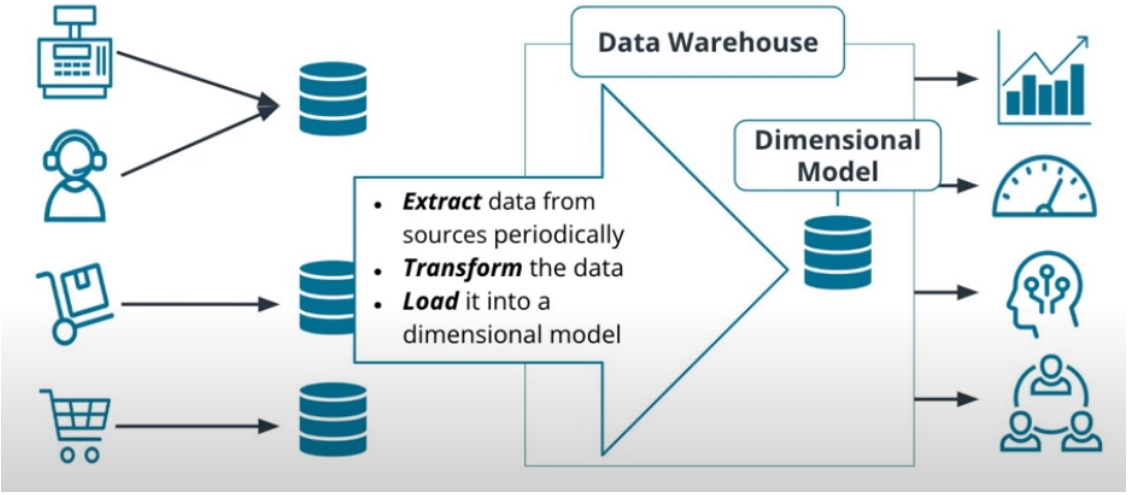
Data Warehouse architecture

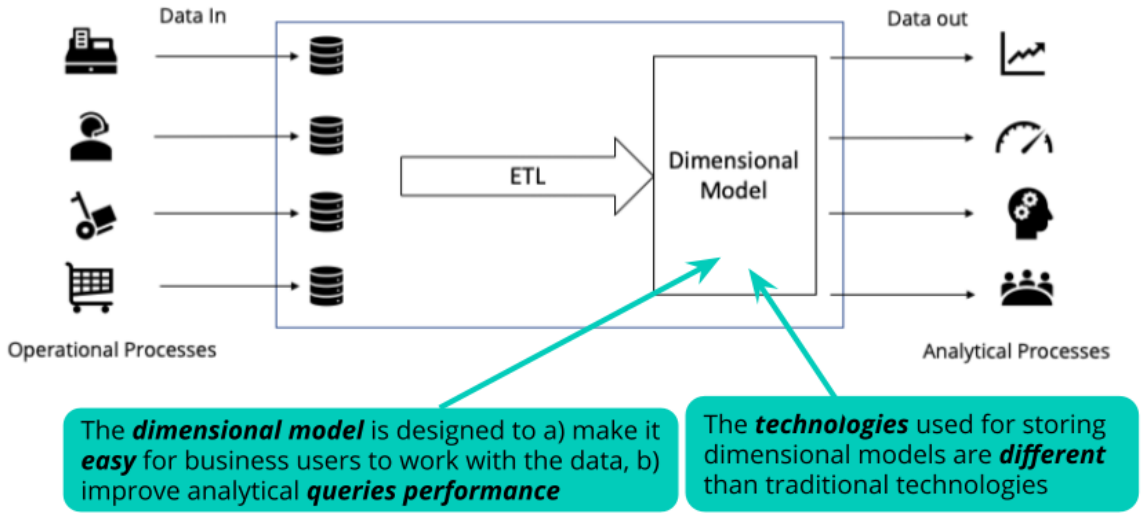
Data Warehouse design: copy data
"A data warehouse is a copy of transaction data specifically structured for query and analysis", Ralph Kimball.

Data Warehouse design: batch updating
A DW retrieves and consolidates data periodically.
"A data warehouse is a subject-oriented, integrated, nonvolatile, and time-variant collection of data in support of management's decisions", Bill Inmon

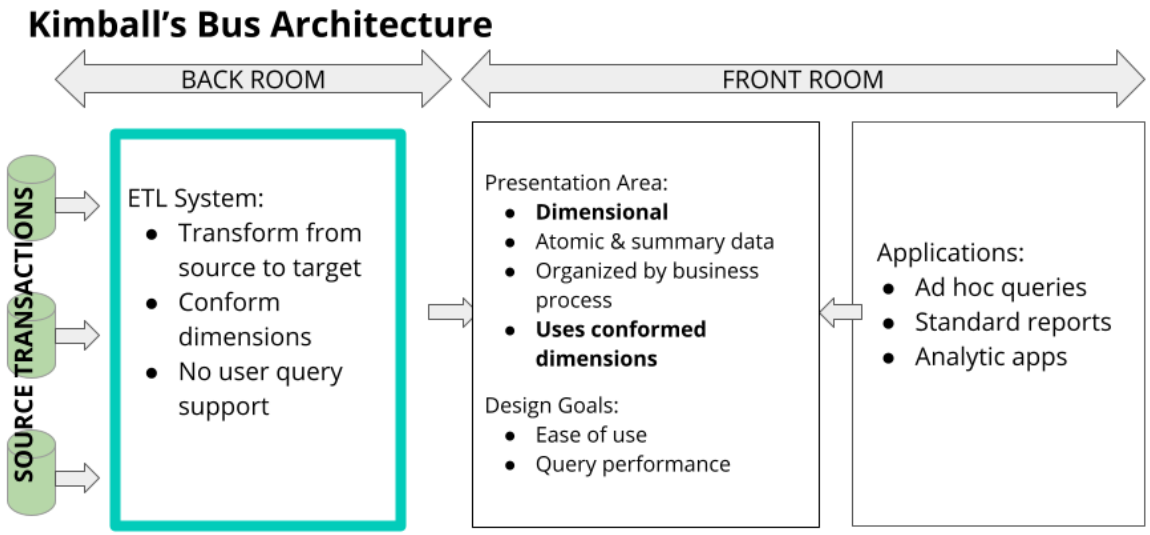
Data Warehouse design: dimensional model
The dimensional model is optimized for analytical query performance.

A data warehouse 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.





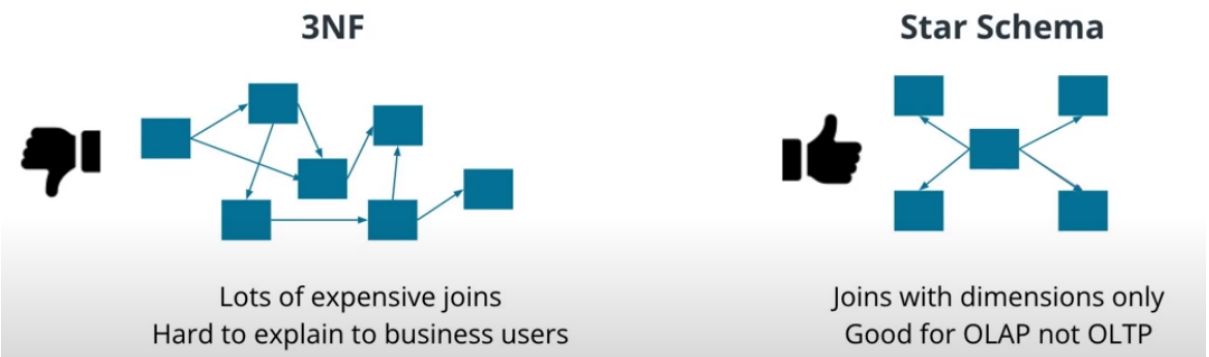
Kimball's Bus Architecture



ETL and Dimensional Modeling for Data Warehouse

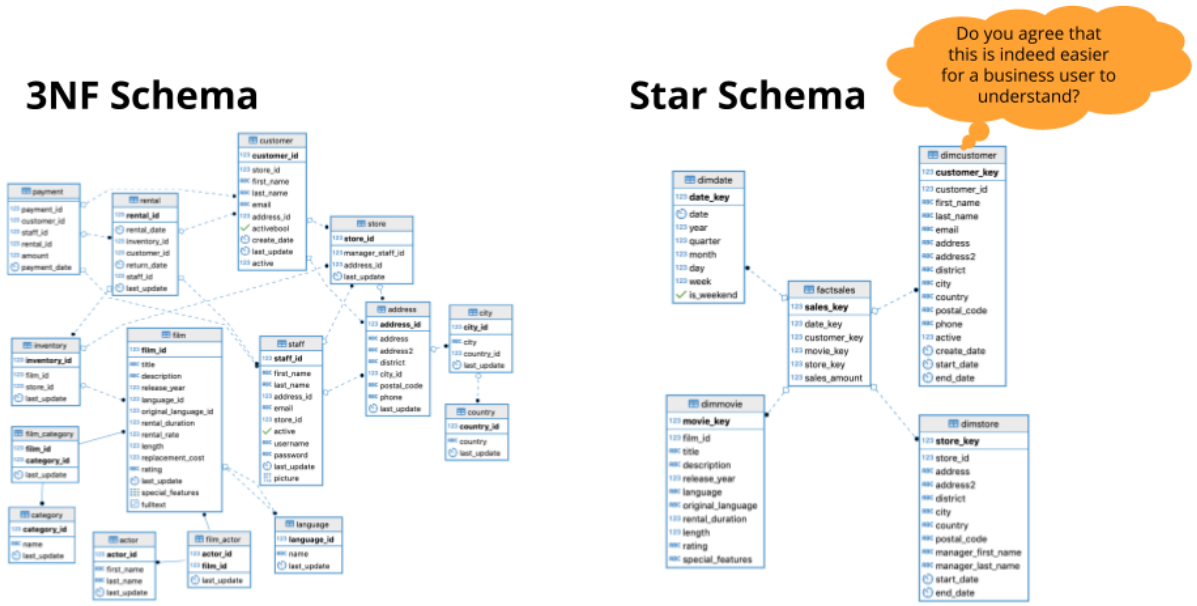
Dimensional Model review

- Dimensional modeling (Star schema) goals:
  - Easy to understand
  - Fast analytical query performance



- Fact tables:
  - Record business events, like an order, a phone call, a book review.
  - Fact tables columns record events recorded in quantifiable metrics like quantity of an item, duration of a call, a book rating.
- Dimension tables:
  - Record the context of the business events, e.g. who, what, where, why, etc.
  - Dimension tables columns contain attributes like the store at which an item is purchased or the customer who made the call, etc.

3rd Normal Form Schema vs Star Schema



ETL: a closer look

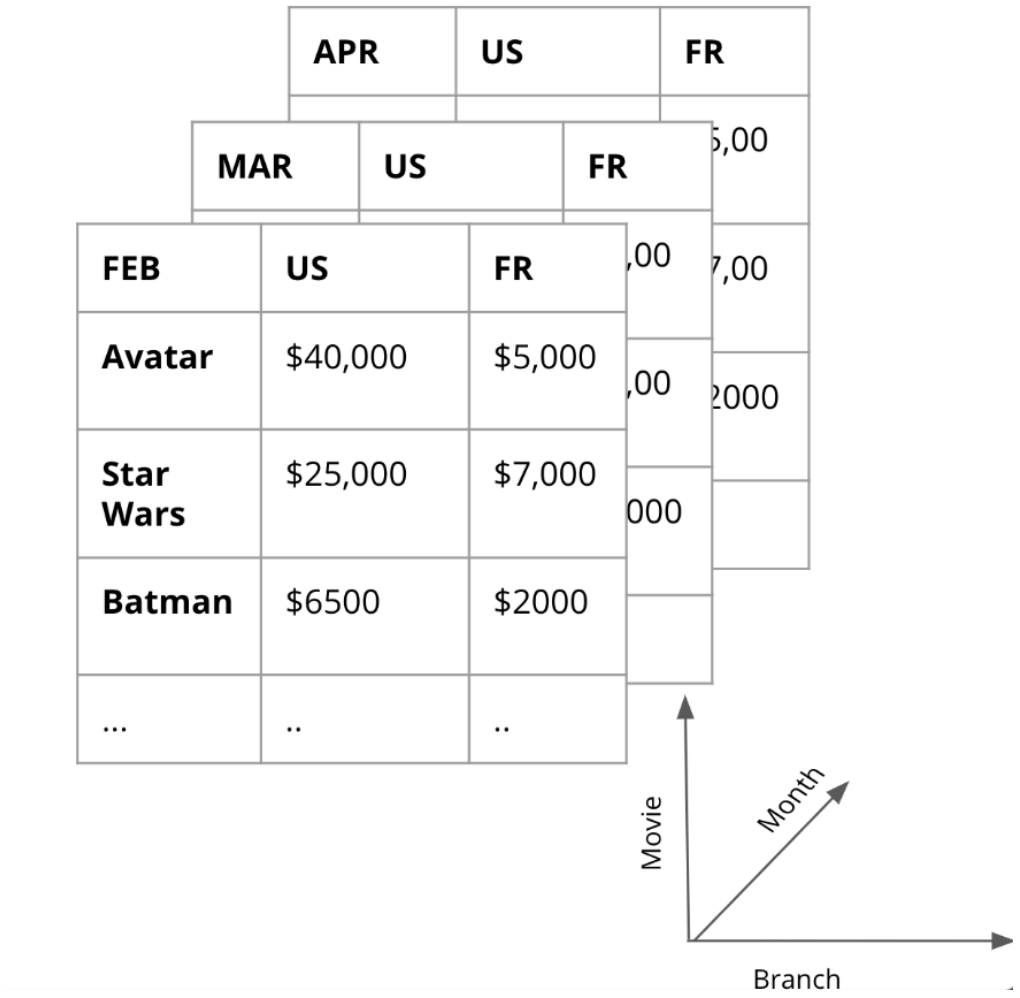
- Extract
  - Transfer data to the warehouse.
- Transform
  - Integrates many sources together.
  - Possibly cleansing: inconsistencies, duplication, missing values, etc.
  - Possibly producing diagnostic metadata.
- Load
  - Structuring and loading the data into the dimensional data model.

OLAP Cubes

- OLAP Cubes are queries that return multiple dimensions of data in a fact and dimensional dataset.
- An OLAP Cube is an aggregation of a fact metric on a number of dimensions.
- It's a model for querying data that is easy to communicate to business users.

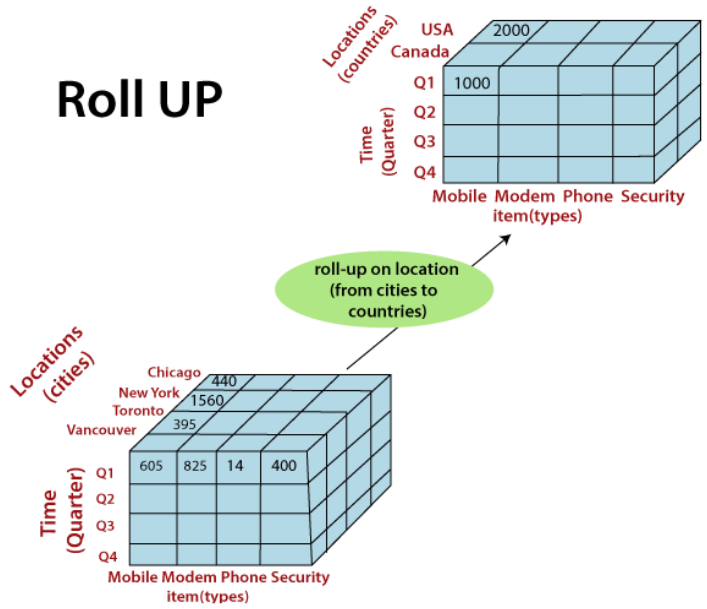
Example:

Dimensions: movie, month, branch.

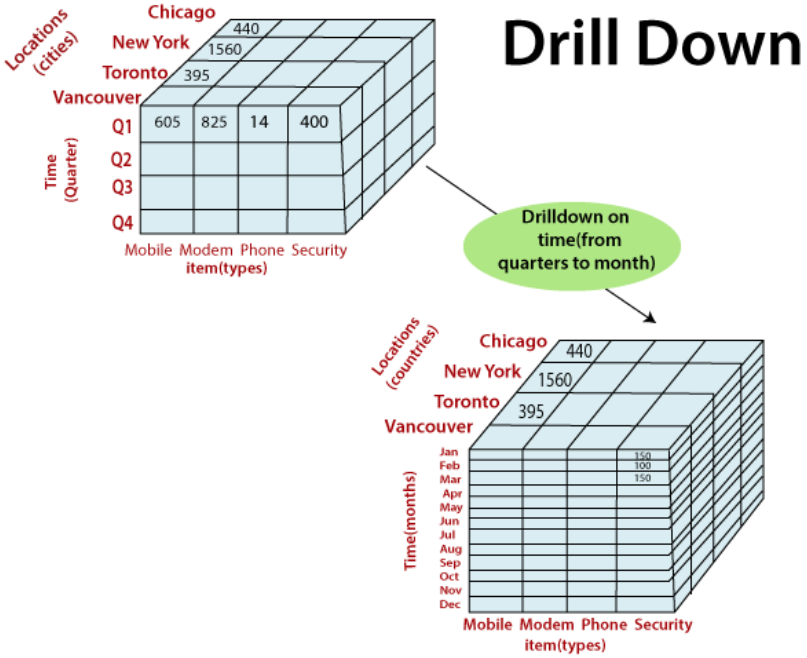


OLAP Cube operations

- **Roll-Up**
  - Group (aggregate) data by one dimension.
  - Zoom-out on the data cube.
  - Eg.: sum up the sales of each city by country.

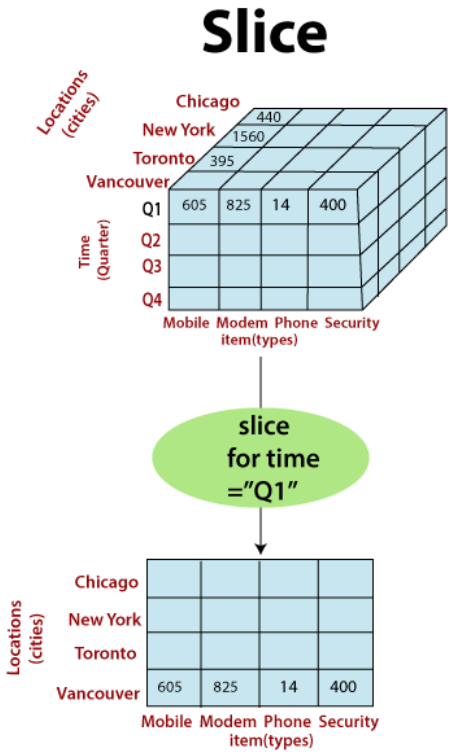


- **Drill-Down**
  - Reverse of Roll-Up operation: decompose data in one dimension.
  - Zoom-in on the data cube.

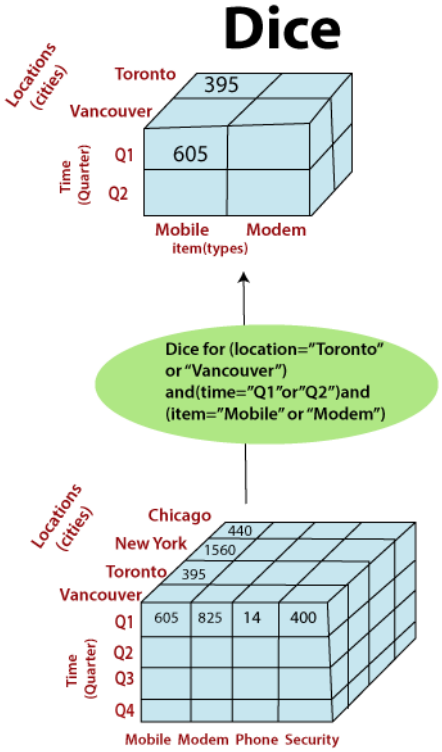


- **Slice**
  - Subset of the cubes corresponding to a single value for one or more members of the dimension.

- Reduce N dimensions to N-1 dimensions by restricting one dimension to a single value.

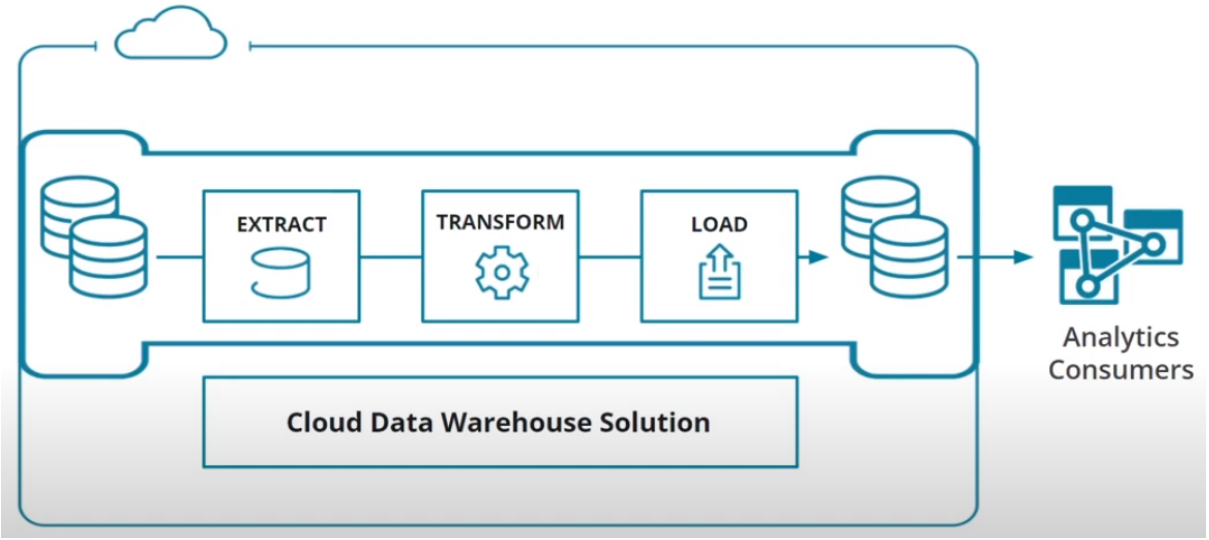


- **Dice**
  - Subcube by operating a selection on two or more dimensions.
  - Same dimensions but computing a subcube by restricting some of the values of the dimensions.



2.3 ELT and Data Warehouse technology in the cloud

Introduction to Cloud Data Warehouses



In a Cloud Data Warehouse (CDW) you'll be working with:

- Database storage technologies for ingesting data as well as making it available to analytics consumers.
- Data pipeline technologies to move data from source to warehouse, as well as between the stages of the Extract, Transform and Load (ETL) processes.
- End-to-end data warehouse solution that provides the ability to manage the various parts of a data warehouse from a single application.

Lesson outline

- ELT (Extract, Load, Transform):
  - Introduction to ELT.
  - ELT vs ETL.
  - When to use ELT.
- Cloud managed data storage:
  - Cloud SQL storage.
  - Cloud NoSQL storage.
- Cloud pipeline services:
  - Cloud ETL pipeline services.
  - Cloud streaming pipeline services.
  - Cloud Data Warehouse solutions.

Key benefits of moving from On-premise to Cloud Data Warehouses

- **Scalability**. Large amounts of data can be loaded into the DW environment, processed, and stored faster and with relative ease

- **Flexibility**. Comes from the ability to add and remove different types of resources from data pipelines as needed. This allows for flexibility in the process as business needs change.
- **Cost shifting** is a result of leveraging cloud database technology to perform the costly and time-consuming transformation of data as the last part of the process rather than doing it earlier as is typical with on-premise DW. By pushing it later, the business can prioritize which transforms they want to complete “just in time” for the business.

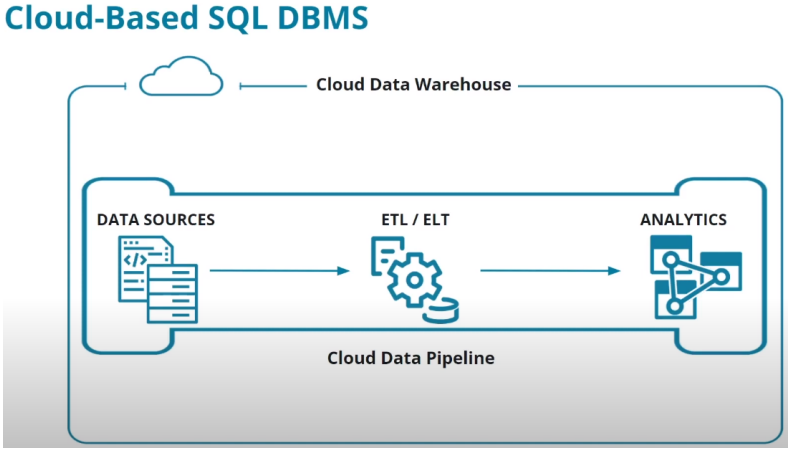
From ETL to ELT

- Traditionally, Data Warehouse ETL processes transform data on a transformation engine running on an intermediate server between extracting and loading.
- Modern cloud based DW pipelines often use ELT instead of ETL. The difference is where the transformation step happens:
    - ETL: transformation on an intermediate server.
    - ELT: transformation on the destination server.
- This means rather than loading data directly into the final format of the destination data warehouse, data is loaded into the destination as either raw data or staging tables (or sometimes both). Only after loading is the transformation performed.

Some benefits of ELT:

- Scalability: massive amounts of data can be loaded into the DW environment with relative ease.
- Flexibility: the Transform step takes place using the same tech stack as the DW runs on allowing for more flexibility in the process as business needs change.
- Cost shifting: the Transform step is often the most costly and by doing it last, Data Engineers can perform Just In Time transformations to meet the highest priority business needs first.
- Better performance for large datasets
- More flexibility for unstructured (NoSQL) datasets

Cloud managed SQL storage

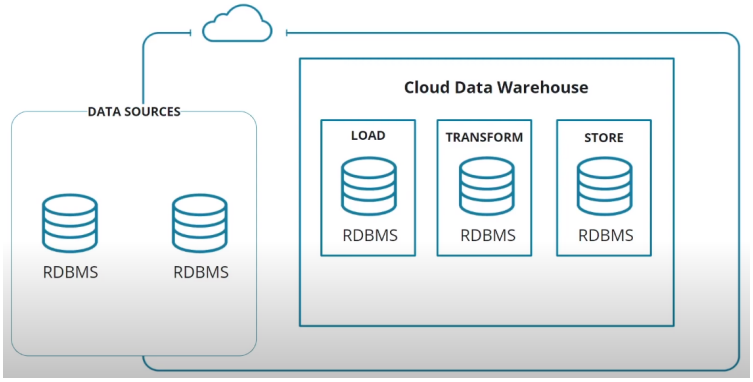


Key components of a Cloud Data Warehouse:

- Data sources.
- ETL or ELT processes.
- Analytics applications.
- Cloud data pipeline to connect all of them.

RDBMS come into play in several parts:

- They are used as data sources, both On-premise and in the Cloud.
- Also for loading, transforming and storing data in the Cloud.



On the Cloud, these RDBMS leverage many of the same SQL style relational databases that are used for OLTP systems. This includes popular RDBMS, such as: Oracle, Microsoft SQL Server, PostgreSQL, MySQL, MariaDB.

The major Cloud providers offer fully managed SQL database systems, which means the user doesn't have to manage the hardware resources to gain optimal performance.

When you run an unmanaged DB, it means you're responsible for:

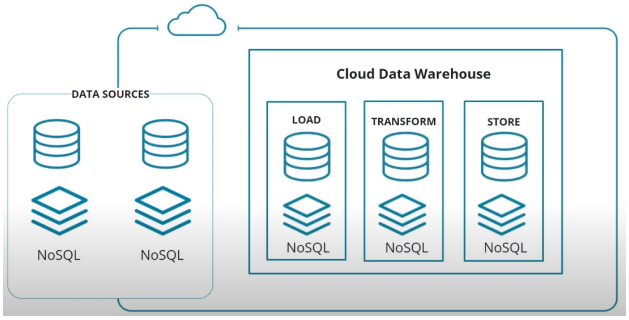
- Running.
- Monitoring.
- Patching.
- Maintaining.

the database server, as well as the underline OS that runs on.

Fully managed system means the configuration and maintenance of the underline server is performed by the Cloud Service Provider and you only need to configure certain aspects of the database itself.

It's also up to the Data Engineer to determine which SQL DBMS is right for the project we're working on.

Cloud managed NoSQL storage



As RDBMS, NoSQL MS come into play in several parts:

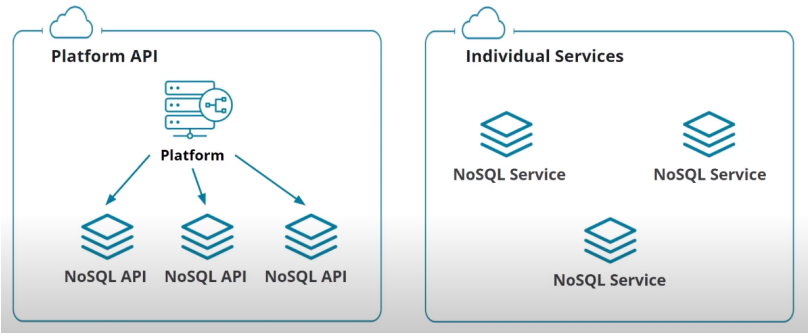
- They are used as data sources, both On-premise and in the Cloud.
- Also for loading, transforming and storing data in the Cloud.

ELT makes it easier to use many NoSQL database management systems in Data Warehousing scenarios. These database come in many styles such as:

- Key value.
- Document.
- Column oriented.
- Graph.
- Time series.

There are two primary ways Cloud service providers offer NoSQL database systems:

- A single platform that provides multiple APIs to the various types of NoSQL databases.
- In other cases, each type of NoSQL database is offered as a separate service.

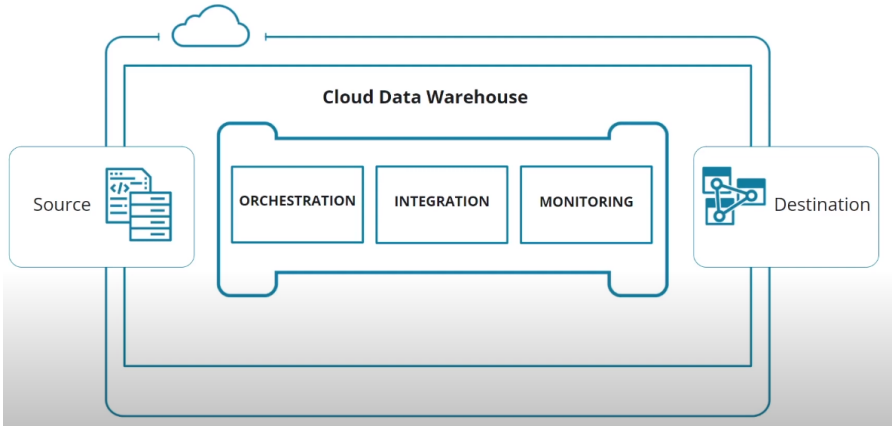


Cloud-based ETL/ELT pipeline services

ETL / ELT processes rely on data pipelines often built using Cloud-based tools. This data pipeline tools can be viewed as a combination of:

- Orchestration tools.
- Integration tools.
- Monitoring tools.

that helps a Data Engineer move and combine data from source to destination.



Major Cloud providers for pipeline tools:

- Azure Data Factory.
- AWS Glue.
- GCP Dataflow.

In addition to these tools, a large number of companies offer cloud-based tools for solving ETL / ELT challenges. These tools offer a variety of connections to both On-premise and Cloud-based data sources and destinations.

Cloud Data Warehouse solutions

Modern cloud data warehouse solutions combine elements from Cloud Storage and Cloud Pipelines with powerful analytics capabilities. Each of the three major cloud providers has its own flavor of Cloud Data Warehouse that works seamlessly with its other cloud data engineering offerings.

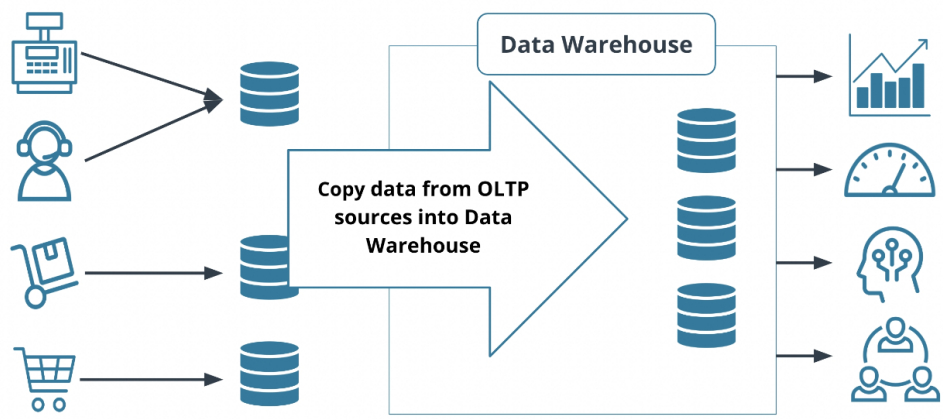
Major Cloud providers for DW systems:

- Azure Synapse.
- Amazon Redshift.
- GCP Big Query.

2.4 AWS Data Warehouse technologies

Building Data Warehouses with AWS Redshift

We'll use AWS Redshift and ETL techniques to build and query the dimensional model required to support analytical processes required in a data warehouse.



Lesson outline

Information to implement a Data Warehouse on AWS. Reference material about:

- Creating IAM Roles.
- Creating IAM Users.
- Creating Security Groups.
- Launching Redshift Clusters.
- Deleting Redshift Clusters.
- Creating S3 buckets.

AWS Billing Dashboard

Amazon Web Services

Services can be accessed in 3 different ways, which can be used in combination:

- AWS Management Console.
- Command Line Interface (CLI).
- Software Development Kits (SDKs). For example, Boto3: AWS SDK for Python

Identity and Access Management (IAM)

- Service that manages access to AWS resources securely.
- IAM is divided into 2 basic operations:
    - Authentication: who can sign in.
    - Authorization: who can do what and to which resource.
- IAM supports creating identities, such as users, groups, and roles.

- Access is granted through IAM policies, which are either "built-in", "custom", or "inline".
- IAM policies are usually of the format, "resource:action" (e.g. "s3:ListBucket").
- Policies can be applied directly to resources (outside of IAM) or identities within IAM.
- There is no charge for the IAM service.

IAM users

- An IAM user is an identity that allows access to AWS resources, via 2 ways:
    - Console (username and password authentication).
    - Programmatically (access key and secret key authentication).
- Users should be granted the least permissions they need.
    - This reduces the potential for error or malfeasance.
- Permissions can be attached to the user by built-in policies or by inline policies.

IAM policies

- IAM permissions are granted via "policies".
- A policy defines who can access a resource and what actions are allowed on that resource.
- Each IAM policy is composed of actions, resources, and principals.
    - principal => either an IAM user or a role.
    - actions => what the principal can / can't do.
    - resources => where can the principal perform these actions.
- There are three types of IAM policies:
    - "Built-in" policies provided by AWS.
    - "Custom" policies created by AWS users.
    - "Inline" policies.
- Formatted as JSON documents with multiple fields.
    - Version field: used internally by AWS; it doesn't change.
    - Statement field: list of statements, each of them with the following subfields:
        - Sid: optional statement ID field to document the statement's purpose.
        - Effect: must be either "Allow" or "Deny".
        - Action: list of resources permissions.
        - Resource: list of **Amazon resource names (arn)**.
          An arn uniquely identifies the resource and its subresources across all AWS accounts.
- **Least Privilege Principle**
    - Security best practice.
    - Grant only the minimum permissions required.
    - Grant necessary permissions gradually.
- Policies are classified as: List, Read and Write.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowReadAccessToBucket",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::bucket-name",
                "arn:aws:s3:::bucket-name/*"
            ]
        }
    ]
}
```

IAM roles

- An IAM role has predefined permissions and allows you to delegate access to trusted entities.
- Defines a set of permissions for accessing AWS services.
- One benefit of the IAM role is the ability to provide access without sharing long-term credentials.
- Can be assumed by a trusted service and operate on your behalf.
- As an example, the AWS EC2 service is a trusted entity.
- We can create an IAM role that an EC2 instance can use to access an S3 bucket.
- EC2 will automatically request temporary credentials and will renew these credentials every 15 minutes. This is why using an IAM role is far more secure than creating an IAM user with long-term credentials.
- IAM roles are the best practice for delegating access to AWS services.
- Designed for service-to-service communication.

Simple Storage Service (S3)

- AWS S3 is central cloud object storage available in many geographic locations ("regions").
- Permissions to access S3 are set via AWS IAM policy.
- An S3 bucket is an object (file) store with high availability. Like a directory.
- S3 can store static files, or even a static website, without any need of a web server.
- Within an S3 bucket, you can create folders and upload files into specific folders.
- A path to a file within a bucket is called “a prefix”.

- Buckets are globally available, but must be created in a specific region.
- IAM permissions control access to S3 buckets.
- An IAM policy can apply to a specific bucket.
- S3 storage is very economical.

AWS Elastic Compute Cloud (EC2)

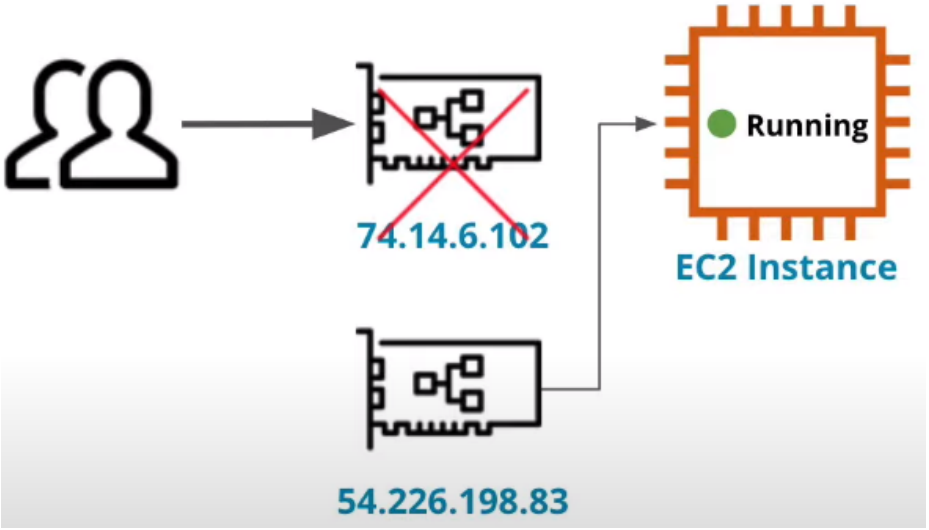
- Resizeable, scalable compute capacity.
- Instance sizes are predefined by instance types.
- Configuring an instance can be done within the EC2 Launch Wizard:
    - Security groups (i.e. firewall rules).
    - Networking (public or private).
    - Storage size and type.
    - Custom provisioning scripts.
- **Init script**: script that you can use to configure or install software on your instance.
    - By default, Init scripts run only on the first boot of the instance (configurable).

Security group

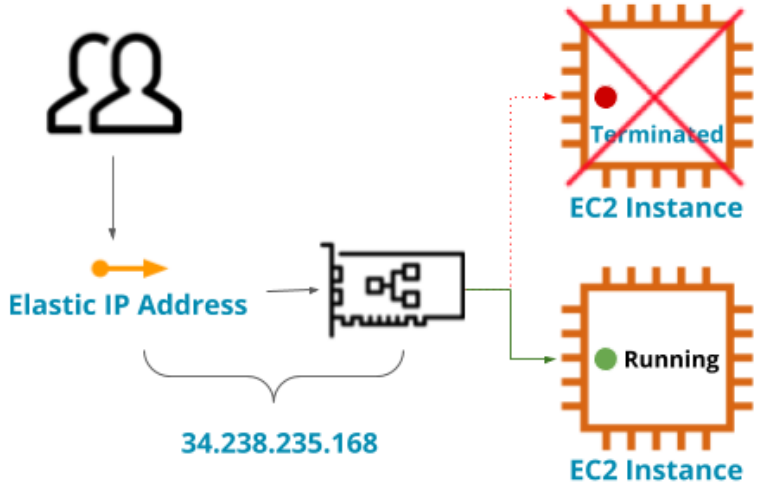
- Security groups are collections of rules that define who can access your AWS resources and how.
- Acts as a virtual firewall.
- With security groups you can control inbound and outbound traffic.
- Security groups are scoped to an instance level, not to a network or subnet.
    - You can assign a different set of security groups to each instance.
    - Each instance can have up to 5 security groups attached.
- Each security group is made of:
    - Inbound rules (“ingress”), which define incoming traffic.
    - Outbound rules (“egress”), which define outgoing traffic.
- A rule consists of port and IP addresses that are allowed to communicate with that port, either ingress or egress.
- You can specify ALLOW rules but you can't specify DENY rules.
- Like for IAM policies, Least Privilege Principle applies.
- It’s best practice to specify a security group for the purpose it serves. For example, SSH, web, etc.
- Security groups are always free resources, so there is no charge for creating or keeping them.

Elastic IP

- In Cloud computing, **IP addresses** are **dynamic**, and can change frequently.
- Cloud providers assign an IP address from a shared pool.
- When an instance’s stopped, restarted or terminated, that IP address goes back to the pool.
- If the instance resumes, AWS will assign a new IP address from the pool, which is almost always different from the previous one.



- A static IP address, by contrast, doesn't change.
- You can use static elastic IP address to mask a failure of an instance by remapping the address to a new instance.
- This means that the internet will always know where your instance is.
- An elastic IP address is public and reachable from the internet.



- You can't have 2 public IP addresses per instance.
- An elastic IP address doesn't incur charges as long as there is network traffic flowing through it.
    - If the address is not attached to any EC2 instance, or it's attached to an instance that's stopped, then the network traffic wouldn't flow through that elastic IP address, so that IP address would incur charges.

Exercise: create an IAM role

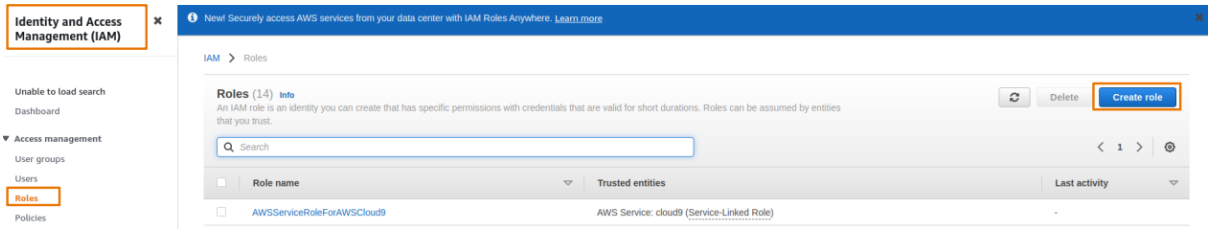
All AWS services require identity and access management (IAM) policies, roles, and their associated permissions. In this exercise, you'll create an IAM user role that has permission to use the Redshift service on AWS.

You will later attach this role to your Redshift cluster to enable your cluster to load data from Amazon S3 buckets.

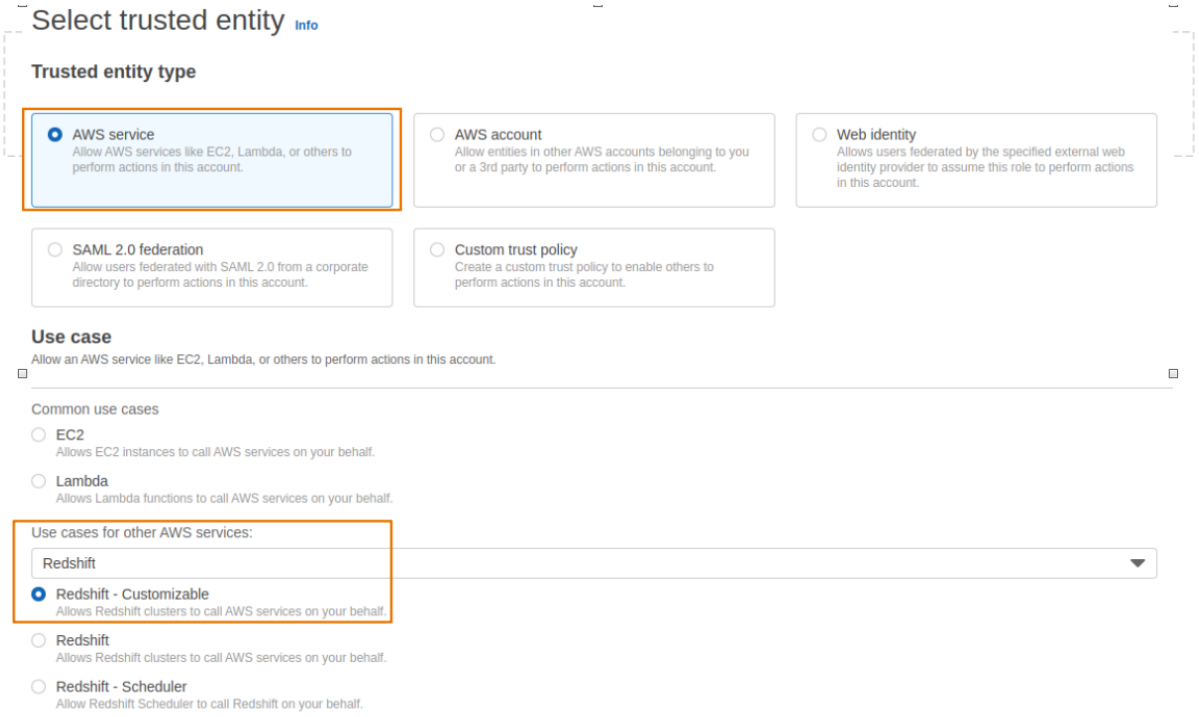
More information: [Getting started with Amazon Redshift]

Steps:

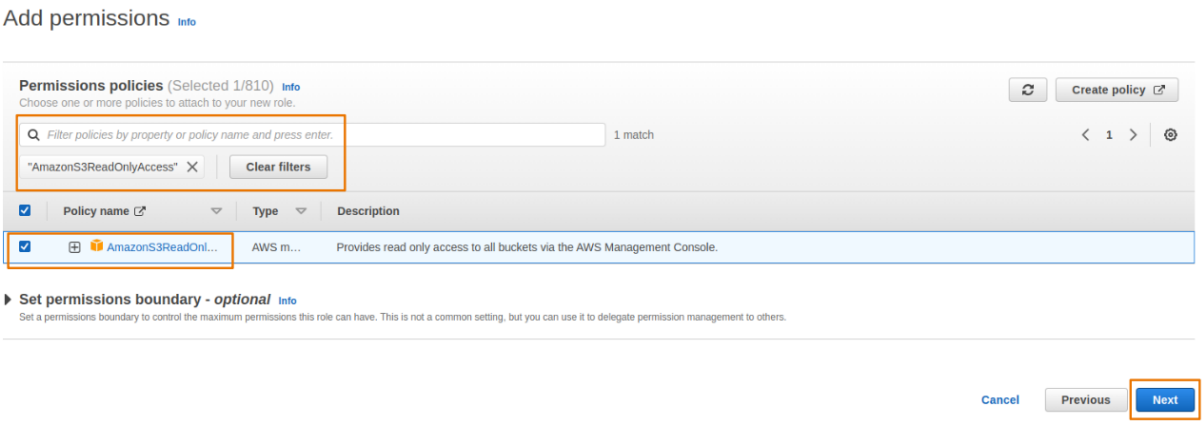
1. Sign into the AWS Management Console.
2. Navigate to the [IAM service dashboard].
3. In the left navigation pane, **Access management** section, choose **Roles**.
4. Choose **Create role**.



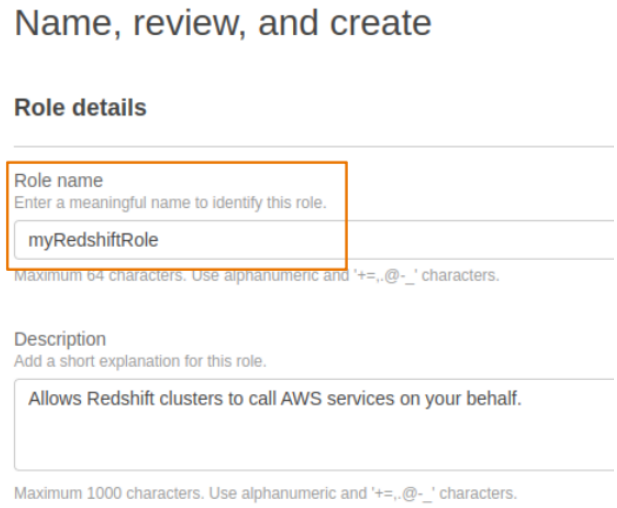
5. **Select trusted entity page (step 1)**: select **AWS Service** group as the trusted entity, and choose **Redshift** service as the **Use case**. Among the options, choose **Redshift - Customizable**, and then click **Next.**



6. On the **Add permissions page (step 2)**, search for and select the **AmazonS3ReadOnlyAccess** policy, and then click **Next**.



7. On the **Name, review and create page (step 3)**, For **Role name**, enter **myRedshiftRole**, and then choose **Create Role** (**Tags** are optional).



8. You will see a success message when the new role will be created.

Next is to attach this role to a new/existing cluster.

Exercise: create a Security group for Redshift

In this exercise, you'll create a security group you will later use to authorize access to your Redshift cluster. A security group will act as firewall rules for your Redshift cluster to control inbound and outbound traffic.

Steps:

1. Navigate to the EC2 service.
2. Under **Network and Security** in the left navigation pane, select **Security Groups**. Click the **Create security group** button to launch a wizard.
3. In the **Create security group** wizard, enter the basic details.

4. In the **Inbound rules** section, click on **Add Rule** and enter the following values:
    a. Type: Custom TCP
    b. Protocol: TCP
    c. Port range: 5439
        i. The default port for Amazon Redshift is 5439, but your port might be different.
    d. Source type: Custom → Anywhere-IPv4
    e. Source: 0.0.0.0/0
        i. **Important**: Using 0.0.0.0/0 is not recommended for anything other than demonstration purposes because it allows access from any computer on the internet. In a real environment, you would create inbound rules based on your own network settings.
5. **Outbound rules** section: leave default values (allow traffic to anywhere).
    a. Type: All traffic
    b. Protocol: All
    c. Port range: All
    d. Destination type: Custom
    e. Destination: 0.0.0.0/0
6. **Tags** are optional. Click on the **Create security group** button at the bottom. You will see a success message.

Exercise: create an IAM user for Redshift

In this exercise, you'll create an IAM user that you will use to access your Redshift cluster.

Steps:

1. Navigate to the IAM service dashboard.
2. In the left navigation pane, **Access management** section, choose **Users**.
3. Click on the **Add users** button. It will launch a new wizard.
4. **Set user details** section: enter a name for your user , say **airflow_redshift_user**, and choose **Access key - Programmatic access**. Then click on the **Next: Permissions** button.
5. **Set permissions** section: choose **Attach existing policies directly**.
    a. Search for redshift and select **AmazonRedshiftFullAccess**.
    b. Search for S3 and select **AmazonS3ReadOnlyAccess**.
    c. After selecting both policies, choose **Next: Tags**.

6. **Tags** are optional. Skip this page and choose **Next: Review**.
7. Review your choices and finally click on the **Create user** button.
8. **SAVE YOUR CREDENTIALS**
   This is the only time you can view or download these credentials on AWS. Choose **Download .csv** to download these credentials and then save this file to a safe location. You'll need to copy and paste both **Access key ID** and **Secret access key** contained in the CSV in the next step.
   IMPORTANT: keep this Access key ID and Secret access key closely guarded, including not putting them in a GitHub public repo, etc.

   After downloading the credentials csv file, you can close the wizard. You'll see a success message.

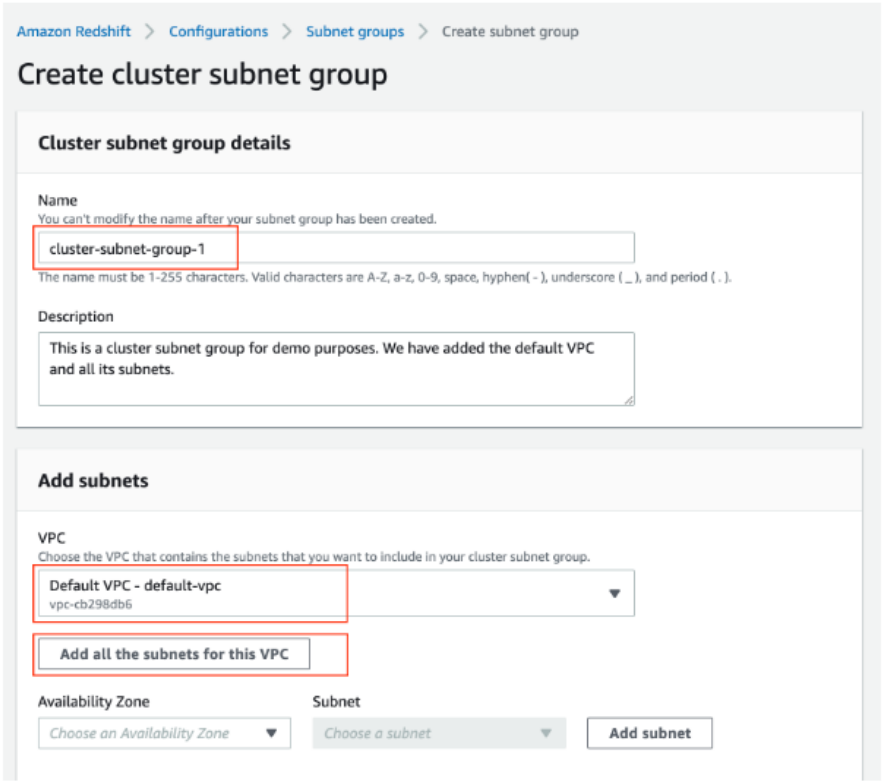
Exercise: launch a Redshift Cluster

An Amazon Redshift data warehouse is a collection of computing resources called nodes, which are organized into a group called a cluster. Each cluster runs an Amazon Redshift engine and contains one or more databases.

WARNING
Each time you create a cluster, you are charged the standard Amazon Redshift usage fees for the cluster until you delete it. Make sure to delete it each time you're finished working to avoid large, unexpected costs.

Steps:
1. Sign in to the AWS Management Console and open the Amazon Redshift console.
2. On the **Amazon Redshift Dashboard**, choose **Create cluster**. It will launch the Create cluster wizard.
3. **Cluster configuration**:
   Provide a unique identifier, such as **redshift-cluster-1**, and choose the **Free trial** option. It will automatically choose the following configuration:
   a. 1 node of dc2.large hardware type. It is a high performance with fixed local SSD storage.
   b. 2 vCPUs.
   c. 160 GB storage capacity.
4. **Database configurations**:
   A few fields will be already filled up by default. Ensure to have to the following values:
   a. Admin user name: awsuser
   b. Adminuser password: <a_password_of_your_choice>
   IMPORTANT: keep these passwords closely guarded, including not putting them in a GitHub public repo, etc.
   Create cluster → It will take a few minutes.
5. Click on the **Clusters** menu item from the left navigation pane, and look at the cluster that you just launched. Make sure that the **Status** is **Available** before you try to connect to the database later.
6. Click on *redshift-cluster-1* that you've just created → **Properties.**
7. Database configurations: previously completed, leave as it is.

8. **Network and security settings → Edit**
    a. Virtual private cloud (VPC): Default VPC.
    You will have to create a [cluster subnet group](#).



Create a cluster subnet group from a default VPC

    b. VPC security groups: Choose the *redshift_security_group* created earlier.
    c. Cluster subnet groups: Choose the default (the one you have just created).
    d. Availability zone: No preference.
    e. Enhanced VPC routing: Turn off.
    f. Publicly accessible: Enable.
9. **Cluster permissions**: **Associated IAM roles**
    Click on the Associate IAM role button and choose the IAM role created earlier, *myRedshiftRole*. Set it as the default for this cluster.
10. Leave the rest of the values as default.

Exercise: delete a Redshift Cluster

Steps:

1. On the **Clusters** page of your Amazon Redshift console, click on the **check-box** next to your cluster name. Then click on the **Actions** drop-down button on top → select **Delete**.
2. You can choose to not **Create final snapshot**, and click on the **Delete cluster** button.
3. Your cluster will change its status to **deleting**, and then disappear from your Cluster list once it's finished deleting. You'll no longer be charged for this cluster.

Exercise: create an S3 bucket

A bucket is a container for objects stored in S3. Let's learn how to create a bucket in Amazon S3, and view a few properties of an existing bucket.

Create a bucket
We create a bucket first, and later we upload files and folders to it.

1. Navigate to the S3 dashboard, and click on the **Create bucket** button. It will launch a new wizard.
2. **General configuration**
   Choose bucket name and AWS region.
3. **Object ownership**
   Leave default value (ACLs disabled).
4. **Block Public Access settings for this bucket**
   You can choose public visibility. Let's uncheck the **Block all public access** option.
5. **Bucket versioning**
   Disable
6. **Tags** (optional)
7. **Default encryption**
   If enabled, it will encrypt the files being stored in the bucket.
   Let 's disable it.
8. **Advance settings: Object lock**
   If enabled, it will prevent the files in the bucket from being deleted or modified.
   Let 's disable it.
9. Click on the **Create bucket** button.

We have created a public bucket. Let's see how to upload files and folders to the bucket, and configure additional settings.

Upload File/Folders to the Bucket

1. From the S3 dashboard, click on the name of the bucket you have created in the step above.
2. Click on the **Upload** button to upload files and folders into the current bucket.
3. Drag and drop some files. Click **Upload**.

Details of an existing bucket

- **Properties**
  There are several properties that you can set for S3 buckets, such as:
  - Bucket Versioning - Allows you to keep multiple versions of an object in the same bucket.
  - Server access logging - Log requests for access to your bucket.
  - Requester pays - Make the requester pay for requests and data transfer costs.
  - Static website hosting - Mark if the bucket is used to host a website. S3 is a very cost-effective and cheap solution for serving up static web content.

- **Permissions**
  It shows who has access to the S3 bucket, and who has access to the data within the bucket. Here, we can write an access policy (in JSON format) to provide access to the objects stored in the bucket.

- **Metrics**
  View the metrics for usage, request, and data transfer activity within your bucket, such as, total bucket size, total number of objects, and storage class analysis.
- **Management**
  It allows you to create life cycle rules to help manage your objects. It includes rules such as transitioning objects to another storage class, archiving them, or deleting them after a specified period of time.
- **Access points**
  Here, you can create access endpoints for sharing the bucket at scale. Using an endpoint, you can perform all regular operations on the bucket.

Exercise: AWS Relational Database Service (RDS)

When you're working with OLTP data in the cloud, you might need to use AWS Relational Database Services (RDS) as a data source. **AWS RDS** is a managed **PostgreSQL** service. In this exercise, you'll see how to create a PostgreSQL database and view the details of an existing database on RDS.

Amazon RDS is a relational database service that manages common database administration tasks, resizes automatically, and is cost-friendly.

Create a PostgreSQL database

1. Navigate to the RDS Dashboard. **Databases** → click on the **Create Database** button. It will launch a new wizard.
2. **Choose a database creation method**: AWS provides two options to choose from:
   a. Standard create - You have set all of the configuration options, including ones for availability, security, backups, and maintenance.
   b. Easy create - You use the industry best-practice configurations. All configuration options, except the Encryption and VPC details, can be changed after the database is created.

   The steps below will show you the **Standard create** fields/options.
3. **Engine options**: select PostgreSQL option. It will pick up the latest stable release by default, though you can select a version of your choice as well.
4. **Templates**: Use either the RDS Free Tier or Dev/Test template. On free-tier resources, you can develop and test applications to gain hands-on experience with Amazon RDS. The free tier will offer you 750 hrs of Amazon RDS in a Single-AZ *db.t2.micro* Instance, 20 GB of General Purpose Storage (SSD), and 20 GB for automated backup storage and any user-initiated DB Snapshots.
   We choose **Free tier.**
5. **Settings**:
   a. **DB instance identifier**, for example, **postgreSQL-test.**
   b. Master credentials:

    i. username
    ii. password - Take note of this password, as you'll need it later.
6. **Instance configuration**: **DB instance class**. The options here present the options for processing power and memory requirements. Since we have selected the Free tier option above, the only available options are *db.t3.micro* and *db.t4g.micro*, which have 2 vCPUs and 1 GiB RAM.
7. **Storage**: choose the default options here. It will offer you 200 GiB SSD storage, expandable up to 1000 GiB, by default.
8. **Connectivity**: Choose/ensure the following values:
    a. VPC: default VPC.
    b. Subnet group: default.
    c. Public access: YES. Once you get familiar, choose NO next time. You will need additional configuration to do so.
    d. VPC security group: either choose default or create a new one.
    e. Availability zone: no preference.
    f. Database port: 5432 (default).
9. **Database authentication**: leave the default value (password authentication).
10. **Monitoring**: default values.
11. **Additional configuration**:
    a. Initial database name.
    b. Backup retention period: select 1 day, since this is for demonstration purposes.
    c. Leave the default values for the rest.
12. Click on the **Create database** button.
13. **Success**: You should land on a confirmation page. It will take a few minutes to launch the database. Wait a few minutes for the status to change to **Available**.

Infrastructure as Code (IaC) on AWS

Some advantages of using IaC over the GUI:

- Sharing: one can share all the steps with others easily.
- Reproducibility: one can be sure that no steps are forgotten.
- Multiple deployments: one can create a test environment identical to the production environment.
- Maintainability: if a change is needed, one can keep track of the changes by comparing the code.

Boto3

- Python SDK for programmatically accessing AWS.
- It enables developers to create, configure, and manage AWS services.

Lesson review

Identity and Access Management (IAM)

- Create an IAM role.
    - Create a role which has permission to use the Redshift service on AWS.

    - Later attach this role to your Redshift cluster to enable your cluster to load data from Amazon S3 buckets.

- Create an EC2 security group.
    - Create a security group which will be used to authorize access to your Redshift cluster.
    - A security group will act as firewall rules for your Redshift cluster to control inbound and outbound traffic.
- Create an IAM user.
    - Create a user that you will use to access your Redshift cluster.
- Launch a Redshift cluster.
    - Redshift DW is a collection of computing resources called nodes.
    - These nodes are organized into a group called cluster.
    - Each cluster runs a Redshift engine and contains one or more databases.
- Delete a Redshift cluster.
- Create an S3 bucket.
    - A bucket is a container for objects stored in S3.
- Create an Amazon Relational Database Service (RDS).
    - Amazon RDS is a web service that makes it easier to set up, operate, and scale a relational database in the AWS Cloud.
    - We make use, in this case, of a Postgres DB engine (others available).

Glossary

IAM
Identity and Access Management is the service for creating identities and policies to govern access to resources.

IAM user
A type of identity designed for console and programmatic access to the AWS services.

IAM role
An IAM identity that has no long-term credentials and acts on behalf of the trusted user or service.

IAM policy
A document defining who can do what on which resource.

S3
Simple Storage Service, which is AWS's service to store files in the cloud.

Bucket
A storage resource within S3, similar to a folder.

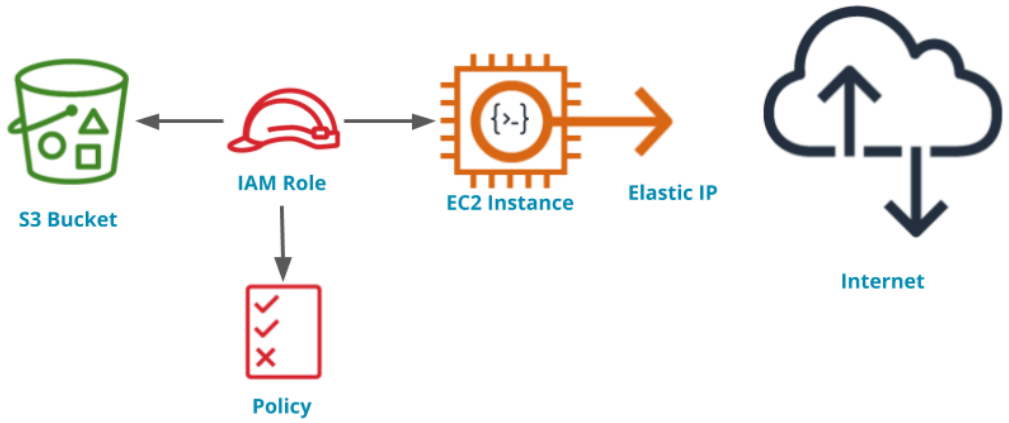
Object
A unit of storage in S3, typically an object is a file.

EC2
Elastic Compute Cloud, this is AWS's foremost compute service.

EC2 instance
A virtual server running applications or software in AWS. An instance type consists of CPU, memory, storage, and networking capacity.

Elastic IP
A static IP allocated to your account that can be associated and dissociated from EC2 instances. Instances' public IP addresses are dynamic and will change when you restart your instance. An Elastic IP Address is static and can always remain attached to the instance you want. An Elastic IP is public and that way reachable from the Internet.

Security Group
A firewall ruleset to control traffic to and from AWS resources. Access to servers is enforced with firewall rules known as “security groups”.

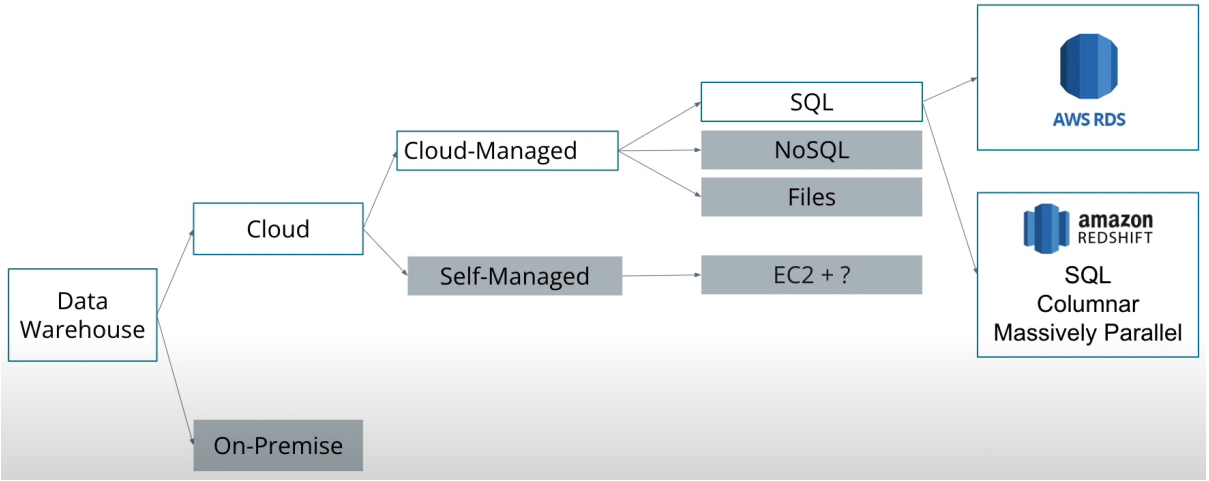


2.5 Implementing a Data Warehouse on AWS

Lesson outline

- Amazon Redshift Data Warehouse solution
    - Redshift architecture
    - Columns vs Row oriented RDBMS
    - Redshift clusters
- ETL in Amazon Redshift
    - SQL to SQL ETL
    - S3 staging
    - Ingesting at scale
- Optimizing Redshift table design
    - Distribution strategies for loading and partitioning data
    - Distribution keys
    - Sorting keys

Amazon Redshift

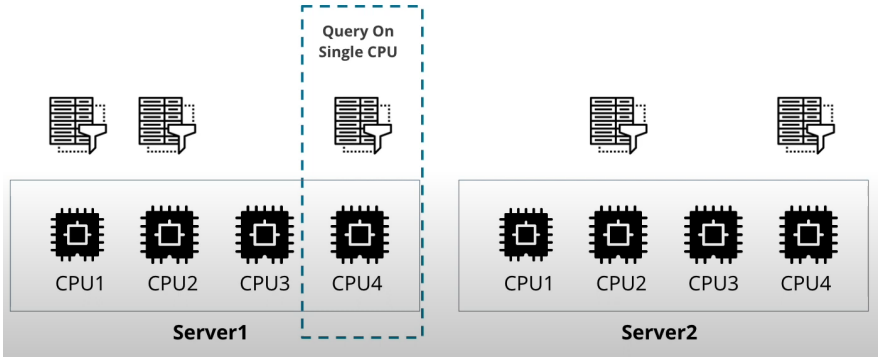


Data warehousing requires a database. We're going to build a Cloud-based data warehouse solution that can be managed using AWS tools.

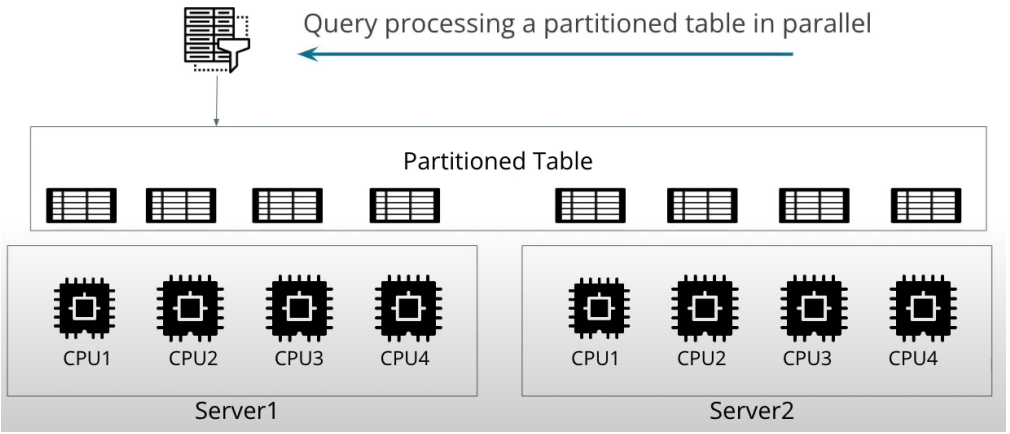
Amazon Redshift vs Amazon RDS

- Both are RDBMS.
- Amazon RDS: row oriented.
- Redshift: column oriented.
    - Best suited for storing OLAP workloads.

- A key difference is the way queries are executed.
    - **Queries in row-oriented RDBMS:**
        - Most relational databases execute multiple queries in parallel, if they have access to many cores or servers.
        - If you have an RDBMS database cluster, many queries can run at the same time, but each query runs on only one CPU.
        - If there are lots of concurrent users running queries, they can be scheduled on CPUs.
        - This is appropriate for OLTP applications because there are a lot of concurrent users, each running small queries, like an update or retrieving a few rows.

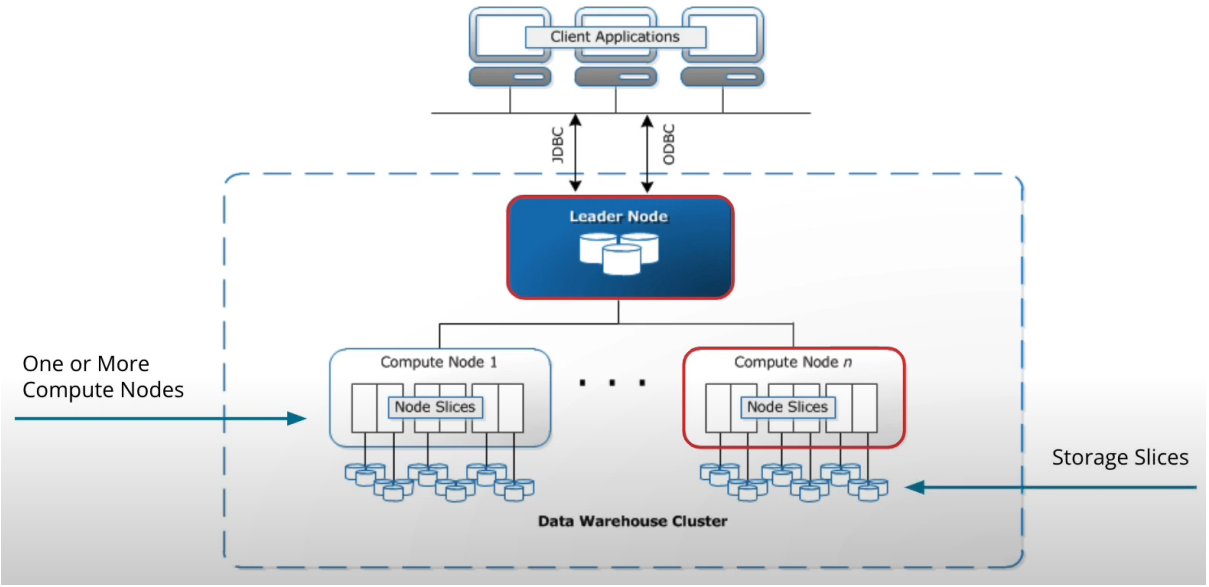


    - **Redshift technology:**
        - When you're dealing with a DW solution and dealing with large amounts of data, you'll need a database capable of massive parallel processing (MPP).
        - MPP databases (like Amazon Redshift) execute one query on multiple CPUs in parallel.
        - Tables in MPP databases are partitioned into smaller partitions and distributed across CPUs, and each CPU has its own associated storage.
        - One query can process a whole table in parallel, and each CPU is processing only one partition of the data.



Redshift Cluster architecture

- Redshift is a cluster of machines composed of one leader node and one or more compute nodes (it's possible to run a cluster with a single node).
- **Leader node:**
  - Coordinates the work of the compute nodes.
  - Handles external communication.
    - Client applications talk to the leader node using protocols like JDBC or ODBC → To them, it's like any other normal database.
  - Optimizes query execution.
- **Compute nodes:**
  - Each with its own CPU, memory and disk (determined by the node type).
  - Each node is an AWS EC2 instance.
  - Scale up: get more powerful nodes.
  - Scale out: get more nodes.
- **Node slices:**
  - Each compute node is logically divided into a number of slices.
  - A slice has dedicated storage and memory.
  - A cluster with n slices can process n partitions of a table simultaneously.
  - The sum of all slices across all compute nodes is the unit of parallelization.



Redshift node types and slices

Computer Optimized Nodes

Node Size	vCPU	RAM (GiB)	Slices Per Node	Storage Per Node	Node Range	Total Capacity
dc1.large	2	15	2	160 GB SSD	1–32	5.12 TB
dc1.8xlarge	32	244	32	2.56 TB SSD	2–128	326 TB
dc2.large	2	15.25	2	160 GB NVMe-SSD	1–32	5.12 TB
dc2.8xlarge	32	244	16	2.56 TB NVMe-SSD	2–128	326 TB

Storage Optimized Nodes

Node Size	vCPU	RAM (GiB)	Slices Per Node	Storage Per Node	Node Range	Total Capacity
ds2.xlarge	4	31	2	2 TB HDD	1–32	64 TB
ds2.8xlarge	36	244	16	16 TB HDD	2–128	2 PB

ETL in Amazon Redshift

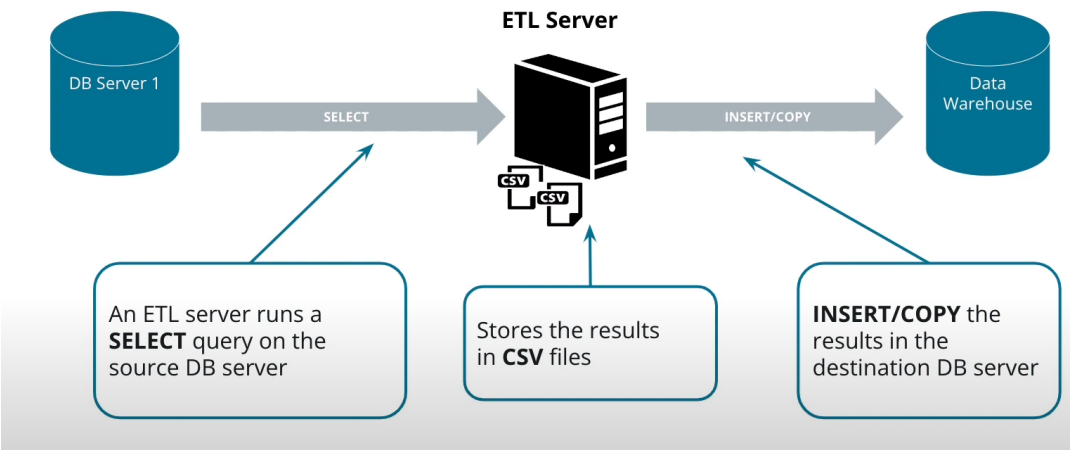
Example:

- We're extracting data from multiple OLTP databases.
- Transforming it from 3NF to a star schema in the Data Warehouse.

We can use SQL queries to select data out of a database. But what do we do if we want to copy the results to another table on a totally different database server like Redshift?

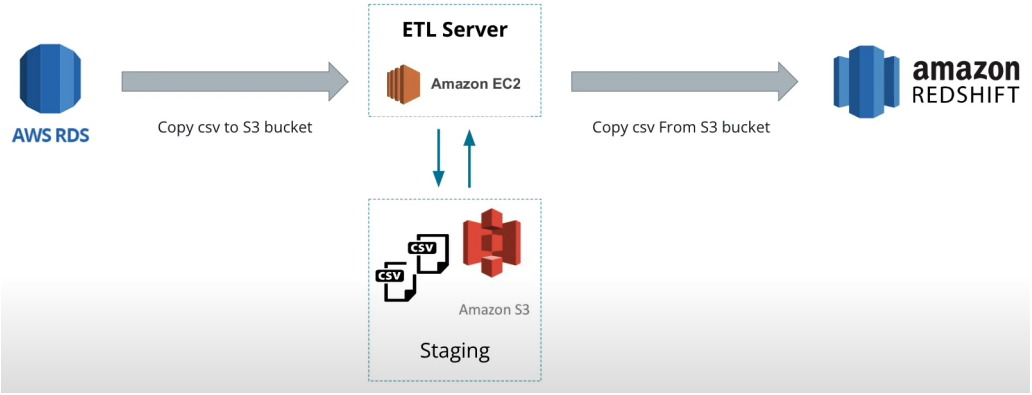
One way to approach SQL to SQL ETL is to put an ETL server between the source database and the destination data warehouse.

- We can use SQL statements to get the data.
- The data is written to CSVs in the local storage or network-attached storage of that ETL server
- And you would insert or copy the data to the data warehouse.



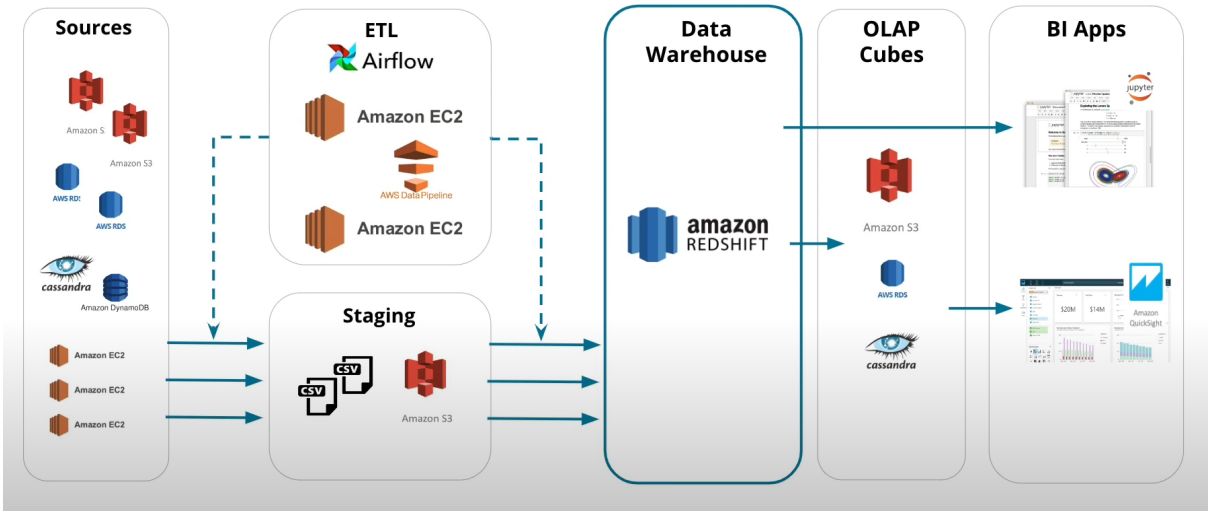
Redshift SQL to SQL ETL

- On the AWS platform, we could use EC2 instances as ETL servers.
- But there is a better way to take advantage of the platform → S3 distributed storage buckets can be used as a staging area for data to be loaded into Redshift.
- Most of the products on the AWS platform, especially the managed service products, are able to communicate with S3 storage.
- You can extract data from multiple data sources within ETL server into S3 as a staging area, use SQL to transform it and load it into Redshift.



Typical Redshift solution architecture and dataflow

- Diverse number of data sources:
  - CSV files.
  - Managed and unmanaged relational data stores like Cassandra or DynamoDB.
  - There could also be EC2 machines.
- Then we would have our ETL servers, a class of products that communicate with all the data sources.
  - ETL server scripts and products issue commands to extract data from the sources and into S3 staging, as well as pulling the data into Redshift.
- Once the data is loaded into Redshift, we're able to connect business intelligence apps and visualizations to it.
  - Data cubes containing pre-aggregated data can also be materialized into Amazon S3.
  - BI apps can work directly from Redshift or faster from these pre-aggregated OLAP cubes.



Ingesting at scale

- Transfer large amounts of data from the S3 staging area to Redshift with the **COPY** command, which performs a bulk upload of data.
    - Inserting data row by row by using INSERT would be very slow.
- Ingest files in parallel:
    - Really big files: break them into multiple smaller files to load in parallel.
    - This will parallelize the copying effort.
- How to know that a group of files belongs together? Two ways to do this:
    - Either the files have a common prefix.
    - Or you can use a manifest file.
- Use compression (gzip).
- Minimize network traffic.
    - Staging area and Redshift cluster should be in the same AWS region.

Example: ingesting with prefix

```
COPY sporting_event_ticket FROM 's3://udacity-labs/tickets/split/part'
CREDENTIALS 'aws_iam_role=arn:aws:iam::464956546:role/dwhRole'
gzip DELIMITER ';' REGION 'us-west-2';
```

```
s3.ObjectSummary(bucket_name='udacity-labs', key='tickets/split/part-00000.csv.gz')
s3.ObjectSummary(bucket_name='udacity-labs', key='tickets/split/part-00001.csv.gz')
s3.ObjectSummary(bucket_name='udacity-labs', key='tickets/split/part-00002.csv.gz')
s3.ObjectSummary(bucket_name='udacity-labs', key='tickets/split/part-00003.csv.gz')
s3.ObjectSummary(bucket_name='udacity-labs', key='tickets/split/part-00004.csv.gz')
s3.ObjectSummary(bucket_name='udacity-labs', key='tickets/split/part-00005.csv.gz')
s3.ObjectSummary(bucket_name='udacity-labs', key='tickets/split/part-00006.csv.gz')
s3.ObjectSummary(bucket_name='udacity-labs', key='tickets/split/part-00007.csv.gz')
s3.ObjectSummary(bucket_name='udacity-labs', key='tickets/split/part-00008.csv.gz')
s3.ObjectSummary(bucket_name='udacity-labs', key='tickets/split/part-00009.csv.gz')
```

Example: ingesting with manifest

```
COPY customer
FROM 's3://mybucket/cust.manifest'
IAM_ROLE 'arn:aws:iam::0123456789012:role/MyRedshiftRole'
manifest;
```

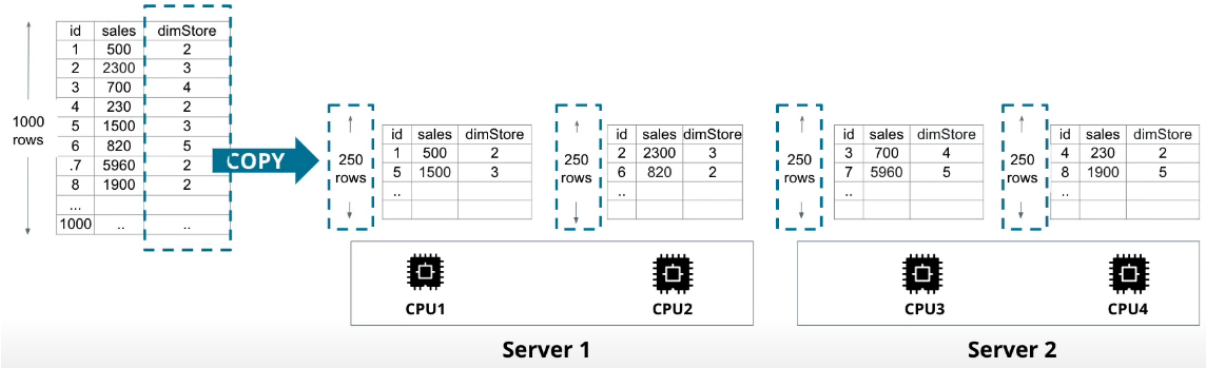
```
{
  "entries": [
    {"url":"s3://mybucket-alpha/2013-10-04-custdata", "mandatory":true},
    {"url":"s3://mybucket-alpha/2013-10-05-custdata", "mandatory":true},
    {"url":"s3://mybucket-beta/2013-10-04-custdata", "mandatory":true},
    {"url":"s3://mybucket-beta/2013-10-05-custdata", "mandatory":true}
  ]
}
```

Optimizing table design with distribution styles

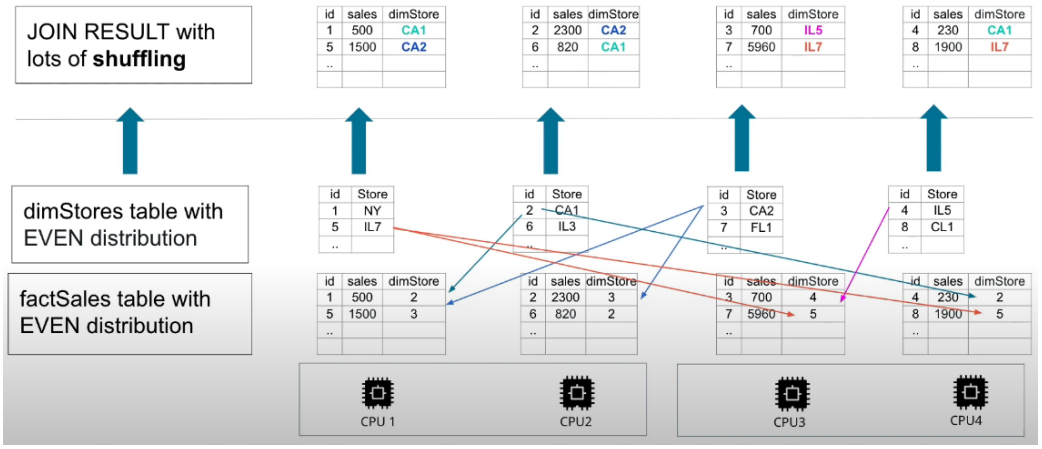
- We want to optimize the design of our tables where we ingest data in order to speed up queries.
- Big tables are partitioned into smaller partitions so that we can access them in parallel and parallelize them in slices.
- If you have information about the frequent access pattern of a table, you can choose a more optimized strategy by configuring different distribution options for your cluster.
- Two possible strategies:
    - Distribution style.
    - Sorting key.
- There are 4 distribution styles:
    - EVEN distribution.
    - ALL distribution.
    - AUTO distribution.
    - KEY distribution.

Uses for each distribution

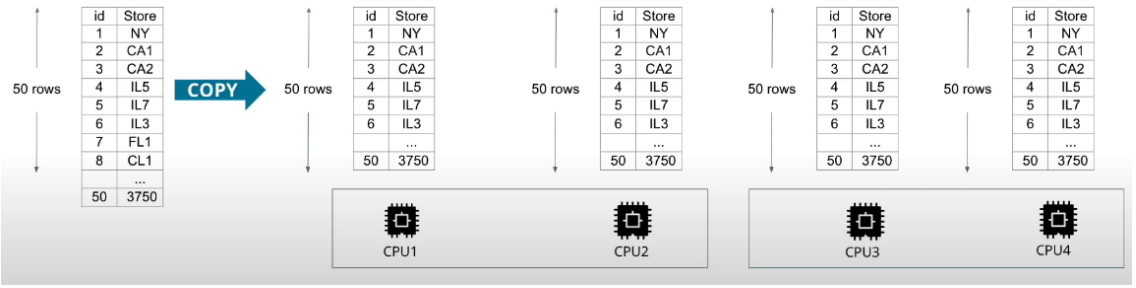
- EVEN distribution
    - Example: table factSales containing 1000 rows, with a primary key and a reference to a dimension (dimStores).
    - We start copying this into a table. We copy it in parallel, and Redshift will balance the amounts of data being copied.
    - Ideally, each server is given the same amount of rows and that evens out the load, using an even or round-robin distribution style.
    - Good if the table won't be joined.



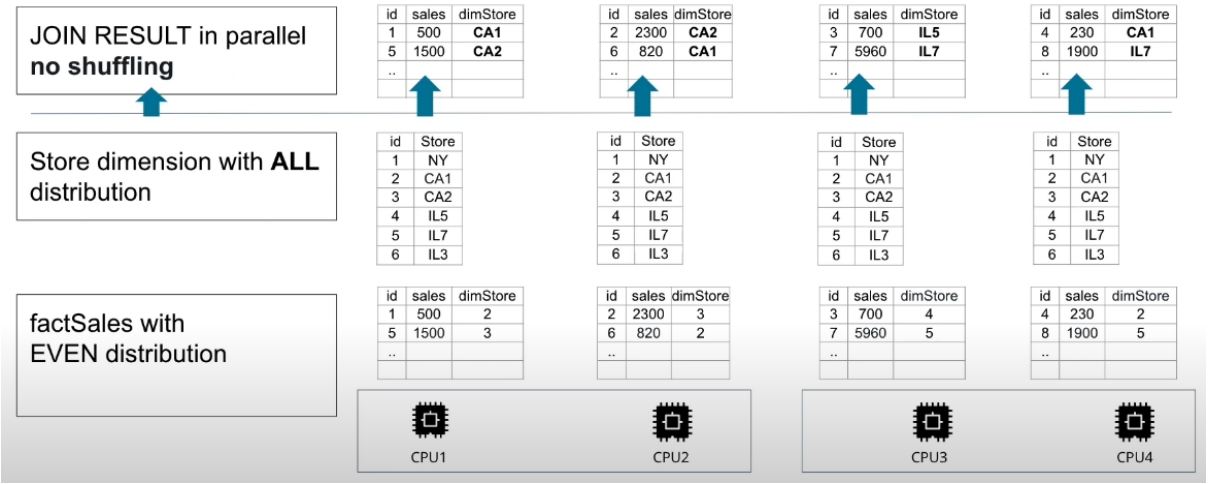
- Another example: we also have a smaller table, which is a dimension table for the stores, with only 40 rows (dimStores).
- If we have dimStores table copied with EVEN distribution, too, when we join the facts and the dimension to get the store information, it will result in lots of shuffling.
- There are other distribution styles to optimize this shuffling.



- <u>ALL distribution</u>
  - Small tables are replicated on all slices to speed up joins.
  - Used frequently for dimension tables (in general, they don't have a lot of rows compared to fact tables).
  - The ALL distribution style is also known as BROADCASTING, because it broadcasts the replicated table across the cluster.

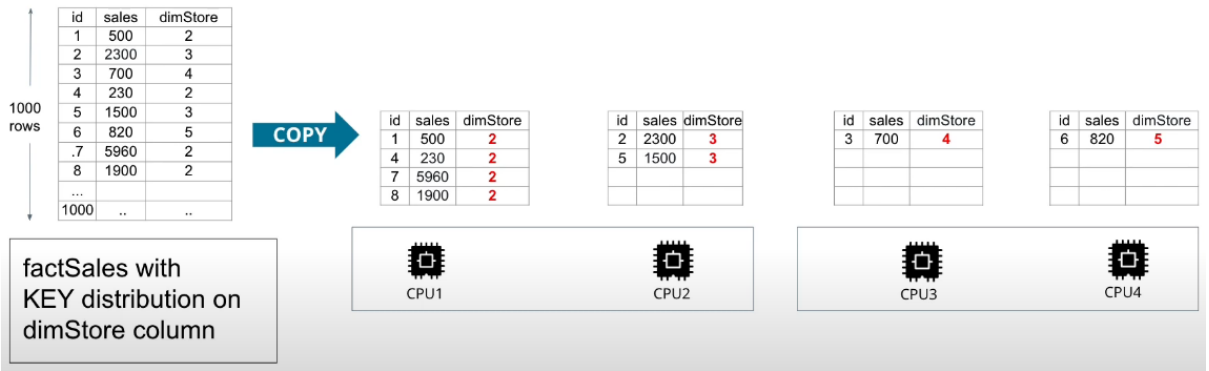


- Using the previous example, the factSales tables are still replicated using the even strategy, but the dimStores table is replicated using the all strategy.



- AUTO distribution
  - Leaves the decision to Redshift.
  - Small tables are distributed with an all strategy.
    - Redshift does the calculations to determine which small tables are optimal to broadcast.
  - Large tables are distributed using an even strategy.

- KEY distribution
  - Rows having similar values are placed in the same slice.
  - In the example below, a fact table is distributed on the dimStore key.
    - Keys are grouped on partitions, which can sometimes lead to a skewed distribution if some values of the key are more frequent than others.



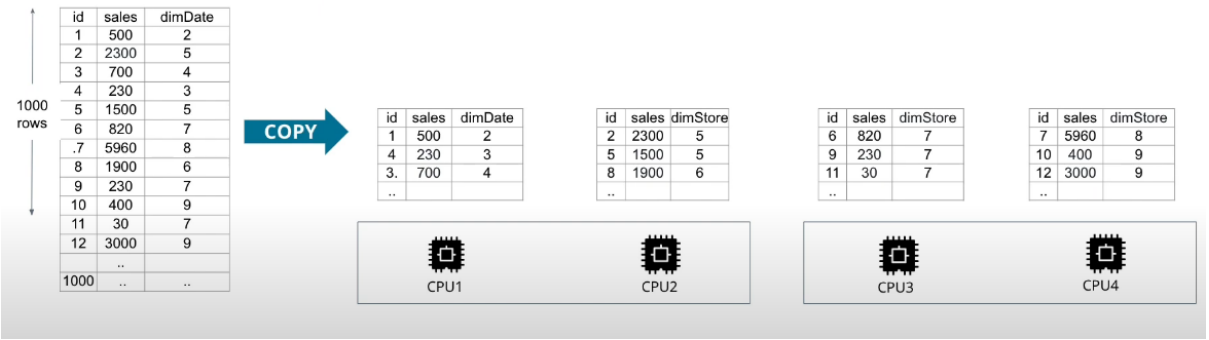
- Key distribution style syntax

```
CREATE TABLE lineorder (
  lo_orderkey           integer       not null,
  lo_linenumber         integer       not null,
  lo_custkey            integer       not null,
  lo_partkey            integer       not null distkey,
  lo_suppkey            integer       not null,
  lo_orderdate          integer       not null sortkey,
  lo_orderpriority      varchar(15)   not null,
  lo_shippriority       varchar(1)    not null,
  lo_quantity           integer       not null,
  lo_extendedprice      integer       not null,
  lo_ordertotalprice    integer       not null,
  lo_discount           integer       not null,
  lo_revenue            integer       not null,
  lo_supplycost         integer       not null,
  lo_tax                integer       not null,
  lo_commitdate         integer       not null,
  lo_shipmode           varchar(10)   not null
);
```

```
CREATE TABLE part (
  p_partkey       integer       not null  sortkey distkey,
  p_name          varchar(22)   not null,
  p_mfgr          varchar(6)    not null,
  p_category      varchar(7)    not null,
  p_brand1        varchar(9)    not null,
  p_color         varchar(11)   not null,
  p_type          varchar(25)   not null,
  p_size          integer       not null,
  p_container     varchar(10)   not null
);
```

Sorting key

- Another strategy for distribution optimization, besides distribution styles.
- You can define a column as a sort key.
- Then, when data is loaded/copied, rows are sorted before distribution to slices.
- This minimizes query time, since each node already has contiguous ranges of rows based on the sorting keys.
- Useful for columns that are used frequently in sorting (ORDER BY) like the date dimension and its corresponding foreign key in the fact table
- A column can be both a distribution key and a sort key (as in the image above).



3 Spark and Data Lakes

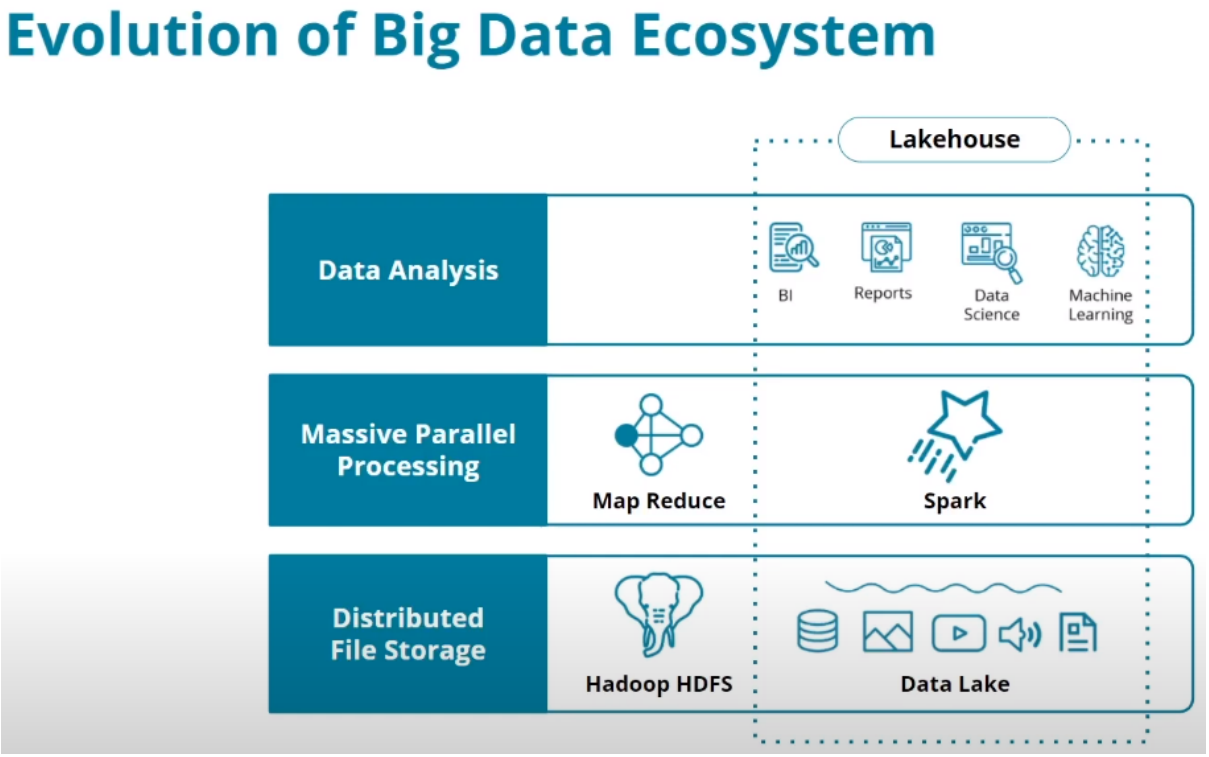
3.1 Course outline

- Big Data ecosystem.
    - Hadoop.
    - MapReduce.
    - Spark.
    - Data lakes and lakehouse architecture.
- Spark essentials.
    - How Spark works.
    - Wrangling data.
    - Spark SQL.
- Using Spark in AWS.
    - Data lakes.
    - How to configure roles and resources for working with AWS Glue.
    - Spark jobs with Gluse Studio.
- Ingesting and organizing data in a lakehouse.
    - Personally identifiable information and data privacy in lakehouse zones.
    - Handling streaming data.
    - Curating data in zones for analytics.

3.2 Big Data ecosystem, Data Lakes and Spark

Introduction to Big Data ecosystem, Spark and Data Lakes

- The modern big data ecosystem is an evolution of **data processing** on **distributed architecture** necessary to handle the sheer volume of data.
- To be effective, Data Science and AI need a big data ecosystem to **move**, **store**, **clean**, **merge**, and **tidy up data**.
- Early efforts at processing large amounts of structured, semi-structured, and unstructured data led to the development of **Hadoop**. Hadoop incorporates two key components:
    - The **Hadoop Distributed File System** (or HDFS) provides distributed storage with high-throughput access to data.
    - **MapReduce** provides a way to conduct massive parallel processing for large amounts of data.
- The next step in the evolution was **Apache Spark**.
    - Spark built on the ideas of Hadoop and provided multiple programming APIs for processing data as well as providing an interactive interface for iteratively developing data engineering and data science solutions.
- Hadoop and Spark have led to the development and popularity of **data lakes** to process large amounts of both structured and unstructured data.
- Finally, the latest step in the evolution of big data ecosystems is the **lakehouse architecture**., which seeks to combine the strengths of both data lakes and data warehouses.

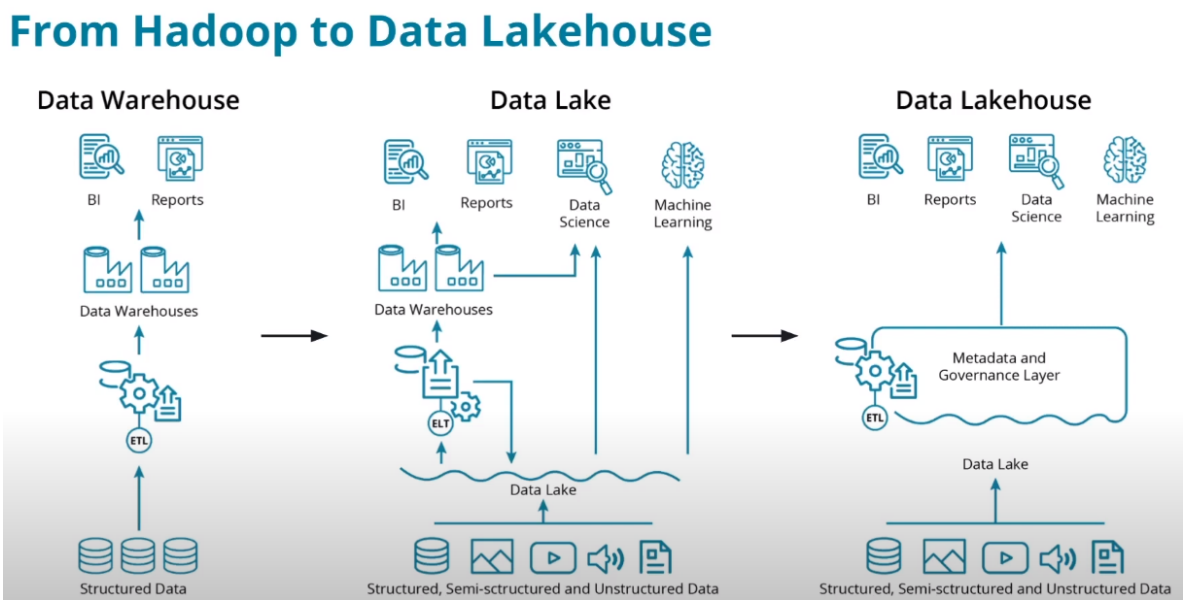


Lesson outline

- Hadoop.
    - Technologies.
    - MapReduce.
- Spark.
    - What is Spark.
    - Spark cluster.
    - When to use Spark.
- Data lakes.
    - What a data lake is.
    - Features of data lakes.
    - Data lakehouse architecture.

From Hadoop to lakehouse architecture

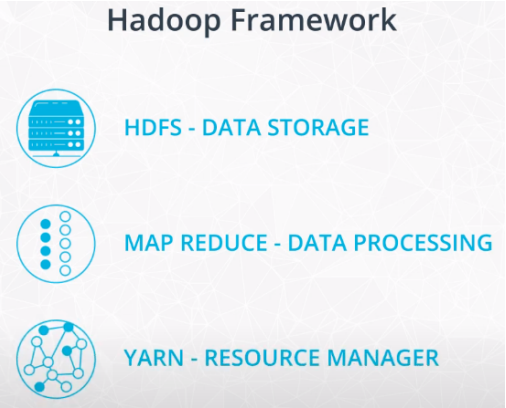
- Hadoop and Spark enabled the evolution of the data warehouse to the data lake.
- **Data warehouses** are based on specific and explicit data structures that allow for highly performant business intelligence and analytics, but they do not perform well with unstructured data.
- **Data lakes** are capable of ingesting massive amounts of both structured and unstructured data with Hadoop and Spark providing processing on top of these datasets.
- Data lakes have several shortcomings that grew out of their flexibility.
    - They are unable to support transactions and perform poorly with changing datasets.
    - Data governance became difficult due to the unstructured nature of these systems.
- Modern **lakehouse architectures** seek to combine the strengths of data warehouses and data lakes into a single, powerful architecture.



The Hadoop ecosystem

Hadoop

- Ecosystem of tools for big data storage and data analysis.
- Older system than Spark but is still used by many companies.
- The major difference between Spark and Hadoop is how they use memory:
    - Hadoop writes intermediate results to disk.
    - Spark tries to keep data in memory whenever possible. This makes Spark faster for many use cases.



Hadoop Distributed File System (HDFS)
A big data storage system that splits data into chunks and stores the chunks across a cluster of computers.

Hadoop MapReduce
A system for processing and analyzing large data sets in parallel.

Hadoop YARN
A resource manager that schedules jobs across a cluster. The manager keeps track of what computer resources are available and then assigns those resources to specific tasks.

Other tools were developed to make Hadoop easier to work with. For example, **Apache Hive** or **Apache Pig**, which both are SQL-like interfaces that run on top of Hadoop MapReduce.

HOW IS SPARK RELATED TO HADOOP?

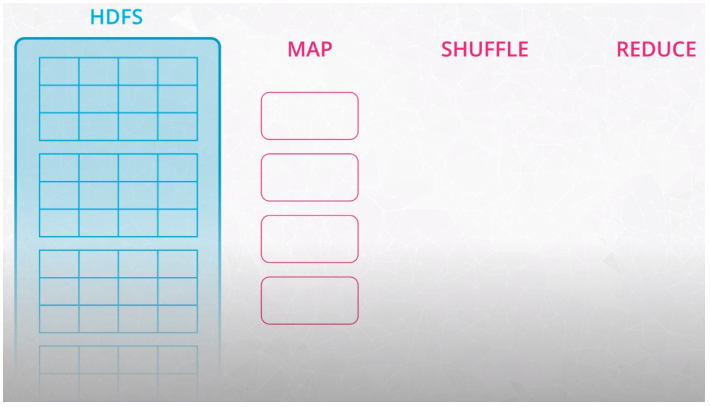
- Spark is another big data framework.
- Contains libraries for data analysis, machine learning, graph analysis, and streaming live data.
- Generally faster than Hadoop, since, unlike Hadoop, which writes intermediate results to disk, Spark tries to keep intermediate results in memory whenever possible.
- Storage system:
    - The Hadoop ecosystem includes a distributed file storage system: HDFS.
    - Spark does not include a file storage system.

- You can use Spark on top of HDFS but you do not have to.
- Spark can read in data from other sources as well.

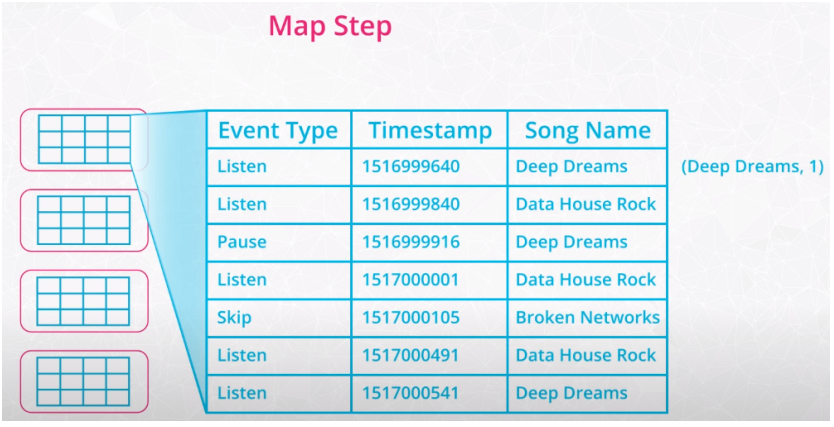
MapReduce

Programming technique for manipulating large data sets. Hadoop MapReduce is a specific implementation of it.

- In the MapReduce job we have 3 well defined steps: map, shuffle and reduce.
- Let's assume that we stored a big log file in HDFS.
- The technique works by first dividing up a large dataset and distributing the data chunks (partitions) across a cluster, so commodity machines can handle the computational load.
    - HDFS takes care of this.

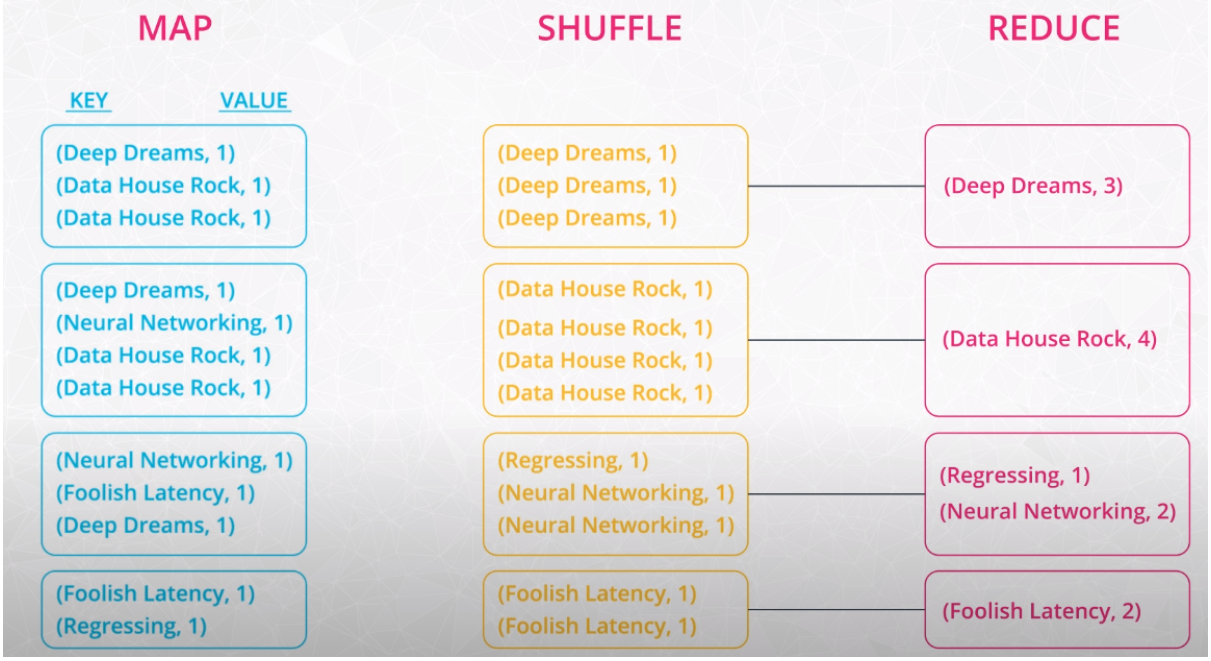


- In the **map** step, each data is analyzed and converted into a **(key, value) pair**.
    - Each map processor is given a partition.
    - Transforms each record in that partition.
    - Then raises modified records to an intermediate file.
    - At the end of the step we have multiple intermediate files containing (key, value) pairs or tuples.



- Then these key-value pairs are **shuffled** across the cluster so that all pairs with the same key are on the same machine.
    - This way, when the node aggregates the values for a key, it can be sure that it has all the corresponding data for each key and can compute the correct final result.

- In the **reduce** step, the values with the same keys are combined together.
- Finally, the count results are written to the output files.



While Spark doesn't implement MapReduce, you can write Spark programs that behave in a similar way to the map-reduce paradigm.

The Spark cluster

- When we talk about distributed computing, we generally refer to a big computational job executing across a cluster of nodes.
- Each node is responsible for a set of operations on a subset of the data.
- At the end, we combine these partial results to get the final answer.

But how do the nodes know which task to run and demote order? Are all nodes equal? Which machine are you interacting with when you run your code?

Most computational frameworks are organized into a **master-worker hierarchy**:

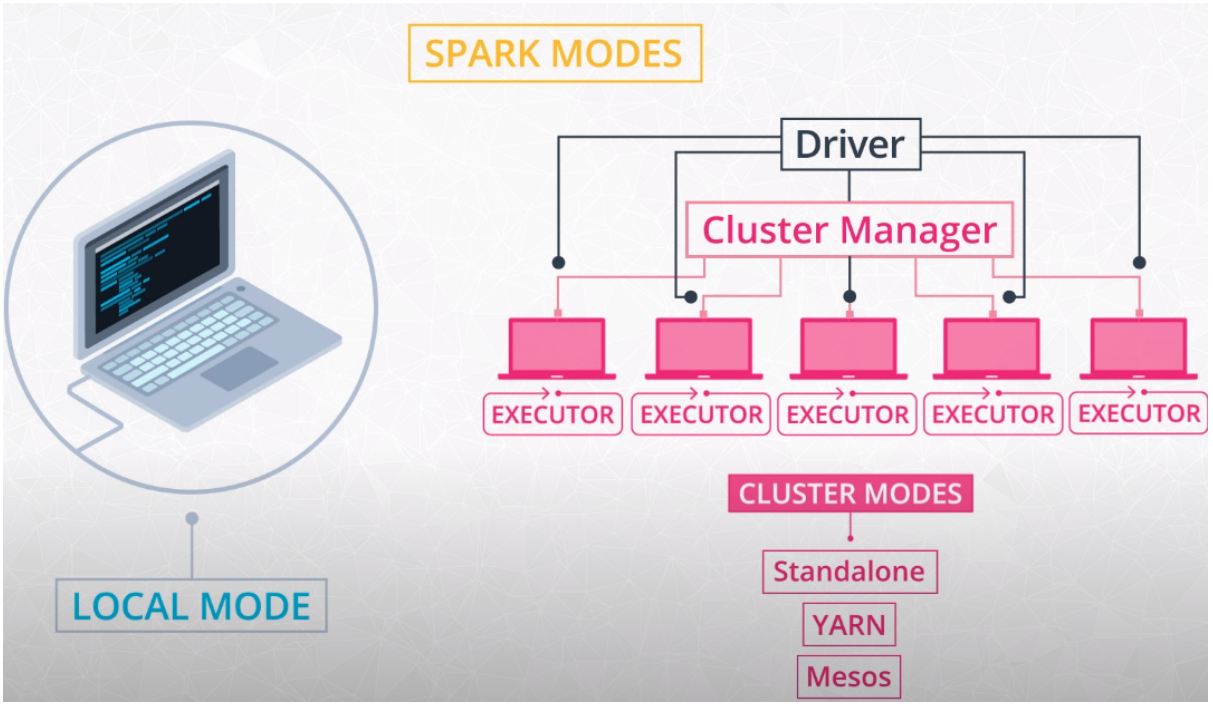
- The master node is responsible for orchestrating the tasks across the cluster.
- Workers are performing the actual computations.

There are **four different modes to setup Spark**:

- **Local mode**
  - Everything happens on a single machine.
  - While we use spark's APIs, we don't really do any distributed computing.
  - Can be useful to learn syntax and to prototype your project.

The other three modes are **distributed** and declare a **cluster manager**. The cluster manager is a separate process that monitors available resources and makes sure that all machines are responsive during the job. There are three different options of cluster managers:

- Spark's own **Standalone Customer Manager** (used in this course).
    - There is a Driver Process.
    - If you open a Spark shell, either Python or Scala, you are directly interacting with the driver program. It acts as the master and is responsible for scheduling tasks.
- **YARN** from the Hadoop project.
- **Mesos**, Another open-source manager from UC Berkeley's AMPLab Coordinators.



Spark use cases

Spark use cases and resources

- Data Analytics.
- Machine Learning.
- Streaming.
- Graph Analytics.

You don’t always need Spark

Spark is meant for big data sets that cannot fit on one computer. But you don't need Spark if you are working on smaller data sets. In the cases of data sets that can fit on your local computer, there are many other options out there you can use to manipulate data, such as:

- AWK - a command line tool for manipulating text files.
- R - a programming language and software environment for statistical computing.

- Python data libraries, which include Pandas, Matplotlib, NumPy, and scikit-learn among other libraries.

Sometimes, you can still use pandas on a single, local machine even if your data set is only a little bit larger than memory. Pandas can read data in chunks. Depending on your use case, you can filter the data and write out the relevant parts to disk.

If the data is already stored in a relational database such as MySQL or Postgres, you can leverage SQL to extract, filter and aggregate the data. If you would like to leverage pandas and SQL simultaneously, you can use libraries such as SQLAlchemy, which provides an abstraction layer to manipulate SQL tables with generative Python expressions.

The most commonly used Python Machine Learning library is scikit-learn. It has a wide range of algorithms for classification, regression, and clustering, as well as utilities for preprocessing data, fine tuning model parameters and testing their results. However, if you want to use more complex algorithms - like deep learning - you'll need to look further. TensorFlow and PyTorch are currently popular packages.

Spark limitations

Spark Streaming's latency is at least 500 milliseconds since it operates on micro-batches of records, instead of processing one record at a time. Native streaming tools such as Storm, Apex, or Flink can push down this latency value and might be more suitable for low-latency applications. Flink and Apex can be used for batch computation as well, so if you're already using them for stream processing, there's no need to add Spark to your stack of technologies.

Another limitation of Spark is its selection of machine learning algorithms. Currently, Spark only supports algorithms that scale linearly with the input data size. In general, deep learning is not available either, though there are many projects that integrate Spark with Tensorflow and other deep learning tools.

Hadoop vs Spark

The Hadoop ecosystem is a slightly older technology than the Spark ecosystem. In general, Hadoop MapReduce is slower than Spark because Hadoop writes data out to disk during intermediate steps. However, many big companies, such as Facebook and LinkedIn, started using Big Data early and built their infrastructure around the Hadoop ecosystem.

While Spark is great for iterative algorithms, there is not much of a performance boost over Hadoop MapReduce when doing simple counting. Migrating legacy code to Spark, especially on hundreds of nodes that are already in production, might not be worth the cost for the small performance boost.

Beyond Spark for storing and processing Big Data

Keep in mind that Spark is not a data storage system, and there are a number of tools besides Spark that can be used to process and analyze large datasets.

Sometimes it makes sense to use the power and simplicity of SQL on big data. For these cases, a new class of databases, known as NoSQL and NewSQL, have been developed.

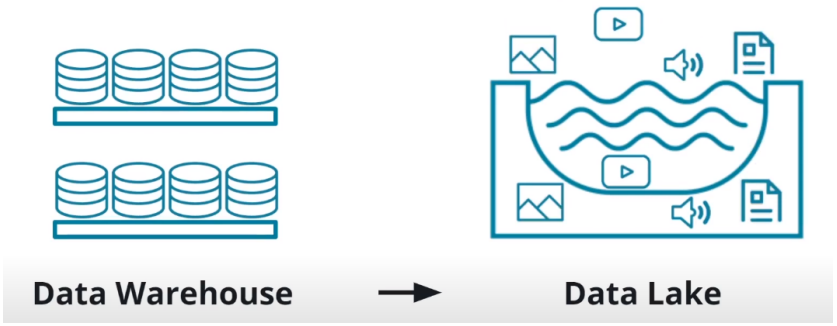
For example, you might hear about newer database storage systems like HBase or Cassandra. There are also distributed SQL engines like Impala and Presto. Many of these technologies use query syntax.

Data Lakes

Data lakes are an evolution beyond data warehouses and allow an organization to ingest massive amounts of both structured and unstructured data into storage.

One of the key differences between data warehouses and data lakes is the inclusion of structured versus unstructured data.

- Data warehouses consist of only highly structured data that is suitable for business intelligence and reporting needs.
- Often data science and machine learning efforts need access to all data, including unstructured data.
- Data lakes provide the ability to serve up both types of data from a single data store.



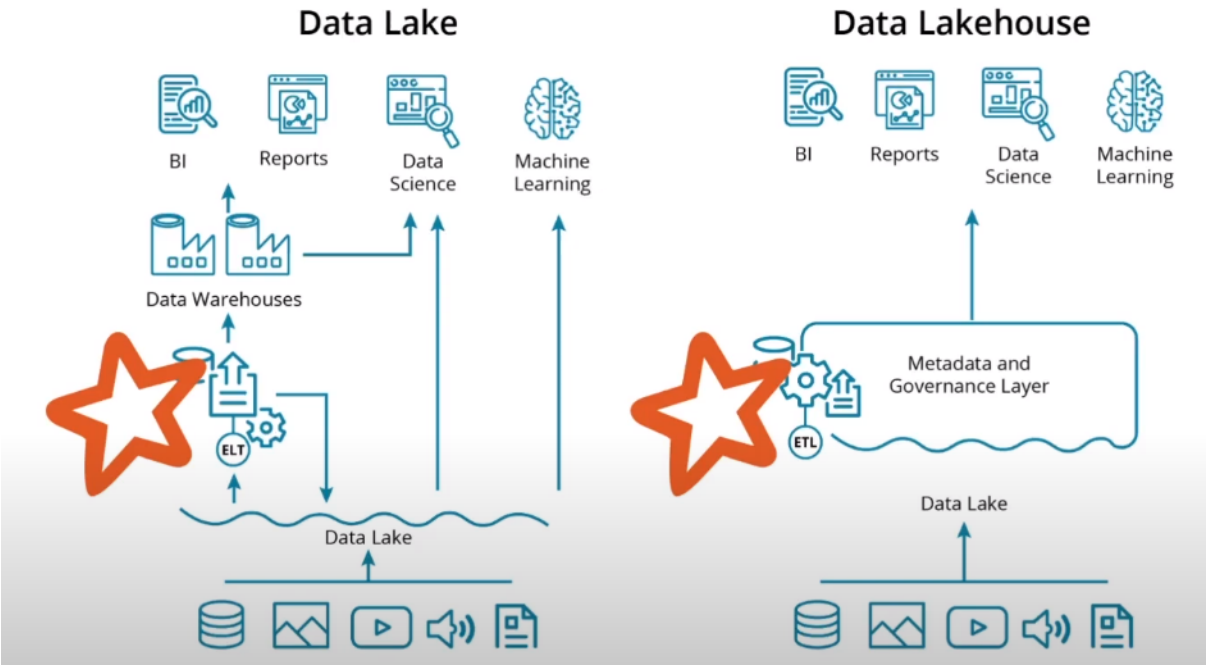
The **key features of data lakes** include:

- Lower costs associated with using big data tools for ETL / ELT operations.
- Data lakes provide schema-on-read rather than schema-on-write which lowers the cost and work of ingesting large amounts of data.
- Data lakes provide support for structured, semi-structured, and unstructured data.

Data Lakes, Lakehouse and Spark

Apache Spark can be used to perform data engineering tasks for building both data lakes and lakehouse architectures. Most often, these data engineering tasks consist of ETL or ELT tasks. Apache Spark allows the data engineer to perform these tasks, along with raw data

ingestion using the language of their choice with support for Python, R, SQL, Scala, and Java.



Data Lakehouse architecture

Data lakes were a huge step forward from **data warehouses,** but some of their key features led to weaknesses. With the need to ingest a large amount of unstructured data, **we lost**:

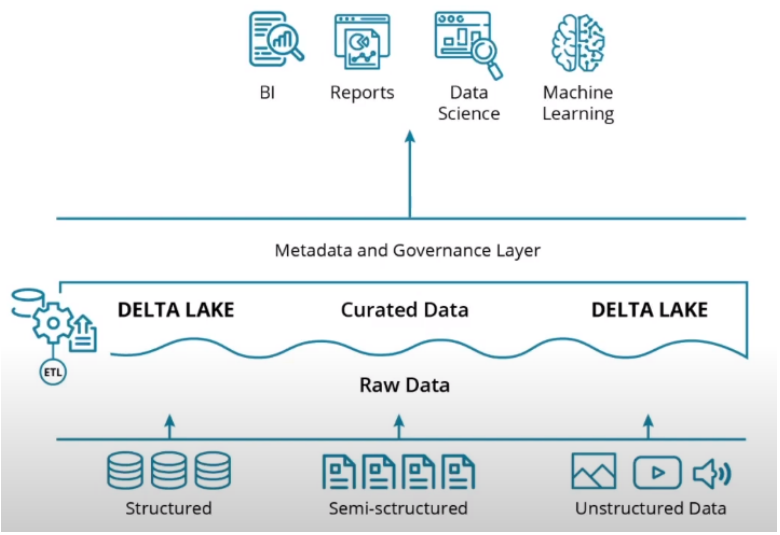
- Atomic transactions: failed production jobs left data in a corrupted state.
- Quality enforcement: inconsistent and therefore unusable data.
- Consistency in data structures: impossible to stream and batch process data while ingesting.

These shortcomings led the industry to seek better solutions.

Lakehouse Architecture

The key innovation of the lakehouse architecture is the creation of a metadata and data governance layer on top of the data lake.

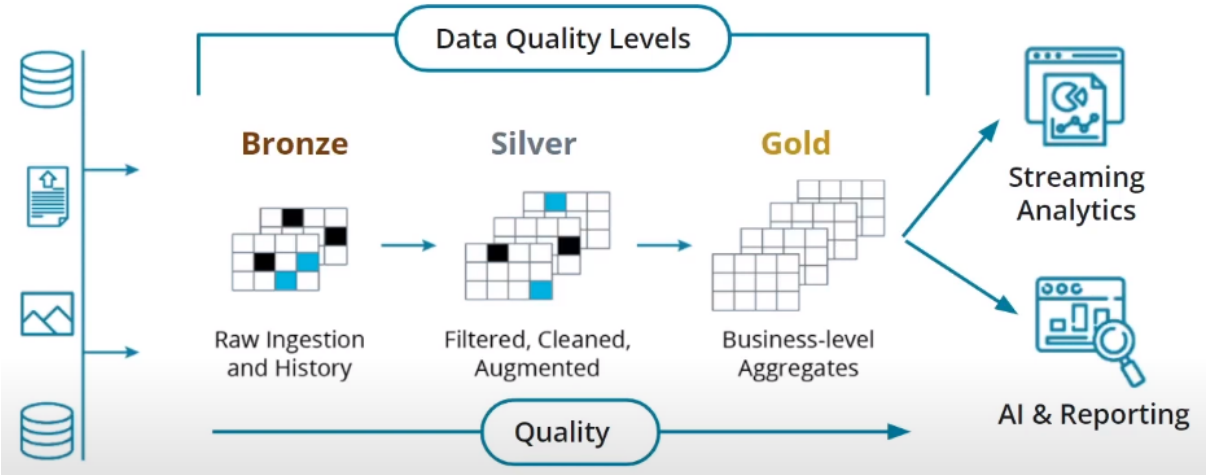
- This creates a pool of raw data as well as a curated set of data.
- This provides the flexibility and benefits we previously saw with data lakes, and it also provides solutions to the weaknesses in data lakes.



Lakehouse Architecture Features

One of the important features of a lakehouse architecture is the ability to quickly ingest large amounts of data and then incrementally improve the quality of the data. We can think of this process using the colors we often associate with Olympic medals.

- Raw ingested data can be considered bronze.
- After some filtering, cleaning, and augmenting, the data can be considered silver.
- Finally, with the addition of business-level aggregates such as you might see with a star schema, data can be considered gold and ready for analytics and reporting needs.



Differences Between Data Lakes and Lakehouse Architecture

The key difference between data lakes and data lakehouse architectures is the inclusion of the metadata and governance layer. This is the crucial ingredient that provides atomicity, data quality, and consistency for the underlying data lake.

3.3 Spark essentials

Useful links

[Spark docs]
[Spark cluster overview]
[Spark Python API docs]
[Spark SQL guide]
[Spark SQL built-in functions]
[RDD programming guide]
[RDDs vs Dataframes and Datasets]
[HDFS architecture]

Lesson outline

- Introduction to Spark.
    - Spark directed acyclic graph (DAG).
    - Resilient Distributed Datasets (RDD).
- Maps and lambda functions.
    - SparkContext object.
    - Lazy evaluation.
    - Anonymous lambda functions.
- Data wrangling with Spark.
    - Reading and writing data into Spark DataFrames.
    - Imperative programming (Python DataFrames) vs declarative programming (SQL).
    - DataFrame functions.
- Spark SQL.
    - Querying with SQL.
    - DataFrames vs SQL.

The Spark DAG

Apache Spark distributes data processing tasks over a cluster of distributed computing resources. How does it accomplish this?

An **idempotent program** can run multiple times without any effect on the result. Some programs depend on prior state in order to execute properly. This is not considered idempotent, because they depend on that state existing before starting.

Dividing the Work
One goal of idempotent code is that data can be processed in parallel, or simultaneously. This is done by calling the same code repeatedly in different threads, and on different nodes

or servers for each chunk or block of data. If each program has no reliance on prior execution, there should be no problem splitting up processing.

When writing Spark code, it is very important to avoid reading data into regular lists or arrays, because the amount of data your code deals with can be very large. Instead, you will use special datasets called **Resilient Distributed Datasets (RDDs)** and **DataFrames**. Much like an SQL query cursor, they don't actually hold all of the data in memory. These datasets give your Spark job access to the shared resources of the cluster in a very controlled way that is managed outside of your Spark job.

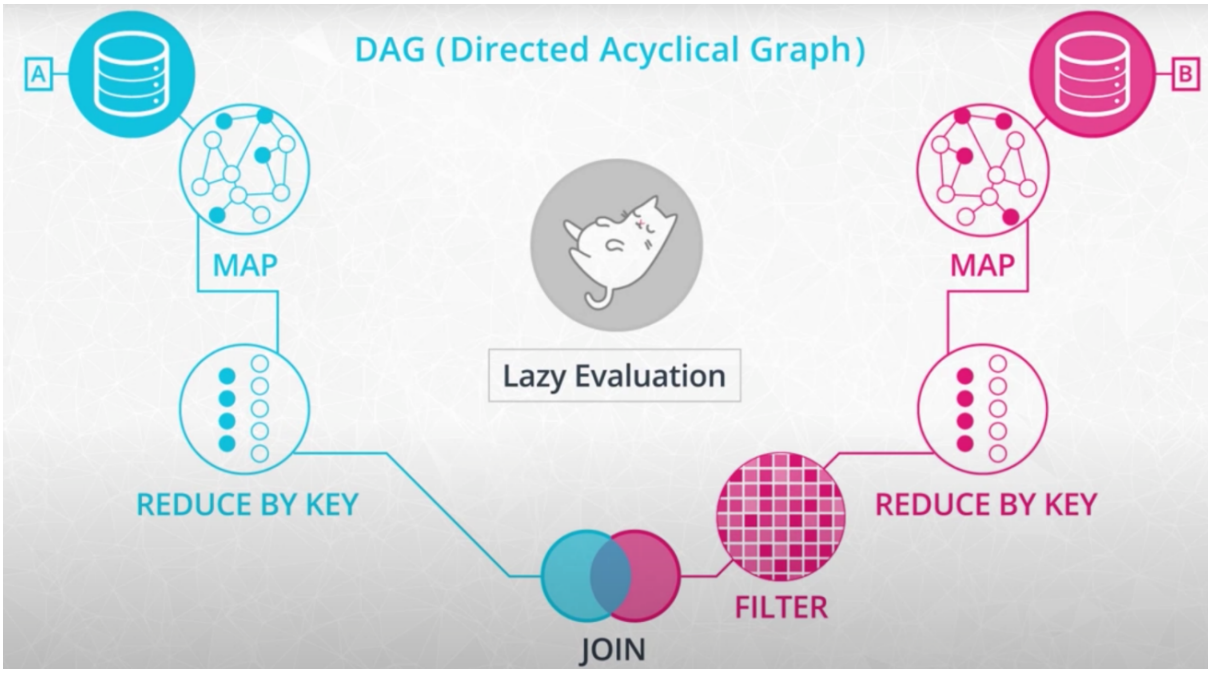
Directed Acyclic Graph (DAG)

Every Spark program makes a copy of its input data and never changes the original parent data. Because Spark doesn't change or mutate the input data, it's known as immutable. But what happens when you have lots of function calls in your program?

- In Spark, you do this by chaining together multiple function calls that each accomplish a small chunk of the work.
- It may appear in your code that every step will run sequentially.
- However, they may be run more efficiently if Spark finds a more optimal execution plan.

Spark uses a programming concept called **lazy evaluation**. Before Spark does anything with the data in your program, it first builds step-by-step directions of what functions and data it will need.

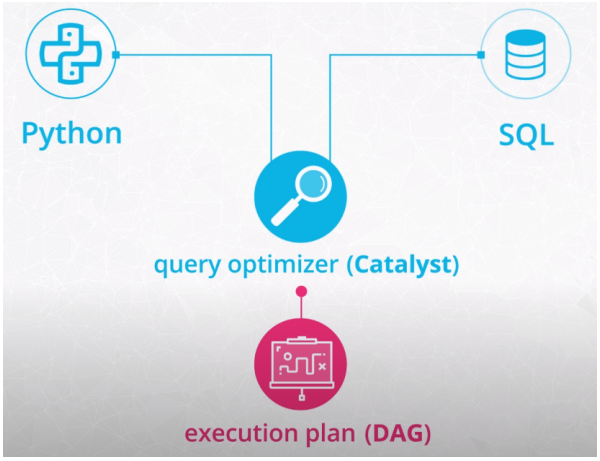
In Spark, and in other similar computational processes, this is called a Directed Acyclic Graph (DAG). The reference is made to the fact that no explicit repetition is inherent in the process. For example, if a specific file is read more than once in your code, Spark will only read it one time. Spark builds the DAG from your code, and checks if it can procrastinate, waiting until the last possible moment to get the data.



Resilient Distributed Datasets (RDDs)

- Spark processes data using a cluster of distributed computing resources.
- Source data is loaded from a database, CSV files, JSON files, or text files.
- Spark converts this data into an immutable collection of objects for distributed processing called a **Resilient Distributed Dataset** or **RDD**.
- The features of the RDD are designed for efficient data processing:
    - Fault-tolerant (recomputed in case of failure).
    - Cacheable.
    - Partitioned.
- RDDs are a low-level abstraction of the data.
- You can think of RDDs as long lists distributed across various machines.
- In the first version of Spark, you had to work directly with RDDs.
- You can still use RDDs as part of your Spark code although working with DataFrames and SQL is easier.

We can do data wrangling in two different ways: by using Python and SQL. The code we write in these higher level APIs first goes through a query optimizer (Spark's optimizer is called *Catalyst*) to turn it into an actual execution plan before it can be run. Under the hood, Catalyst will translate our code to the same DAG. So when we run our application, we won't notice much difference between using Python or SQL. The code generated based on the execution plan operates on RDDs.



Pyspark and SparkSession

- Python is one of many languages you can use to write Spark Jobs.
- If you choose to use Python, then you will use the **PySpark** library.
- PySpark gives you access to all the important Spark data constructs like:

    - RDDs.
    - DataFrames.
    - Spark SQL.
- You can write Spark code that runs in either a Spark Cluster, in a Jupyter Notebook, or on your laptop.
- When you write code on your Jupyter Notebook or a laptop, Spark creates a temporary Spark node that runs locally.
- Spark uses Java, so it is necessary to install the **JDK** on a computer used to run PySpark code.

The SparkSession

- **SparkContext**: main entry point for Spark functionality; connects the cluster with the application.
- To create a SparkContext, we first need a **SparkConf** object to specify some information about the application, such as:
    - Name.
    - Master's nodes' IP address.
        - If we run Spark in local mode, we can just put the string *"local"* as master.
- To read data frames, we need to use Spark SQL equivalent, the **SparkSession**. We can specify some parameters to create a SparkSession.
    - **getOrCreate()**, for example, means that if you already have a SparkSession running, instead of creating a new one, the old one will be returned and its parameters will be modified to the new configurations.

Maps and Lambda Functions in Spark

One of the most common functions in Spark is **map**. It simply makes a copy of the original input data and transforms that copy according to whatever function you pass to map. You can think of it as directions for the data telling each input how to get to the output.

Data formats

The most common data formats you might come across are CSV, JSON, HTML, and XML.

Distributed data stores

When we have so much data that we need distributed computing, the data itself often needs to be stored in a distributed way.

Distributed file systems, storage services, and distributed databases store data in a **fault-tolerant** way. So if a machine breaks or becomes unavailable, we don't lose the information we have collected.

Hadoop has a Distributed File System, **HDFS**, to store data. HDFS splits files into 64 or 128 megabyte blocks and replicates these blocks across the cluster. This way, the data is stored in a fault-tolerant way and can be accessed in digestible chunks.

On your local machine, Spark simulates a distributed file store. If you are working on a project or in a company that runs its own cluster resources, they might use HDFS.

If you're working on the cloud, you can use a distributed data storage service on a cloud provider. Examples of distributed data services on cloud providers include:

- Amazon Simple Storage Service, or S3.
- Azure Blob Storage.
- Google Cloud Storage.

Imperative vs declarative programming

Imperative programming

- Concerned about the “how”.
- Focus on the exact steps, how we get to the result.
- Data transformations with DataFrames.

Declarative programming

- Cares about the “what”.
- Concerned about the result we want to achieve.
- Abstraction layer of an imperative system.

Data wrangling with DataFrames tips

General functions

- **select()**: returns a new DataFrame with the selected columns.
- **filter()**: filters rows using the given condition.
- **where()**: is just an alias for filter().
- **groupBy()**: groups the DataFrame using the specified columns, so we can run aggregation on them.
- **sort()**: returns a new DataFrame sorted by the specified column(s). By default the second parameter 'ascending' is True.
- **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 column or replacing the existing column that has the same name. The first parameter is the name of the new column, the second is an expression of how to compute it.

Aggregate functions
Spark SQL provides built-in methods for the most common aggregations in the **pyspark.sql.functions** module, such as:

- **count()**
- **countDistinct()**
- **avg()**
- **max()**
- **min()**
- ….

These methods are not the same as the built-in methods in the Python Standard Library, where we can find min() for example as well, hence you need to be careful not to use them interchangeably.

In many cases, there are multiple ways to express the same aggregations:

- Aggregate method after a groupBy().
- **agg()** when using different functions on different columns.
    - Example: agg({"salary": "avg", "age": "max"}).

User defined functions (UDF)
In Spark SQL we can define our own functions with the **udf** method from the **pyspark.sql.functions** module. The default type of the returned variable for UDFs is string. If we would like to return another type we need to explicitly do so by using the different types from the **pyspark.sql.types** module.

Window functions
Window functions are a way of combining the values of ranges of rows in a DataFrame. When defining the window we can choose how to sort and group (with the **partitionBy** method) the rows and how wide of a window we'd like to use (described by **rangeBetween** or **rowsBetween**).

Spark SQL

Used to query data with a declarative approach. Spark comes with a SQL library that lets you query DataFrames using the same SQL syntax you'd use in a tool like MySQL or Postgres.

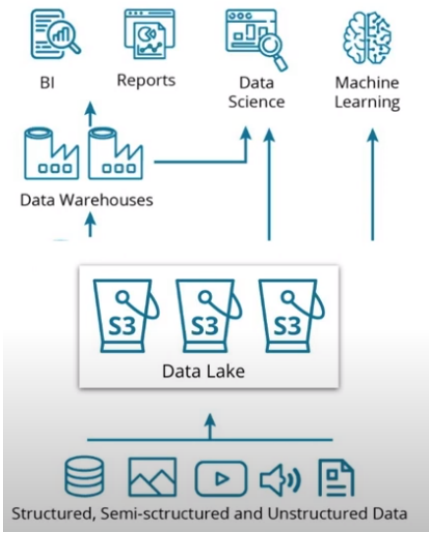
Spark automatically optimizes your SQL code, to speed up the process of manipulating and retrieving data.

3.4 Using Spark in AWS

Lesson outline

- Data Lakes in AWS.
    - Structured and unstructured data.
    - AWS S3 buckets as data lakes to store data for Spark jobs.
- Using Spark on AWS.
    - Spark with the AWS EMR service.
    - EC2 instances to run Spark jobs.
    - AWS Glue Service to run Spark jobs.
- AWS Glue.
    - Components needed to to set up and configure AWS Glue.
    - How to use Spark in Glue to process data with Spark Glue jobs.
    - Create and run Glue jobs.

Data Lakes in AWS



- Data Lakes are not a specific technology. They can be implemented using many types of file storage systems.
- In AWS, the most common way to store files is S3, so we can implement data lakes using S3 storage.

S3 Data Lakes

- Similar to HDF, AWS created S3, the Simple Storage Service, whose top level is **S3 Bucket.**
- S3 does not require the maintenance required by most file storage systems.
- Almost unlimited storage capacity and very inexpensive → ideal location to store data for our data lake.

    - Much cheaper than other more sophisticated data locations, such as RDS, EC2.
    - No computational cost associated with S3 by default.

Using Spark on AWS

Several choices for running Spark on AWS:

- **EMR**
    - AWS managed Spark service which makes use of a scalable set of EC2 machines already configured to run Spark.
    - The user only needs to configure the necessary cluster resources.
- **EC2**
    - Self-managed Spark.
    - Use EC2 machines and install and configure Spark and HDFS yourself.
- **Glue**
    - Serverless Spark environment with added libraries (such as Glue context and Glue Dynamic Frames).
    - It also interfaces with other AWS data services like Data Catalog and AWS Athena.

We will focus on AWS Glue to run Spark scripts.

	AWS EMR	AWS Glue	Self Managed Spark
Distributed Computing	Yes	Yes	Yes
Serverless	No	Yes	No
HDFS Installed	Yes	Yes	Yes
Billing Model	EC2 costs	Job Duration	EC2 costs
Provisioning	Automated	Zero Provisioning	Self Installed

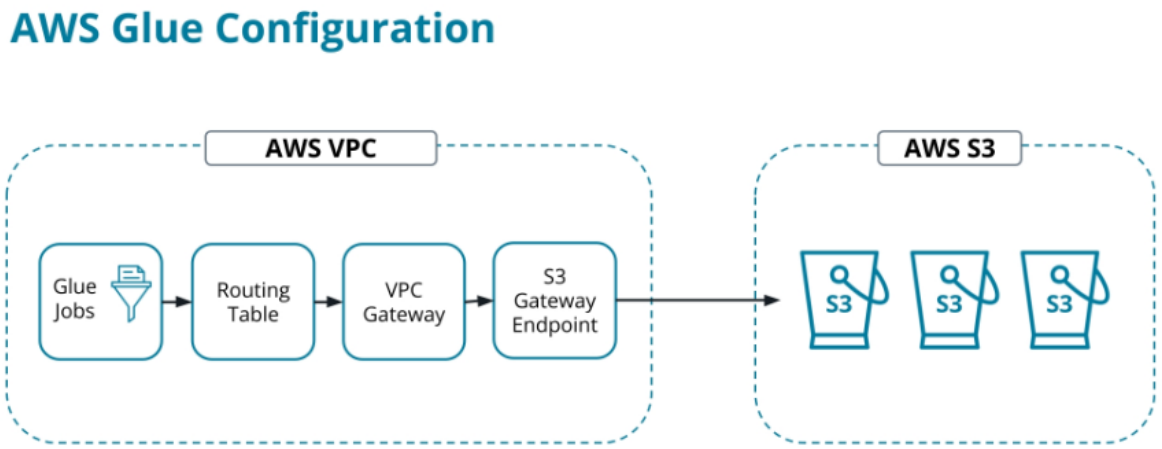
MapReduce system
Hadoop Distributed File System (HDFS) uses MapReduce system as a resource manager to allow the distribution of the files across the hard drives within the cluster. Think of it as the MapReduce System storing the data back on the hard drives after completing all the tasks.

Spark, on the other hand, runs the operations and holds the data in the RAM memory rather than the hard drives used by HDFS. Spark, unlike Hadoop, lacks a file distribution system to organize, store and process data files, so Spark tools are often installed on Hadoop and can make use of the HDFS.

Why would you use an EMR cluster?
Since a Spark cluster includes multiple machines, in order to use Spark code on each machine, we would need to download and install Spark and its dependencies. This is a manual process. **AWS EMR** is a service that negates the need for you, the user, to go through the manual process of installing Spark and its dependencies for each machine.

Introduction to AWS Glue

Glue is an AWS service that relies on Spark. Using AWS Glue requires the following resources and configuration:



Routing table

- Entity that stores the network paths to various locations (e.g., path to S3 from within your VPC).
- You will need a routing table to configure with your VPC Gateway.

VPC Gateway

- Your cloud project runs resources within a Virtual Private Cloud (VPC).
- This means your Glue job runs in a Secure Zone without access to anything outside your Virtual Network (unless you specifically require resources on the internet).
- A VPC Gateway is a network entity that gives access to outside networks and resources.
- S3 is a shared service, so it does not reside in your VPC.

S3 Gateway Endpoint

- By default, Glue Jobs can't reach any networks outside of your Virtual Private Cloud (VPC).
- Since the S3 Service runs in a different network, we need to create what is called an S3 Gateway Endpoint.
- This allows S3 traffic from your Glue Jobs into your S3 buckets.
- Once you have created the endpoint, your Glue Jobs will have a network path to reach S3.

S3 Buckets

- Storage locations within AWS, that have a hierarchical directory-like structure.
- Once you create an S3 bucket, you can create as many sub-directories, and files as you want.
- The bucket is the "parent" of all of these directories and files.

HDFS vs AWS S3

Since Spark does not have its own distributed storage system, it leverages HDFS or AWS S3, or any other distributed storage. Let's see some differences between these two systems:

- **AWS S3** is an **object storage system** that stores the data using key value pairs, and **HDFS** is an **actual distributed file system** that guarantees fault tolerance. HDFS achieves fault tolerance by duplicating the same files at 3 different nodes across the cluster by default (it can be configured to reduce or increase this duplication).
- **HDFS** has traditionally been installed in on-premise systems which had engineers on-site to maintain and troubleshoot the Hadoop Ecosystem, **costing more than storing data in the cloud.** Due to the flexibility of location and reduced cost of maintenance, cloud solutions have been more popular. With the extensive services AWS provides, S3 has been a more popular choice than HDFS.
- Since **AWS S3** is a **binary object store**, it can store **all kinds of formats**, even images and videos. **HDFS** strictly **requires a file format** - the popular choices are avro and parquet, which have relatively high compression rates making it useful for storing large datasets.

3.5 Ingesting and organizing data in a Lakehouse

Lesson outline

- Lakehouse architecture.
    - Lakehouse zones.
    - How to build and configure lakehouse zones on the AWS platform.
    - How to use ETL and ELT to move and process data into these zones.
    - Personally Identifiable Information (PII) and privacy issues.
- Streaming data.
    - Streaming message brokers.
    - Streaming data lakehouse lifecycle.
- Curated data.
    - Data attributes.
    - Lakehouse zones.
    - Processing.

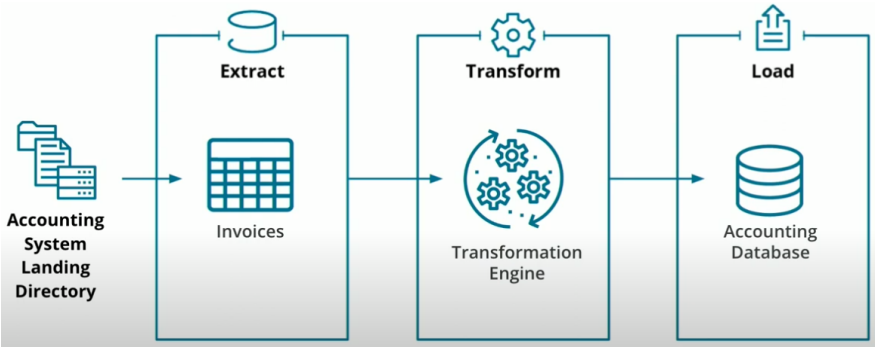
Lakehouse architecture

Lakehouse is another evolution of data storage. The purpose of a Lakehouse is to separate data processing into stages. Like an oil refinery, data is staged and processed step by step until it becomes available for querying.

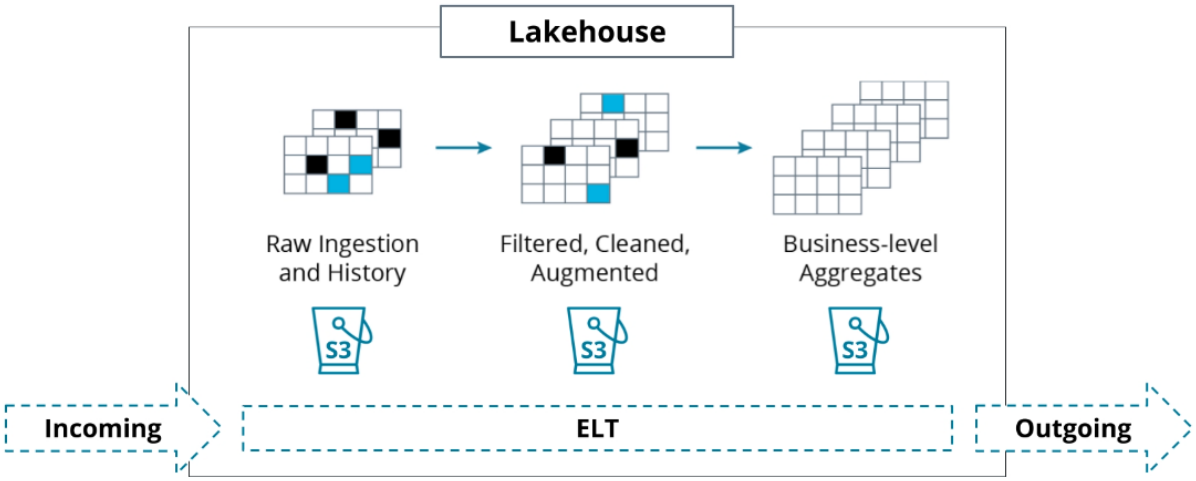
Lakehouse is not a specific technology. It can be implemented using any file storage and processing layer. In AWS, the most common way to store files is in S3, so we can implement the Lakehouse using S3 storage.

ETL vs ELT

Think about our invoice example earlier. An accounting system is the destination for the files in the landing directory. That accounting system is responsible for extracting the invoices from the invoice files, transforming them into the correct format, and loading them into the accounting database where they can be paid. These steps are known as ETL (Extract, Transform, and Load). With ETL, usually data is going from a semi-structured (files in directories) format to a structured format (tables).



With ELT, however, and with a Lakehouse, the data stays in semi-structured format, and the last zone contains enriched data where it can be picked up for processing later. Deferring transformation is a hallmark of Data Lakes and Lakehouses. In this way, keeping the data at multiple stages in file storage gives more options for later analytics, because it preserves all of the format. What if the accounting system had a defect that didn't load data properly for certain vendors. The original data is still available to be analyzed, transformed, and re-processed as needed.



Lakehouse zones

- Data is ingested from the source data lake into the **landing zone** as is. The data goes through processing for validation and is stored in the **raw zone** for historical long-term storage.
- The data from the raw zone goes through processing to bring it into a conformed state and is stored in the **trusted zone**. This layer of processing applies the schema, partitioning and other transformations to the data.
- As the last step, the data is processed in the trusted zone by modeling it and joining it with other datasets to create a curated dataset and store it in the **curated zone**. Datasets from the curated layer are then ingested into a Data Warehouse or other data analytics endpoints that serve use cases with low latency access requirements or need to run complex SQL queries.

Source

Streaming data

Streaming Message Brokers

Spark is intended to process data that was previously generated. It doesn't process data in real time. **Spark Streaming** gives us the option of processing data in near real-time. In this course, we won't go into Spark Streaming in-depth, but we'll cover it briefly for awareness.

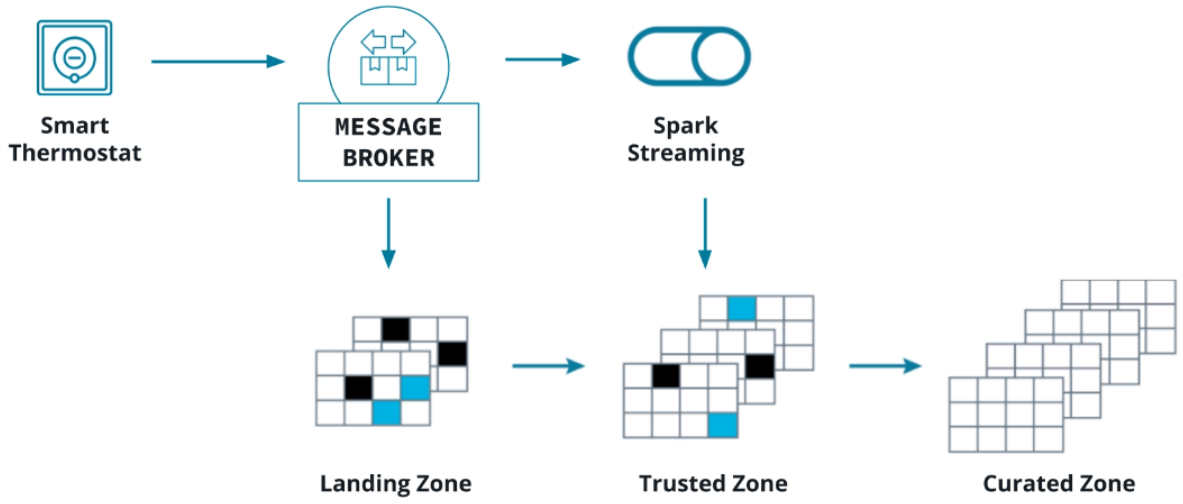
Often data is generated in real-time and then is stored for later processing. A common example is IoT or Internet of Things, which consists of small devices sending internet-connected messages. These devices send messages to convey meaning about the world. A smart thermostat is an IoT device. It continually communicates back with a server to send usage statistics to the end user.

Because servers are not always designed to handle large volumes of real-time data, **message brokers** were created. They are intended to "broker" connections between systems and make near real-time processing of data possible.

Some examples of message brokers are:

- Kafka.
- Simple Queue Services (AWS SQS).
- Amazon Kinesis.

However, the data message brokers store doesn't last forever. Brokers are intended to facilitate a message being received and re-transmitted. This event typically should happen within seven days. Then the data will be deleted from the Raw Zone. To keep messages longer, we move them into a Landing Zone. This is where the data can be loaded and transformed for later use in the Trusted and Curated Zone.



Using Glue to Process Streaming Data

Glue can load data directly from Kafka or Kinesis. AWS doesn't offer Glue support for SQS at this time. Using Spark Streaming, we can load data from Message Brokers into a Spark DataFrame or Glue DynamicFrame.

We can then join data from the Message Broker with other data sources as part of the streaming job to create Trusted or Curated data. Kafka can be configured to load data into S3 using a Kafka Connector as a Landing Zone, avoiding the need to connect Glue to Kafka directly.

Curated data

Curated Data Attributes

- High quality (meeting organizational data standards).
- Filtered for privacy (PII data should be removed and data should only be stored with consent).
- Can be a composition of multiple data sources with these data qualities.

We can join multiple trusted data sources, and apply other transformations to create curated data.

4 Automate Data Pipelines

4.1 Data Pipelines

Lesson outline

Directed Acyclic Graphs (DAGs).

- Data validation.
- How Apache Airflow uses DAGs.
- How Airflow works.
- Schedules in Airflow.
- Operators and Tasks in Airflow.
- Context and Templating in Airflow.

DAGs and Data Pipelines

Data pipeline
A data pipeline describes, in code, a series of sequential data processing steps. Depending on the data requirements for each step, some steps may occur in parallel. Data pipelines also typically occur on a schedule. Extract, transform and load (ETL), or extract, load, and transform (ELT), are common patterns found in data pipelines, but not strictly required. Some data pipelines perform only a subset of ETL or ELT.

What is a graph?
Graphs describe entities and the relationships that exist between them.

Directed Acyclic Graphs (DAGs)
DAGs are a special subset of graphs in which the edges between nodes have a specific direction, and no cycles exist (i.e., the nodes can't create a path back to themselves).

Node
A step in the data pipeline process.

Edge
Dependencies or other relationships between nodes.

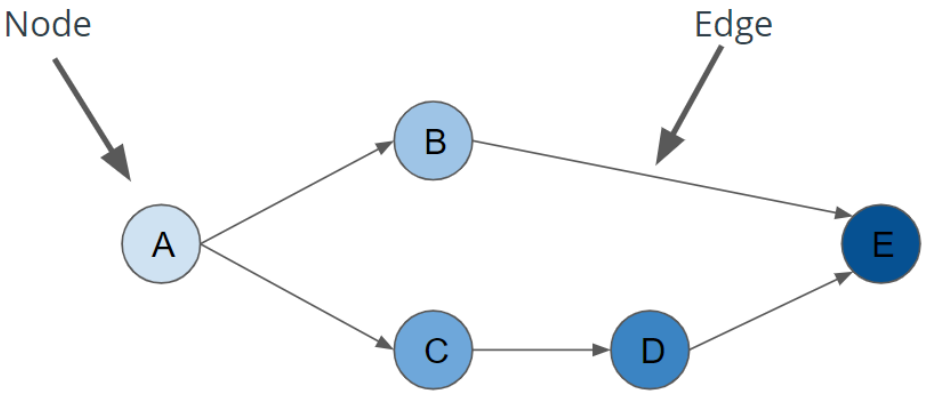


Diagram of a Directed Acyclic Graph

Data pipelines as DAGs
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.

Data validation

Process of ensuring that data is present, correct and meaningful. Ensuring the quality of your data through automated validation checks is a critical step in building data pipelines at any organization.

Examples of validation steps (bikesharing)
After loading from S3 to Redshift:

- Validate the number of rows in Redshift match the number of records in S3.

Once location business analysis is complete:

- Validate that all locations have a daily visit average greater than 0.
- Validate that the number of locations in our output table match the number of tables in the input table.

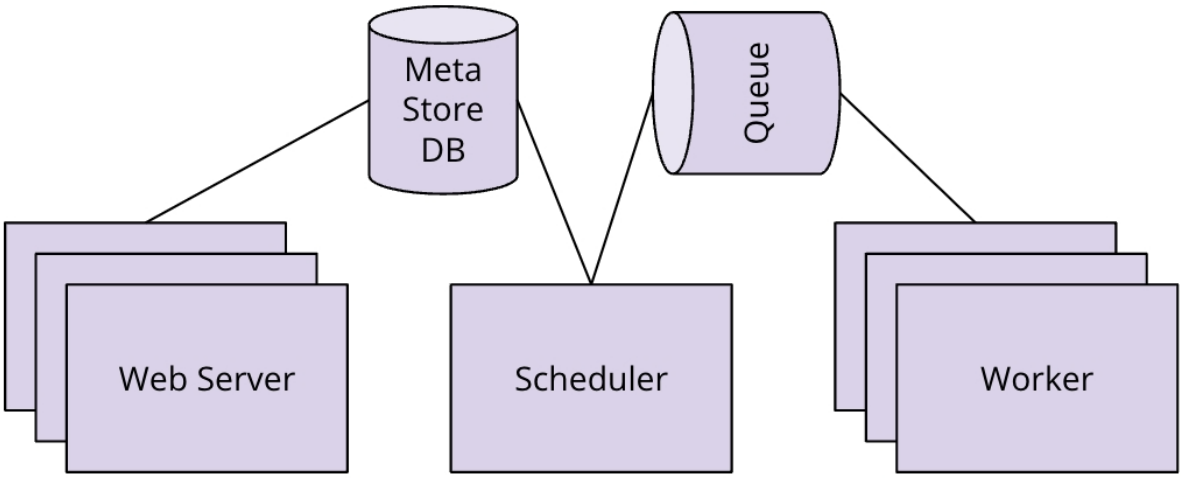
Introduction to Apache Airflow

- Open-source tool which structures data pipelines as DAGs.
- Allows users to write DAGs in Python that run on a schedule and/or from an external trigger.
- It's simple to maintain and can run data analysis itself, or trigger external tools (Redshift, Spark, Presto, Hadoop, etc) during execution.
- It provides a web-based UI for users to visualize and interact with their data pipelines.

Apache Airflow website.

How Airflow works

Components of Airflow



Airflow component diagram

- **Scheduler** orchestrates the execution of jobs on a trigger or schedule. The Scheduler chooses how to prioritize the running and execution of tasks within the system. You can learn more about the Scheduler from the official Apache Airflow documentation.
- **Work queue** is used by the scheduler in most Airflow installations to deliver tasks that need to be run to the **workers**.
- **Worker** processes execute the operations defined in each DAG. In most Airflow installations, workers pull from the **work queue** when it is ready to process a task. When the worker completes the execution of the task, it will attempt to process more work from the **work queue** until there is no further work remaining. When work in the queue arrives, the worker will begin to process it.
- **Metastore database** saves credentials, connections, history, and configuration. The database, often referred to as the metadata database, also stores the state of all tasks in the system. Airflow components interact with the database with the Python ORM, SQLAlchemy.
- **Web interface** provides a control dashboard for users and maintainers.

Order of operations for an Airflow DAG

1. The Airflow scheduler starts DAGs based on time or external triggers.
2. Once a DAG is started, the scheduler looks at the steps within the DAG and determines which steps can run by looking at their dependencies.
3. The scheduler places runnable steps in the queue.
4. Workers pick up those tasks and run them.
5. 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.
6. Once all tasks have been completed, the DAG is complete.

4.2 Airflow and AWS

Lesson outline

Configuration:

- AWS IAM.
- AWS Redshift Serverless.
- AWS S3.

Connection:

- Airflow to AWS using IAM credentials.
- Airflow to AWS Redshift Serverless.
- Airflow to S3.

Querying Redshift data using AWS Console.

4.3 Data Quality

Lesson outline

- Data lineage and the movement of data in our pipelines.
- Scheduling, analysis and catchup.
- Data partitioning.
- Requirements and determining and measuring data quality.

Data lineage

Definition
The data lineage of a dataset describes the discrete steps involved in the creation, movement, and calculation of that dataset.

Why is Data Lineage important?

- **Instilling confidence**: being able to describe the data lineage of a particular dataset or analysis will build confidence in data consumers (engineers, analysts, data scientists, etc.) that our data pipeline is creating meaningful results using the correct datasets. If the data lineage is unclear, it's less likely that the data consumers will trust or use the data.
- **Defining metrics**: Another major benefit of surfacing data lineage is that it allows everyone in the organization to agree on the definition of how a particular metric is calculated.
- **Debugging**: Data lineage helps data engineers track down the root of errors when they occur. If each step of the data movement and transformation process is well described, it's easy to find problems when they occur.

In general, data lineage has important implications for a business. Each department or business unit's success is tied to data and to the flow of data between departments. For e.g., sales departments rely on data to make sales forecasts, while at the same time the finance department would need to track sales and revenue. Each of these departments and roles depend on data, and knowing where to find the data. Data flow and data lineage tools enable data engineers and architects to track the flow of this large web of data.

Visualizing data lineage

- Task dependencies in Airflow can be used to reflect and track data lineage.
- Another way is to trace the input and output data at every step of the way.
- In Airflow, the graph view of a DAG can let us see the success or failures of individual tasks.

Data pipeline schedules

Data in time ranges
Pipeline schedules can be used to enhance our analyses by allowing us to make assumptions about the scope of the data we're analyzing. What do we mean when we say the scope of the data?

When a data pipeline is designed with a schedule in mind, we can use the execution time of the pipeline run to only analyze data from the time period since this data pipeline last ran.

In a naive analysis, with no scope, we would analyze all of the data at all times.

Schedules
Pipelines are often driven by schedules which determine what data should be analyzed and when.

Why Schedules:

- Pipeline schedules can reduce the amount of data that needs to be processed in a given run. It helps scope the job to only run the data for the time period since the data pipeline last ran. In a naive analysis, with no scope, we would analyze all of the data at all times.
- Using schedules to select only data relevant to the time period of the given pipeline execution can help improve the quality and accuracy of the analyses performed by our pipeline.
- Running pipelines on a schedule will decrease the time it takes the pipeline to run.
- An analysis of larger scope can leverage already-completed work. For. e.g., if the aggregates for all months prior to now have already been done by a scheduled job, then we only need to perform the aggregation for the current month and add it to the existing totals.

Selecting the time period
Determining the appropriate time period for a schedule is based on a number of factors which you need to consider as the pipeline designer.

1. **What is the size of data, on average, for a time period?** If an entire year's worth of data is only a few kb or mb, then perhaps it's fine to load the entire dataset. If an hour's worth of data is hundreds of mb or even in the gbs then likely you will need to schedule your pipeline more frequently.
2. **How frequently is data arriving, and how often does the analysis need to be performed?** If our bikeshare company needs trip data every hour, that will be a driving factor in determining the schedule. Alternatively, if we have to load hundreds of thousands of tiny records, even if they don't add up to much in terms of mb or gb, the file access alone will slow down our analysis and we'll likely want to run it more often.
3. **What's the frequency on related datasets?** A good rule of thumb is that the frequency of a pipeline's schedule should be determined by the dataset in our

pipeline which requires the most frequent analysis. This isn't universally the case, but it's a good starting assumption. For example, if our trips data is updating every hour, but our bikeshare station table only updates once a quarter, we'll probably want to run our trip analysis every hour, and not once a quarter.

Data partitioning

Schedule partitioning
Not only are schedules great for reducing the amount of data our pipelines have to process, but they also help us guarantee that we can meet timing guarantees that our data consumers may need.

Logical partitioning
Conceptually related data can be partitioned into discrete segments and processed separately. This process of separating data based on its conceptual relationship is called logical partitioning. With logical partitioning, unrelated things belong in separate steps. Consider your dependencies and separate processing around those boundaries.

Also worth mentioning, the data location is another form of logical partitioning. For example, if our data is stored in a key-value store like Amazon's S3 in a format such as: `s3://<bucket>/<year>/<month>/<day>` we could say that our date is logically partitioned by time.

Size partitioning
Size partitioning separates data for processing based on desired or required storage limits. This essentially sets the amount of data included in a data pipeline run. Size partitioning is critical to understand when working with large datasets, especially with Airflow.

Why data partitioning?
Pipelines designed to work with partitioned data fail more gracefully. Smaller datasets, smaller time periods, and related concepts are easier to debug than big datasets, large time periods, and unrelated concepts. Partitioning makes debugging and rerunning failed tasks much simpler. It also enables easier redos of work, reducing cost and time.

Another great thing about Airflow is that if your data is partitioned appropriately, your tasks will naturally have fewer dependencies on each other. Because of this, Airflow will be able to parallelize execution of your DAGs to produce your results even faster.

Data quality

Data quality is the measure of how well a dataset satisfies its intended use.

Adherence to a set of **requirements** is a good starting point for measuring data quality.

Requirements should be defined by you and your data consumers before you start creating your data pipeline.

Examples of quality requirements:

- 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.

4.4 Production data pipelines

Lesson outline

In this lesson, we'll be learning about how to build maintainable and reusable pipelines in Airflow.

- Focus on elevating your DAGs to reliable, production-quality data pipelines.
- Extending Airflow with custom plug-ins to create your own hooks and operators.
- How to design task boundaries so that we maximize visibility into our tasks.
- subDAGs and how we can actually reuse DAGs themselves.
- Monitoring in Airflow.

Extending Airflow Hooks and Contrib

Custom Hooks allow Airflow to integrate and consolidate common interactions with non-standard datastores and systems.

Airflow has a rich and vibrant open source community. This community is constantly adding new functionality and extending the capabilities of Airflow. As an Airflow user, you should always check Airflow contrib and/or Airflow providers before building your own airflow plugins, to see if what you need already exists.

Operators and hooks for common data tools like Apache Spark and Cassandra, as well as vendor specific integrations for Amazon Web Services, Azure, and Google Cloud Platform can be found in Airflow contrib. If the functionality exists and it's not quite what you want, that's a great opportunity to add that functionality through an open source contribution.

Extending Airflow with Plugins

Airflow was built with the intention of allowing its users to extend and customize its functionality through plugins. The most common types of user-created plugins for Airflow are Operators and Hooks. These plugins make DAGs reusable and simpler to maintain.

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

To create custom operator, follow the steps:

1. Identify Operators that perform similar functions and can be consolidated.
2. Define a new Operator in the plugins folder.
3. Replace the original Operators with your new custom one, re-parameterize, and instantiate them.

Here is the Official Airflow Documentation for custom operators.

Best Practices for Data Pipeline Steps - Task Boundaries

DAG tasks should be designed such that they are:

- Atomic and have a single purpose.
- Maximize parallelism.
- Make failure states obvious.

Every task in your dag should perform **only one job**.

Benefits of Task boundaries

- Re-visitable: Task boundaries are useful for you if you revisit a pipeline you wrote after a 6 month absence. You'll have a much easier time understanding how it works and the lineage of the data if the boundaries between tasks are clear and well defined. This is true in the code itself, and within the Airflow UI.
- Tasks that do just one thing are often more easily parallelized. This parallelization can offer a significant speedup in the execution of our DAGs.

Pipeline Monitoring

Airflow can surface metrics and emails to help you stay on top of pipeline issues.

SLAs

Airflow DAGs may optionally specify an SLA, or “Service Level Agreement”, which is defined as a time by which a DAG must complete. For time-sensitive applications these features are critical for developing trust amongst your pipeline customers and ensuring that data is delivered while it is still meaningful. Slipping SLAs can also be early indicators of performance problems, or a need to scale up the size of your Airflow cluster.

Emails and alerts

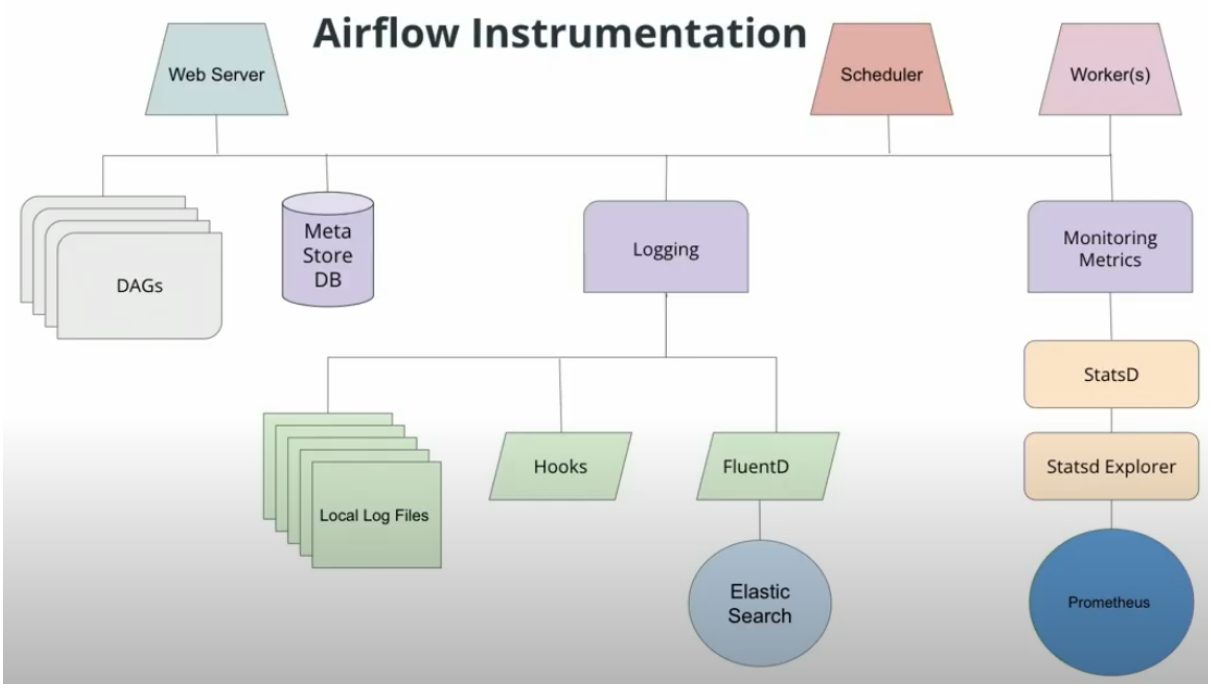
Airflow can be configured to send emails on DAG and task state changes. These state changes may include successes, failures, or retries. Failure emails can allow you to easily trigger alerts. It is common for alerting systems like PagerDuty to accept emails as a source of alerts. If a mission-critical data pipeline fails, you will need to know as soon as possible to get online and get it fixed.

Metrics

Airflow comes out of the box with the ability to send system metrics using a metrics aggregator called StatsD. StatsD can be coupled with metrics visualization tools like Grafana and monitoring tools like Prometheus to provide you and your team high level insights into the overall performance of your DAGs, jobs, and tasks. These systems can be integrated into your alerting system, such as **pagerduty**, so that you can ensure problems are dealt with immediately. These Airflow system-level metrics allow you and your team to stay ahead of issues before they even occur by watching long-term trends.

Logging
By default, Airflow logs to the local file system. You probably sifted through logs so far to see what was going on with the scheduler. Logs can be forwarded using standard logging tools like fluentd.

See Airflow Logging and Monitoring architecture here.



Additional resources: list of pipeline orchestrators

Awesome Pipeline