1. Introduction to Bigdata & Hadoop ecosystem…

Hadoop is the framework to process and analyze Big Data. Hadoop's popularity speaks for itself.

Big Data Challenges

We can describe the Big Data challenges into four Parts.

- `Storage` - We need to know how to store the huge data efficiently.
- `Computational Efficiency` - Efficiency in storing the data that is suitable for computation.
- `Data Loss` - Due to disruption, hardware failure. We need to have a proper recovery strategy in place.
- `Cost` - The solution that we propose should be cost-effective.

Why Hadoop?

What is the need for Big Data solutions like Hadoop?

Why traditional database is not a solution?

- It is **not horizontally scalable** because we cannot add resources or more computational node.
- A database is designed for **structured** data. This database is not a good choice when we have a variety of data.

Hadoop: A Good Solution

- It **support huge volume** of data.
- It **stores** the data efficiently.
- Data Loss is unavoidable. The proposed solution gives **good recovery strategies.**
- The solution should be **horizontally scalable** as the data grows.
- It should be **cost-effective**.
- Minimize the learning code. It should be **easy** for the programmer and non-programmers.

What is Hadoop?

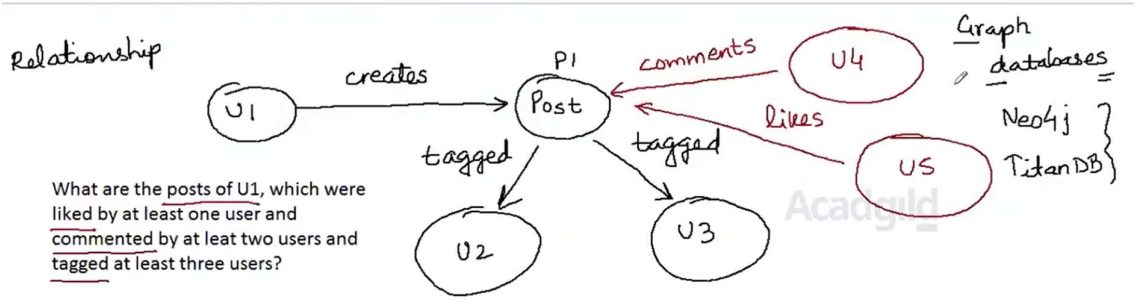
- Apache Hadoop was born out as a **solution to Big Data.**
- Hadoop was created by `Doug Cutting` and `Mike Cafarella.`
- Apache Hadoop software library is a **framework** that ***allows for the distributed processing of large data sets across clusters of computers using simple programming model.***

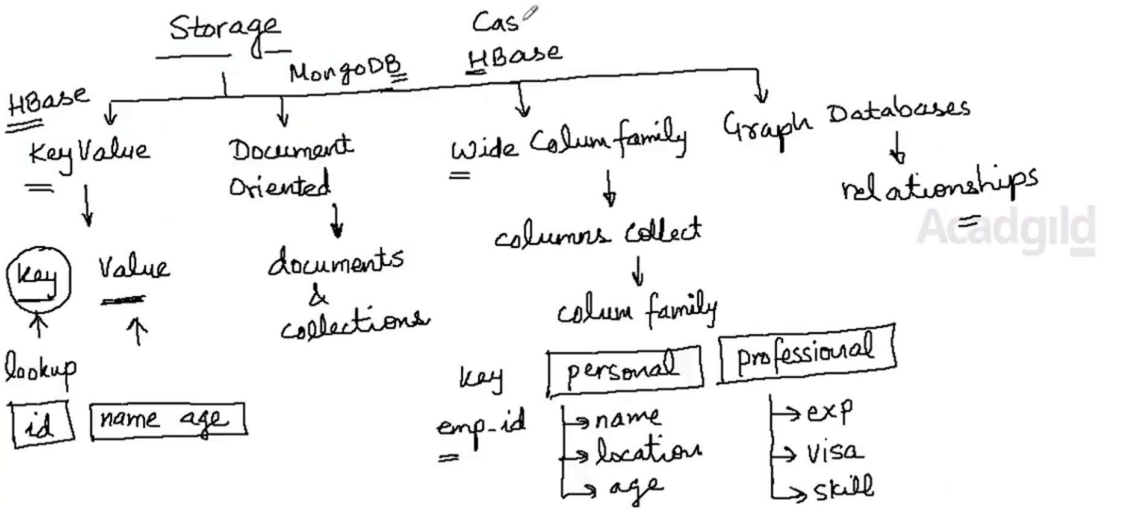
Let's check out the characteristics of Hadoop.

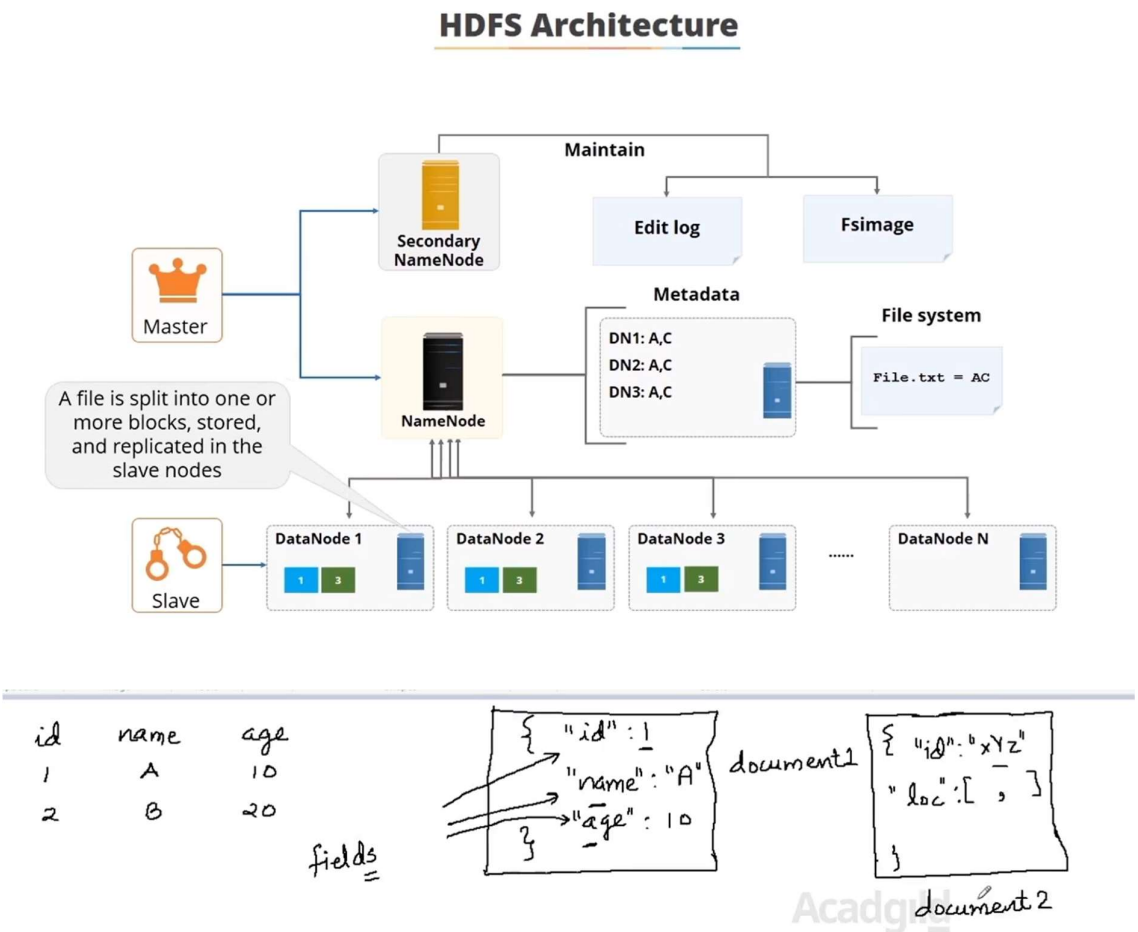
- `High Availability` - The data is highly available/accessible despite hardware failure. It keeps multiple copies of data.
- `Scalability` - It provides horizontal scalability.
- `Fault tolerant` - By default, three replicas of each block is stored across the cluster in Hadoop.
- `Economic` - It is not so expensive as it runs on a cluster of commodity hardware.

Introduction to Mapreduce

- It is the core building blocks to processing in Hadoop Framework.
- **MapReduce** is a *`programming framework`* **that allows us to perform distributed and parallel processing on a large data set in the distributed environment.**
- Users specify the computation in terms of a *`map`* and *`reduce`* function.
- MapReduce model performs *`parallel computation`* across large-scale clusters of a machine.







Data Integrity

- When the data volumes flowing through the system are as large as those that Hadoop can handle, the chances of data corruption are high.
- The usual way to **detect corrupted data** is through **calculating the data checksum**.
- It helps only in error detection not to fix it.
- ***CRC-32*** (*cyclic redundancy check*), **is the commonly used error detecting code** .*It calculates a 32-bit integer checksum for any size of the input.*

Local FileSystem

- The **Hadoop LocalFileSystem** performs *client-side checksumming*.
- When a file is written named as filename, the filesystem client creates a hidden file namely `filename.crc`, in the same directory with the checksum for each chunk of the file transparently.
- ***Raw LocalFile System***

- Disable checksums.
- Used when checksums are not required.

- `Checksum FileSystem`

- Its a wrapper around the File System.
- THe underlying FS is known as `raw FS` .

Compression

When you need to deal with large volumes of data, we need compression to:

- reduce the space required for storing files.
- increase the transfer rate of data.

Codecs

The implementation of ***compression and decompression*** algorithm is known as **codecs**.

Compression format	Hadoop CompressionCodec
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
LZO	com.hadoop.compression.lzo.LzopCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec

The LZO libraries might not be added in Apache distributions as they are GPL-licensed. So Hadoop codecs need to be downloaded separately from here.

Serialization

What is the need of serialization in Hadoop?

Although you can see **data** in a **structured** form, the ***raw data form is a set or a bits stream.*** This *raw data* is the one which travels through the network and later stored.

Now let's come to the point what is serialization ?

`Serialization` is the **method of transforming structured objects into a byte stream** for network transmission or writing to permanent storage. `Deserialization` is the **reverse process of converting a byte flow into a series of structured objects.**

- You can see **serialization** in two areas of **distributed data processing**:

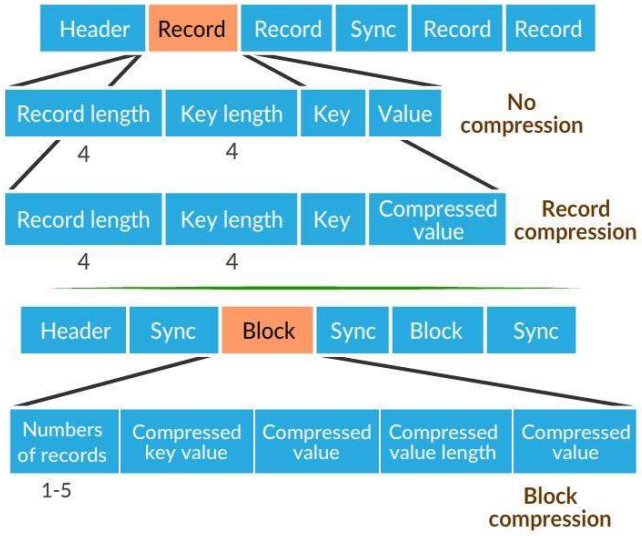
- **Interprocess communication**
- **Permanent storage.**

File-Based Data Structure

*You need a **specialized** data structure for some applications to store your data. To perform **MapReduce-based** processing, each binary data blob does not **scale in its file**, so Hadoop has introduced many high-level containers for these conditions.*

Let's imagine a ***log file*** in which ***each log is a new text line***. If you would like to ***input binary types***, ***plain text is not an appropriate format***. *The* `Hadoop SequenceFile` *class ticks all the boxes in this scenario*, offering the *binary key-value pairs* with a continual data structure. You would use it as a log file format.

Sequence File Format



- The **sequence file** is a **header** and following one or more records.
- The first three bytes of the sequence file is bytes **SEQ**, that function as a magic number, accompanied by a **single byte.**
- The **header** contains **other** fields, including **key and value class names**, compression details, **user-defined metadata, and the sync marker.**
- **Sync** markers appear in the **sequence file between records**.

`Record Format:`

- No compression.
- Record Compression.
- Block Compression.

Here is an Example.

MapFile

A **MapFile** is a *sequence file with an index that allows key searches*.

- MapFile can be considered as a persistent form of `java.util.Map`, which can grow beyond the size of memory kept.
- **Keys** must be instances of **WritableComparable** and **values** as **Writable**.
- **MapFile** is **directory** containing **two** files – **index file and data file**. -- **Index** file contains the **key** and **value** (**key** for the **position** and **value** for the **offset**) of the data file. This **index** file works as a **lookup file**.
- When you need to get /read/process a file, this index file is first referred, and the data is returned based on the result.

Different Types of Files in Hadoop

File should :

- Get Read fast.
- Get Written fast.
- Be Splittable i.e. multiple tasks can run parallel on parts of file.
- Support Schema evolution, allowing us to change schema of file.
- Support advanced compression through various available compression codecs (Bzip2,LZO,Snappy etc).

Text Files (Csv,Tsv)

Behaviour - Each line is a record/data, and lines are terminated by a newline character (\n).

Read/Write - Good write performance but slow reads.

Compression - Do not support Block compression.

Splittable - Text-files are inherently splittable on \n character.

Schema Evolution - Limited Schema evolution (New fields can only be appended to existing fields while old fields can never be deleted).

Sequence File

Behaviour - Each record is stored as a key value pair in binary format.

Read/Write - Good write performance than Text files.

Compression - Support Block compression.

Splittable - Sequence files are splittable.

Schema Evolution - Limited Schema evolution (New fields can only be appended to existing fields while old fields can never be deleted).

Avro File

Behaviour – It is a file format plus a serialization and deserialization framework. Avro uses JSON for defining data types and serializes data in a compact binary format.

Read/Write – Average read/write performance.

Compression - Support Block compression.

Splittable – Avro files are splittable.

Schema Evolution – Was mainly designed for Schema evolution. Fields can renamed, added, deleted while old files can still be read with the new schema.

Columnar File Formats

- In columnar file format instead of just storing rows of data adjacent to one another we also store column values adjacent to each other.
- So datasets are partitioned both horizontally and vertically.

RC File

Behaviour – These are flat files consisting of binary key/value pairs, and it shares much similarity with Sequence File.

Read/Write – Was developed for faster reads but with a compromise with write performance.

Compression - Provides significant Block compression, can be compressed with high compression ratios.

Splittable – RC files are splittable.

Schema Evolution – Was mainly designed for Faster reads so NO schema evolution.

ORC File

Behaviour – A better version of RC file.

Read/Write – Was developed for faster reads but with a compromise with write performance *(Better than RC file)* .

Compression - Provides significant Block compression, can be compressed with high compression ratios *(Better than RC file).*

Splittable – ORC files are splittable at stripe level.

Schema Evolution – Was mainly designed for Faster reads so NO schema evolution.

Parquet File

Behaviour - It is a columnar file format, similar to RC and ORC. Parquet stores nested data structures in a flat columnar format.

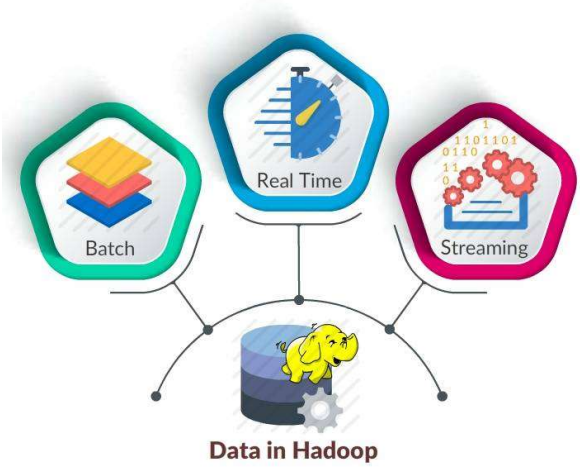
Read/Write - Faster reads with slow writes.

Compression - Support compression mostly with snappy algorithm.

Splittable - Parquet files are conditionally splittable.

Schema Evolution - Limited Schema evolution (New fields can only be appended to existing fields while old fields can never be deleted).

Data Ingestion



Before starting with Sqoop and Flume, lets understand the concept of **Data Ingestion** .

In order to transform data into information or insights, it needs to ingested into Hadoop.

It's the **process of importing, transferring, loading and processing data for for future use in database is known as DATA INGESTION.**

WHAT IS APACHE SQOOP?

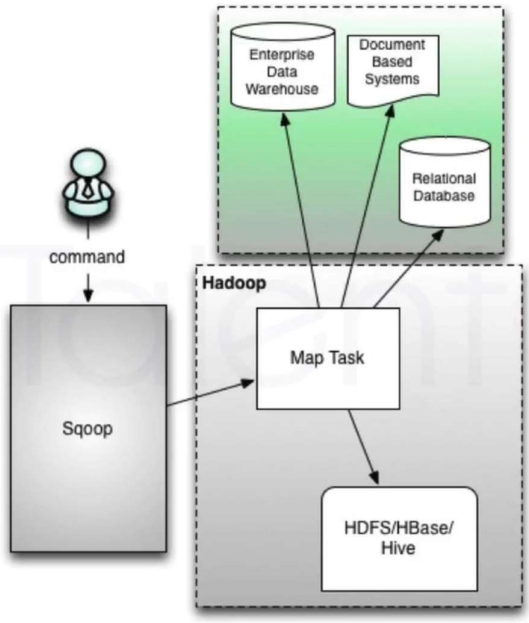
- Apache Sqoop is a tool which is designed to transfer data between relational databases and Hadoop
- Sqoop can be used to transfer data between RDMS like MySQL or Oracle or Teradata etc., to Hadoop
- Sqoop can store data on to HDFS or Hive or HBase or Accumulo
- Sqoop uses MapReduce for importing or exporting data
- Sqoop provides parallel operations and fault tolerance

SQOOP USAGE

- Input to sqoop is either database table or mainframe datasets
- Sqoop will read table row by row into HDFS
- For mainframe datasets sqoop will read records from each mainframe datasets to HDFS
- Output of this import process is a set of files containing a copy of the imported table or datasets
- Import process is performed in parallel, hence there will be multiple files in the target directory
- Output of sqoop can be a text file or binary Avro or sequence files containing serialised record data

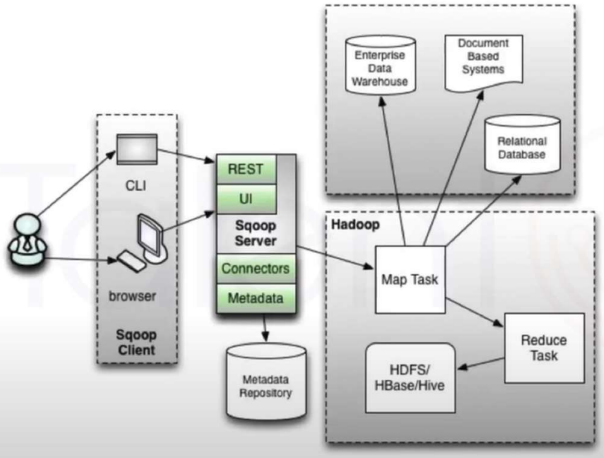
- After applying transformations on the imported data, results can be exported back to relational databases
- Sqoop's export process will read a set of delimited text files from HDFS in parallel, parse them into records and insert them as new records in target database table
- Sqoop provides mechanism to inspect the database we want to work with
- You can list available databases and tables using sqoop commands
- Sqoop provides primitive SQL execution shell (sqoop-eval)

- Sqoop has two major versions Sqoop 1 and Sqoop 2
- Both versions have significant differences in the architecture
- Sqoop 2 is still a work in progress model and the design is subject to change



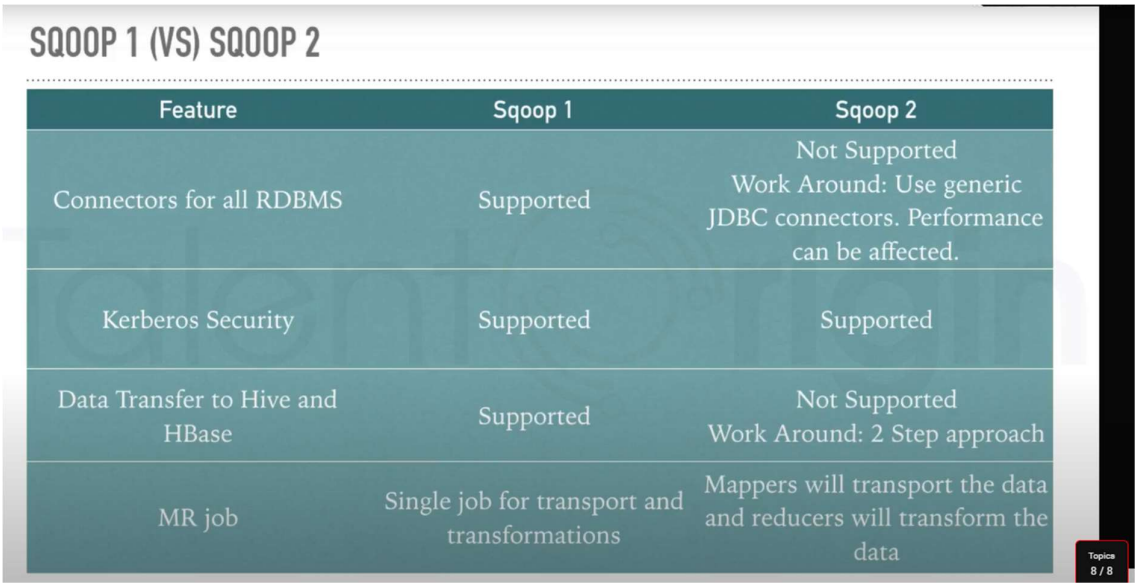
SQOOP VERSIONS –SQOOP 1

- Requires client side installation and configuration
- Provides command line interface (CLI) only
- Launches single map only job which does both data transport and transformation



SQOOP 2 – SQOOP AS SERVICE

- Server side installation and configuration
- Provides both CLI and UI
- REST API exposed for external integration
- Uses Map Task for Transport and Reduce Task for transformations
-



Why Choose Sqoop?

Why do we need Apache Sqoop in the world of big data?

- Using Sqoop, you can **transfer** legacy systems into Hadoop.
- Leverage the **parallel processing** capabilities of Hadoop to process huge amounts of data.
- The results of Hadoop analysis can be again stored back to relational data storage using Sqoop **export** functionality.

```
sqoop import --connect jdbc:mysql://localhost/oozie --
username root --password cloudera --table country --
target-dir /PRA_MV
```

```
Sqoop query sqoop export \ --connect
jdbc:mysql://localhost/cloudera\ --
username cloudera -P \ --table exported \
--export-dir /user/country_imported/part-
m-00000
```

How Sqoop Processes?

Sqoop is:

- Designed to **import/export** individual tables or entire databases.

- Generates **Java classes** (Java classes are packaged into a jar file and deployed in Hadoop cluster to be executed by MapReduce job).
- Job is submitted to Hadoop using **Command Line Tool.**
- By default, four mappers are run with each mapper importing 25% of data.
- The default database is **MySQL.**

NOTE: When a large data is transferred from RDBMS to Hadoop, it is called *import*, and vice versa is called *export*.

Import Tool

- The **import** tool helps to **import RDBMS data into HDFS.**
- Each row in the table is taken as a **record** in HDFS. **SYNTAX:-**

```
sqoop import (generic-args) (import-args)
```

Example:

```
sqoop import-all-tables --connect jdbc:mysql://localhost/<database_name> --
username <username> --password <password> -m 1
```

Importing to Desired Directory

Using the Sqoop **import tool**, the target directory can be specified while importing data into HDFS.

SYNTAX:

```
--target-dir <directory in HDFS>
```

Example:

```
sqoop import --connect jdbc:mysql://localhost/<database_name> --username username
--password <password> -m 1  --target-dir /<directory_name>


sqoop import-all-tables --connect jdbc:mysql://localhost/<database_name> --
username username --password <password> -m 1  --warehouse-dir /<directory_name>
```

Import Subset of Table Data

The subset of a table can be imported using **where** clause in Sqoop import tool.

Syntax:

```
--where "condition"
```

Example:

```
sqoop import --connect jdbc:mysql://localhost/<database_name> --username root --
password root --table emp --m 1 --where "deptno = '20'" --target-dir /sqoop_data/1
```

Use Cases: ELT

- One of the most common uses of Sqoop is **ELT (Extract, Load, Transform)**.
- **Real use case in this scenario is**:
    - Copying the billing data from a database to Hadoop for **billing cycle processing.**

      Instead of processing the batch processing of billing data, we can copy the data to Hadoop for processing in parallel and return the final result, i.e., summarized billing data for a customer back to the database.
    - **Extract** and **load** data from a database and then **transform, modify and process** the data in Hadoop leveraging all the **parallel processing capabilities.**

ETL from DWH

Many companies use RDBMS for **data warehousing.** ETL means to extract the data from the relational database, transform it and place it in data warehouse database, which is also relational for business intelligence and data analytics.

- ETL process is limited.
- Sqoop can help by **copying/extracting** the relational data from our operational database to Hadoop. Also, use Hadoop as an intermediate **parallel processing engine**, which is a part of the overall ETL process.
- The result can be then **copied/load** to our relational database by **Sqoop**.

Data Analysis

For some data analysis, we might require combining current data from the relational database as well as raw data from Hadoop.

We can use Sqoop to copy on a regular interval, current data from the relational database to Hadoop. Make that data accessible along with raw data already present in Hadoop.

This approach allows any data processing jobs in Hadoop to access current data and compare it to historical data. This approach is useful while predicting and analysis jobs.

Why is copying process done always?

Relational database is not very scalable, so it is suitable to store raw data. Hadoop is a backend data platform. It doesn't support transactions which are critical from any front-end type application.

Problems in Real world

So, first let's see what some of the problems that we face in the real world are.

There are multiple ways in which you can transfer data from an external system into Hadoop.

We have already seen one tool called **Sqoop**. Well Sqoop *allows an RDBMS a system to connect to Hadoop and to transfer the structure data.* You can also *manually* copy the data.

However, many times, you might face a situation where you have to take data from external systems and continuously dump it into Hadoop. You call such type of data as **streaming data**. For example, think about something like application server logs which is being generated every second.

To analyze these log files, a person can manually copy the log files and store it into Hadoop or write a script. The above methods are highly inefficient.

What is Flume?

We need a capable system that can transfer the data as generated without losing it into Hadoop. You need a system that can bundle the data over a period of time and dump it into Hadoop. ***So, the solution is Flume.***

Flume is a **tool used to collect, aggregate and transport large amounts of data streaming from many different sources to a centralized data store such as events, log files, etc.**

It also allows for **geo-analytical application.**

- Flume gets **multi-source streaming data for storage and analysis to Hadoop.**
- **To store the data**, Flume gives many options. You can either **directly store it into your Hadoop, which is HDFS or a real-time system such as HBase.**
- Flume provides **horizontal scalability** in case if data streams and volume increases.
- It offers **buffer storage** for real-time spikes.
- It uses a **cross-platform** operating system.

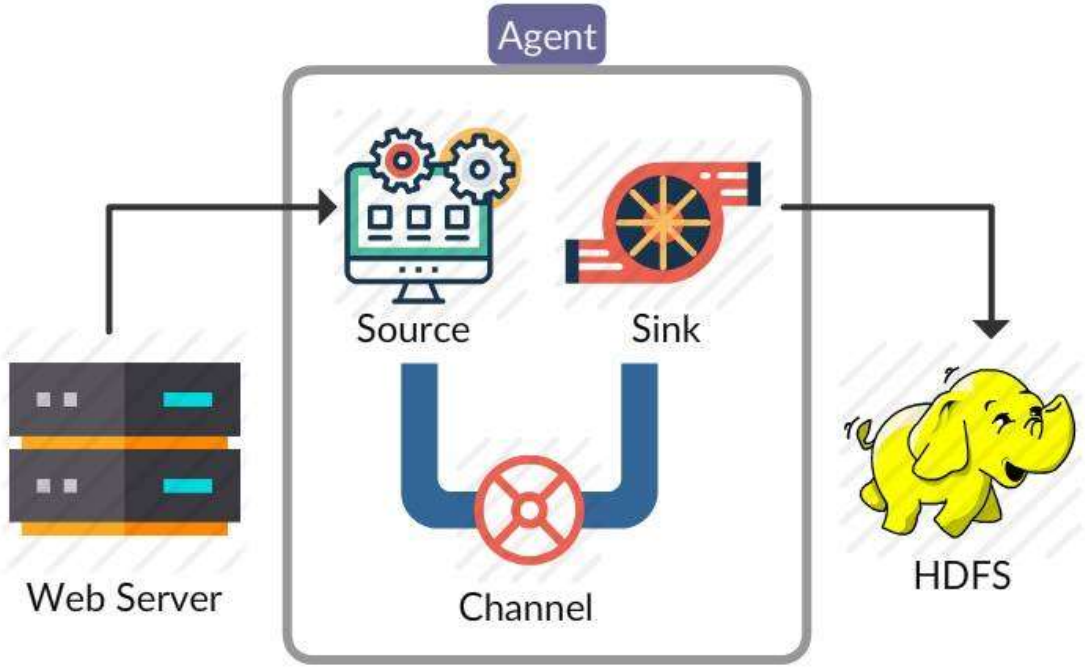
Advantages of Flume

- **Scalable**- log data can be multiplexed and scale to a really large number of servers.
- **Reliable**
- **Customizable**- Flume can customize according to your needs.
- **Declarative and Dynamic Configuration** - Configuration can be written in a file called agent configuration file and is dynamic.

Architecture of Flume

In the Flume Architecture, you can see three things:

- **Web Server**
- **Flume Agent.**
- **HDFS**



Flume considers **data** as **event**. e.g., log entry.

- Events generated by external source **(Web Server)** are consumed by Flume Data source.
- The **format** of the **event** should be in such a format that the **target source should recognize it.**

Flume Agent

Flume agent is a kind of **JVM process** or can be said as an important part of Flume deployment. So, each flume agent has **three components**

- **Source**
- **Channel**
- **Sink**

Source

- It is responsible for sending the event to the channel it is connected to
- It has no control how data is being stored in the channel.
- It supports Netcat, exec, , TCP, and UDP as source data files.

Channel

- The channel acts as a **storehouse** that keeps the events until the flume sink consumes them.
- The channel may use a local file system to store these events.
- There can be **more than one Flume agent**. In this case, flume sink forwards the events to flume source of the other flume agent in the data flow.

Sink

The flume sink removes the events from channels and stores it into an external repository like HDFS or to another flume agent.

- Sink waits for events.
- It is responsible for sending the event to the desired output.
- It manages issues like time out.
- As long as one sink is available, the channel will function.

File Configuration - 1

- Flume agents are **configured using text configuration files.**
- Uses the properties file format.

Now, you will see a single node flume configuration.

`Name the components of the agent`

```
agent1_name.sources = source1_name
agent1_name.sinks = sink1_name
agent1_name.channels = channel1_name
```

`Describing the Source`

```
agent_name1.sources. source_name1.type = value
agent_name1.sources. source_name1.property2 = value
agent_name1.sources. source_name1.property3 = value
```

`Describing the Sink`

```
agent_name1.sinks. sink_name.type = value
```

```
agent_name1.sinks. sink_name.property2 = value
agent_name1.sinks. sink_name.property3 = value
```

`Describing the Channel`

```
agent_name1.channels.channel_name.type = value
agent_name1.channels.channel_name. property2 = value
agent_name1.channels.channel_name. property3 = value
```

`Binding the Source and the Sink to the Channel`

```
agent_name.sources.source_name.channels = channel_name
agent_name.sinks.sink_name.channels = channel_name
```

Starting a Flume Agent

After the configuration of the Flume agent, you need to start the flume agent.

```
./flume-ng agent_name1 -n na  -c conf -f
../conf/<agent_configuration_filename>.properties
```

Why Oozie?

Apache Hadoop, an open source implementation of Google's MapReduce paper and Google File System, has become a de facto platform for processing and storing Big Data within a very short period.

Most of the time all required processing cannot be performed with a single MapReduce, Pig or Hive job. Multiple MapReduce, Pig or Hive jobs always need to be linked, intermediate data is produced and consumed and their execution flow coordinated.

Why Oozie?

Some developers have written **shell scripts** to start one job after another in Hadoop. Some have used the **JobControl class** of Hadoop, which performs multiple MapReduce jobs with topological sorting.

As the above solutions began to be widely used, several problems arose. The **tracking of errors was difficult, and it was difficult to recover from failures**. The progress cannot be monitored easily.

It was clear that a **general purpose system needs to run on a multi-state basis.**

Oozie

Oozie, opened in **2010** by Yahoo, was submitted to the Apache in **2011**. One year later, Apache Oozie became a top project.

Oozie is a **server-based workflow scheduling system to manage Hadoop jobs.**

- It **simplifies** workflow and coordination between jobs.
- You can **schedule** jobs as well as **reschedule** jobs on failure.
- Allows creating **DAG** of workflows.
- **Tightly integrated** with Hadoop stack supporting various Hadoop jobs.

Features of Oozie

Let's discuss some of the important features of Oozie:

- **Execute** and **monitor workflows** in Hadoop.
- **Periodic scheduling** of workflows.
- **Trigger execution** by data availability.
- **HTTP** and **command line interface** + **web console.**

What is Spark?

Apache Spark is an open source cluster computing framework that provides an interface for entire programming clusters with implicit data parallelism and fault-tolerance.

Apache Spark is devised to serve as a `general-purpose` and `fast cluster computing platform`.

- Spark runs **computations in memory** and provides a quicker system for complex applications operating on disk.
- Spark covers various **workloads needing a dedicated distributed systems** namely streaming, interactive queries, iterative algorithms, and batch applications.

Key Features

Performance:

- Faster than Hadoop MapReduce up to 10x (on disk) - 100x (In-Memory)
- Caches datasets in memory for interactive data analysis
- In Spark, tasks are threads, while in Hadoop, a task generates a separate JVM.

Rich APIs and Libraries

- Offers a deep set of high-level APIs for languages such as R, Python, Scala, and Java.
- Very less code than Hadoop MapReduce program because it uses functional programming constructs.

Scalability and Fault Tolerant

- Scalable above 8000 nodes in production.
- Utilizes Resilient Distributed Datasets (RDDs) a logical collection of data partitioned across machines, which produces an intelligent fault tolerance mechanism.

Supports HDFS

- Integrated with Hadoop and its ecosystem
- It can read existing data.

Realtime Streaming

- Supports streams from a variety of data sources like Twitter, Kinesis, Flume, and Kafka.
- We defined a high-level library for stream processing, utilizing Spark Streaming.

Interactive Shell

- Provides an Interactive command line interface (in Python or Scala) for horizontally scalable, low-latency, data exploration.
- Supports structured and relational query processing (SQL), via Spark SQL.

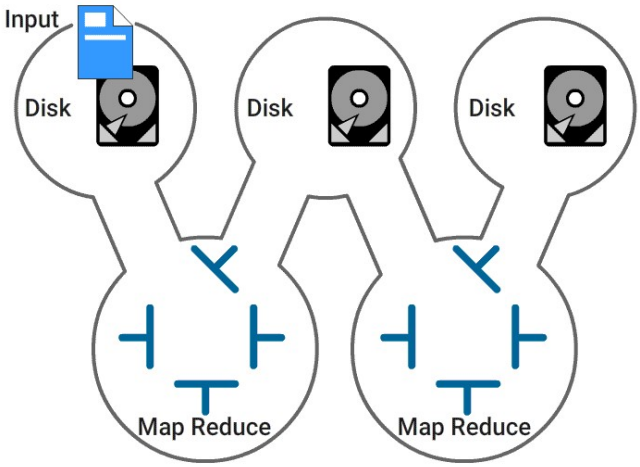
Machine Learning

- Higher level libraries for graph processing and machine learning.
- Various machine learning algorithms such as pattern-mining, clustering, recommendation, and classification.

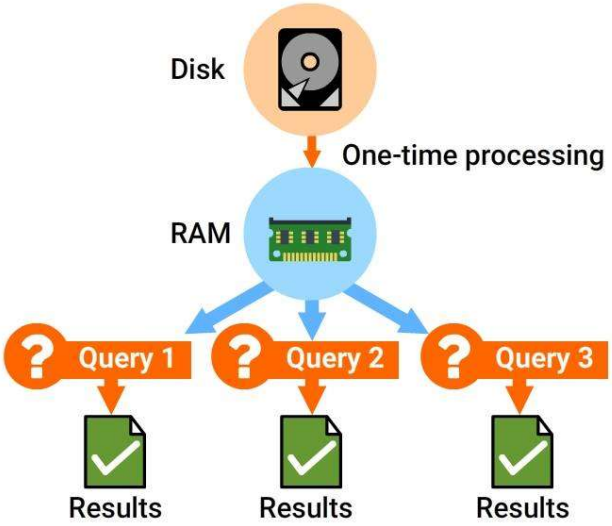
Advantages of Spark Over MapReduce

Let us compare `MapReduce` and `Spark` based on the following essential aspects in detail.

- **Solving Iterative problems**
- **Solving Interactive problems**



Mapreduce Process



Spark vs MapReduce

The other aspects by which Spark differs from MapReduce are summarized below.

Difficulty: Apache Spark is a simpler to program and does not require any abstractions whereas MapReduce is hard to program with abstractions.

Interactivity: Spark provides an interactive mode whereas MapReduce has no inbuilt interactive mode except for Pig and Hive.

- `Streaming`: Hadoop MapReduce offers batch processing on historical data whereas Spark provides streaming of data and processing in real-time.
- `Latency`: Spark caches partial results over its memory of distributed workers thereby ensuring lower latency computations. In contrast to Spark, MapReduce is disk-oriented.

- **Speed**: Spark places the data in memory, by storing the data in Resilient Distributed Databases (RDD). Spark is 100X quicker than Hadoop MapReduce for big data processing.

Supported Languages

- Apache Spark currently supports multiple programming languages, including **Java, Scala, R** and **Python**. The final language is chosen based on the efficiency of the functional solutions to tasks, but most developers prefer **Scala.**
- Apache Spark is built on Scala, thus being proficient in Scala helps you to dig into the source code when something does not work as you expect.
- Scala is a multi-paradigm programming language and supports functional as well as object oriented paradigms. It is a JVM based statically typed language that is safe and expressive.
- Python is in general slower than Scala while Java is too verbose and does not support Read-Evaluate-Print-Loop (REPL).

Applications of Spark

- **Interactive analysis** – MapReduce supports batch processing, whereas Apache Spark processes data quicker and thereby processes exploratory queries without sampling.
- **Event detection** – Streaming functionality of Spark permits organizations to monitor unusual behaviors for protecting systems. Health/security organizations and financial institutions utilize triggers to detect potential risks.
- **Machine Learning** – Apache Spark is provided with a scalable Machine Learning Library named as **MLlib**, which executes advanced analytics on iterative problems. Few of the critical analytics jobs such as sentiment analysis, customer segmentation, and predictive analysis make Spark an intelligent technology.

SparkConf

- **SparkConf** stores configuration parameters for a Spark application.
- These configuration parameters can be properties of the Spark driver application or utilized by Spark to allot resources on the cluster, like memory size and cores.
- SparkConf object can be created with **new SparkConf()** and permits you to configure standard properties and arbitrary key-value pairs via the set() method.

SparkConf

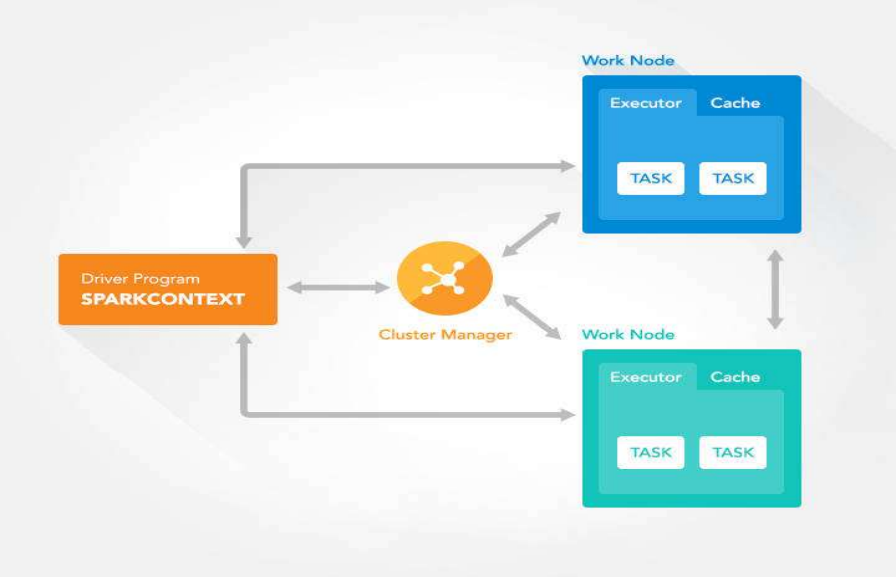
```
val conf = new SparkConf()
             .setMaster("local[4]")
             .setAppName("FirstSparkApp")
```

```
val sc = new SparkContext(conf)
```

Here, we have created a `SparkConf` object specifying the master URL and application name and passed it to a `SparkContext`.

SparkContext

- Main entry point for Spark functionality
- **SparkContext** can be utilized to create broadcast variables, RDDs, and accumulators, and denotes the connection to a Spark cluster.
- To create a **SparkContext**, you first have to develop a **SparkConf** object that includes details about your application.



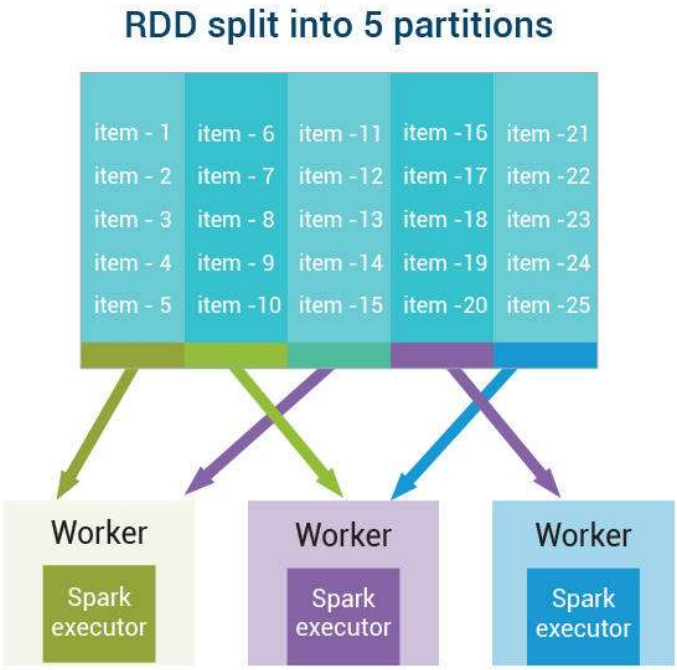
- As shown in the diagram, the Spark driver program uses **`SparkContext`** to connect to the cluster manager for resource allocation, submit Spark jobs and knows what resource manager (YARN, Mesos or Standalone) to communicate.
- Via **`SparkContext`**, the driver can access other contexts like StreamingContext, HiveContext, and SQLContext to program Spark.

`There may be only one SparkContext active per JVM`. Before creating a new one, you have to stop() the active SparkContext.

- In the Spark shell, there is already a special interpreter-aware **SparkContext** created in the variable named as ***sc***.

```
val sc = new SparkContext(conf)
```

RDD



- `Resilient distributed datasets (RDDs)` are the known as the main abstraction in Spark.
- It is a partitioned collection of objects spread across a cluster, and can be persisted in memory or on disk.
- Once created, RDDs are immutable.

Features of RDDs

- **Resilient**, i.e. tolerant to faults using RDD lineage graph and therefore ready to recompute damaged or missing partitions due to node failures.
- **Dataset** - A set of partitioned data with primitive values or values of values, For example, records or tuples.
- **Distributed** with data remaining on multiple nodes in a cluster.

Creating RDDs

Parallelizing a collection in driver program.

E.g., here is how to create a parallelized collection holding the numbers 1 to 5:

```
val data = Array(1, 2, 3, 4, 5)
```

```
val newRDD = sc.parallelize(data)
```

Here, *newRDD* is the new RDD created by calling SparkContext's *parallelize* method.

Referencing one dataset in an external storage system, like a shared filesystem, HBase, HDFS, or any data source providing a Hadoop InputFormat.

For example, text file RDDs can be created using SparkContext's *textFile* method. This method takes an URI for the file (either a local path on the machine, or a hdfs://, s3n://, etc URI) and reads it as a collection of lines to produce RDD *newRDD*.

```
val newRDD = sc.textFile("data.txt")
```

DataFrames

Similar to an RDD, a **DataFrame** is an immutable distributed set of data.

Unlike an RDD, data is arranged into named columns, similar to a table in a relational database.

Created to make processing simpler, **DataFrame** permits developers to impose a structure onto a distributed collection of data, enabling higher-level abstraction.

Creating DataFrames

`DataFrames` can be created from a wide array of sources like existing RDDs, external databases, tables in Hive, or structured data files.

`Applications can create DataFrames from a Hive table, data sources, or from an existing RDD with an SQLContext.`

The subsequent example creates a DataFrame based on the content of a JSON file:

```
val sc: SparkContext // An existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
val df = sqlContext.read.json("home/spark/input.json")
// Shows the content of the DataFrame to stdout
df.show()
```

SQL on DataFrames

The **sql** function on a SQLContext allows applications to run SQL queries programmatically and returns the result as a DataFrame.

```
val sc: SparkContext // An existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
val df = sqlContext.read.json("home/spark/input.json")
```

```
input.registerTempTable("students")

val teenagers = sqlContext.sql("SELECT name, age FROM students WHERE age >= 13 AND
age <= 19")
```

Dataset

- Dataset is a new interface included in Spark 1.6, which provides the advantages of RDDs with the advantages of Spark SQL's optimized execution engine.
- It is an immutable, strongly-typed set of objects that are mapped to a relational schema.
- A DataFrame is a known as a Dataset organized into named columns.
- Dataset will act as the new abstraction layer for Spark from Spark 2.0.

Creating a Dataset

```
val lines = sqlContext.read.text("/log_data").as[String]

val words = lines

  .flatMap(_.split(" "))

  .filter(_ != "")
```

Here, we have created a dataset **lines** on which RDD operations like *filter* and *split* are applied.

Benefits of Dataset APIs

1. Static-typing and runtime type-safety
2. High-level abstraction and custom view into structured and semi-structured data
3. Higher performance and Optimization

SparkSession - a New Entry Point

- `SparkSession`, introduced in Apache Spark 2.0, **offers a single point of entry** to communicate with underlying Spark feature and enables programming Spark with Dataset APIs and DataFrame.
- In previous versions of Spark, **spark context** was the entry point for Spark. For streaming, you required *StreamingContext* for hive *HiveContext* and for SQL *SQLContext*.
- As Dataframe and DataSet APIs are the new standards, Spark 2.0 features `SparkSession` as the new entry point.

- SparkSession is **a combination of** ***HiveContext***, ***StreamingContext***, **and** ***SQLContext***. All the APIs available on these contexts are available on SparkSession also. It internally has a spark context for actual computation.

Creating SparkSession

A **SparkSession** can be built utilizing a **builder pattern**. The builder will automatically reuse an existing SparkContext if one exists; and create a SparkContext if it does not exist.

```
val dataLocation = "file:${system:user.dir}/spark-data"
// Create a SparkSession
val spark = SparkSession
   .builder()
   .appName("SparkSessionExample")
   .config("spark.sql.data.dir", dataLocation)
   .enableHiveSupport()
   .getOrCreate()
```

Configuring Properties

Once the `SparkSession` is instantiated, you can configure the runtime config properties of Spark. E.g: In this code snippet, we can alter the existing runtime config options.

```
//set new runtime options
spark.conf.set("spark.executor.memory", "1g")
spark.conf.set("spark.sql.shuffle.partitions", 4)
//get all settings
val configMap:Map[String, String] = spark.conf.getAll()
```

Running SQL Queries

SparkSession is the entry point for reading data, akin to the old `SQLContext.read`. It can be utilized to execute SQL queries across data, getting the results back as a DataFrame.

```
val jsonData = spark.read.json("/home/user/employee.json")

display(spark.sql("select * from employee"))
```

Access to Underlying SparkContext

SparkSession.sparkContext returns the subsequent SparkContext, employed for building RDDs and managing cluster resources.

```
spark.sparkContext

res17: org.apache.spark.SparkContext = org.apache.spark.SparkContext@2debe9ac
```

Shared Variables

Usually, when a function passed to a Spark operation is run on a remote cluster node, it runs on individual copies of all the variables used in the function.

These variables are copied to every machine, and no updates to the variables on the remote machine are delivered back to the driver program.

Spark offers two limited types of shared variables for two common usage patterns: **accumulators** and **broadcast variables**.

Broadcast Variables

- Enables the programmer to keep a read-only variable cached on each machine instead of shipping a copy of it with tasks.
- Generated from a variable *v* by calling ***SparkContext.broadcast(v)***
- Its value can be accessed by calling `the value method`. The subsequent code shows this:

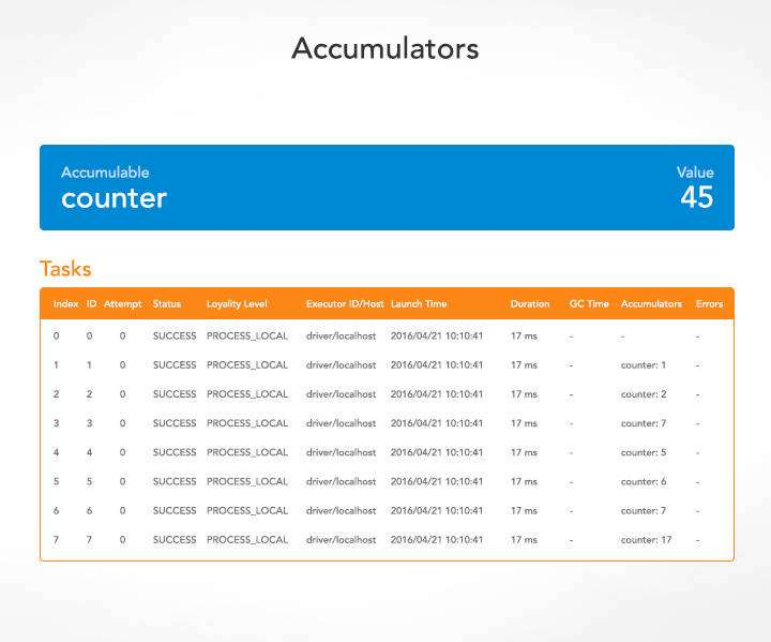
```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3, 4, 5))

broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)

scala> broadcastVar.value

res0: Array[Int] = Array(1, 2, 3, 4, 5)
```

Accumulators

- Accumulators are known as the variables that are only “added” via an associative and commutative operation and can, hence, be efficiently supported in parallel.
- They can be utilized to implement sums or counters.
- Programmers can include support for new types.
- Spark natively offers support for accumulators of numeric types.

You can create unnamed or named accumulators as a user. As seen in the image, a named accumulator (here, **counter**) will be displayed in the web UI for the stage that modifies that

accumulator. Spark shows the value for each accumulator modified by a task in the “Tasks” table.



Introduction to RDD Operations

- The following example shows the implementation of Spark - MapReduce’s example, word count. You can view that Spark offers support for **operator chaining**.
- It turns out to be handy when doing pre- or post-processing on data, like filtering data before running a complex MapReduce job.

```
val file = sc.textFile("hdfs://.../wordcounts-*.gz")
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://.../wordcountOutput")
```

- Here `flatMap`, `map`,`reduceByKey` and `saveAsTextFile` are the operations on the RDDs.

Transformations

- Transformations are functions that use an RDD as the input and return one or more RDDs as the output.
- *randomSplit, cogroup, join, reduceByKey, filter, and map* are examples of few transformations.

- Transformations do not change the input RDD, but always create one or more new RDDs by utilizing the computations they represent.
- By using transformations, you incrementally create an RDD lineage with all the parent RDDs of the last RDD.
- Transformations are *lazy*, i.e. are not run immediately. Transformations are done on demand.
- Transformations are executed only after calling an action.

Example of Transformations

- **filter(func)**: Returns a new dataset (RDD) that are created by choosing the elements of the source on which the function returns true.
- **map(func)**: Passes each element of the RDD via the supplied function.
- **union()**: New RDD contains elements from source argument and RDD.
- **intersection()**: New RDD includes only common elements from source argument and RDD.
- **cartesian()**: New RDD cross product of all elements from source argument and RDD.

Actions

- Actions return concluding results of RDD computations.
- Actions trigger execution utilizing *lineage graph* to load the data into original RDD, and then execute all intermediate transformations and write final results out to file system or return it to Driver program.
- `Count, collect, reduce, take, and first` are few actions in spark.

Example of Actions

- **count()**: Get the number of data elements in the RDD
- **collect()**: Get all the data elements in an RDD as an array
- **reduce(func)**: Aggregate the data elements in an RDD using this function which takes two arguments and returns one
- **take (n)**: Fetch first n data elements in an RDD computed by driver program.
- **foreach(func)**: Execute function for each data element in RDD. usually used to update an accumulator or interacting with external systems.
- **first()**: Retrieves the first data element in RDD. It is similar to take(1).
- **saveAsTextFile(path)**: Writes the content of RDD to a text file or a set of text files to local file system/HDFS.

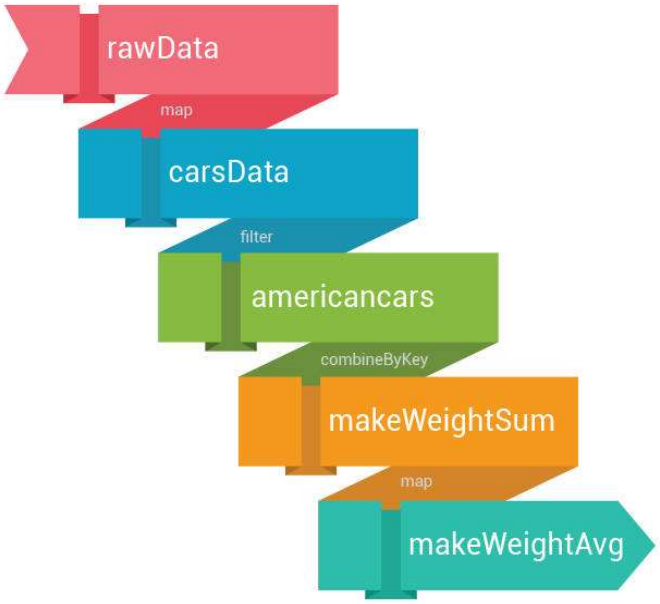
Sample Code Snippet

Let us look at the outline of a code snippet that uses transformations and actions on a RDD.

```
records = spark.textFile(“hdfs://...”)
errors = records.filter(_.startsWith(“ERROR”))
messages = errors.map(_.split(‘\t’)(2))
cachedMessages = messages.cache()
cachedMessages.filter(_.contains(“400”)).count
```

In this program, **records** is the Base RDD and **errors** is a transformed RDD created by applying the filter transformation.

Count is the action called upon which the transformations start to execute.

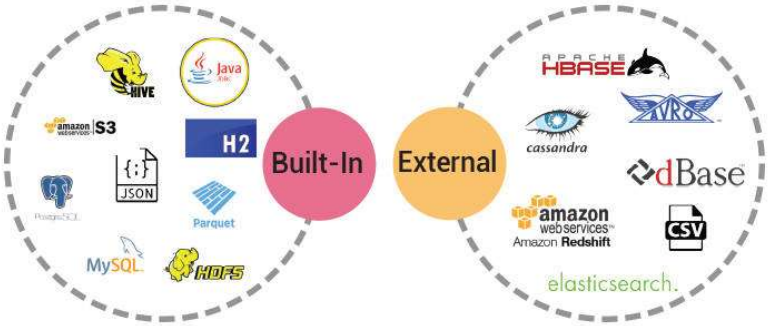


Lazy Evaluation

- When we call a transformation on RDD, the operation is not immediately executed. Alternatively, Spark internally records meta-data to show this operation has been requested. It is called as `Lazy evaluation`.
- Loading data into RDD is lazily evaluated as similar to how transformations are.
- In Hadoop, developers often spend a lot of time considering how to group together operations to minimize the number of MapReduce passes. It is not required in case of Spark.

- Spark uses lazy evaluation to reduce the number of passes it has to take over our data by grouping operations together. Hence, users are free to arrange their program into smaller, more controllable operations.

Spark Sources



- The Data Sources API offers a single interface for storing and loading data using Spark SQL.
- In addition to the sources that come prepackaged with the Apache Spark distribution, this API offers an integration point for external developers to add support for custom data sources.

File Formats

The following are some of the file formats supported by Spark.

- Text
- JSON
- CSV
- Sequence File
- Parquet
- Hadoop InputOutput Format

Storage/Source Integrations

Although often linked to the Hadoop Distributed File System (HDFS), Spark can combine with various open source or commercial third-party data storage systems, including:

- Google Cloud
- Elastic Search
- JDBC
- Apache Cassandra
- Apache Hadoop (HDFS)
- Apache HBase
- Apache Hive

Developers are most expected to choose the data storage system they are previously utilizing elsewhere in their workflow.

Hive Integration

- Hive comes packaged with the Spark library as **HiveContext** that inherits from `SQLContext`. Utilizing HiveContext, you can create and find tables in the **HiveMetaStore** and write queries on it using **HiveQL**.
- When `hive-site.xml` is not configured, the context automatically produces a metastore named as **metastore_db** and a folder known as **warehouse** in the current directory.

Data Load from Hive to Spark

Now let us see an example code to load data from Hive in Spark.

Consider the following example of `employee` in a text file named `employee.txt`. We will first create a hive table, load the employee record data into it using HiveQL language, and apply some queries on it.

Use the subsequent command for initializing the HiveContext into the Spark Shell

```
scala> val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

Let us now create a table named employee with the fields id, name, and age using HQL.

```
scala> sqlContext.sql("CREATE TABLE IF NOT EXISTS employee(id INT, name STRING,
age INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'")
```

Now we shall load the employee data into the employee table in Hive.

```

```

```
scala> sqlContext.sql("LOAD DATA LOCAL INPATH 'employee.txt' INTO TABLE employee")
```

Next, we shall fetch all records using HiveQL select query.

```
scala> val result = sqlContext.sql("FROM employee SELECT id, name, age")
```

To show the record data, call the show() method on the result DataFrame.

```
scala> result.show()
```

Spark Cluster

- A Spark application includes a **single driver process** and a **collection of executor processes** scattered over nodes on the cluster.
- Both the executors and the driver usually run as long as the application runs.

Spark Driver

- Program that **produces the SparkContext**, connecting to a given Spark Master.
- **Declares the actions and transformations** on RDDs of data.

Spark Executors

- **Runs the tasks, return results** to the driver.
- **Offers in memory storage** for RDDs that are cached by user programs.
- Multiple executors per nodes possible

Cluster Managers

- **Standalone** – a simple cluster manager added with Spark that makes it simple to establish a cluster.
- **Apache Mesos** – a cluster manager that can run service applications and Hadoop MapReduce.
- **Hadoop YARN** – the resource manager in Hadoop 2.

Commonly Used Options

- **class**: entry point for your application (e.g. org.apache.spark.examples.SparkPi)
- **master**: master URL for the cluster (e.g. spark://23.195.26.187:7077)
- **deploy-mode**: Whether to deploy your driver on the worker nodes (cluster) or locally as an external client (client) (default: client)
- **conf**: Arbitrary Spark configuration property in key=value format.
- **application-jar**: Path to a bundled jar with the application and dependencies.
- **application-arguments**: Arguments passed to the main method of your main class, if any.

Launching Applications with Spark-submit

./bin/spark-submit

--class <main-class>

--master <master-url>

--deploy-mode <deploy-mode>

--conf <key>=<value>

... # other options

<application-jar>

[application-arguments]

Deployment Modes

Choose which mode to run using the --deploy-mode flag

1. **Client** - Driver runs on a dedicated server (e.g.: Edge node) inside a dedicated process. Submitter starts the driver outside of the cluster.
2. **Cluster** - Driver runs on one of the cluster's Worker nodes. The Master selects the worker. The driver operates as a dedicated, standalone process inside the Worker.

Running a Spark Job

- Spark submit program initiated with spark driver; creates logical DAG.
- Spark Driver program checks with the cluster manager-YARN (Mesos or Standalone) for resource availability for executors and launches it.
- Executors created in Nodes, register to spark driver.

- Spark driver converts the actions and transformations defined in the main method and allocate to executors.
- Executors performs the transformations and; actions return

values to the driver.

- While reading from an HDFS, each executor directly applies the subsequent Operations, to the partition in the same task.

RDD Caching and Persisting

- In Spark, you can utilize few RDDs multiple times. If we repeat the same process of RDD evaluation every time, it is needed or taken into action, this task can be time-consuming and memory-consuming, especially for iterative algorithms that look at data multiple times.
- To resolve the problem of repeated computation, the method of **caching** or **persistence** came into the picture.
- RDDs can be cached with the help of cache operation. They can also be persisted using persist operation.
- **Cache persists** with default storage level **MEMORY_ONLY**.
- RDDs can also be unpersisted to eliminate RDD from a permanent storage like memory and disk.

How to Assign a Storage Level

- Let us see how to persist an RDD to a storage level.

```
result = input.map(<Computation>)

result.persist(LEVEL)
```

- By default, Spark uses the algorithm of **Least Recently Used (LRU)** to remove old and unused RDD to release more memory.
- We can also manually remove remaining RDD from memory by using **unpersist()**.

Storage Levels

Persistence in Spark

Persistence Level	Description
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, DISK_ONLY_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.

Spark SQL is a component of the Spark ecosystem and is used to access **structured** and **semi-structured** information.

- It is used for relational big data processing in integration with Scala, Python, Java, etc.
- It also provides the SQL query execution support to Spark. The process of querying data can be done on data stored in internal Spark RDDs and other external data sources.

Spark SQL is the solution for data analysis from the Spark family of tool sets. It runs SQL on top of Spark.

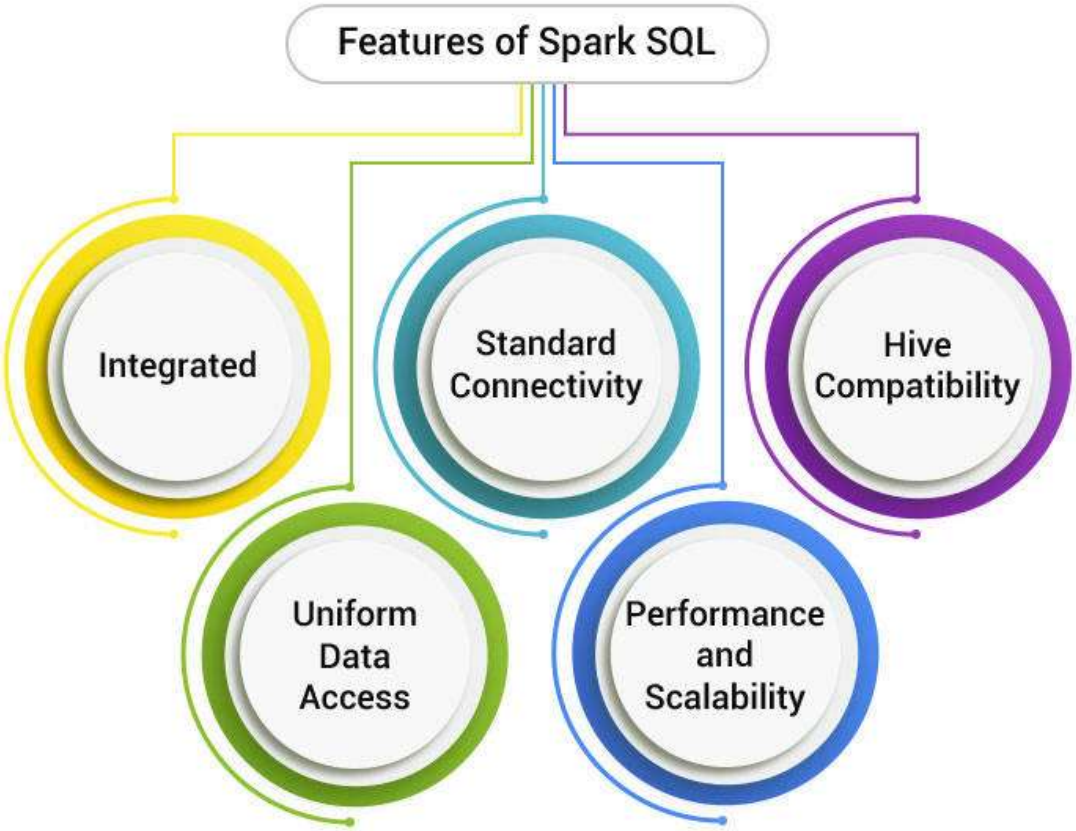
- Spark SQL can deal with data from a wide variety of data sources such as:
  - Structured data files
  - Tables in Hive
  - External Databases
  - Existing RDD
- Spark SQL uses easy-to-use and straightforward Domain Specific Language (DSL) for selecting, filtering, and aggregating data.

Goals of Spark SQL

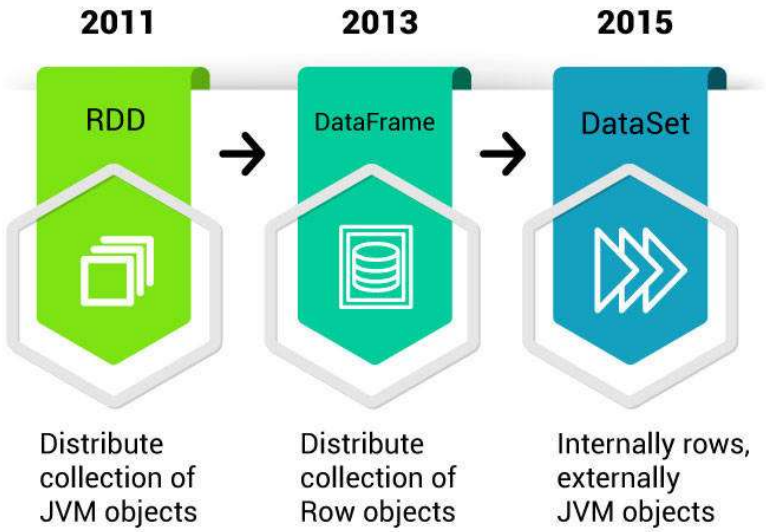
- **Relational processing** can be done on both native RDDs and external sources.
- DBMS techniques ensure **high performance** processing.

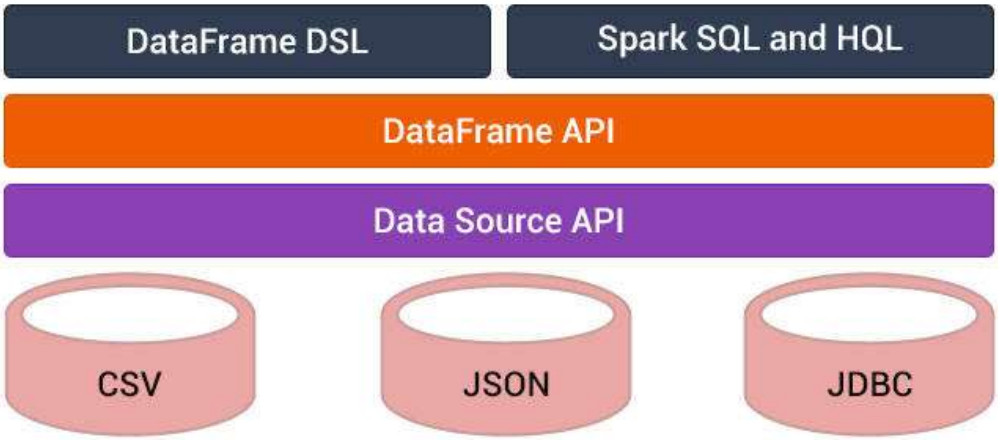
- Easy to process **new data sources** (both semi-structured data and external sources).
- Supports **advanced analytics** algorithms such as machine learning and graph processing.

Features of Spark SQL



- **Integrated**: Spark SQL provides the integration of SQL queries with Spark programs.
- **Unified Data Access**: Spark SQL allows users to load and query data from different data sources in a variety of data formats including Hive, JSON, Parquet, Avro, JDBC, and ORC.
- **Hive Compatibility**: Spark SQL reuses the Hive frontend, and Hive MetaStore provides the compatibility with the execution of Hive queries and UDFs.
- **Standard Connectivity**: Spark SQL provides the connection through JDBC or ODBC through the server mode.
- **Scalability**: Spark SQL is highly scalable and supports the mid-query fault tolerance, allowing it scale to large jobs. The same execution engine is used for interactive and long queries.


2011
2013
2015
RDD
DataFrame
DataSet
Distribute collection of JVM objects
Distribute collection of Row objects
Internally rows, externally JVM objects


DataFrame DSL
Spark SQL and HQL
DataFrame API
Data Source API
CSV
JSON
JDBC

Spark SQL is composed of four libraries that are used to interact with relational and procedural processing. They are:

- DataSource API
- DataFrame API
- SQL Interpreter and Optimizer
- SQL Service

Architecture - DataSource API

- The bottom layer in the architecture of Spark SQL.
- The universal layer used for reading and storing structured and semi-structured data into Spark SQL.
- Built-in support to read data from various input formats such as Hive, Avro, JSON, JDBC, and Parquet.
- Fetches the data from different sources and moves to next layer DataFrame API.

Architecture - DataFrame API

A **DataFrame** is considered as a distributed set of data organized into named columns. The data read through DataSource API is converted to a tabular column to execute the SQL operations.

- Similar to a relational table in SQL used for storing data into tables.
- Processes data with a size ranging from Kilobytes to Petabytes on a single node or multi-node clusters.
- Integrated easily with all Big data tools and frameworks.
- Provides API for Scala, Python, Java and R programming.
- Supports several data formats and storage systems.

SQL Interpreter and Optimiser:

- Constructed using Scala functional programming and provides pattern matching abilities.
- Supports run-time and resource utilization optimization
- Rule-based optimization enables queries to run faster than RDD's.
- Converts the dataframe based code to catalyst expressions and then to Java bytecode.

SQL Service:

- The starting point to work with structured data in Spark.
- Creates DataFrame objects and executes SQL queries.
- Includes ANSI SQL parser that supports subqueries.

There are two ways of interacting with the Spark SQL Library. They are:

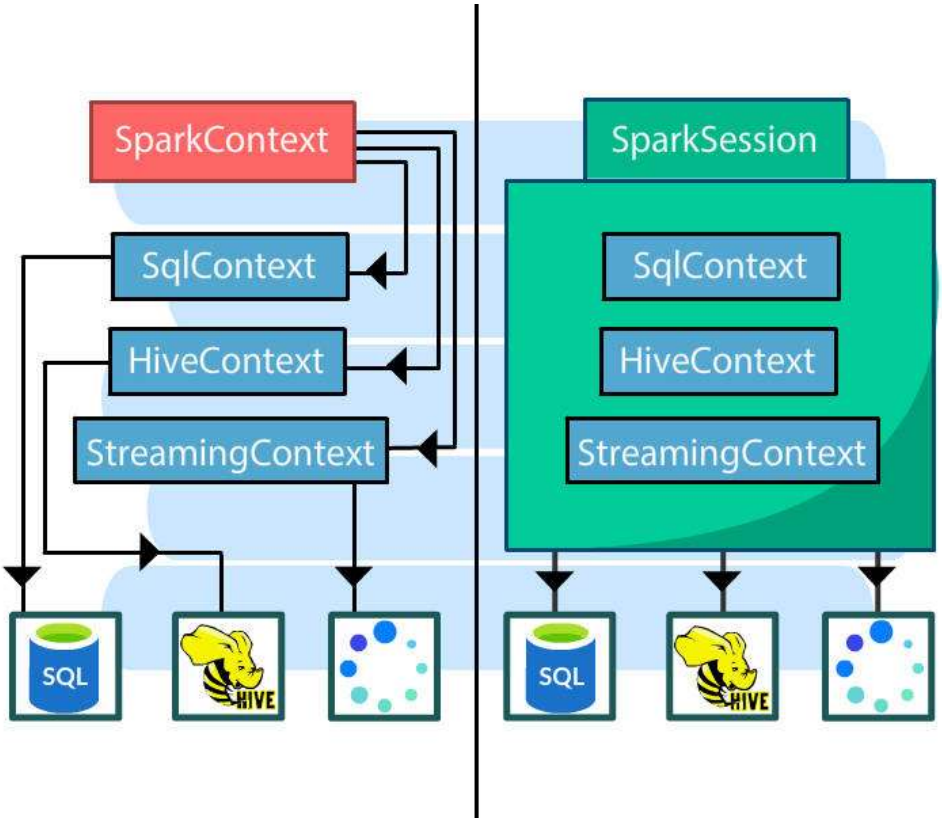
- SQL queries
- DataFrame API calls

The queries/calls are converted to *language-neutral expressions* called ***unresolved logical plan***. A ***logical plan*** is derived from the unresolved logical plan by performing validations against the metadata in DataFrame.

By applying standard optimization rules to the logical plan, it is further transformed to a ***optimized logical plan***. Finally, the optimized logical plan is split into multiple ***physical plans*** that are pushed down to RDDs to operate on the data.

The best out of multiple physical plans are chosen based on the optimal cost. `Cost` is the runtime and resource utilization.

Introducing SparkSession



In the earlier versions of Spark, RDD being the main API, `SparkContext` was considered as the entry point to Spark. SparkContext uses SparkConf to determine the configuration details for the given application. There were different connection objects like `StreamingContext` for Streaming, `SqlContext` for SQL, `HiveContext` for Hive and so on. SparkContext connects to any of these Contexts processing.

In Spark 2.0, there is a new entry point for Dataset and Dataframe APIs, known as, **SparkSession**. It is a combination of SqlContext, HiveContext, and StreamingContext. These contexts need not be created explicitly as SparkSession has access to all these contexts.

Creating SparkSession

SparkSession uses builder design pattern to create an instance. This pattern will reuse a Spark context if it exists, otherwise, creates a new Spark context.

The following code is to create a Spark session.

```
val sparkSession = SparkSession.builder.
  master("local")
  .appName("Spark session in Fresco")
  .getOrCreate()
```

The value `local` for the master is used as a default value to launch the application in test environment locally.

Setting Config Options

Used to set and get runtime configuration options.

```
spark.conf.set("spark.fresco.config", "play")


spark.conf.get("spark.fresco.config")
```

Reading Spark SQL from Hive

The following code is used to read from Hivetables using SparkSession object.

```
//drop the table if it already exists in the same name.


spark.sql("DROP TABLE IF EXISTS sample_hive_table")


//save as a hive table


spark.table("sample_table").write.saveAsTable("sample_hive_table")


// query on hive table


val resultsHiveDF = spark.sql("SELECT name, rollno,totalmark,division FROM
sample_hive_table WHERE totalmark > 40000")


resultsHiveDF.show(10)
```

DataFrame - Introduction

In Spark SQL, the programming abstraction is achieved using the concept of **DataFrames** (starting from Spark 1.3 release). It is based on the data frame concept in R language.

Dataframe allows **data selection, filtering, and aggregation**. Since data frames store more information about the structure of the data, the processing is done efficiently as compared to RDD.

DataFrame is similar to relational database tables with rows of named columns.

Creating DataFrame

With a SparkSession, applications can create DataFrame easily from

- **an existing RDD**
- **a hive table**
- **Spark data sources**.

To create a DataFrame from a List or Seq using spark.createDataFrame:

```
    val df = spark.createDataFrame(List(("John", 35), ("Thomas", 30), ("Martin", 15)))
```

Here `spark` mentioned is the **SparkSession** object.

To create a DataFrame based on the content of a JSON file:

```
    val df = spark.read.json("examples/src/main/resources/data.json")
```

DataFrame Content

To show the content of the DataFrame:

```
    df.show()
```

Output:

```
// +----+-------+
// | name   |age|
// +----+-------+
```

```
// |  John  | 35|
// |  Thomas| 30|
// |  Martin| 15|
// +----+-------+
```

Querying DataFrames

It is possible to run SQL queries programmatically using a dataframe and return the result as a DataFrame.

To do that, the **dataframe has to be first registered as a temporary view**.

```
val df =  spark.read.json("examples/src/main/resources/data.json")


df.createOrReplaceTempView("data")


val sqlDF = spark.sql("SELECT * FROM data")
 sqlDF.show()
```

Output:

```
 // +----+-------+
 // | name   |age|
 // +----+-------+
 // |  John  | 35|
 // |  Thomas| 30|
 // |  Martin| 15|
 // +----+-------+
```

DataFrames across Sessions

To query DataFrames using SQL, one can also choose to create a temporary global view using **createGlobalTempView**. This way the temporary view can be made available for querying across the sessions.

However, scope of dataframe view created using **createOrReplaceTempView** is session restricted and is not visible outside of the session where the view is created.

```
val df =  spark.read.json("examples/src/main/resources/data.json")


df.createGlobalTempView("data")
```

```
val sqlDF = spark.sql("SELECT * FROM data")
sqlDF.show()
```

DataFrames - Summary

Dataframe API in Spark SQL:

- Improves **performance and scalability**.
- Avoids **garbage-collection cost** for individual objects for every row in the dataset.

In spite of the advantages mentioned above, there are few challenges while using DataFrames.

- Since **compile-time type safety** provision is not available with DataFrame API, the data cannot be manipulated if the structure is not known.
- Another challenge is that a domain object cannot be regenerated once it is converted into dataframe.

Dataset

`Dataset` is known as a data structure in Spark SQL that provides type safety and object-oriented interface.

Dataset:

- Acts as an extension to DataFrame API.
- Combines the features of DataFrame and RDD.
- Serves as a functional programming interface for working with structured data.
- Overcomes the limitations of RDD and DataFrames - the **absence of automatic optimization** in RDD and **absence of compile-time type safety** in Dataframes.

Features of Dataset

- Offers **optimized query** using **`Tungsten`**and `Catalyst Query optimizer`. These will be discussed in upcoming topics.
- Capable of **analyzing during compile-time**.
- Capable of **converting type-safe dataset to untyped DataFrame** using methods - `toDS():Dataset[A]`, `toDF():DataFrame` and `toDF(columnnames:String):DataFrame`.
- Delivers **high performance** due to faster computation.
- Low memory consumption.
- Provides **unified API** for Java and Scala.

Creating Datasets

Dataset uses a specialized Encoder to serialize the objects for processing or transmitting over the network.

Dataset can be created in the following ways:

Encoders are created for case classes. Call `.toDS()` on a sequence to convert the sequence to a Dataset.

```
    val dsCaseClass = Seq(Person("John", 35)).toDS()
    dsCaseClass.show()
```

Output:

```
// +----+---+
// |name|age|
// +----+---+
// |John| 35|
// +----+---+
```

Dataset from RDD

A dataset can be created from RDD as shown:

```
    val ds=rdd.DS()
    ds.show()
```

Output:

```
// +----+-------+
// | name   |age|
// +----+-------+
// |  John  | 35|
// |  Thomas| 30|
// |  Martin| 15|
// +----+-------+
```

Dataset from DataFrame

Dataset can be created from DataFrame as well. Call `df.as[SomeCaseClass]` to convert the DataFrame to a Dataset.

```
 val peopleDS =
spark.read.json("examples/src/main/resources/data.json").as[Person]

 peopleDS.show()
```

Output:

```
 // +----+-------+

 // | name   |age|

 // +----+-------+

 // |  John  | 35|

 // |  Thomas| 30|

 // |  Martin| 15|

 // +----+-------+
```

Spark API Release

1

- DataFrames released in Spark 1.3 on 2013.
- DataSet released in Spark 1.6 on 2015.

SPARK API Data Representation

2

- DataFrames is an **organized form of distributed data** into named columns that is similar to a table in a relational database.
- Dataset is an **upgraded release of DataFrame** that includes the functionality of object-oriented programming interface, type-safe and fast.

Data Format

3

- In DataFrames, data is represented as a **distributed collection of row objects**.
- In DataSets, data is represented as **rows internally** and **JVM Objects externally**.

Data Transformation

4

- After RDD transformation into **dataframe**, it **cannot be regenerated** to its previous form.
- After RDD transformation into a **dataset**, it is **capable of converting back** to original RDD.

Type-Safety Check

5

- In dataframe, a runtime error will occur while accessing a column that is not present in the table. It does not provide compile-time type safety.
- In dataset, compile time error will occur in the same scenario as dataset provides the compile-time type safety.

Data Serialization

6

- The Dataframe API provides a Tungsten execution backend that handles the memory management explicitly and generates bytecode dynamically.
- The Dataset API provides the encoder that handles the conversion from JVM objects to table format using Spark internal Tungsten binary format.

Supported Programming Languages

7

- Dataframe supports the programming languages such as Java, Python, Scala, and R.
- Dataset supports Scala and Java only.

Datatypes in Spark SQL

Numeric Datatypes

- **ByteType** - Represents one-byte signed integer numbers ranging from -128 to 127.
- **IntegerType** - Represents four-byte signed integer numbers.
- **FloatType** - Represents four-byte single precision floating point numbers.
- **DecimalType** - Represents arbitrary-precision signed decimal numbers.
- **StringType** - Represents character string values.
- **BinaryType** - Represents byte sequence values.
- **BooleanType** - Represents boolean values.
- **TimestampType** - Represents date and time values including values of fields - year, month, day, hour, minute, and second.

Complex Datatypes - ArrayType

ArrayType(elementType, containsNull)

- Refers to sequence of elements with the type of `elementType`.
- `containsNull` is used to indicate if elements in a ArrayType value can have null values.

Let's create a DataFrame with an ArrayType column to list best cricket players of some countries.

```
val playersDF = spark.createDF(
```

```
List(
 ("India", Array("Sachin", "Dhoni")),
 ("Australia", Array("Ponting"))),
List(
("team_name", StringType, true),
("top_players", ArrayType(StringType, true), true)
)
)
```

playersDF.show()

```
|team_name   |  top_players  |
-----------------------------------
|India      ||[Sachin, Dhoni]   |
|Australia  |   [Ponting]      |
-------------------------------------
```

MapType(keyType, valueType, valueContainsNull)

- Contains a set of key-value pairs.
- `keyType` denotes the data type of keys, and `valueType` denotes the data type of values.
- Keys cannot have null values.
- `valueContainsNull` is used to verify whether values of a MapType value can have null values.

DataFrame with MapType

Let's create a DataFrame with a MapType column to list the good and bad songs of some singers.

```
  val singerDF = spark.createDF(
List(
("sublime", Map(
  "good_song" -> "santeria",
  "bad_song" -> "doesn't exist")
),
("prince_royce", Map(
  "good_song" -> "darte un beso",
  "bad_song" -> "back it up")
```

```
)
),
List(
("name", StringType, true),
("songs",MapType(StringType,
 StringType, true), true)
 )
)
```

singerDF.show()

```
     | name|               | songs|
 ------------+--------------------+
   |sublime|Map(good_song -> ...|
   |prince_royce|Map(good_song -> ...|
  +------------+--------------------+
```

singerDF.printSchema()

```
 root
|-- name: string (nullable = true)
|-- songs: map (nullable = true)
|    |-- key: string
||-- value:
string (valueContainsNull = true
```

StructType(fields)

- Represents values with the structure described by a sequence of StructFields (fields). *StructField(name, dataType, nullable)*:
- Represents a field in a StructType.
- `name` indicates the name of a field, and `datatype` indicates the data type of a field.
- `nullable` is used to indicate if values of this fields can have null values.

Let's create a DataFrame with a StructType.

```
val rdd = sc.parallelize
```

```
    (Array(Row(ArrayBuffer(1,2,3,4))))
val df = sqlContext.createDataFrame(
        rdd,StructType(Seq(StructField
      ("list", ArrayType(IntegerType,
      false), false)
)
```

df.printSchema

```
root
  -- list: array (nullable = false)

 |-- element: integer (containsNull = false)
```

df.show

```
  |list|

 |[1, 2, 3, 4]|
```

Aggregations

An **aggregation** is an act of collecting together.

Spark SQL has:

- Built-in aggregations such as functions designed for DataFrames - `avg()`, `max()`, `min()`,`count()`,`approx_count_distinct()`, etc.
- User-defined aggregations - both UnTyped and Type-Safe.

Untyped Aggregation Functions

To implement a custom untyped user-defined aggregation function, the user has to extend the `UserDefinedAggregateFunction` abstract class.

Creating a custom aggregation function to calculate the average.

```
 object CalculateAverage extends UserDefinedAggregateFunction
```

Aggregate function can have input arguments

```
def inputSchema: StructType = StructType(StructField("inputColumn", LongType) ::
Nil)
```

Values in the aggregation buffer with data types

```
def bufferSchema: StructType = {
StructType(StructField("total", LongType) :: StructField("count", LongType) :: Nil)
}
```

Mention the data type of the returned value of the aggregation function

```
 def dataType: DataType = DoubleType
```

Initialize the values of aggregation buffer

```
def initialize(buffer: MutableAggregationBuffer): Unit = {
buffer(0) = 0L
buffer(1) = 0L
}
```

Here, the given aggregation buffer is updated with new input data given:

```
 def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
if (!input.isNullAt(0)) {
  buffer(0) = buffer.getLong(0) + input.getLong(0)
  buffer(1) = buffer.getLong(1) + 1
}
}
```

Spark SQL Optimization

Optimization refers to the fine-tuning of a system to make it more efficient and to reduce its resource utilization.

Spark SQL Optimization is achieved by **Catalyst Optimizer**.

Catalyst Optimizer is:

- The core of Spark SQL.
- Provides advanced programming language features to build a query optimizer.
- Based on functional programming construct in Scala.
- Supports **rule-based optimization** (defined by a set of rules to execute the query) and **cost-based optimization** (defined by selecting the most suitable way to execute a query).

To manipulate the tree, the catalyst contains the **tree** and the set of **rules**.

Tree

Tree is the main datatype in Catalyst that contains node objects.

Every node will have a node type and zero or more children. The new nodes created are immutable in nature and are defined as subclasses of `TreeNode` class in Scala. These objects can be manipulated using functional transformations as explained in the following example.

Consider three node classes:

Constant value:

```
Literal(value:Int)
```

Attribute value from an input:

```
attribute(name:String)
```

Subtraction of two expressions:

```
Sub(left:TreeNode,right:TreeNode)
```

Rule

Trees can be manipulated using rules which can be defined as a function from one tree to another tree. This rule can run the arbitrary code on the input tree.

The common approach is to use a **pattern matching function** and further the subtree replaced with a specific structure.

Using **transform function** on trees, we can recursively apply pattern matching on all nodes of the tree, transforming the ones that match each pattern to a result.

Example:

```
tree.transform
{
case
   Sub(worth(c1),worth(c2))
       => worth(c1+c2)
 }
```

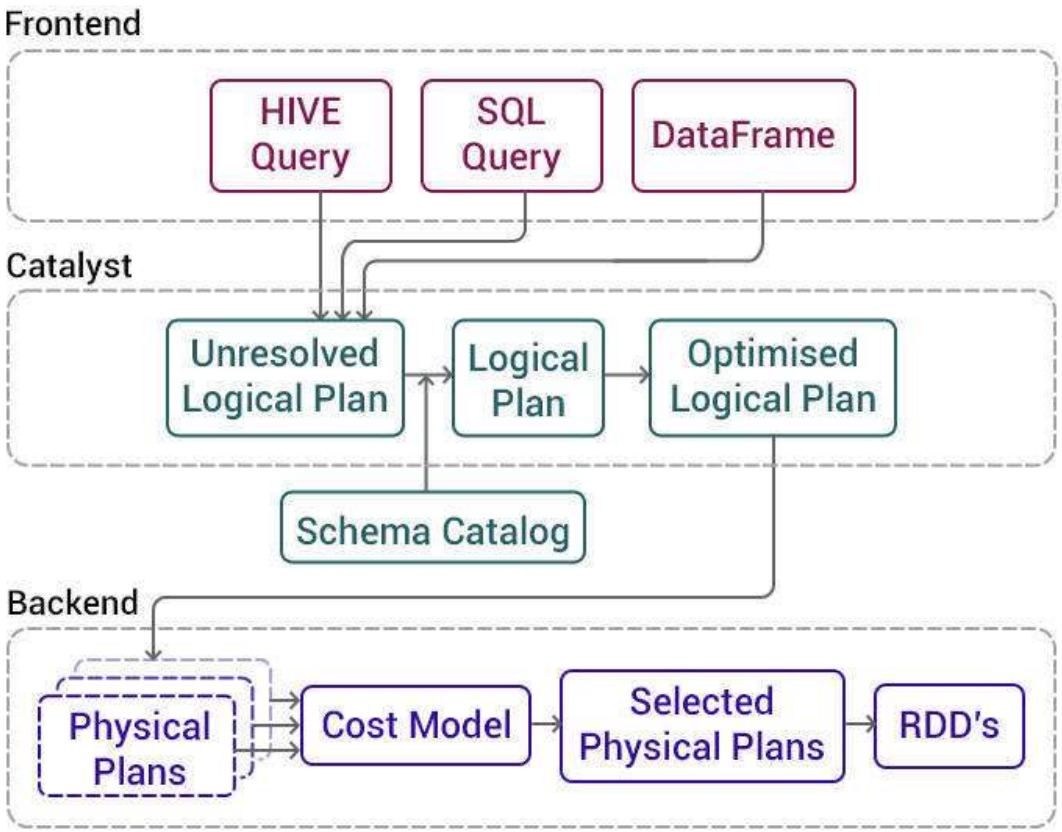
The pattern matching expression that is passed to transform is a partial function. It only needs to match to a subset of all possible input trees.

The catalyst will check to that part of the tree the given rule applies and then skip automatically over the trees that do not match.

The rule can match the multiple patterns with the same transform call.

Example:

```
tree.transform
 {
   case Sub(worth(c1), worth(c2))
        => worth(c1-c2)
   case Sub(left , worth(0)) => left
   case Sub(worth(0), right) => right
}
```



In Spark SQL, the Catalyst performs the following functions:

- **Analysis**
- **Logical Optimization**
- **Physical Planning**
- **Code Generation**

1] Analysis Phase

The beginning of Spark SQL optimization is with relation to be computed either from abstract syntax tree returned by SQL parser or dataframe object created using API.

Here in both cases,the query may contains unresolved attributes which means the type of attribute is unknown and have not matched it to an input table.

```
SELECT value from PROJECT
```

In the above query,the type of *value* is not known and not even sure whether its valid existing column.

`Catalyst rules` and `Catalog object` is used in Spark SQL to track all data sources to resolve these unresolved attributes. This is done by creating an unresolved logical plan and then apply the below steps.

- Lookup the relation BY NAME FROM CATALOG.
- Map the named attribute like 'value' as in example above.
- Determine the attribute which refer to the same value to give them unique ID.
- Propagating and pushing types through expressions.

2] Logical Optimization Phase

In this phase of Spark SQL optimization applies the standard rule-based optimization to the logical plan. It includes-

- constant folding
- predicate pushdown
- project pruning
- null propagation

It is extremely easy to add rules for a wide variety of situations.

3] Physical Planning Phase

In this phase of Spark SQL Optimization,one or more physical plan is formed from the logical plan,using physical operator matches the Spark execution engine.

Here `Cost-based optimization` is used to select join algorithms. The framework supports broader use of cost-based optimization for small relation SQL uses broadcast join.

Using the rule, it can estimate the cost recursively for the whole tree.

4] Code Generation Phase

In code generation phase of Spark SQL optimization,java byte code is generated to run on each machine.

Its very tough to build code generation engines. Catalyst make use of the special feature of Scala language,`QUASIQUOTES` to make code generation easier.

Quasiquotes allow the programmatic construction of AST (Abstract syntax tree)in scala language.At runtime this can then fed to the Scala compiler to generate bytecode.

Catalyst can transform a tree which represents an expression in SQL to AST for Scala code to evaluate that expression,compile and run the generated code.

Performance Tuning Options in Spark SQL

Following are the different Spark SQL performance tuning options available.

1. spark.sql.codegen

- Used to improve the performance of large queries.
- The value of `spark.sql.codegen` should be true.
- The default value of `spark.sql.codegen` is false.
- Since it has to run a compiler for each query, performance will be poor for very short queries.

2. spark.sql.inMemorycolumnarStorage.compressed

- We can compress the in-memory columnar storage automatically based on statistics of data.
- The value of `spark.sql.inMemorycolumnarStorage.compressed` should be true.
- Its default value will be false.

3. spark.sql.inMemoryColumnarStorage.batchSize

- We can boost up memory utilization by giving the larger values for this parameter `spark.sql.inMemoryColumnarStorage.batchSize`.
- The default value will be 10000.

4. spark.sql.parquet.compression.codec

- We can enable high speed and reasonable compression using the parameter `spark.sql.parquet.compression.codec`.
- The snappy library can be used for compression and decompression.

Spark Streaming

The **live input data** streams received by Spark Streaming are divided into several **micro batches**. **Spark Engine** take up these batches and **process** it to generate the final streams of results in batches.

Streaming Abstraction

The high-level abstraction that Spark Streaming provides is called **Discretized stream or DStream**.

DStream represents a continuous stream of data.

DStreams can be created in two different ways:

- From **input** data streams **from sources** such as Kafka, Flume etc.
- By applying **high-level operations** on other DStreams.

DStreams

A **DStream** represents a sequence of data which arrives over time.

It is otherwise called **discretized streams** because internally, a DStream is represented as a sequence of RDD's arriving at *discrete time intervals*.

Operations that can be applied on DStreams are similar to those of RDD operations.

- **Transformations** : Yield a new DStream.
- **Output operations** : Write data to an external system

Transformations on DStreams

map(func) :

Returns a new **DStream** by *passing each element of the source DStream* through a function func.

flatMap(func) :

Each input item can be mapped to 0 or more output elements.

filter(func) :

Return a new DStream by selecting only those records of the source DStream which returns a true value for the **func**.

There are many more transformations. Click here to learn more.

Output Operations on DStreams

print() :

Prints the first ten elements of every batch of data in a DStream.

saveAsTextFiles(prefix, [suffix]) :

Save this DStream's contents as text files.

saveAsObjectFiles(prefix, [suffix]) :

Save this DStream's contents as SequenceFiles of serialized Java objects.

Click here to learn more.

DStreams - Classification

Spark Streaming has **two** categories of sources.

- Basic sources : Sources that are directly available in the StreamingContext API. e.g., File systems, Socket connections, and Akka actors.

- Advanced sources : Sources like Kafka, Flume, Twitter, etc. These are available through extra utility classes.

Significance of Shared Variables

While designing *distributed tasks*, we may come across situations where we want **each task** to access some **shared variables or values** instead of independently returning intermediate results back to the driver program.

In such situations you can use a shared variable which can be accessed by all of the tasks.

Spark Shared Variables

Spark provides **two** different types of **shared variables** to two known usage patterns.

- Accumulators
- Broadcast variables

Creating Accumulators

Accumulators are created from SparkContext(sc) as: val acc = sc.accumulator(0, "test") You can create built-in accumulators for *longs, doubles, or collections*. You are free to create accumulators with or without a name, but only named accumulators are displayed in **Spark UI**.

Accumulators for Executors & Driver

For an Exector/worker node,accumulators are write-only variables. Task running on the executor nodes can manipulate the value set to an accumulator. However, they *cannot read* it's value.

```
e.g.,
```

```
scala>sc.parallelize(Array(1, 2)).foreach(x => acc += x)
```

The driver, can read the value of the accumulator, using the **value** method as:

```
scala> acc.value
res4: Int = 3
```

Creating Broadcast Variables

Broadcast variables are created with **SparkContext.broadcast** function as:

```
scala>val broadVar = sc.broadcast(Array(1, 2))
```

Note : Explicitly create broadcast variables only if *tasks across multiple stages are in need of same data*.

value function is used to get the *value* assigned to a broadcast variable.

```
scala> broadcastVar.value
res2: Array[Int] = Array(1, 2)
```

Workflow Stages

Stage 1 : *When the **Spark Streaming Context** starts*, the driver will execute **task** on the executors/worker nodes.

Stage 2 : Data Streams generated at the streaming sources will be received by the **Receivers** that sit on top of executor nodes. The *receiver is responsible for dividing the stream into blocks* and for keeping them in memory.

Stage 3 : In order to avoid data loss, these blocks are also **replicated** to another executor.

Stage 4 : **Block Management Master** on the driver *keeps track of the block ID* information.

Stage 5 : For every batch interval configured in Spark Streaming Context (commonly in **seconds**), the **driver** will **launch tasks** to process the blocks.

After the processing, the resultant data blocks are persisted to any number of target data stores including **cloud** storage , **relational** data stores , and **NoSQL** stores."

Caching / Persistence

DStreams can be persisted in as stream's of data.

You can make use of the **persist() method** on a DStream which persist every RDD of that particular *DStream in memory.*

This is useful if the data in the DStream is computed **multiple times**.

Default persistence level for input streams is set to *replicate* the data to two nodes for *fault-tolerance*.

Creating StreamingContext

The first step in creating a streaming application is to create a **StreamingContext**.

This is the primary **entry point** for all streaming functionality.

```
//creating configuration object
val conf = new sparkConf().setMaster(""local"").setAppName(""MyStreamingApp"")

//creating streaming context and batch interval is set to 1 second
val ssc = new StreamingContext(conf, Seconds(1))
```

Required imports:

```
import org.apache.spark._
import org.apache.spark.streaming._
```

Creating DStreams

You can create a **DStream** that represents streaming data by using the StreamingContext - **ssc**.

```
val lines = ssc.socketTextStream(""samplehost"", 007)
```

`lines` DStream represents a **stream of data** that you will receive from the data server. Each record in `lines` corresponds to a line of text.

Let us Count the Words!

Now that we have extracted the data into 'lines', We are good to go.

You can now transform the data as per requirements.

Let's perform a word count program.

```
// splitting the line at <space>
val words = lines.flatMap(_.split(" "))

// creating (<word>,1) as a pair
val pairs = words.map(word => (word, 1))

//summing up all values(1's) for a particular word
```

```
val wordCounts = pairs.reduceByKey(_ + _)


//printing the result
wordCounts.print()
```

Most Important Step!!!

Streaming is **not yet started**!

All the above computations are performed only at the **start of StreamingContext(ssc)**

```
// Start computation
ssc.start()


// Wait for computation to terminate
ssc.awaitTermination()
```

NoSQL - Database Revolution

In 1970, E.F. Codd envisioned a new model of DBMS through his paper titled **A Relational Model of Data for Large Shared Data Banks** which paved the way for the emergence of **Relational DBMS (RDBMS)**.

- RDBMS formulated a new methodology for storing data and processing large databases.
- The records (data) would be stored in 'table' with fixed-length records unlike the free-form list of linked records in IDS, IMS.
- Later, databases like Ingres, query language like SQL got evolved.

Era of Database Wars

The nuances and benefits of RDBMS had a wider reach, resulting in buy-in from different vendors, setting a stage for an **era of Database wars**.

Many RDBMS such as `Sybase, Microsoft SQL Server, Informix, MySQL, DB2, Oracle` got launched around the same time claiming better

- **Performance**

- **Availability**
- **More functionalities**
- **Cost of storage**
- **Economy of usage**.

With no alternates, the roots of RDBMS got **completely entrenched** by early 2000s.

NoSQL Explosion

Later in 2005, the difference and change in architectures design of applications between the **client-server era** and the **era of massive web-scale applications** triggered `lot of pressure` on the

- **Level of usage**
- **Volume of data considered**
- **knack of handling/monitoring change**

on RDBMS that couldn't upscale through incremental innovation.

This started the era of **Distributed Non-Relational Database Management System**, later coined as `'NoSQL'`, which was more aligned to *`New-Age applications`*.

'NoSQL' grabbed the attention on the database system that broke the practice of the traditional SQL database.

NoSQL - Where it Scores?

Key features of NoSQL which makes it the most sought DB.

- **Distributed computing system**
- **Higher scalability**
- **Reduced Costs**
- **Flexible schema design**
- **Process unstructured and semi-structured data**
- **No complex relationship**
- **Open-sourced**

NoSQL Traits

- Not based on `Relational Data Model`.
- Not a standard NoSQL Model.
- No schema in NoSQL.
- Often uses `aggregates`.

Focus - Key Points

Eventual Consistency

- Asynchronous inserts and updates.
- Eventually all changes will propagate through the system.
- Eventually the updates reaches every node.

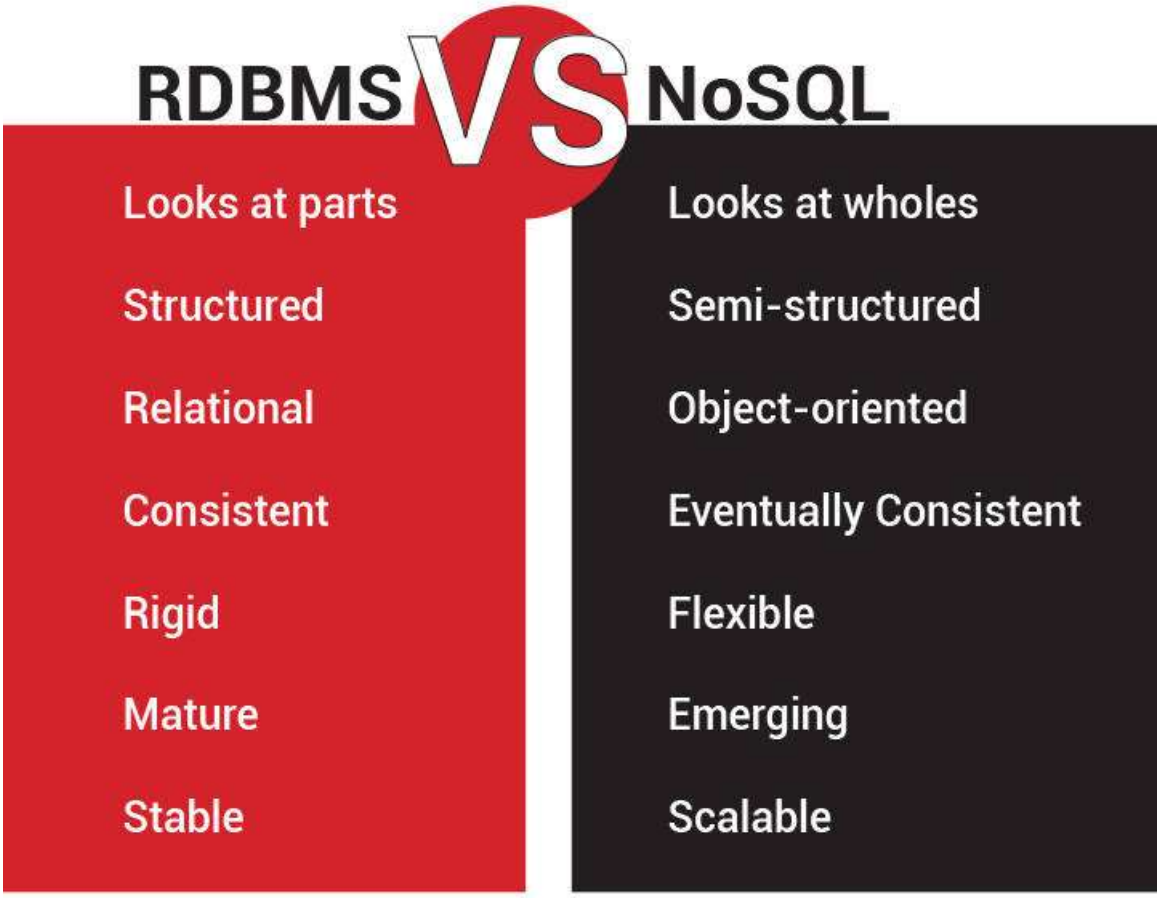
BASE

`Basically Available` Possibilities of faults but not fault of whole system.

`Soft State` Copies of data may be inconsistent.

`Eventual Consistency` Copies become consistent Eventually.

RDBMS vs NoSQL



Scaling

RDBMS - Vertical Scaling

- Architecture design runs well on a single machine.
- To handle larger volumes of operations is to upgrade the machine with a faster processor or more memory.
- There is a limitation to size/level of scaling.

NoSQL - Horizontal Scaling

- NoSQL databases are intended to run on clusters of comparatively low-specification servers.
- To handle more data, add more servers to the cluster.
- Calibrated to **operate with full throttle even with low-cost hardware**.
- **Relatively cheaper approach** to handle increased
    - Number of operations
    - Size of the data.

Maintenance

RDBMS - High Maintenance

- Maintaining high-end RDBMS systems is **expensive** and requires **trained workforce** for database management.

NoSQL - Low Maintenance

- NoSQL databases require minimal management, and it supports many features, which makes the need for **administration and tuning requirements** becomes less. This covers
    - ***Automatic repair***
    - ***Easier data distribution***
    - ***Simpler data models***

Data Model

RDBMS - Rigid Data Model:

- RDBMS requires data in structured format as per defined data model.
- As change management is a big headache in SQL with a strong dependency on primary/foreign keys, ad-hoc data insertion becomes tougher.

NoSQL - No Schema/Data model:

- NoSQL database is schema-less so that data can be inserted into a database with ease, even without any predefined schema.
- The format or data model could be changed anytime, without application disruption.

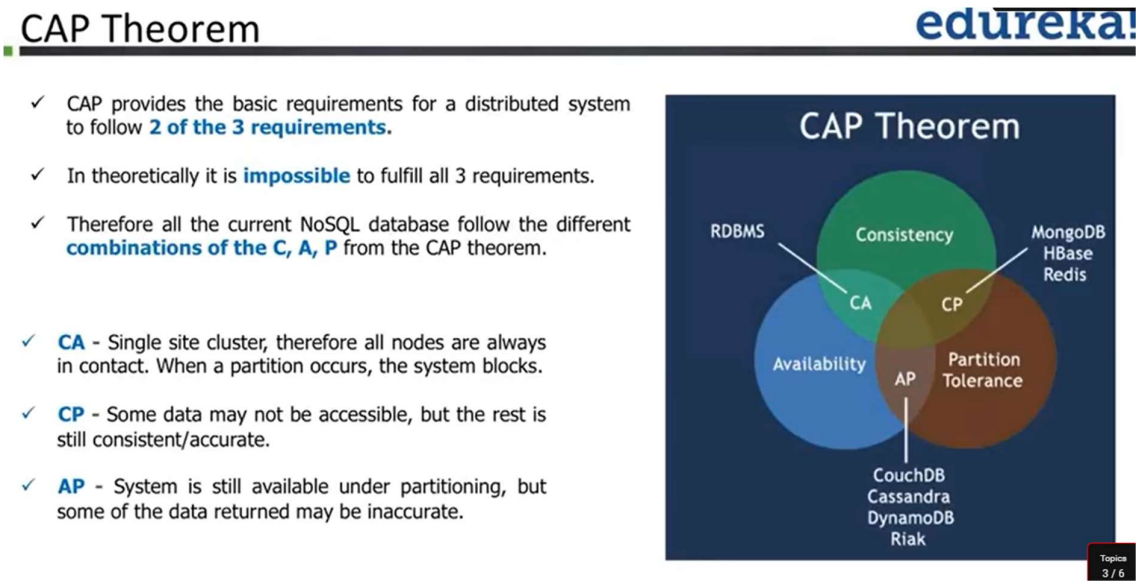
Caching

RDBMS - Separate Hardware

- The caching in typical RDBMS database requires separate infrastructure.
- As there is overhead, the logic of retrieval involves little delay.

NoSQL - Integrated:

- NoSQL database supports caching in `system memory`, so it increases data output performance.



Changing pH measure

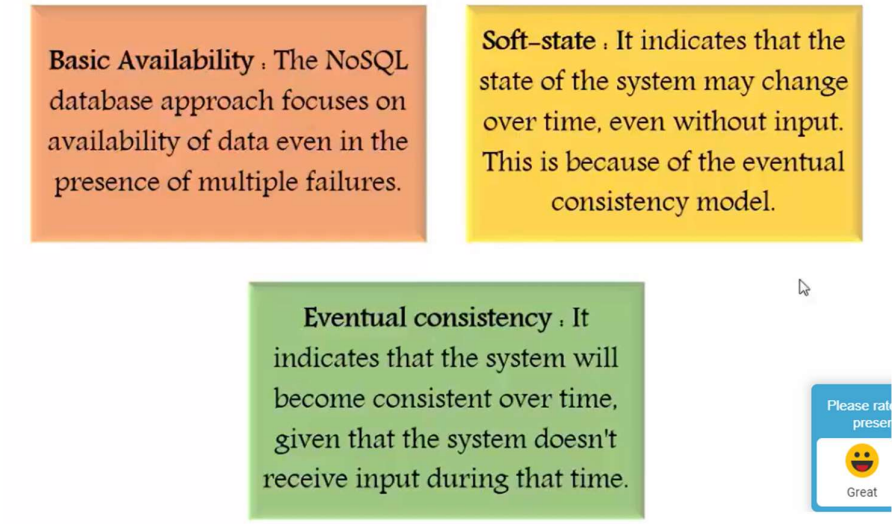
Here you will get to know about core principles of DB processing.

RDBMS - ACID

- `Atomicity`: If any one element of a transaction fails then the entire transaction fails.
- `Consistency`: The transaction must adhere to all protocols/rules at all times.
- `Isolation`: No transaction has access to any other transaction that is in an intermediate or unfinished state.
- `Durability`: Once the transaction is complete, it will continue to persist as complete and cannot be undone.

NoSQL - BASE

- `Basically Available`: System does guarantee the availability of the data as per CAP (**Consistency, Availability and Partition Tolerance**) Theorem.
- `Soft state`: The state of the system could change over time, so even during times without input.
- `Eventual consistency`: The system would eventually become consistent as it stops receiving input.



NoSQL database is not viewed as a replacement to RDBMS but, rather, a complementary addition to RDBMS and SQL.

Data Replication

Data replication is all about having your data **geo-distributed** through a non-interactive and reliable process as a contingency measure to avoid loss of data.

Most of the NoSQL systems have data replication feature built-in.

Data replication in RDBMS is little difficult as they have not adopted **Horizontal scaling**.

NoSQL data replication is homogenous, in the sense data cannot be replicated from a NoSQL system to RDBMS SQL system.

Three types of Data Replication include -

- **Sharding Replication**

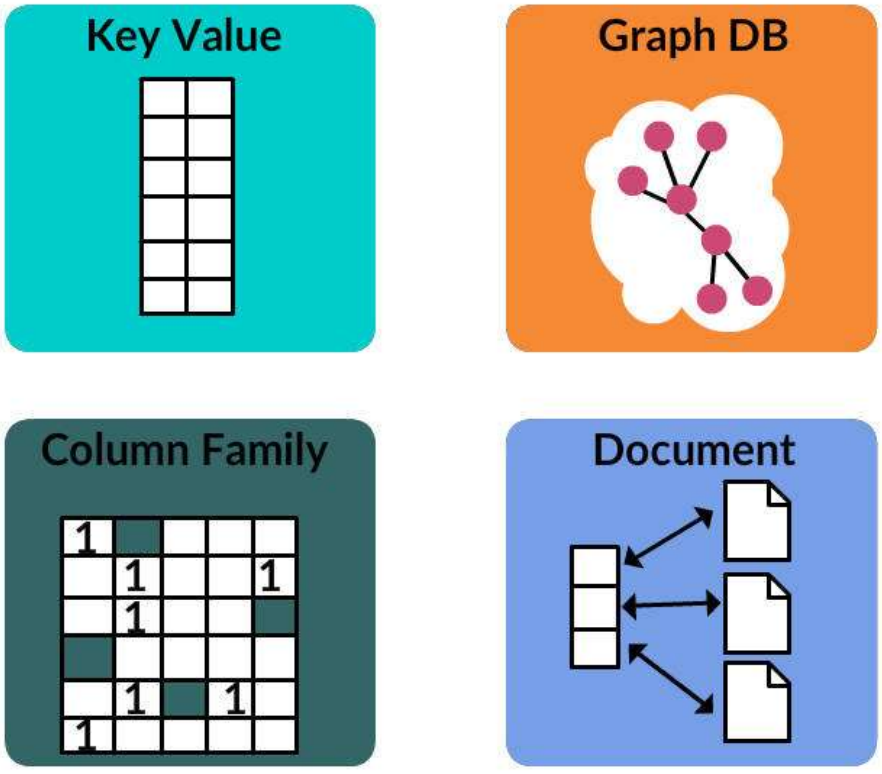
- **Master-Slave Replication**
- **Peer-to-Peer Replication**.

You will understand more about these in the following set of cards.

Peer-to-Peer Replication Model

- Data is replicated across many nodes.
- Each node is `equal`.
- All nodes accept `reads and writes`.

Flavors



NoSQL just means **Not only SQL**, reconfirming NoSQL works as an alternative database to RDBMS.

Stores

- **Most simplest NoSQL database** among all.

- The data is stored in **key-value pairs**.
- It provides better performance.
- Easy to access data via API and the client could
    - get value for a key
    - put value for a key
    - delete a key

Few key-value databases - **Riak, Redis, Memcached, Berkeley DB, Amazon DynamoDB (not open-source) and so on.**

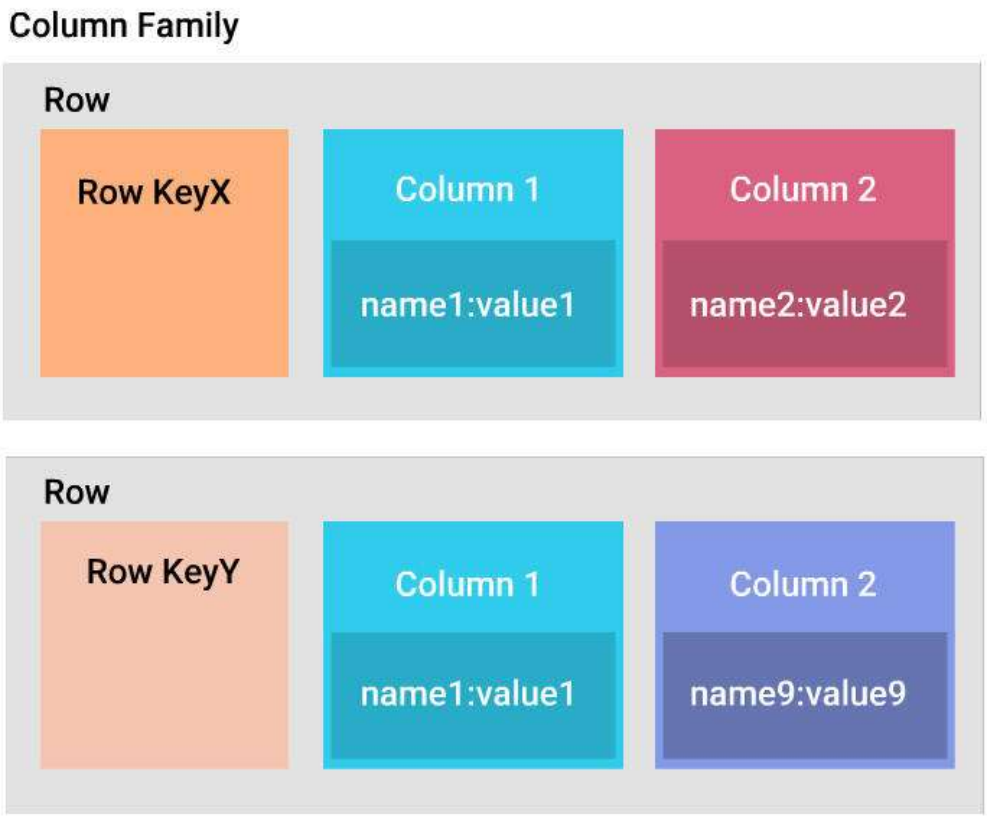
Key Value Store Example

Let's take an example of Key-Value pair.

```
Key        value
Harry    "Gryffindor, sneaker, 14"
Malfoy   "Slytherin, chaser, 15"
Luna     "Ravenclaw, Singer,13"
Cedric   "Hufflepuff,'', 17"
```

Based on the key, the corresponding values could be fetched and tokenized accordingly.

Columnar Stores



- Stores data as **sections of data columns (`column families`)**.
- Column families are rows with many columns associated.
- Column families are chunks of related data often accessed together.

Popular columnar databases include **Cassandra, HBase, Hypertable and Google Bigtable**.

- Stores data as **sections of data columns (`column families`)**.
- Column families are rows with many columns associated.
- Column families are chunks of related data often accessed together.

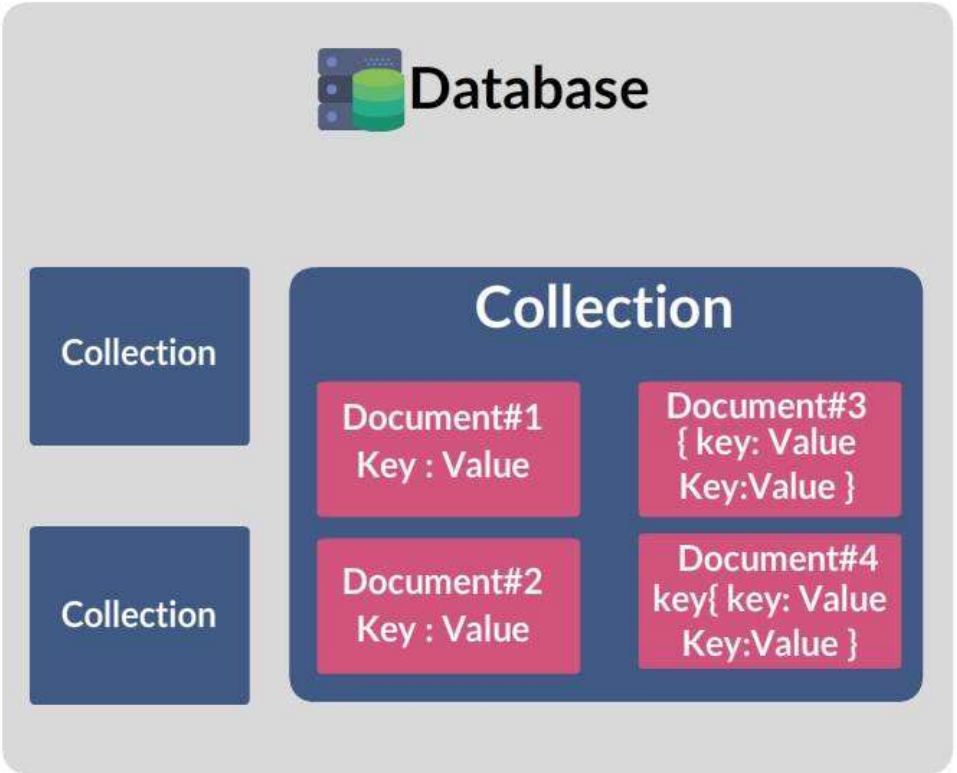
Popular columnar databases include **Cassandra, HBase, Hypertable and Google Bigtable**.

Columnar Store Example

```
RowId StuID Lastname  Firstname  Score

001   10    Weasley   Ron        48000
```

```
002   11    Granger   Hermione   58000

003   15    Potter    Harry      42000


10:001,11:002,15:003;

Ron:001,Hermione:002,Harry:003;

Weasley:001,Granger:002,Potter:003;

48000:001,58000:002,42000:003;
```

Document Stores



- Stores and retrieves documents of formats XML, JSON, BSON, and so on.
- Documents consist of maps, collections, and scalar values.

- Document Store is mainly categorized into -

```
- ***XML based Databases***

- ***JSON based Databases***
```

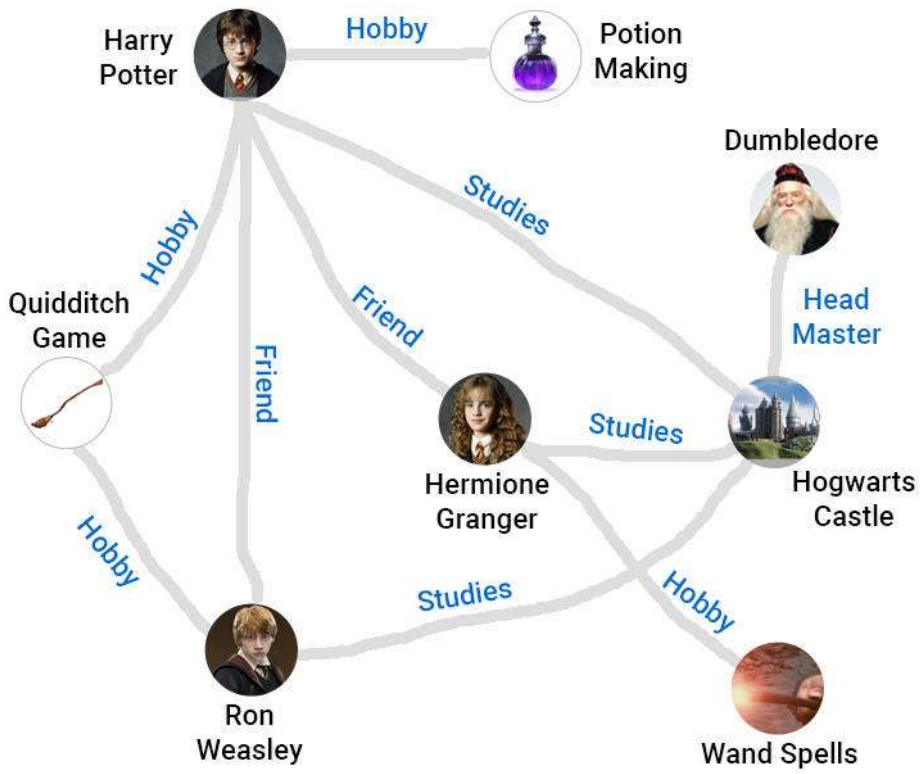
Few of the document databases include **MongoDB, CouchDB, Terrastore, RavenDB, OrientDB.**

Document Store Example

```
{

 "FirstName": "Harry",

 "LastName": "Potter",

 "phone":{

    "cell":"095 - 000 - 100 - 110",

    "work":"099 - 800 - 100 - 110"

  },

 "Address": "15 Hogwarts School of Wizardry",

 "Hobby": ["Potion","Quidditch"]

}



<contact>

  <firstname>Harry</firstname>
```

```
  <lastname>Potter</lastname>

  <phone type="Cell">095 - 000 - 100 - 110</phone>

  <phone type="Work">099 - 800 - 100 - 110</phone>

  <address>

    <Room No>15</Room No>

    <University>Hogwarts School of Wizardry</University>

  </address>

</contact>
```

Graph Store



- Uses relationship, nodes, and properties to represent data
- All nodes are **connected through relationships**.

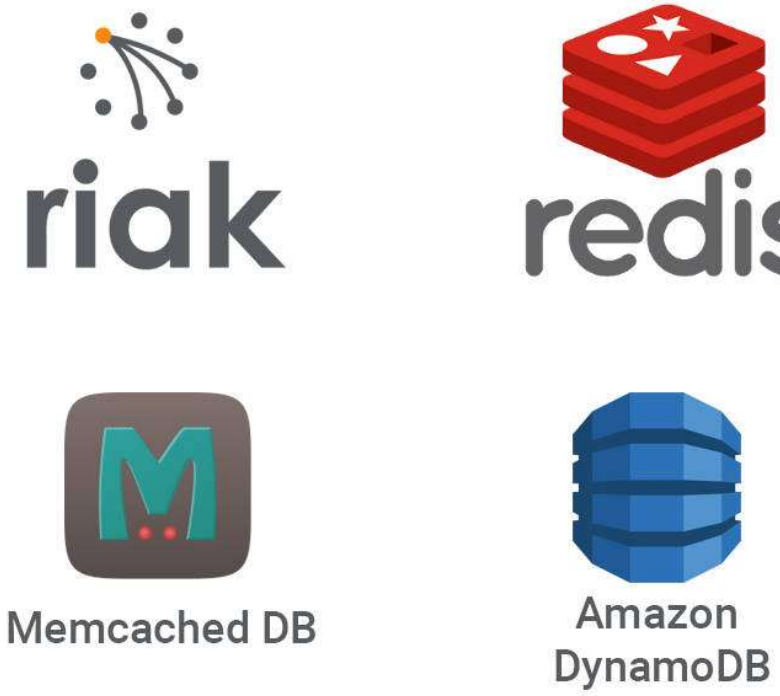
- The relationship has a direction, type, start node and end node.
- Uses `Graph Theory` to store, map and query relationships.

Few popular Graph Databases are **Neo4j, AllegroGraph, Oracle Spatial and Graph, Teradata Aster, ArangoDB, and Graphbase**.

Key-Value - Zeroed

- The **Key-Value pair** design is similar to simple hash table design. It can be compared to RDBMS table with two columns (ID/Name).
- The value could be **blob, text, JSON, XML**, and so on.
- Popular Dbs include - **Riak, Redis, Memcached DB, and Amazon DynamoDB.**
- 'Key' must be unique and should not be too long.
- Redis DB allows performing many superior functions like range, diff, and intersection.

Popular Key Value Databases



Few widely used `Key-Value databases` are highlighted above.

Key Features to be considered while selecting Key Value Store are listed below

- **Consistency**
- **Transactions**
- **Querying ability**
- **Scalability**

Consistency

Consistency

- Applicable on single Key as it involves **get, put, or delete**.
- Although **optimistic writes** could be performed, they are expensive to implement.
- In **Distributed key-value store implementations** like Riak, values will be replicated to other nodes.

Buckets are like namespace keys, which reduces key collisions. Example - All Student keys may reside in the Student bucket.

- With Buckets - `'write'` is considered good only when the data is consistent across all the nodes where the data is stored.

'Buckets' in Key-Value Dbs are similar to 'Tables' in RDBMS.

Transactions

- Different key-value databases have different specifications of transactions, and they implement transactions in different ways.
- There is **No guarantee** on the write operations.
- Riak has to write tolerance by leveraging **quorum concept using W value—replication factor**.

`Example`: Consider a Riak cluster,

- Replication factor of 7 (determines # of data copies be maintained across multiple nodes)
- Value of W value 4.

Then write is only reported as successful, when writes happen in at least four nodes.

Query Features

- Design of ***'Key'*** plays prominent role and this is achieved by
    - using some Algorithm
    - with user inputs (user-id, name, email-id)

    - from timestamps/external data.
- Could be queried by the key/value associated with it.
- Querying based on an attribute of value column is not possible from DB.
- In some DBs, the value of the key is retrieved using the fetch API. Ex: Riak.

Scaling

- Scalability of Key-Value database is achieved through **sharding**.
- In sharding, the value of the key determines on which node the key is stored.

For example, say you are sharding by the first character of the key.

```
if the key is k76151487d, which starts with an 'k', will be sent to a different
node than the key dgh396542.
```

Benefits

- Increase performance as more nodes can be added to the cluster.

Impact

- If the node used to store 'f' goes down, the data stored on that node becomes unavailable, nor can new data be written with keys that start with f.

How to overcome this issue?

Riak DB leverages CAP Theorem to improve its scalability:

- **N** - # of nodes to store the key-value replicas.
- **R** - # of nodes to fetch data from.
- **W** - # of nodes to write data to.

For example, consider **5-node Riak cluster**. And if you configure,

```
N = 3 => all data should be replicated to at least 3 nodes.

R = 2 => Any 2 nodes must respond to GET request to be considered successful.

W = 2 => PUT request is written to 2 nodes before the write is considered
successful.
```

- Best practice is to **choose a W value to match your consistency** needs during bucket creation

Usecase - Storing Session Details

- Every web session is assigned a unique session-id value, which the applications **store on disk(logfile) / DB(RDBMS)**.
- Moving this to key-value DB will improve performance to great extent as every info about the session could be
    - Stored by a PUT request
    - Retrieved using GET request
- The operation is very fast, as session info are stored in a **single object**.

Usage:

- **Memcached** for caching web applications and microapps,
- **Riak** when availability is an important criteria.

Usecase - Storing Profile/Preferences

Key-Value would be best-fit to store `user profile`

- ***userId***
- ***username*** and
- ***additional attributes***

and `user preferences`

- ***language***
- ***country***
- ***timezone*** and
- ***user favorites*** and so on

All these information could be stored in a single object, so getting preferences of a user would just take single GET operation.

On similar lines, **product profiles** could be stored as well.

Usecase - Shopping Cart Details

- All e-commerce websites have shopping carts deeply linked with the user.
- The shopping cart details should be available at all times, across different browsers, devices, machines, and sessions.
- Key-Value would be best-fit for this scenario, with all shopping related information put into 'value' where the 'key' is the userid.

Usage

- Amazon uses its DynamoDB for storing its user's shopping cart details.

Limitations of Key-Value Store

Key-value databases **would not be the best fit** in the few scenarios highlighted below.

- **Relationships among Multiple-Data** - There exist relationships between different sets of data or correlation and the data between different sets of keys.
- **Multi-operation Transactions** -

If you are storing many keys and when there is a failure to save one of the keys, and you want to roll back the rest of the operations.

- **Query Data by *'value'*** - Searching the 'keys' based on some info found in the 'value' part of the key-value pairs. Some exceptions include Riak Search or indexing engines such as Lucene or Solr.
- **Operation by groups** - As operations are confined to one key at a time, there exists no way to run several keys simultaneously.

Types of NoSQL Databases

There are four types of NoSQL Databases.

- **Graph databases:** Stores information about networks, such as social connections. Examples: `Neo4J` and `HyperGraphDB`
- **Document databases:** Stores semi-structured data such as documents, typically in JSON or XML format. Examples: `MongoDB`
- **Key-value databases:** Also known as `key-value store` uses a simple key/value method to store data. Examples: `Riak` and `Redis`
- **Wide-column stores:** Stores data together as columns instead of rows and are optimized for queries over large data sets. Examples: `Cassandra` and `HBase`

Cassandra Brass Tacks

Introducing Apache Cassandra

Cassandra is a wide column store NoSQL database that is massively scalable.

It was originally developed at Facebook, was **open sourced** in 2008, and became a **top-level Apache project** in 2010.

It offers:

- Continuous availability
- Linear scale performance
- Operational simplicity
- Easy data distribution across multiple data centers/cloud availability zones
- No single point of failure

Cassandra Features

- **Scalable architecture** – A masterless design with same nodes, offering easy scale-out and operational simplicity.
- **Universally active design** – All nodes can be written to and also read from.
- **Linear scale performance** – Capability to include nodes without going down creates likely boost in performance.

Now that you know what Cassandra is, we will see where Cassandra is being used.

Cassandra is ideal for IoT Applications for using faster incoming data from sensors, devices, and related mechanisms in various locations.

Cassandra is the preferred database for retailers who require quick product catalog lookups, strong shopping cart protection, and related retail app support.

Several media and entertainment companies deploy Cassandra for monitoring and tracking their user interaction activity with music, movies, online and website applications.

Cassandra acts as the database mainstay for several messaging provider and mobile phone applications.

Numerous social media providers, websites, and online companies adopt Cassandra to ingest, examine, and offer recommendations and analysis to their customers.

Who is Using Cassandra?

Cassandra is being adopted by numerous large and foremost companies for multiple requirements.

- **Twitter** uses Cassandra for **data mining** across the whole user store, places of interest data, geolocation and **real-time analytics**.
- **Mahalo and Digg** are deploying Cassandra for their **primary near-time data store**.
- **Facebook** continues to adopt Cassandra for **inbox search**, despite utilizing a proprietary fork.
- **Rackspace** are employing it for **cloud service, logging, and monitoring**.
- **Reddit** deploys it as a **persistent cache**.

Cassandra Overview

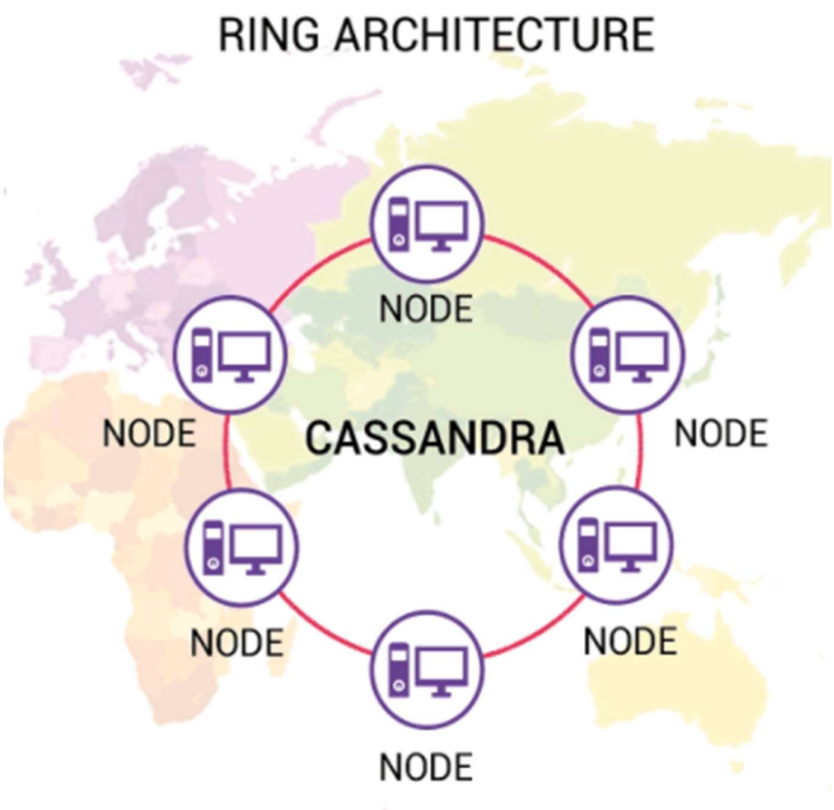
Cassandra has been developed to handle **huge data** and it offers high write and read throughput.

In a Cassandra cluster, there are no slave or master nodes - i.e. **All nodes are considered as equal.**

Unlike few conventional relational databases, Cassandra can read from, and write to any node. **In addition, this enables Cassandra to be available without having any single point of failure**.

Cassandra partitions data with all nodes within its cluster and the partitioned data can be searched utilizing **partition key**.

Ring Architecture



Cassandra features a masterless `ring architecture` that can be set up and maintained with ease.

In Cassandra, **all nodes have a similar role**; with all nodes interacting with each other through a scalable, distributed protocol known as `gossip`.

Gossip Protocol

Cassandra can be a considered as a `peer-to-peer` system, in which the transmission of cluster topology details is done through the Gossip protocol.

The `Gossip protocol` resembles the common gossip, where a node (X) conveys information about the node (Y) to few members in the cluster. These nodes talk about X to few other nodes, and gradually, all the nodes are aware of Y.

Cassandra Replication

One of the key features that boost the acceptance of Cassandra is it's **built-in and customizable replication**.

For a piece of data, a single node or more nodes within the cluster serves as a replica.

If few nodes replied with a **out-of-date value**, then Cassandra returns the most current value to the client, after which it executes a **read repair** in updating the stale value.

Generally, a replication factor of three is recommended. Replication is supposed to work in **several cloud availability zones, numerous data centers, and a data center**.

Key Components

- **Node** − Place where data is stored.
- **Data center** − a collection of associated nodes.
- **Cluster** − A component that comprises one or more data centers.
- **Commit log** − A crash-recovery mechanism found in Cassandra. Each write operation is written for committing log first.
- **Mem-table** − A memory-resident data structure. Once commit log is done, the data is written to the mem-table. Usually, there is only one active memtable for a single-column family. Eventually these memtables are flushed onto disk and become immutable SSTables.
- **SSTable** − A disk file to which the information is flushed from the mem-table while its contents arrive at a threshold value.

Bloom Filter

A Bloom filter is known as a generic data structure, a type of a cache for checking if an **element is available in a set or not**.

It is a fast and non-deterministic algorithm designed at the cost of **risking to return false positives**.

Role of Bloom Filter in Cassandra

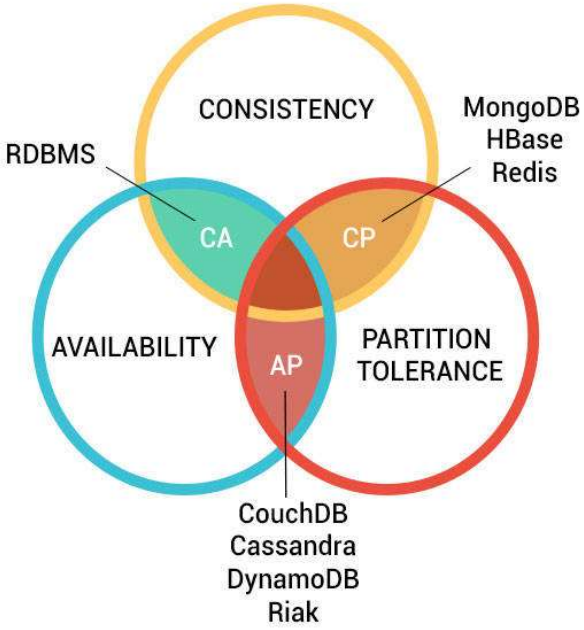
Cassandra utilizes bloom filters for testing if any SSTable is anticipated to include or not to include the requested partition key, without the need to read their contents, thereby evading expensive IO operations.

- If a bloom filter shows **false** for a partition key, then the **partition key is not available in the related SSTable**.
- If it shows **true**, the SSTable is **supposed to include the partition key**. Cassandra will move to more advanced methods to discover if it requires to read the SSTable or not.

Brewer's CAP Theorem

The theorem says that in a large-scale distributed data system, three requirements have a relationship of sliding dependency: Partition Tolerance, Availability, and Consistency.

To understand the design of Cassandra and its label as a consistent database, we have to know the CAP theorem.



- **Availability**: Database clients can read and write information anytime.
- **Consistency**: Database clients has to read the corresponding values for the corresponding queries and simultaneous updates.
- **Partition Tolerance**: The database can be divided into many machines; it can remain operating in the face of network segmentation breaks.

Sliding Mutual Dependency

Brewer's theorem says that, in a system, one can support `only two among the three`. You have to pick between them due to **sliding mutual dependency**.

If you **demand more consistently** from the system, it is expected to be **less partition-tolerant**, until any concessions are made around availability.

Despite being an AP system,**Consistency is accomplished** to a level by **consistency tuning mechanisms** and replication in Cassandra.

When a client declares a write operation.

At first, Cassandra writes the data into Commit log and then into the memtable. Cassandra then transmits the acknowledgment back to the client thereby confirming the write operation.

As more write operations arrive.

Cassandra will add the records to the Commit Log in sequential order and later to the memtable.

Data Flush

Memtable is present in memory and cannot keep on growing infinitely, Cassandra then flushes the records from the memtable to disk, this causes the creation of an SSTable.

Every single flush of the memtable creates a new SSTable in the given disk.

Furthermore, the records collected in the SSTable are sorted with the partitioning key while they were present in the memtable. This helps in **sequential access** of data from SSTable.

What Happens After Data Flush?

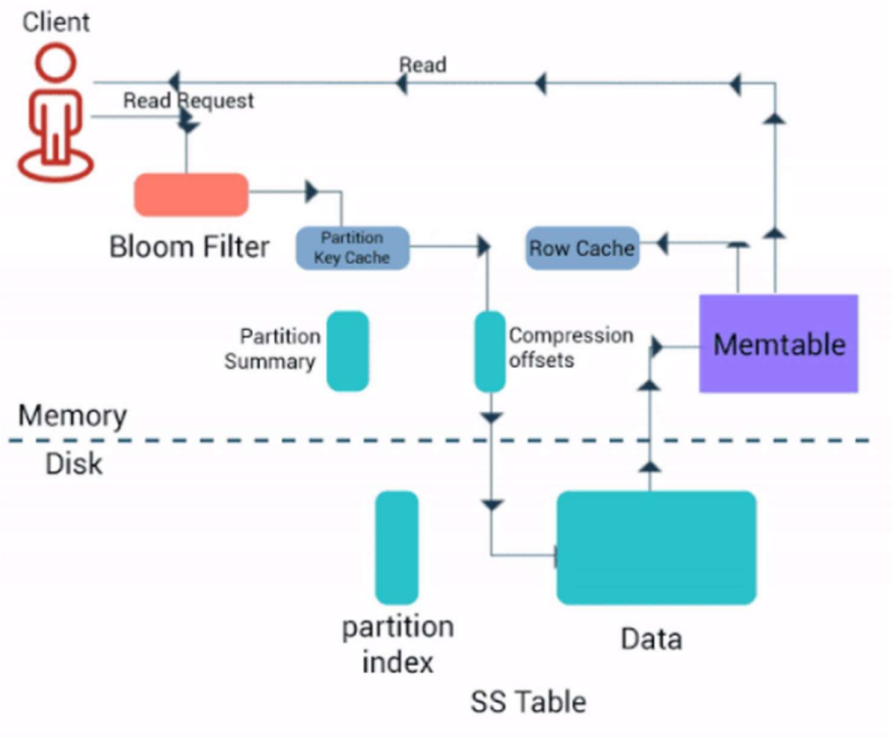
- If data is flushed from memtable to SSTable, the Commit log and memtable is not required for data durability.
- During this time, the memtable is cleared, commit log (file) is excluded (related to the records) and new commit log is generated for incoming records.

Note: Cassandra will reload the data that is unflushed from commit log to memtable at system startup, if the data is not flushed to disk prior to system failure.

Compaction

Periodically, Cassandra does a background operation called `compaction` to merge SSTables to optimize the read operation.

The read path noticed in Cassandra database is complex than the write path, because there is a broad range of components in reading data from a Cassandra database.



Components in Read Path

- **Row Cache**: It is an in-memory cache that stores recently read records (rows). (optional in Cassandra database)
- **(Partition) Key Cache**: Maps the newly read partition keys to a particular SSTable offset.
- **Partition Indexes**: They are generated during SSTable creation and stay on the disk. They are sorted partition keys that are mapped to SSTable offsets.
- **Partition Summaries**: This is utilized to accelerate the access to index on disk. This is regarded as an off-heap in-memory sampling of the Partition Indexes.
- **Compression offsets:** Here, for compressed blocks, compaction offsets keep the offset mapping data.

Tables in Cassandra are compressed by default. When Cassandra necessitates to read data, it searches the in-memory compression offset maps and then unpacks the data chunks.

If Record is Not Available in Row Cache...

- Cassandra will take help of **Bloom filters** that shows in which SSTables the requested record is not present.
- Once Cassandra identifies the SSTables to be scanned, it reaches out to the **(Partition) Key Cache**.

- If we obtain the partition key (of the demanded record) in the key cache, you can fetch the record directly from the SSTable because the **key cache will point directly to the particular record offset** in the SSTable.

Role of Partition Summary

If the partition key is not available in the key cache, Cassandra will search the **Partition Summaries**.

Partition Summary aids in jumping into a specific offset in the Partition Index.

Once in Partition Index, we will have the offset of the partition key in SSTable and can directly fetch the record from that offset of SSTable.

Note: No matter how Cassandra fetches the record from SSTables, it always needs to check the **Compression offsets** to be capable of reading the data from compressed blocks.

Merging Multiple Versions of Record

- In Cassandra, there might be many versions of the same record because Cassandra does not execute in-place modification.
- Cassandra links a **timestamp to every version of the record**. Cassandra utilizes this to *merge* records from several SSTables and memtable to display the **current version** of the complete record.

Data Modeling in RDBMS World

In the relational environment, data is modeled by generating entities and connecting them with relationships based on the guidelines directed by the relational theories.

This intends that developers can focus on the structure or logical view of the data without considering how the application manipulates and accesses the information.

How Different is Cassandra Data Modeling?

Cassandra data model is non-relational.

Here you need to focus on the application in addition to the data itself. You need to think about how you will query the data in the application.

Cassandra promotes **"read-before-write"** operation. **Read-before-write** or **ready-for-read** design is used to analyze your data read requirement first and store it in the same way.

Cassandra Data Modeling Considerations

Other important considerations while data modeling are:

- **Take the physical topology of a Cassandra cluster into account**.

- Cassandra is designed to work in a large-scale, distributed environment in which ACID compliance is difficult to achieve, and **replication is a must**.
- Unlike RDBMS world there are no join operations in Cassandra and hence has to **follow a denormalized structure**.

Developers must be aware of such differences in the process of data modeling in Cassandra.

Cassandra Data Structure Overview

Following are the basic Cassandra Data Structures

Cluster

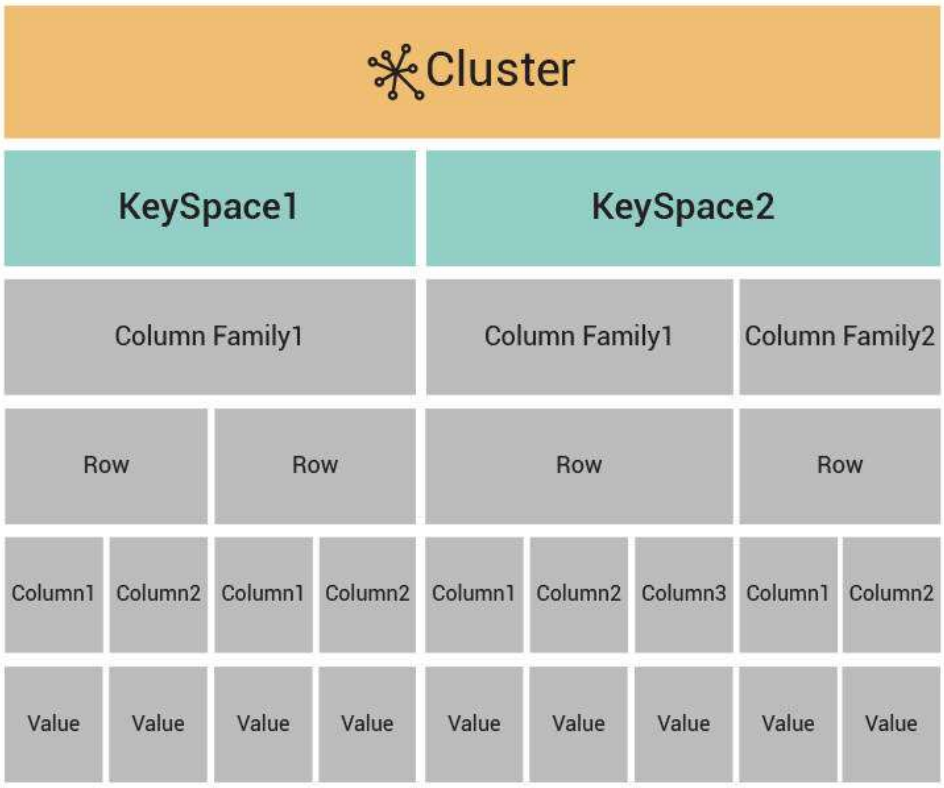
Keyspace

Column Family

Row

Column

Clusters



Cassandra database is particularly developed to be distributed over numerous machines working together that seem to be a single instance to the end user.

Cluster is the outermost structure in Cassandra.

Keyspace

In Cassandra, Keyspace is regarded as the outermost container for data.

Keyspace is a collection of super column families and column families. It is similar to a database or schema in the relational world.

Column Family

Column families describe the structure of the data.

Each keyspace has a minimum of one and often several column families. In a relational database, a single column family is similar to a table.

A column family offers better flexibility by enabling diverse columns in diverse rows. Any column can be easily included to any column family at any time.

Column families are stored physically in particular files on a disk. Hence, it is necessary to **retain relevant columns in the same column family** for enhancing performance and saving disk I/O.

Column

Column is the smallest storage unit and data model element in Cassandra.

A column is a name-value pair with a timestamp and an optional Time-To-Live (TTL) value.

Time-To-Live is an optional expiration value used to mark the column deleted after expiration. The column is then physically removed during compaction.

Install Apache Cassandra

Time to Start Coding

The best way to learn is to **Try as you Learn**

For the hands-on exercises in this course, you may use the Katacoda playground.

Just login to Katacoda and execute the terminal commands.

Let us start Coding!

Step 2 — Installing Cassandra

```
$ echo "deb http://www.apache.org/dist/cassandra/debian 22x main" | sudo tee -a
/etc/apt/sources.list.d/cassandra.sources.list


$ echo "deb-src http://www.apache.org/dist/cassandra/debian 22x main" | sudo tee -
a /etc/apt/sources.list.d/cassandra.sources.list
```

To avoid package signature warnings during package updates, we need to add three public keys from the Apache Software Foundation

```
$ gpg --keyserver pgp.mit.edu --recv-keys F758CE318D77295D
gpg --export --armor F758CE318D77295D | sudo apt-key add -


$ gpg --keyserver pgp.mit.edu --recv-keys 2B5C1B00
gpg --export --armor 2B5C1B00 | sudo apt-key add -


$ gpg --keyserver pgp.mit.edu --recv-keys 0353B12C
gpg --export --armor 0353B12C | sudo apt-key add -
```

After adding the keys, get the updates and install Cassandra.

```
$ sudo apt-get update
$ sudo apt-get install cassandra
```

Step 3 — Starting and Connecting to cluster

```
$ sudo service cassandra status
```

If you were able to start Cassandra, verify the status of the cluster:

```
$ sudo nodetool status
```

Now connect to it using its interactive command line interface cqlsh.

```
$ cqlsh
```

There you go!!

Creating a Keyspace

In Cassandra, a keyspace is considered as a **namespace** that defines data replication on nodes. *A cluster comprises one keyspace per node.*

CREATE KEYSPACE < identifier > WITH < properties >

The CREATE KEYSPACE statement has two properties:

- replication
- durable_writes

e.g., CREATE KEYSPACE tutorialspoint WITH replication = {'class':'SimpleStrategy', 'replication_factor' : 3};

Alter Keyspace

ALTER KEYSPACE is utilized to change properties such as the durable_writes of a KeySpace and the number of replicas.

Syntax:

ALTER KEYSPACE < identifier > WITH < properties >

e.g., ALTER KEYSPACE company WITH replication = {'class':'SimpleStrategy', 'replication_factor' : 3};

Dropping a Keyspace

You can put a KeySpace with the help of the command DROP KEYSPACE.

Syntax:

DROP KEYSPACE < identifier >

e.g., DROP KEYSPACE company;

Create Your First Table

You can generate a table using the command **CREATE TABLE**.

Given below is the syntax for creating a table.

CREATE (TABLE | COLUMNFAMILY) <tablename>('<column-definition>' , '<column-definition>')(WITH <option> AND <option>)

e.g., CREATE TABLE emp(emp_id int PRIMARY KEY, emp_name text, emp_city text, emp_sal varint, emp_phone varint);

How to define a Column?

You can define a column as given below.

```
(column name1 datatype,
column name2 datatype)
```

e.g.,

(empname text, empcity text)

Primary Key

The prime key is a column that is utilized to identify a row uniquely.

Defining a primary key is mandatory while creating a table. A primary key can be made of one or more columns of a table.

```
CREATE TABLE tablename(
   column1 name datatype PRIMARYKEY,
   column2 name data type,
   column3 name data type.
   )
```

OR

```
CREATE TABLE tablename(
   column1 name datatype,
   column2 name data type,
   column3 name data type,
   PRIMARY KEY (column1)
   )
```

Create Data (Insert Operation)

You can include data into the columns of a row within a table with the help of the command INSERT.

Syntax:

INSERT INTO < tablename >(< column1 name >, < column2 name >....)VALUES (< value1 >, < value2 >....)USING < option >

e.g., INSERT INTO emp(emp_id, emp_name, emp_city) VALUES(1,'Sam', 'Bangalore');

Reading Data (Select Operation)

SELECT clause is utilized to read data from a table in Cassandra. By employing this clause, you can read a particular cell, whole table, or a single column.

Syntax:

SELECT FROM < tablename >

OR

SELECT FROM < tablename > WHERE < condition >

e.g., SELECT emp_name, emp_city from emp;

Update Data

UPDATE is the command for updating data in a table. The following keywords are used while updating data in a table:

- **Where** - This clause is used for choosing the row to be updated.
    - **Set** - Set the value using this keyword.
    - **Must** - Covers all the columns composing the primary key.

If a given row is not available, while updating rows, then UPDATE creates a fresh row.

Syntax:

UPDATE tablename SET < column name > = < new value > < column name > = < value >.... WHERE < condition >

e.g., UPDATE emp SET emp_city='Delhi' where emp_id=2;

Delete Data

Use the DELETE command to delete data from a table.

Syntax:

DELETE FROM < identifier > WHERE < condition >;

e.g., DELETE emp_name FROM emp WHERE emp_id=3;

Let Us Do it

Login to the **Katacoda playground**.

- Create a Keyspace in the name 'University' with replication factor 2.
- Go ahead and create two tables in it- 'Department1' , 'Department2' with dep_id as the primary key.
- Insert data and later try updating any of the fields.

Go a step further and explore!

1. Spark_Case_Study

Spark Test

In this exercise, you will be performing analysis on the given dataset in Apache Spark.

Data Set Description:

The dataset used in this question is the historical data of the h1b visa applications. The input file name is h1_b_dataset.parquet.

Problem Description:

The question is divided into 2 stages.

Stage 1:

- Read the input parquet file using Apache Spark.

- Select the required columns CASE_STATUS, VISA_CLASS, EMPLOYER_NAME, JOB_TITLE, PREVAILING_WAGE, PW_SOURCE_YEAR, WORKSITE_STATE.

- Rename the PREVAILING_WAGE as SALARY, PW_SOURCE_YEAR as FINANCIAL_YEAR.

- Filter all null columns and select only CASE_STATUS == CERTIFIED.

- LLC: Limited liability Company has to be removed from the employer name since the government decided to not investigate their case(hint: use endswith).

- Change the type of FINANCIAL_YEAR as Integer Type and SALARY as DoubleType.

- Drop the null in SALARY Columns if any

- Save the output dataframe as cleaned_data in csv type.

Schema of cleaned data:

```
 |-- CASE_STATUS: string (nullable = true)
 |-- VISA_CLASS: string (nullable = true)
 |-- EMPLOYER_NAME: string (nullable = true)
 |-- JOB_TITLE: string (nullable = true)
 |-- SALARY: double (nullable = true)
 |-- FINANCIAL_YEAR: integer (nullable = true)
 |-- WORKSITE_STATE: string (nullable = true)
```

Stage 2:

Using the cleaned dataframe find the answers for the below questions.

Question 1:

- Top 10 Employers who were given H-1B Visa.

- Save the result to a parquet file named top_approvals.

Output Sample:

```
+--------------------+-------------+
|      EMPLOYER_NAME|APPROVED_VISA|
+--------------------+-------------+
```

| ACCENTURE LLP| 18|

|DELOITTE CONSULTI...| 17|

+--------------------+------------+

Question 2:

- Find the Set1, Set2, and Set3 count in each WORKSITE_STATE?

salary > 10 and salary <= 10000 -> Set1

salary > 10000 and salary <= 50000 -> Set2

salary > 50000 and salary <= 100000 -> Set3

- The output should be in ascending order of WORKSITE_STATE.

- Save the output in csv format with name location_count.

Sample Output

+--------------+----+----+----+

|WORKSITE_STATE|Set1|Set2|Set3|

+--------------+----+----+----+

| AL| 1| 0| 0|

| AR| 0| 0| 4|

| AZ| 1| 1| 15|

| CA| 12| 7| 71|

+--------------+----+----+----+

Instructions for the Hands-On:

- All the necessary packages should be imported.
- Use the parquet file provided as the input.
- Repartition the result into a single before saving it.
- Run->install to install all the packages.