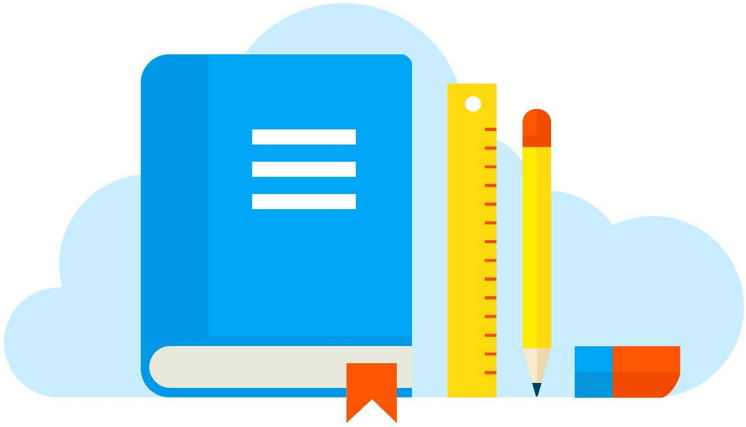


Data Engineering Nanodegree

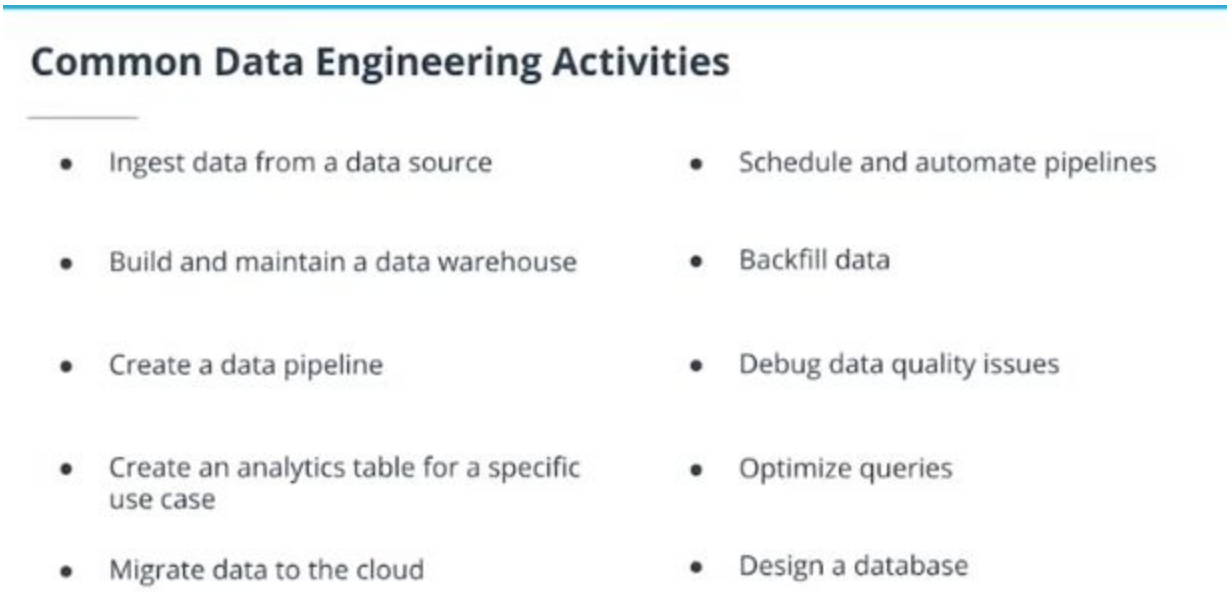
1. Welcome To The Nanodegree Program

Introduction To Data Engineering

- **Data Engineering:** Data engineering comprises all engineering and operational tasks to make data available for end users for different purposes like model building, analytics etc.

- **Data Science Hierarchy:**
    i. **Collect**
    ii. **Move/Store**
    iii. **Explore/Transform**
    iv. **Aggregate/label**
    v. **learn/optimize**

- **Data Engineer:**



- **Brief History:**

  ON the evolution of Data Engineering - Hacking Analytics

  Whitepaper 1

- **Data Engineering Tools:**

  Feel free to look at these resources and infographics describing the different tools used by Data Engineers.

  The Rise of Data Engineering: Common Skills and Tools

  Data Engineering 101: Top Tools And Framework Resources

  This infographic of Big Data tools will blow your mind [infographic]

  The Big Data Open Source Tools Landscape

2. Data Modeling

Introduction To Data Modeling

- **Database:** A database is a structured repository or collection of data that is stored and retrieved electronically for use in applications. Data can be stored, updated, or deleted from a database.
- **Database Management System (DBMS):** The software used to access the database by the user and application is the database management system. Check out these few links describing a DBMS in more detail.
- **What is Data Modeling?** Data modeling is a process used to define and analyze data requirements needed to support the business processes within the scope of corresponding information systems in organizations.
    - Conceptual Data Modeling

      Organize relationships on a high level with different entities.
    - Logical Data Modeling

      Design Entity fill with attributes
    - Physical Data Modeling

      With the DDL (data definition language) create a data model in relational and non relational databases.
- **Why is Data modeling important?**

  To save from complex queries. It is an iterative process.
    - **Data Organization:** The organization of the data for your applications is extremely important and makes everyone's life easier.
    - **Use cases:** Having a well thought out and organized data model is critical to how that data can later be used. Queries that could have been straightforward and simple might become complicated queries if data modeling isn't well thought out.

- **Starting early:** Thinking and planning ahead will help you be successful. This is not something you want to leave until the last minute.
- **Iterative Process:** Data modeling is not a fixed process. It is iterative as new requirements and data are introduced. Having flexibility will help as new information becomes available.

- **Relational Databases: MySQL, Oracle, Teradata, Sqlite, PostgreSQL**

  This model organizes data into one or more tables which consists of columns and rows with a unique key identifying each row.

  Attribute => Column

  Tuple => Row

  Table => Entity

  Database => Collection of tables or entities

- **Relational Model**

  In relational models we organize the data into one or more tables of columns and rows with a unique key to identify each row. Row is also called **Tuples.** Column is also called **Attributes.**

- **RDBMS**

  Relational database management system is a software used to maintain the relational databases.

- **SQL**

  Sql (structured query language) is the language used across almost all relational databases systems for querying and maintaining the databases.

- **Advantages of Using a Relational Database**
  - **Flexibility for writing in SQL queries:** With SQL being the most common database query language.
  - **Modeling the data not modeling queries**

- **Ability to do JOINS**
- **Ability to do aggregations and analytics**
- **Secondary Indexes available :** You have the advantage of being able to add another index to help with quick searching.
- **Smaller data volumes:** If you have a smaller data volume (and not big data) you can use a relational database for its simplicity.
- **ACID Transactions:** Allows you to meet a set of properties of database transactions intended to guarantee validity even in the event of errors, power failures, and thus maintain data integrity.
- **Easier to change to business requirements**

- **ACID Transactions:**

  Properties of database transactions intended to guarantee validity even in the event of errors or power failures.

  - **Atomicity:** The whole transaction is processed or nothing is processed. A commonly cited example of an atomic transaction is money transactions between two bank accounts. The transaction of transferring money from one account to the other is made up of two operations. First, you have to withdraw money in one account, and second you have to save the withdrawn money to the second account. An atomic transaction, i.e., when either all operations occur or nothing occurs, keeps the database in a consistent state. This ensures that if either of those two operations (withdrawing money from the 1st account or saving the money to the 2nd account) fail, the money is neither lost nor created.
  - **Consistency:** Only transactions that abide by constraints and rules are written into the database, otherwise the database keeps the previous state. The data should be correct across all rows and tables
  - **Isolation:** Transactions are processed independently and securely, order does not matter. A low level of isolation enables many users to access the data simultaneously, however this also increases the possibilities of

concurrency effects (e.g., dirty reads or lost updates). On the other hand, a high level of isolation reduces these chances of concurrency effects, but also uses more system resources and transactions blocking each other.

    - **Durability:** Completed transactions are saved to the database even in cases of system failure. A commonly cited example includes tracking flight seat bookings. So once the flight booking records a confirmed seat booking, the seat remains booked even if a system failure occurs.

- **When Not to Use a Relational Database**
    - **Have large amounts of data:** Relational Databases are not distributed databases and because of this they can only scale vertically by adding more storage in the machine itself. You are limited by how much you can scale and how much data you can store on one machine. You cannot add more machines like you can in NoSQL databases.
    - **Need to be able to store different data type formats:** Relational databases are not designed to handle unstructured data.
    - **Need high throughput -- fast reads**: While ACID transactions bring benefits, they also slow down the process of reading and writing data. If you need very fast reads and writes, using a relational database may not suit your needs.
    - **Need a flexible schema:** Flexible schema can allow for columns to be added that do not have to be used by every row, saving disk space.
    - **Need high availability:** The fact that relational databases are not distributed (and even when they are, they have a coordinator/worker architecture), they have a single point of failure. When that database goes down, a fail-over to a backup system occurs and takes time.
    - **Need horizontal scalability:** Horizontal scalability is the ability to add more machines or nodes to a system to increase performance and space for data.

- **What is PostgreSQL?**

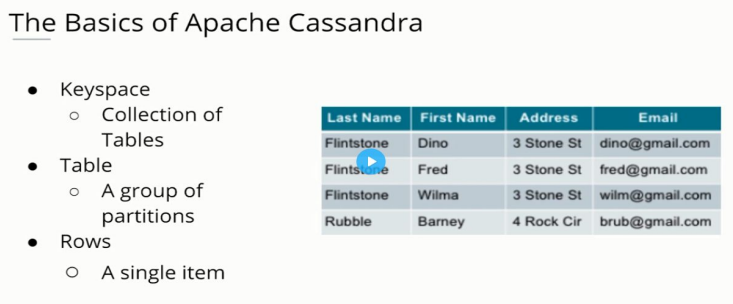
  Open source object-relational database system. It allows us to add duplicate data into the table.

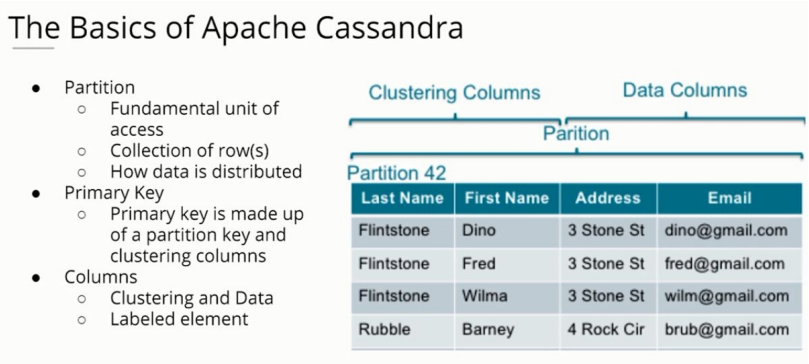
- **NoSQL Database**

  Has a simpler design, horizontal scaling, and finer control of availability. Data structures used are different than those in relational databases and make some operations faster.

- **NoSQL Databases Types:**

  - **MongodB:** Document Store
  - **DynamoDB:** key-value Store
  - **Apache HBase:** Wide Column Store
  - **Neo4j:** Graph Database
  - **Apache Cassandra:** This is a partitioned row store. Data is distributed by partitions across nodes or servers and data is organized in the columns and rows format. A Keyspace in Apache Cassandra is similar to a schema/database in PostgreSQL. No Duplicate data in it.





Provide scalability and high availability without compromising performance.Use its Own query language CQL (Cassandra Query Language)

- **What type of companies use Apache Cassandra?**

  All kinds of companies. For example, Uber uses Apache Cassandra for their entire backend. Netflix uses Apache Cassandra to serve all their videos to customers. Good use cases for NoSQL (and more specifically Apache Cassandra) are : Transaction logging (retail, health care) Internet of Things (IoT) Time series data Any workload that is heavy on writes to the database (since Apache Cassandra is optimized for writes).

- **When to use a NoSQL Database**

  NoSQL is better when you have large amounts of data for which you need high availability or if you need to scale out quickly.

  - **Need to be able to store different data type formats**: NoSQL was also created to handle different data configurations: structured, semi-structured, and unstructured data. JSON, XML documents can all be handled easily with NoSQL.

  - **Large amounts of data:** Relational Databases are not distributed databases and because of this they can only scale vertically by adding more storage in the machine itself. NoSQL databases were created to be able to be horizontally scalable. The more servers/systems you add to the database the more data that can be hosted with high availability and low latency (fast reads and writes).

- **Need horizontal scalability:** Horizontal scalability is the ability to add more machines or nodes to a system to increase performance and space for data
- **Need high throughput:** While ACID transactions bring benefits they also slow down the process of reading and writing data. If you need very fast reads and writes using a relational database may not suit your needs.
- **Need a flexible schema:** Flexible schema can allow for columns to be added that do not have to be used by every row, saving disk space.
- **Need high availability:** Relational databases have a single point of failure. When that database goes down, a failover to a backup system must happen and takes time.

- **When NOT to use a NoSQL Database**
  - **When you have a small dataset:** NoSQL databases were made for big datasets not small datasets and while it works it wasn't created for that.
  - **When you need ACID Transactions:** If you need a consistent database with ACID transactions, then most NoSQL databases will not be able to serve this need. NoSQL databases are eventually consistent and do not provide ACID transactions. However, there are exceptions to it. Some non-relational databases like MongoDB can support ACID transactions.
  - When you need the ability to do JOINS across tables: NoSQL does not allow the ability to do JOINS. This is not allowed as this will result in full table scans.
  - **If you want to be able to do aggregations and analytics**
  - **If you have changing business requirements :** Ad-hoc queries are possible but difficult as the data model was done to fix particular queries
  - **If your queries are not available and you need the flexibility:** You need your queries in advance. If those are not available or you will need to be able to have flexibility on how you query your data you might need to stick with a relational database

- **Caveats to NoSQL and ACID Transactions**

  There are some NoSQL databases that offer some form of ACID transaction. As of v4.0, MongoDB added multi-document ACID transactions within a single replica set. With their later version, v4.2, they have added multi-document ACID transactions in a sharded/partitioned deployment.

  Check out this documentation

  [White Paper] MongoDB Multi-Document ACID Transactions

  MongoDB Multi-Document ACID Transactions are GA

  Another example of a NoSQL database supporting ACID transactions is MarkLogic. How MarkLogic Supports ACID Transactions

Relational Data Models

- **Database**

  A set of related data and the way it is organized

- **Database Management System**

  Consist of computer software that allows users to interact with the databases and provides access to all of the data because of the close relationship the term database is often used to refer to both database and the DBMS used.

- **Importance of Relational Databases:**
  1. **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
  2. **Flexibility in adding and altering tables:** Relational databases give you flexibility to add tables, alter tables, add and remove data.
  3. **Data Integrity:** Data Integrity is the backbone of using a relational database.
  4. **Structured Query Language (SQL):** A standard language can be used to access the data with a predefined language.
  5. **Simplicity :** Data is systematically stored and modeled in tabular format.

6. **Intuitive Organization:** The spreadsheet format is intuitive but intuitive to data modeling in relational databases.

- **OLAP vs OLTP**

  OLAP (On-line Analytical Processing) deals with Historical Data or Archival Data. OLAP is characterized by relatively low volume of transactions. Queries are often very complex and involve aggregations. For OLAP systems a response time is an effectiveness measure. OLAP applications are widely used by Data Mining techniques. In the OLAP database there is aggregated, historical data, stored in multi-dimensional schemas (usually star schema). Sometimes queries need to access large amounts of data in Management records like what was the profit of your company in last year.Databases optimized for these workloads allow for complex analytical and ad hoc queries, including aggregations. These type of databases are optimized for reads

  OLTP (On-line Transaction Processing) is involved in the operation of a particular system. OLTP is characterized by a large number of short on-line transactions (INSERT, UPDATE, DELETE). The main emphasis for OLTP systems is put on very fast query processing, maintaining data integrity in multi-access environments and an effectiveness measured by number of transactions per second. In OLTP databases there is detailed and current data, and the schema used to store transactional databases is the entity model (usually 3NF). It involves Queries accessing individual records like Update your Email in the Company database. Databases optimized for these workloads allow for less complex queries in large volume. The types of queries for these databases are read, insert, update, and delete

  The key to remember the difference between OLAP and OLTP is analytics (A) vs transactions (T). If you want to get the price of a shoe then you are using OLTP (this has very little or no aggregations). If you want to know the total stock of shoes a particular store sold, then this requires using OLAP (since this will require aggregations)

- **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.

For More Information: Database normalization

How to reach normal form?

First Normal Form (1NF):

- Atomic values: each cell contains unique and single values
- Be able to add data without altering tables
- Separate different relations into different tables
- Keep relationships between tables together with foreign keys

Second Normal Form (2NF):

- Have reached 1NF
- All columns in the table must rely on the Primary Key
- No Composite Key

Third Normal Form (3NF):

- Must be in 2nd Normal Form
- No transitive dependencies Remember, transitive dependencies you are trying to maintain is that to get from A-> C, you want to avoid going through B.
- Transitive Dependency in a Database

Other normal forms not for production.

- **Denormalization**

  The 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 Must be done in read heavy workloads to increase performance. 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.

  For More: https://en.wikipedia.org/wiki/Denormalization

- **Fact Table and Dimension Table**

  Work together to create an organized data model. Fact and dimension tables work together to solve business problems. At least one or more fact tables for each dimension table.

  Fact table consists of the measurements, metrics or facts in a business process. Based on the event that actually happened.

  A structure that categorizes facts and measures in order to enable users to answer business questions. Dimensions are people, product, place and time.

  https://en.wikipedia.org/wiki/Dimension_(data_warehouse)

  https://en.wikipedia.org/wiki/Fact_table

- **Star Schema**

  Star schema is the simplest style of data mart schema. The star schema consists of one of more fact tables referencing any number of dimension table

  https://en.wikipedia.org/wiki/Star_schema

- **Snowflake Schema**

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

  Deep Diving in the World of Data Warehousing

  https://en.wikipedia.org/wiki/Snowflake_schema

- **Star Schema vs Snowflake Schema**

  Star schema does not allow for one to many relationships in case of dimensions table but in snowflake we have multiple dimensions of one or more dimensions table.

  Star Schema is a special, simplified case of the snowflake schema.

  Snowflake schema is more normalized than star schema but only in 1nf or 2nf

- **Upsert**

  Upsert In RDBMS language, the term upsert refers to the idea of inserting a new row in an existing table, or updating the row if it already exists in the table. The action of updating or inserting has been described as "upsert".

  PostgreSQL Tutorial - Learn PostgreSQL from Scratch

NO SQL Data Models

- **Apache Cassandra**

  Open source NOSQL DB. Masterless architecture. High availability. Linearly scalable

  Planning and testing DataStax Enterprise Apache Cassandra deployments | DSE Planning guide

  NoSQL Databases Overview, Types and Selection Criteria

  Cassandra - Architecture

  https://docs.datastax.com/en/archived/cassandra/3.0/cassandra/architecture/archTOC.html

- **Eventual Consistency:**

  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:**
  - **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

    - Disambiguating ACID and CAP

- **Which of these combinations is desirable for a production system - Consistency and Availability, Consistency and Partition Tolerance, or Availability and Partition Tolerance?**

  As the CAP Theorem Wikipedia entry says, "The CAP theorem implies that in the presence of a network partition, one has to choose between consistency and availability." So there is no such thing as Consistency and Availability in a distributed database since it must always tolerate network issues. You can only have Consistency and Partition Tolerance (CP) or Availability and Partition Tolerance (AP). Remember, relational and non-relational databases do different things, and that's why most companies have both types of database systems.

- **Does Cassandra meet just Availability and Partition Tolerance in the CAP theorem?**

  According to the CAP theorem, a database can actually only guarantee two out of the three in CAP. So supporting Availability and Partition Tolerance makes sense, since Availability and Partition Tolerance are the biggest requirements.

- **If Apache Cassandra is not built for consistency, won't the analytics pipeline break?**

  If I am trying to do analysis, such as determining a trend over time, e.g., how many friends does John have on Twitter, and if you have one less person counted because of "eventual consistency" (the data may not be up-to-date in all locations), that's OK. In theory, that can be an issue but only if you are not constantly updating. If the pipeline pulls data from one node and it has not been updated, then you won't get it. Remember, in Apache Cassandra it is about Eventual Consistency.

- **Denormalization in Apache Cassandra**
    - Denormalization is not just okay -- it's a must
    - Denormalization must be done for fast reads
    - Apache Cassandra has been optimized for fast writes
    - ALWAYS think Queries first
    - One table per query is a great strategy

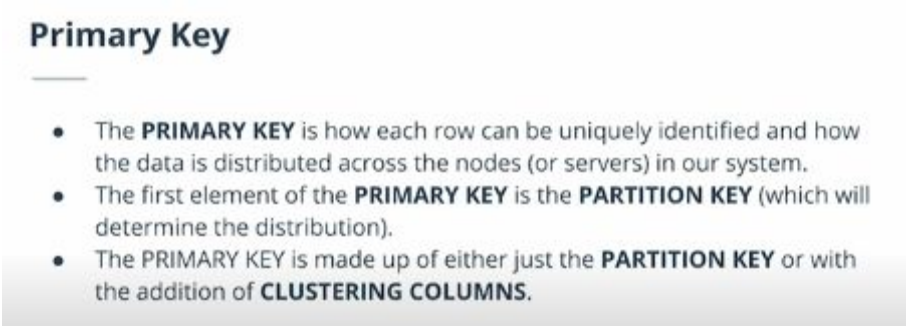
- Apache Cassandra does not allow for JOINs between tables

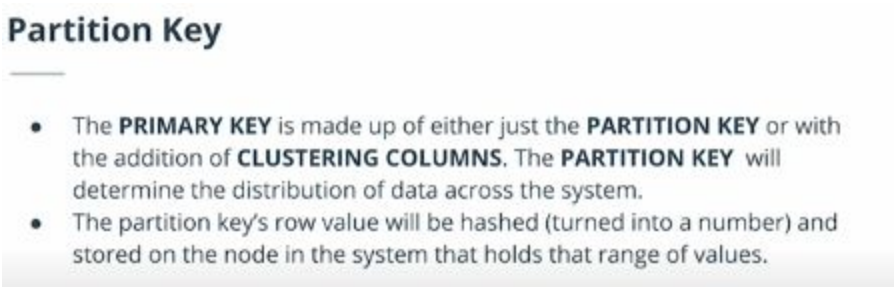
Data Modeling in Cassandra: [Data modeling concepts | CQL for DSE 6.7]

- **CQL**

  Cassandra query language is the way to interact with databases and is very similar to SQL. Joins, GroupBy or sub queries are not in CQL and are not supported by CQL.

- **Primary Key in Apache Cassandra**





- **Clustering Columns**
  - The clustering column will sort the data in 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
  - [Difference between partition key, composite key and clustering key in Cassandra?]

- **Where Clause**

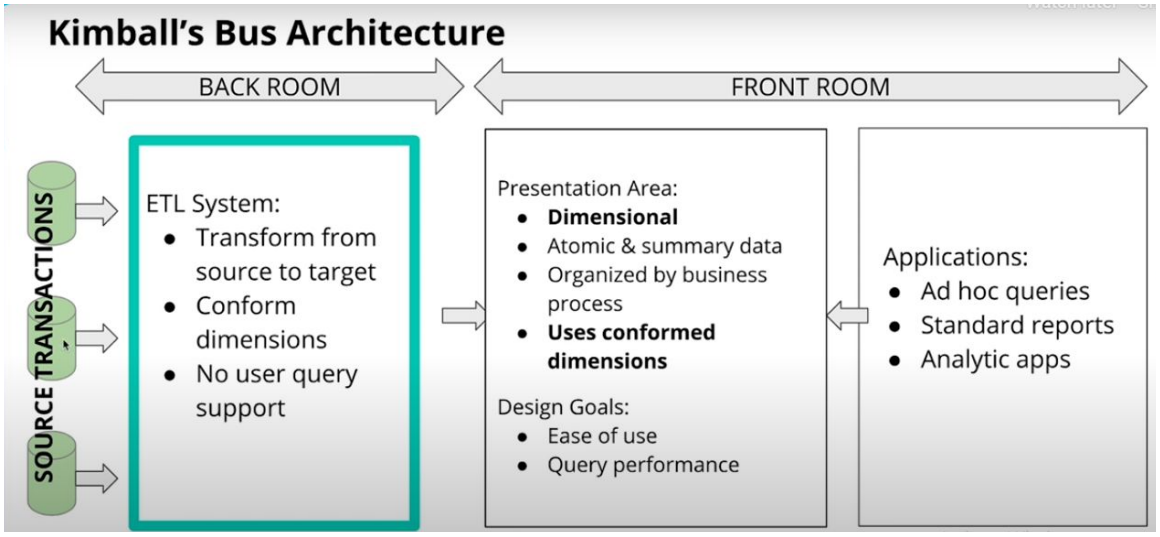
  WHERE clause Data Modeling in Apache Cassandra is query focused, and that focus needs to be on the WHERE clause Failure to include a WHERE clause will result in an error.

3. Cloud Data Warehouses

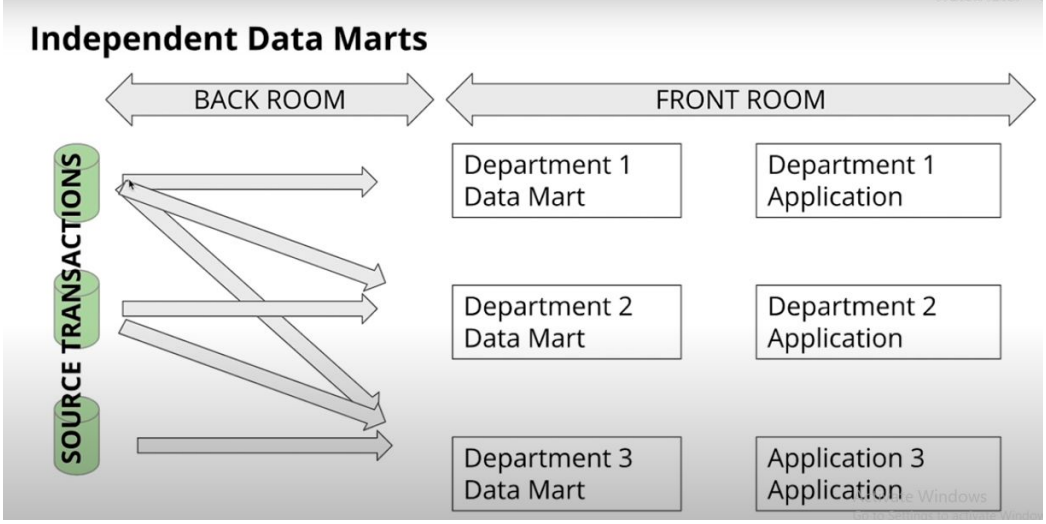
Introduction To Data Warehouse

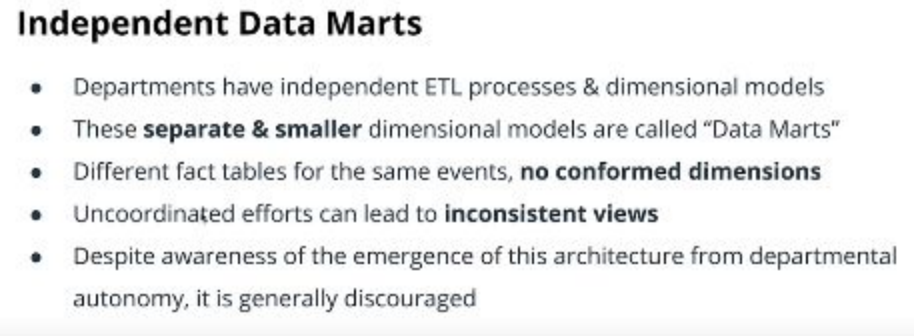
- Data warehouse is a system (including processes technologies and data representations) that enables us to support analytical processes.
- **ETL:** Extract the data and from the source systems used for operations. Transform the data and load it into a dimensional model.
- **Dimensional Model:** The dimensional model is designed to make it easy for business users to work with data. Improve analytical queries performance
- **Kimball's Bus Architecture**

  According to Kimball's Bus Architecture, data is kept in a common dimension data model shared across different departments. It does not allow for individual department specific data modeling requirements.



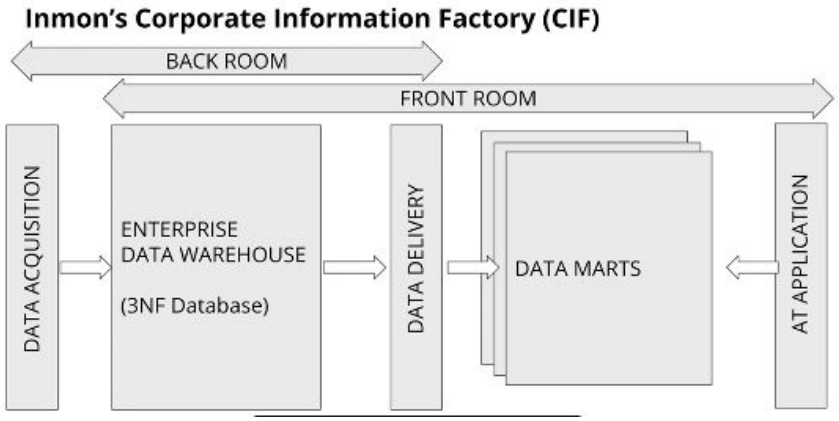
- **Independent Data Marts**





Independent Data Marts are not highly encouraged to help business departments meet their analytic needs.

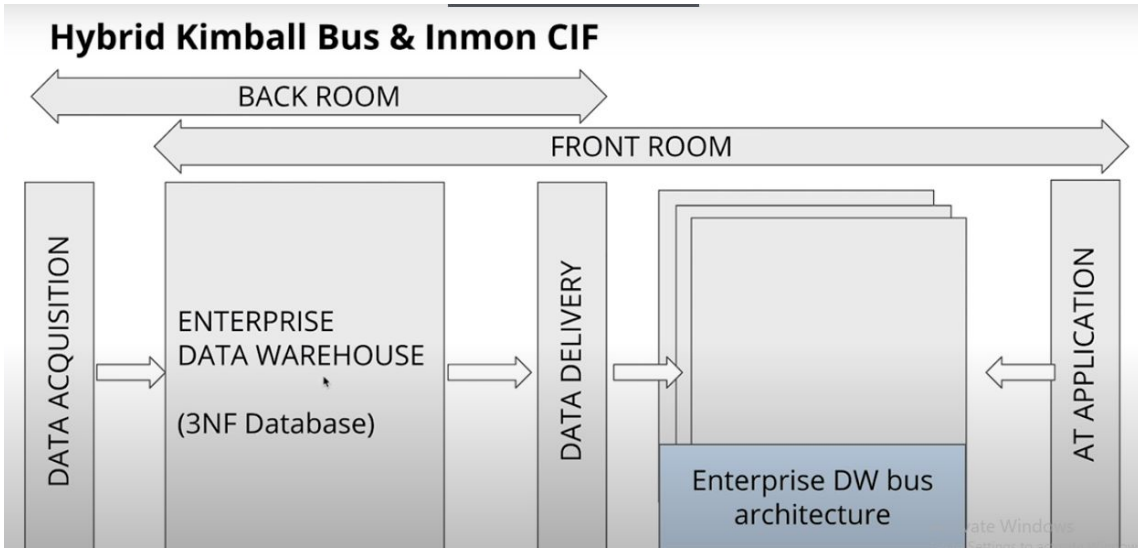
- **Inmon's Corporate Information Factory(CIF)**



Inmon's Corporate Information Factory (CIF)
Data Marts

- 2 ETL Process
  - Source systems → 3 NF DB
  - 3 NF DB → Departmental Data Marts
- The 3NF DB acts an enterprise wide data store.
  - Single integrated source of truth for data-marts
  - Could be accessed by end-users if needed
- Data marts dimensionally modelled & unlike Kimball's dimensional models, they are mostly aggregated

- **Hybrid Kimball Bus and Inmon CIF**

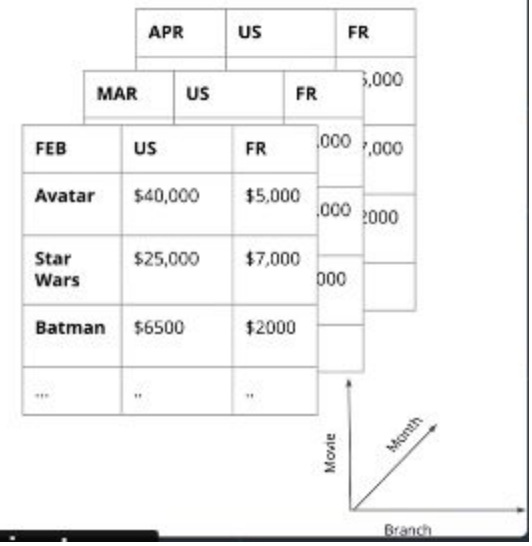


- **OLAP Cubes**

  An OLAP cube is an aggregation of a fact metric on a number of dimensions. Easy to communicate to business users. Common OLAP Operations includes Rollup, DrillDown, Slice and Dice.

OLAP Cubes Operations: Roll-up & Drill Down

- **Roll-up**: Sum up the sales of each city by Country: e.g. US, France (less columns in branch dimension)
- **Drill-Down**: Decompose the sales of each city into smaller districts (more columns in branch dimension)
- The OLAP cubes should store the finest grain of data (atomic data), in case we need to drill-down to the lowest level, e.g. Country → City → District → Street, etc..



OLAP Cubes Operations: Slice

- Reducing N dimensions to N-1 dimensions by restricting one dimension to a single value
- E.g. month='MAR'

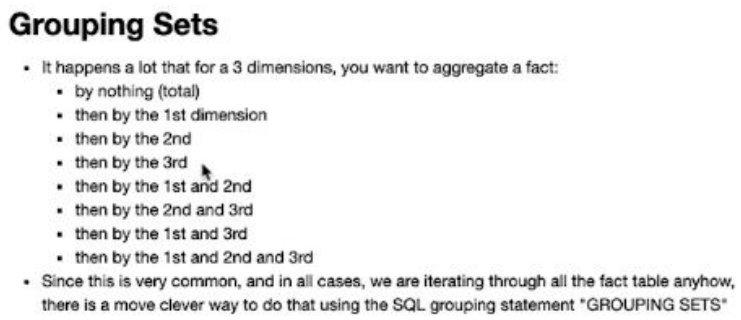
OLAP Cubes Operations: Dice

- Same dimensions but computing a sub-cube by restricting, some of the values of the dimensions
- E.g. month in ['FEB', 'MAR'] and movie in ['Avatar', 'Batman'] branch = 'NY'

Reducing N dimensions to N-1 dimensions by restricting one dimension to a single value.	Slice
Same dimensions but computing a sub-cube by restricting, some of the values of the dimensions.	Dice
Aggregates or combines values and reduces number of rows or columns.	Roll-Up
Decomposes values and increases number of rows or columns.	Drill Down

- **OLAP Cubes query optimization**

- **Grouping Sets**



- **CUBE**
- **MOLAP vs ROLAP**
- **Recommended Books:**

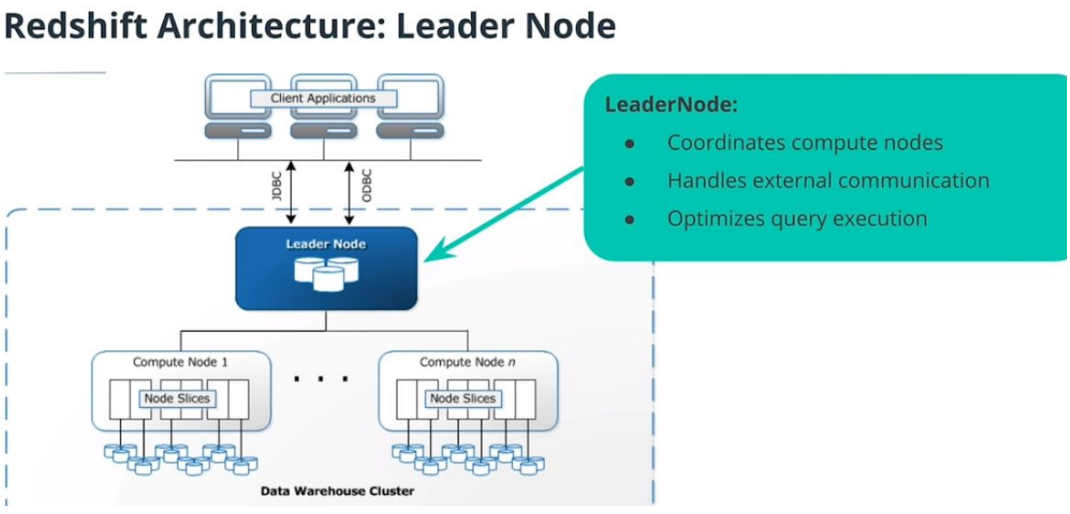
  [The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling]

  [Building the Data Warehouse Fourth Edition: Inmon, WH: 9780764599446]

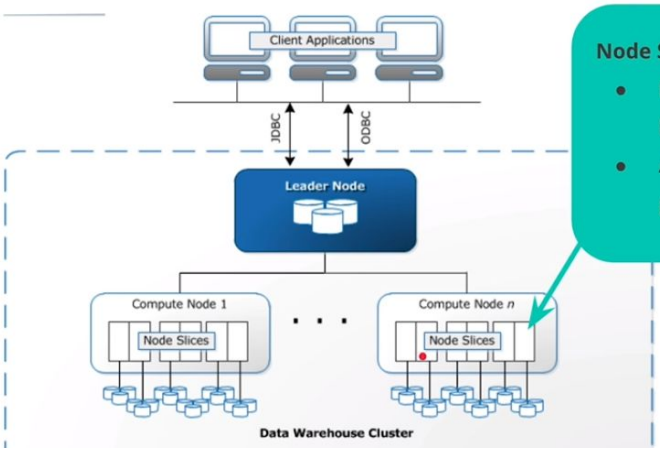
  [Building a Data Warehouse: With Examples in SQL Server (Expert's Voice)]

Implementing Data Warehouses on AWS

- **Redshift Architecture**



Redshift Architecture: Slices



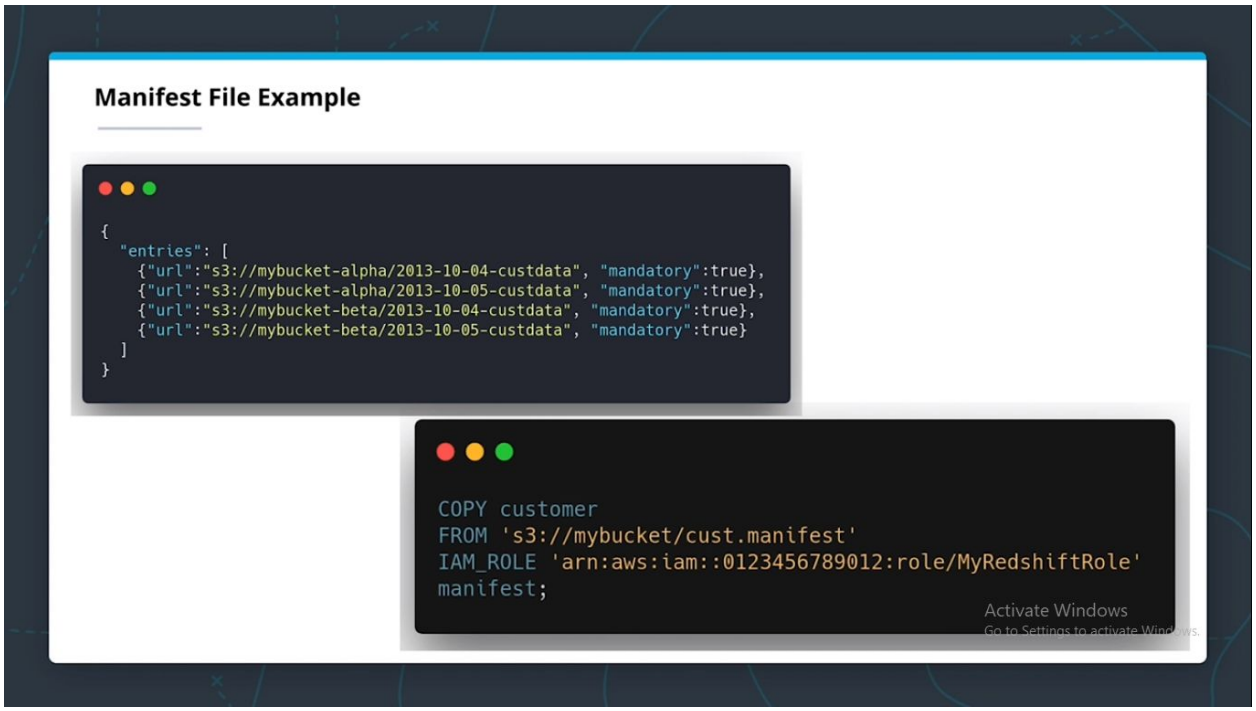
Node Slices:

- Each compute node is logically divided into a number of slices
- A cluster with *n* slices, can process *n* partitions of a tables simultaneously

Common Prefix Example

```
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')
```



Redshift ETL Automatic Compression Optimization

- The optimal compression strategy for each column type is different
- Redshift gives the user control over the compression of each column
- The COPY command makes automatic best-effort compression decisions for each column

ETL from Other Sources

- It is also possible to **ingest directly** using ssh from EC2 machines
- Other than that:
  - S3 needs to be used as a **staging area**
  - Usually, an EC2 ETL worker needs to run the ingestion jobs **orchestrated by a dataflow product** like Airflow, Luigi, Nifi, StreamSet or AWS Data Pipeline

ETL Out of Redshift

- Redshift is accessible, like any relational database, as a JDBC/ODBC source
  - Naturally used by BI apps
- However, we may need to extract data out of Redshift to pre-aggregated OLAP cubes

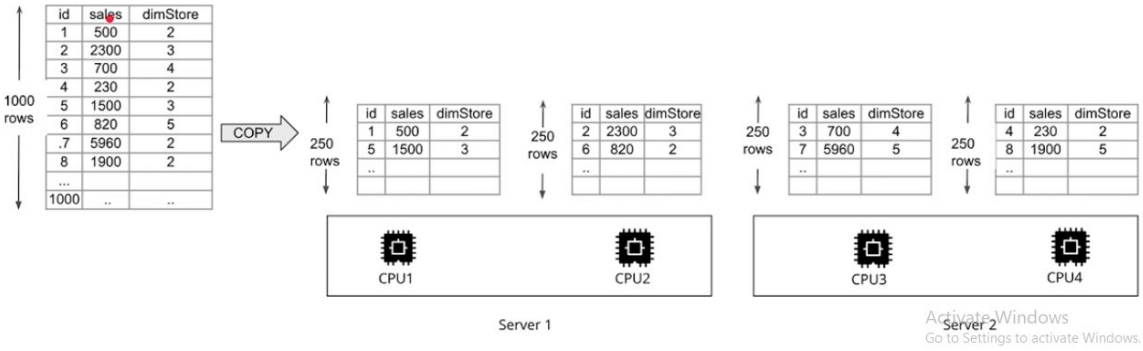
```
UNLOAD ('select * from venue limit 10')
to 's3://mybucket/venue_pipe_'
iam_role 'arn:aws:iam::0123456789012:role/MyRedshiftRole';
```

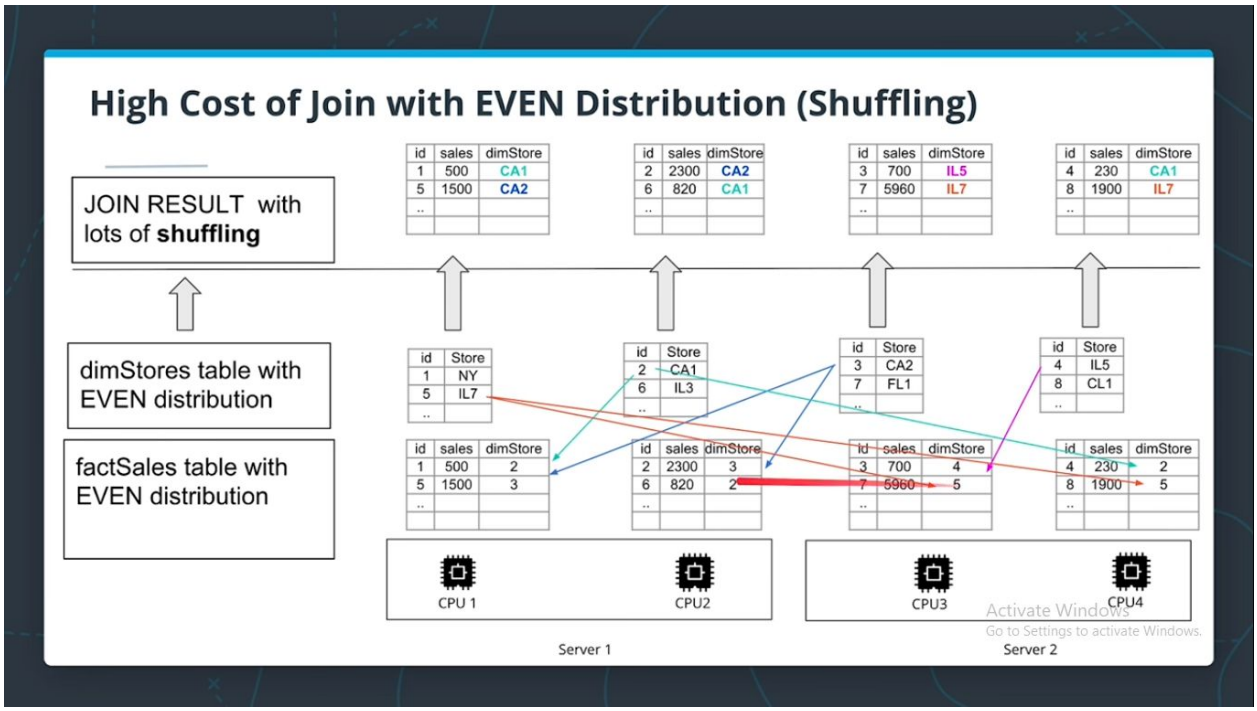
Optimizing Table Design

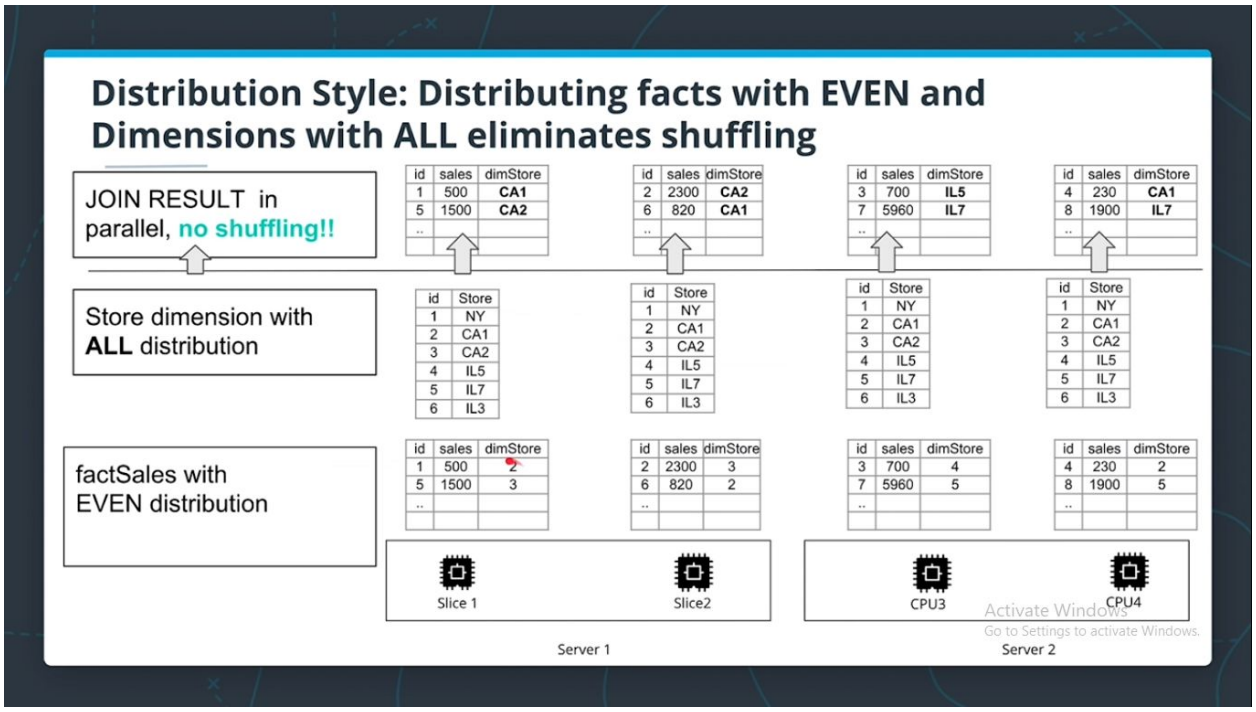
- When a table is partitioned up into many pieces and distributed across slices in different machines, this is done blindly
- If one has an idea about the frequent access pattern of a table, one can choose a more clever strategy
- The 2 possible strategies are:
  - Distribution Style
  - Sorting key

Distribution Style: EVEN, EXAMPLE 1: FACT TABLE

- Round-robin over all slices to achieve **load-balancing**
- Good if a table won't be joined

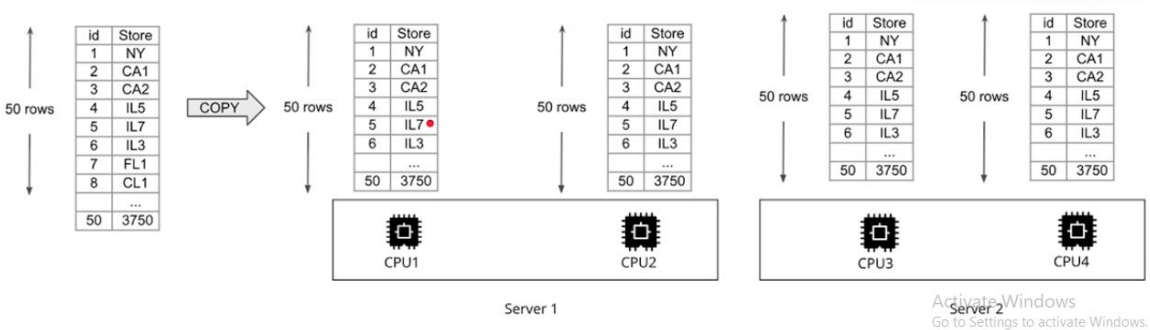



High Cost of Join with EVEN Distribution (Shuffling)
JOIN RESULT with lots of shuffling
dimStores table with EVEN distribution
factSales table with EVEN distribution
CPU 1
CPU2
CPU3
CPU4
Server 1
Server 2
Activate Windows
Go to Settings to activate Windows.


Distribution Style: Distributing facts with EVEN and Dimensions with ALL eliminates shuffling
JOIN RESULT in parallel, no shuffling!!
Store dimension with ALL distribution
factSales with EVEN distribution
Slice 1
Slice2
CPU3
CPU4
Server 1
Server 2
Activate Windows
Go to Settings to activate Windows.

Distribution Style: ALL

- Small tables could be replicated on all slices to speed up joins
- Used frequently for dimension tables
- AKA "broadcasting"

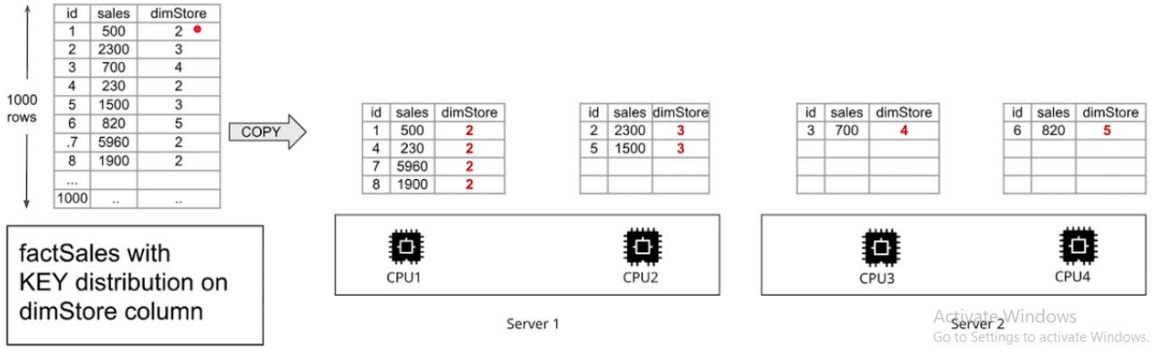


Distribution Style : AUTO

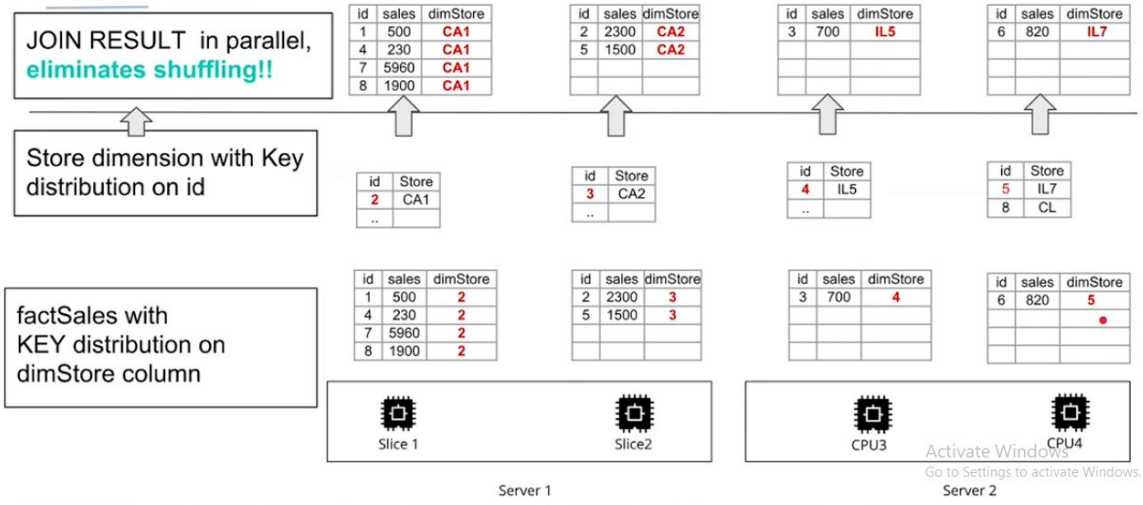
- Leave decision to Redshift
- "Small enough" tables are distributed with an ALL strategy
- Large tables are distributed with EVEN strategy

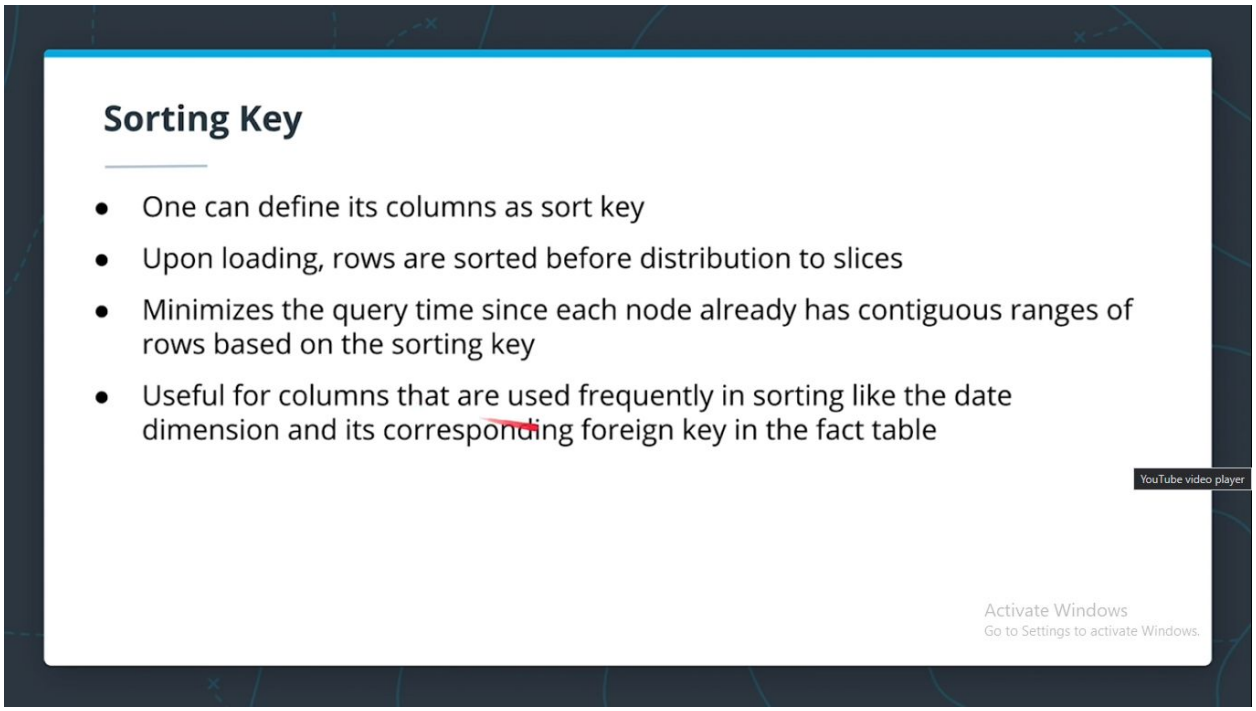
Distribution Style: KEY

- Rows having similar values are placed in the same slice



Distribution Style: Distributing both facts and Dimensions on the joining KEYs eliminates shuffling





3. Data Lake With Spark

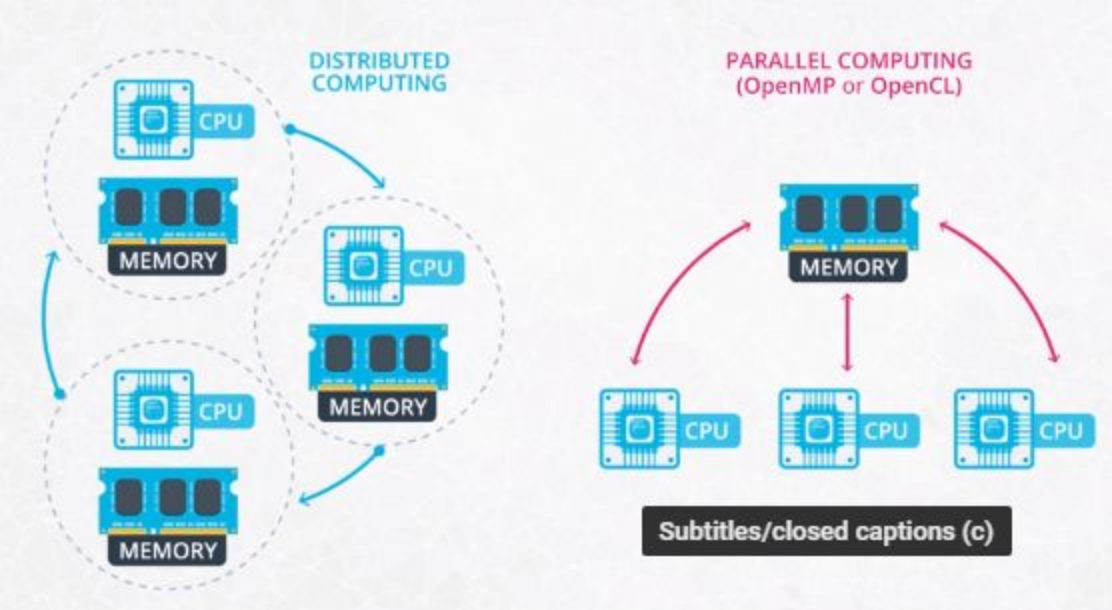
Power of Spark

- **Why Learn Spark?**

  Spark is currently one of the most popular tools for big data analytics. You might have heard of other tools such as Hadoop. Hadoop is a slightly older technology although still in use by some companies. Spark is generally faster than Hadoop, which is why Spark has become more popular over the last few years.

  There are many other big data tools and systems, each with its own use case. For example, there are database systems like Apache Cassandra and SQL query engines like Presto. But Spark is still one of the most popular tools for analyzing large data sets.

- **Distributed and Parallel Computing**



- **Hadoop Vocabulary**

  Here is a list of some terms associated with Hadoop. You'll learn more about these terms and how they relate to Spark in the rest of the lesson.

  - **Hadoop** - an ecosystem of tools for big data storage and data analysis. Hadoop is an 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 whereas Spark tries to keep data in memory whenever possible. This makes Spark faster for many use cases.

  - **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.

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

As Hadoop matured, other tools were developed to make Hadoop easier to work with. These tools included:

- **Apache Pig** - a SQL-like language that runs on top of Hadoop MapReduce
- **Apache Hive** - another SQL-like interface that runs on top of Hadoop MapReduce

Oftentimes when someone is talking about Hadoop in general terms, they are actually talking about Hadoop MapReduce. However, Hadoop is more than just MapReduce.

- **How is Spark related to Hadoop?**

  Spark, which is the main focus of this course, is another big data framework. Spark contains libraries for data analysis, machine learning, graph analysis, and streaming live data. Spark is generally faster than Hadoop. This is because Hadoop writes intermediate results to disk whereas Spark tries to keep intermediate results in memory whenever possible.

  The Hadoop ecosystem includes a distributed file storage system called HDFS (Hadoop Distributed File System). Spark, on the other hand, 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 such as Amazon S3.

- **Streaming Data**

  Data streaming is a specialized topic in big data. The use case is when you want to store and analyze data in real-time such as Facebook posts or Twitter tweets.

  Spark has a streaming library called Spark Streaming although it is not as popular and fast as some other streaming libraries. Other popular streaming libraries include Storm and Flink

- **Map Reduce**

  MapReduce is a programming technique for manipulating large data sets. "Hadoop MapReduce" is a specific implementation of this programming technique.

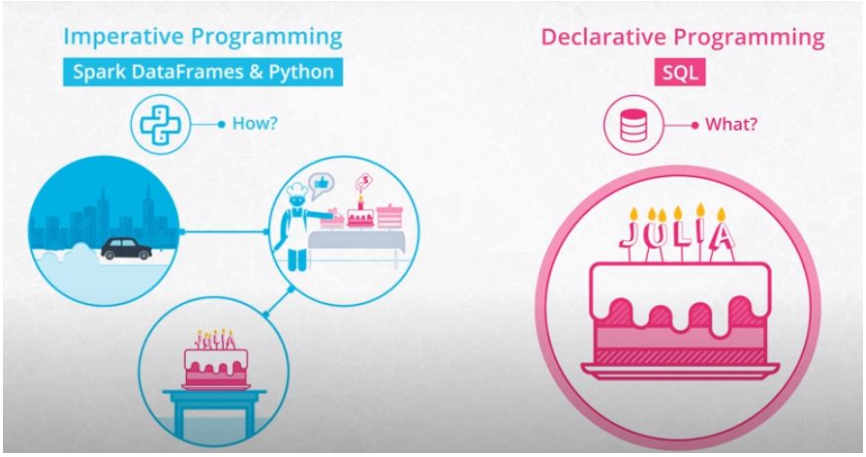
The technique works by first dividing up a large dataset and distributing the data across a cluster. In the map step, each data is analyzed and converted into a (key, value) pair. Then these key-value pairs are shuffled across the cluster so that all keys are on the same machine. In the reduce step, the values with the same keys are combined together.

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

- **Functional Programming**

  Functional programming is good for distributed system.The PySpark API allows you to write programs in Spark and ensures that your code uses functional programming practices. Underneath the hood, the Python code uses py4j to make calls to the Java Virtual Machine (JVM).

- **Imperative Programming vs Declarative programming**



- **Data Wrangling with Dataframe**

  We have used the following general functions that are quite similar to methods of pandas dataframes:

  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 such as count(), countDistinct(), avg(), max(), min(), etc. in the pyspark.sql.functions module. 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. For example, if we would like to compute one type of aggregate for one or more columns of the DataFrame we can just simply chain the aggregate method after a groupBy(). If we would like to use different functions on different columns, agg()comes in handy. For example agg({"salary": "avg", "age": "max"}) computes the average salary and maximum age.

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).

- **Why might you prefer to use SQL over data frames? Why might you prefer data frames over SQL?**

  Both Spark SQL and Spark Data Frames are part of the Spark SQL library. Hence, they both use the Spark SQL Catalyst Optimizer to optimize queries.

  You might prefer SQL over data frames because the syntax is clearer especially for teams already experienced in SQ

  Spark data frames give you more control. You can break down your queries into smaller steps, which can make debugging easier. You can also cache intermediate results or repartition intermediate results

  Spark master node use 7077 for their worker node

  Spark running job info share on port 4040

  Spark web ui is show on port 8080

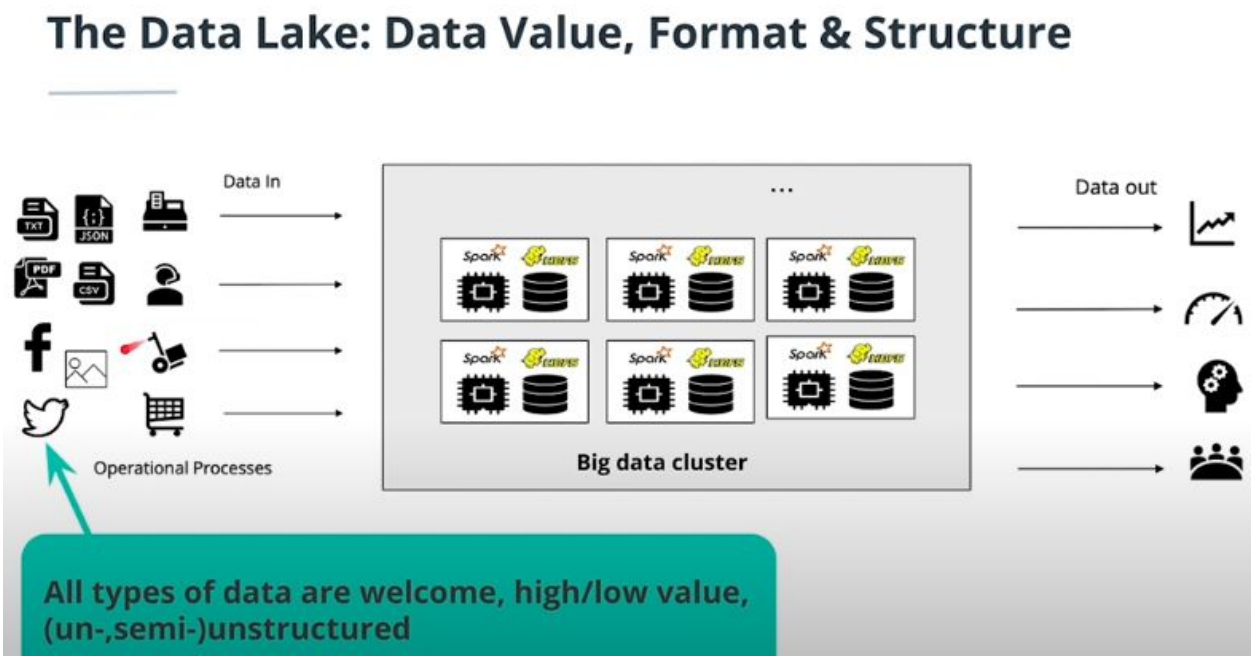
  Spark jupyter notebooks connects on port 8888

- **Cohort Analysis**

  In this type of analysis we segment different user like new ones with old ones

Introduction to Data Lake

- **Data Lake**



All types of data are welcome, high/low value(un,semi) unstructured

Data is stored “as-in” transformations are done later i.e ELT instead of ETL

Data is processed with schema-on-read, no predefined star-schema is there before transformation.

Massive parallelism & scalability come out of the box

Data Lake vs Data Warehouse

	Data Warehouse	Data Lake
Data form	Tabular format	All formats
Data value	High only	High-value, medium-value and to-be-discovered
Ingestion	ETL	ELT
Data model	Star & snowflake with conformed dimensions or data-marts and OLAP cubes	Star, snowflakes and OLAP are also possible but other ad-hoc representations are possible
Schema	Known before ingestion (schema-on-write)	On-the-fly at the time of analysis (schema-on-read)
Technology	Expensive MPP databases with expensive disks and connectivity	Commodity hardware with parallelism as first principle
Data Quality	High with effort for consistency and clear rules for accessibility	Mixed, some data remain in raw format, some data is transformed to higher quality
Users	Business analysts	Data scientists, Business analysts & ML engineers
Analytics	Reports and Business Intelligence visualizations	Machine Learning, graph analytics and data exploration

Data Lake issues

- Data Lake is prone to being a chaotic **data garbage dump**. Efforts are being made to put measures and practices like detailed metadata to reduce this risk.
- Since a major feature of the data lake is the wide accessibility of cross-department data and external data of relevance, sometimes **data governance** is not easy to implement. Telling who has access to what is hard.
- Finally, it is still sometimes unclear, per given case, whether a data lake should **replace, offload or work in parallel** with a data warehouse or data marts. In all cases, dimensional modelling, even in the context of a data lake, continues to remain a valuable practice.

4. Data Pipelines

Introduction to Data Pipelines

- **Extract Transform Load (ETL) and Extract Load Transform (ELT):**

  "ETL is normally a continuous, ongoing process with a well-defined workflow. ETL first extracts data from homogeneous or heterogeneous data sources. Then, data is cleansed, enriched, transformed, and stored either back in the lake or in a

data warehouse. "ELT (Extract, Load, Transform) is a variant of ETL wherein the extracted data is first loaded into the target system. Transformations are performed after the data is loaded into the data warehouse. ELT typically works well when the target system is powerful enough to handle transformations. Analytical databases like Amazon Redshift and Google BigQ."

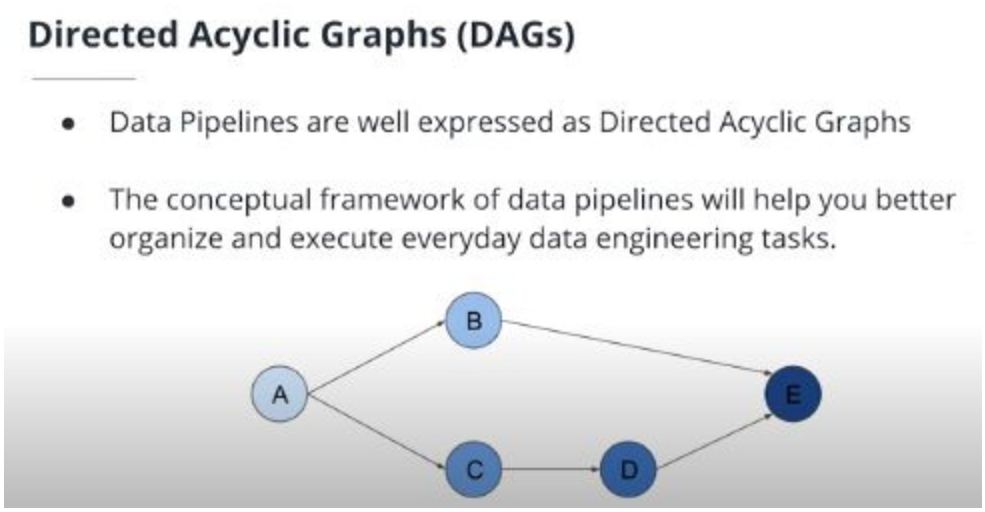
- **What is S3?** "Amazon S3 has a simple web services interface that you can use to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, fast, inexpensive data storage infrastructure that Amazon uses to run its own global network of websites."

- **What is RedShift?**

  "Amazon Redshift is a fully managed, petabyte-scale data warehouse service in the cloud. You can start with just a few hundred gigabytes of data and scale to a petabyte or more... The first step to create a data warehouse is to launch a set of nodes, called an Amazon Redshift cluster. After you provision your cluster, you can upload your data set and then perform data analysis queries. Regardless of the size of the data set, Amazon Redshift offers fast query performance using the same SQL-based tools and business intelligence applications that you use today.

- **Data Validation**

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



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. When we say “no cycles exist” what we mean is the nodes cant create a path back to themselves.

Nodes: A step in the data pipeline process.

Edges: The dependencies or relationships other between nodes.

Are there real world cases where a data pipeline is not DAG?

It is possible to model a data pipeline that is not a DAG, meaning that it contains a cycle within the process. However, the vast majority of use cases for data pipelines can be described as a directed acyclic graph (DAG). This makes the code more understandable and maintainable.

Can we have two different pipelines for the same data and can we merge them back together?

Yes. It's not uncommon for a data pipeline to take the same dataset, perform two different processes to analyze it, then merge the results of those two processes back together.

Apache Airflow

"Airflow is a platform to programmatically author, schedule and monitor workflows. Use airflow to author workflows as directed acyclic graphs (DAGs) of tasks. The airflow scheduler executes your tasks on an array of workers while following the specified dependencies. Rich command line utilities make performing complex surgeries on DAGs a snap. The rich user interface makes it easy to visualize pipelines running in production, monitor progress, and troubleshoot issues when needed. When workflows are defined as code, they become more maintainable, versionable, testable, and collaborative."

- **Building Pipeline in Airflow**

  Creating a DAG

  Creating a DAG is easy. Give it a name, a description, a start date, and an interval.

  from airflow import DAG

  divvy_dag = DAG('divvy', description='Analyzes Divvy Bikeshare Data', start_date=datetime(2019, 2, 4), schedule_interval='@daily')

  Creating Operators to Perform Tasks

  Operators define the atomic steps of work that make up a DAG. Instantiated operators are referred to as Tasks.

  from airflow import DAG

  from airflow.operators.python_operator import PythonOperator

  def hello_world(): print(“Hello World”)

  divvy_dag = DAG(...) task = PythonOperator(task_id=’hello_world’, python_callable=hello_world, dag=divvy_dag)

  Schedules

  Schedules are optional, and may be defined with cron strings or Airflow Presets. Airflow provides the following presets:

    - **@once -** Run a DAG once and then never again

- **@hourly -** Run the DAG every hour
- **@daily -** Run the DAG every day
- **@weekly -** Run the DAG every week
- **@monthly -** Run the DAG every month
- **@yearly-** Run the DAG every year
- **None -** Only run the DAG when the user initiates it

Start Date: If your start date is in the past, Airflow will run your DAG as many times as there are scheduled intervals between that start date and the current date.

End Date: Unless you specify an optional end date, Airflow will continue to run your DAGs until you disable or delete the DAG.

Operators:

Operators define the atomic steps of work that make up a DAG. Airflow comes with many Operators that can perform common operations. Here are a handful of common ones:

- **PythonOperator**
- **PostgresOperator**
- **RedshiftToS3Operator**
- **S3ToRedshiftOperator**
- **BashOperator**
- **SimpleHttpOperator**
- **Sensor**

Task Dependencies

In Airflow DAGs:

- Nodes = Tasks
- Edges = Ordering and dependencies between tasks

Task dependencies can be described programmatically in Airflow using >> and <<

- a >> b means a comes before b
- a << b means a comes after b

hello_world_task = PythonOperator(task_id='hello_world', ...)

goodbye_world_task = PythonOperator(task_id='goodbye_world', ...) ...

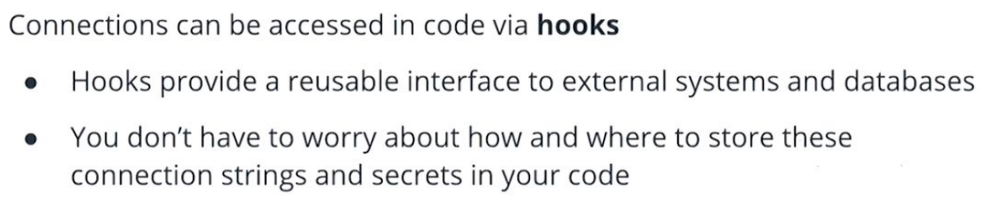
Use >> to denote that goodbye_world_task depends on hello_world_task

hello_world_task >> goodbye_world_task

Tasks dependencies can also be set with "set_downstream" and "set_upstream"

- a.set_downstream(b) means a comes before b
- a.set_upstream(b) means a comes after b

Airflow Hooks



Connections can be accessed in code via hooks. Hooks provide a reusable interface to external systems and databases. With hooks, you don't have to worry about how and where to store these connection strings and secrets in your code.

Airflow comes with many Hooks that can integrate with common systems. Here are a few common ones:

- HttpHook
- PostgresHook (works with RedShift)
- MySqlHook SlackHook
- PrestoHook

[The Zen of Python and Apache Airflow]

- **Airflow Context**

  Context provides access to runtime variables.

- **Data Lineage**

  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, its 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.

- **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.

    - **What is the size of data, on average, for a time period?** If an entire years worth of data is only a few kb or mb, then perhaps its 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.
    - **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.
    - **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.
- **Start Date**

  Airflow will begin running pipelines on the start date selected. Whenever the start date of a DAG is in the past, and the time difference between the start date and now includes more than one scheduled interval, Airflow will automatically schedule and execute a DAG run to satisfy each one of those intervals. This feature is useful in almost all enterprise settings, where companies have established years of data that may need to be retroactively analyzed.

- **End Date** Airflow pipelines can also have end dates. You can use an end_date with your pipeline to let Airflow know when to stop running the pipeline. End_dates can also be useful when you want to perform an overhaul or redesign of an existing pipeline. Update the old pipeline with an end_date and then have the new pipeline start on the end date of the old pipeline.
- **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.

  In practice, it is often best to have Airflow process pre-partitioned data. If your upstream data sources cannot partition data, it is possible to write an Airflow

DAG to partition the data. However, it is worth keeping in mind memory limitations on your Airflow workers. If the size of the data to be partitioned exceeds the amount of memory available on your worker, the DAG will not successfully execute.

- **Data Quality**
    - 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

Airflow 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. To create custom operator, follow the steps: Identify Operators that perform similar functions and can be consolidated Define a new Operator in the plugins folder Replace the original Operators with your new custom one, re-parameterize, and instantiate them

SubDAGs

Commonly repeated series of tasks within DAGs can be captured as reusable SubDAGs. **Benefits include:**

- Decrease the amount of code we need to write and maintain to create a new DAG
- Easier to understand the high level goals of a DAG
- Bug fixes, speedups, and other enhancements can be made more quickly and distributed to all DAGs that use that SubDAG

Drawbacks of Using SubDAGs

- Limit the visibility within the Airflow UI
- Abstraction makes understanding what the DAG is doing more difficult
- Encourages premature optimization

Common Questions Can Airflow nest subDAGs? - Yes, you can nest subDAGs. However, you should have a really good reason to do so because it makes it much harder to understand what's going on in the code. Generally, subDAGs are not necessary at all, let alone subDAGs within subDAGs.

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 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.