Apache Spark

Apache Spark is a unified analytics engine for large-scale data processing. It provides high level APIs like Spark SQL, Spark Streaming, MLib, and GraphX to allow interaction with core functionalities of Apache Spark. Spark also facilitates several core data abstractions on top of the distributed collection of data which are RDDs, DataFrames, and DataSets.

Spark RDD (Resilient Distributed Dataset)

Spark RDD stands for **Resilient Distributed Dataset** which is the core data abstraction API and is available since very first release of Spark (**Spark 1.0**). It is a lower-level API for manipulating distributed collection of data. The RDD APIs exposes some extremely useful methods which can be used to get very tight control over underlying physical data structure. It is an immutable (read only) collection of partitioned data distributed on different machines. RDD enables in-memory computation on large clusters to speed up big data processing in a fault tolerant manner.

To enable fault tolerance, RDD uses **DAG (Directed Acyclic Graph)** which consists of a set of vertices and edges. The vertices and edges in DAG represent the RDD and the operation to be applied on that RDD respectively. The transformations defined on RDD are lazy and executes only when an action is called. Let's have a quick look at features and limitations of RDD:

RDD Features:

1. **Immutable collection:** RDD is an immutable partitioned collection distributed on different nodes. A partition is a basic unit of parallelism in Spark. The immutability helps to achieve fault tolerance and consistency.
2. **Distributed data:** RDD is a collection of distributed data which helps in big data processing by distributing the workload to different nodes in the cluster.
3. **Lazy evaluation:** The defined transformations do not gets evaluated until an action is called. It helps Spark in optimizing the overall transformations in one go.
4. **Fault tolerant:** RDD can be recomputed in case of any failure using DAG(Directed acyclic graph) of transformations defined for that RDD.
5. **Multi-language support:** RDD APIs supports **Python, R, Scala, and Java** programming languages.

Limitation of RDD:

1. **No optimization engine:** RDD does not have an in-built optimization engine. Programmers need to write their own code in order to minimize the memory usage and to improve execution performance.

Spark DataFrame

Spark 1.3 introduced two new data abstraction APIs – **DataFrame and DataSet**. The DataFrame APIs organizes the data into named columns like a table in relational database. It enables programmers to define schema on a distributed collection of data. Each row in a DataFrame is of object type row. Like an SQL table, each column must have same number of rows in a DataFrame. In short, DataFrame is lazily evaluated plan which specifies the operations needs to be performed on the distributed collection of the data. DataFrame is also an immutable collection. Below are the features and limitations of DataFrame:

DataFrame Features:

1. **In-built Optimization:** DataFrame uses **Catalyst** engine which has an in-built execution optimization that improves the data processing performance significantly. When an action is called on a DataFrame, the Catalyst engine analyzes the code and resolves the references. Then, it creates a logical plan. After that, the created logical plan gets translated into an optimized physical plan. Finally, this physical plan gets executed on the cluster.
2. **Hive compatible:** The DataFrame is fully compatible with Hive query language. We can access all hive data, queries, UDFs, etc using Spark SQL from hive MetaStore and can execute queries against these hive databases.
3. **Structured, semi-structured, and highly structured data support:** DataFrame APIs supports manipulation of all kind of data from structured data files to semi-structured data files and highly structured parquet files.
4. **Multi-language support:** DataFrame APIs are available in **Python, R, Scala, and Java**.
5. **Schema support:** We can define a schema manually or we can read a schema from a data source which defines the column names and their data types.

DataFrame Limitations:

1. **Type safety:** Each row in a DataFrame is of object type row and hence is not strictly typed. That is why DataFrame does not support compile time safety.

Spark DataSet

As an extension to the DataFrame APIs, **Spark 1.3** also introduced DataSet APIs which provides strictly typed and object-oriented programming interface in Spark. It is immutable, type-safe collection of distributed data. Like DataFrame, DataSet APIs also uses Catalyst engine in order to enable execution optimization. DataSet is an extension to the DataFrame APIs. Features and limitations of the DataSet are as below:

DataSet Features:

1. **Combination of RDD and DataFrame:** DataSet enables functional programming like RDD APIs and relational queries and execution optimization like DataFrame APIs. Thus, it provides the benefit of best of both worlds – RDDs and DataFrames.
2. **Type-safe:** Unlike DataFrames, DataSet APIs provides compile time type safety. It conforms the specification at compile time using defined case classes (for Scala) or Java beans (for Java).

DataSet Limitations:

1. **Limited language support:** DataSet is only available to JVM based langauges like Java and Scala. Python and R do not support DataSet because these are dynamically typed languages.
2. **High garbage collection:** JVM types can cause high garbage collection and object instantiation cost.

RDD vs DataFrame vs DataSet

Below table represents a quick comparison between RDD, DataFrame and DataSet:

Feature	RDD	DataFrame	DataSet
Immutable	Yes	Yes	Yes
Fault tolerant	Yes	Yes	Yes
Type-safe	Yes	No	Yes
Schema	No	Yes	Yes
Execution optimization	No	Yes	Yes
Level	Low	High	High

RDD, DataFrame, and DataSet – Comparison

References:

https://sqlrelease.com/rdd-dataframe-and-dataset-introduction-to-spark-data-abstraction#:~:text=in%20Apache%20Spark.-,Spark%20Data%20Abstraction,computations%20in%20a%20distributed%20way.

https://data-flair.training/forums/topic/how-many-abstractions-are-provided-by-apache-spark/

https://spark.apache.org/docs/3.4.0/rdd-programming-guide.html#resilient-distributed-datasets-rdds

https://spark.apache.org/docs/latest/quick-start.html

https://spark.apache.org/docs/latest/rdd-programming-guide.html

https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/

Parallelized Collections and RDD examples in Spark:

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

Interactive Analysis with the Spark Shell

Basics

Spark's shell provides a simple way to learn the API, as well as a powerful tool to analyze data interactively. It is available in either Scala (which runs on the Java VM and is thus a good way to use existing Java libraries) or Python. Start it by running the following in the Spark directory:

- **Scala**
- **Python**

```
./bin/pyspark
```

Or if PySpark is installed with pip in your current environment:

```
pyspark
```

Spark's primary abstraction is a distributed collection of items called a Dataset. Datasets can be created from Hadoop InputFormats (such as HDFS files) or by transforming other Datasets. Due to Python's dynamic nature, we don't need the Dataset to be strongly-typed in Python. As a result, all Datasets in Python are Dataset[Row], and we call it `DataFrame` to be consistent with the data frame concept in Pandas and R. Let's make a new DataFrame from the text of the README file in the Spark source directory:

```
>>> textFile = spark.read.text("README.md")
```

You can get values from DataFrame directly, by calling some actions, or transform the DataFrame to get a new one. For more details, please read the *API doc*.

```
>>> textFile.count()  # Number of rows in this DataFrame
126

>>> textFile.first()  # First row in this DataFrame
Row(value=u'# Apache Spark')
```

Now let's transform this DataFrame to a new one. We call `filter` to return a new DataFrame with a subset of the lines in the file.

```
>>> linesWithSpark = textFile.filter(textFile.value.contains("Spark"))
```

We can chain together transformations and actions:

```
>>> textFile.filter(textFile.value.contains("Spark")).count()  # How many
lines contain "Spark"?
15
```

More on Dataset Operations

Dataset actions and transformations can be used for more complex computations. Let's say we want to find the line with the most words:

- **Scala**
- **Python**

```
>>> from pyspark.sql.functions import *
>>> textFile.select(size(split(textFile.value,
"\s+")).name("numWords")).agg(max(col("numWords"))).collect()
[Row(max(numWords)=15)]
```

This first maps a line to an integer value and aliases it as “numWords”, creating a new DataFrame. `agg` is called on that DataFrame to find the largest word count. The arguments to `select` and `agg` are both *Column*, we can use `df.colName` to get a column from a DataFrame. We can also import pyspark.sql.functions, which provides a lot of convenient functions to build a new Column from an old one.

One common data flow pattern is MapReduce, as popularized by Hadoop. Spark can implement MapReduce flows easily:

```
>>> wordCounts = textFile.select(explode(split(textFile.value,
"\s+")).alias("word")).groupBy("word").count()
```

Here, we use the `explode` function in `select`, to transform a Dataset of lines to a Dataset of words, and then combine `groupBy` and `count` to compute the per-word counts in the file as a DataFrame of 2 columns: “word” and “count”. To collect the word counts in our shell, we can call `collect`:

```
>>> wordCounts.collect()
[Row(word=u'online', count=1), Row(word=u'graphs', count=1), ...]
```

Spark RDD Operations- Transformations & Actions with Example

1. Spark RDD Operations

Two types of **Apache Spark** RDD operations are- Transformations and Actions. A **Transformation** is a function that produces new **RDD** from the existing RDDs but when we want to work with the actual dataset, at that point **Action** is performed. When the action is triggered after the result, new RDD is not formed like transformation.

Spark RDD Operations-Transformation & Action with Example

2. Apache Spark RDD Operations

Before we start with Spark RDD Operations, let us deep dive into RDD in Spark.

Apache Spark RDD supports two types of Operations-

- Transformations
- Actions

Now let us understand first what is Spark RDD Transformation and Action-

3. RDD Transformation

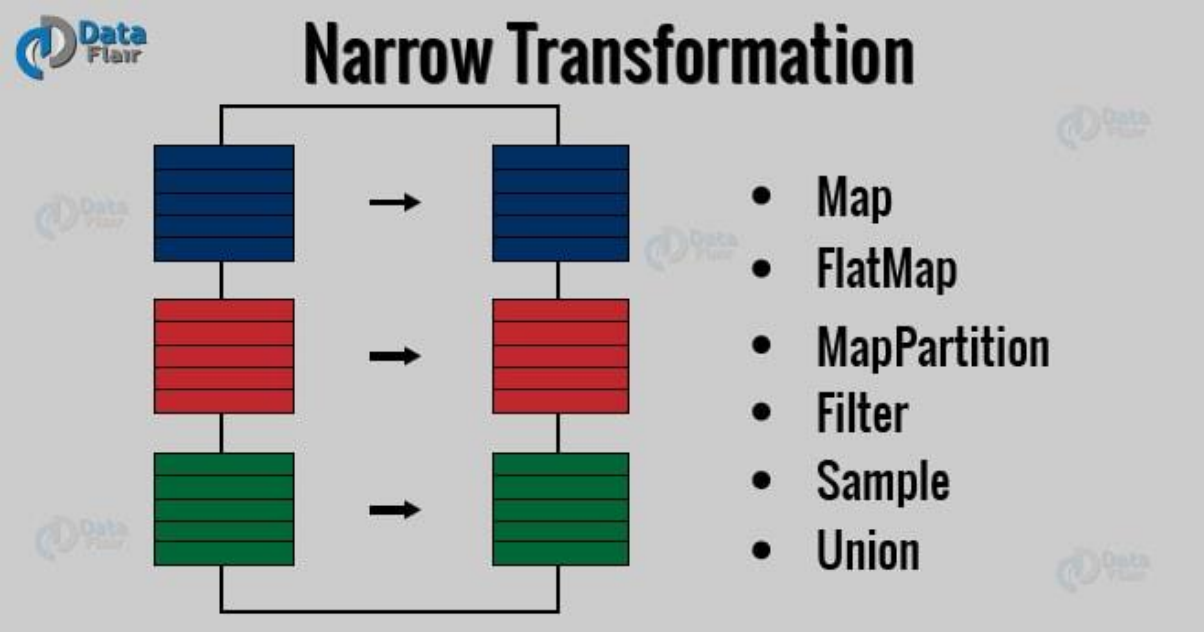
Spark Transformation is a function that produces new RDD from the existing RDDs. It takes RDD as input and produces one or more RDD as output. Each time it creates new RDD when we apply any transformation. Thus, the so input RDDs, cannot be changed since RDD are immutable in nature.

Applying transformation built an **RDD lineage**, with the entire parent RDDs of the final RDD(s). RDD lineage, also known as **RDD operator graph** or **RDD dependency graph.** It is a logical execution plan i.e., it is Directed Acyclic Graph (DAG) of the entire parent RDDs of RDD.

Transformations are lazy in nature i.e., they get execute when we call an action. They are not executed immediately. Two most basic type of

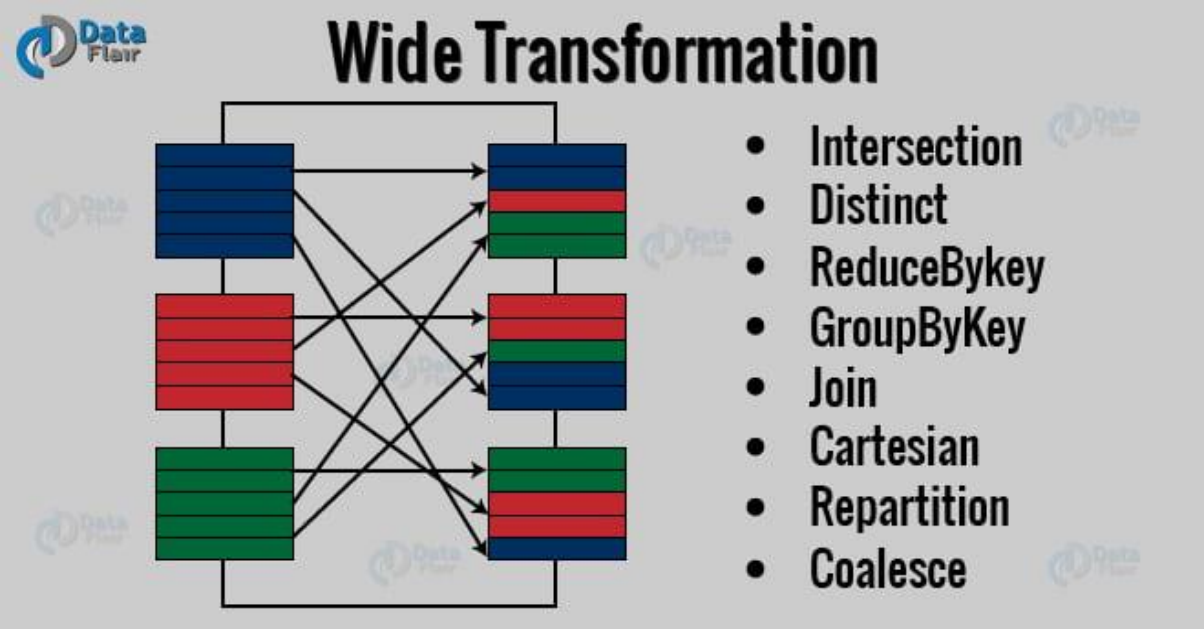
transformations is a map(), filter().
After the transformation, the resultant RDD is always different from its parent RDD. It can be smaller (e.g. filter, count, distinct, sample), bigger (e.g. flatMap(), union(), Cartesian()) or the same size (e.g. map).
There are two types of transformations:

- **Narrow transformation –** In *Narrow transformation*, all the elements that are required to compute the records in single partition live in the single partition of parent RDD. A limited subset of partition is used to calculate the result. *Narrow transformations* are the result of *map(), filter().*



Apache Spark Narrow Transformation Operation

- **Wide transformation –** In wide transformation, all the elements that are required to compute the records in the single partition may live in many partitions of parent RDD. The partition may live in many partitions of parent RDD. *Wide transformations* are the result of *groupbyKey()* and *reducebyKey()*.



Spark Wide Transformation Operations

There are various functions in RDD transformation. Let us see RDD transformation with examples.

3.1. map(func)

The map function iterates over every line in RDD and split into new RDD. Using **map()** transformation we take in any function, and that function is applied to every element of RDD.
In the map, we have the flexibility that the input and the return type of RDD may differ from each other. For example, we can have input RDD type as String, after applying the

map() function the return RDD can be Boolean.

For example, in RDD {1, 2, 3, 4, 5} if we apply "rdd.map(x=>x+2)" we will get the result as (3, 4, 5, 6, 7).

Also Read: **How to create RDD**
Map() example:
[php]import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
import org.apache.spark.sql.SparkSession
object mapTest{
def main(args: Array[String]) = {

val spark =
SparkSession.builder.appName(“mapExample”).master(“local”).getOrCrea
te()
val data = spark.read.textFile(“spark_test.txt”).rdd
val mapFile = data.map(line => (line,line.length))
mapFile.foreach(println)
}
}[/php]

spark_test.txt”

```
hello...user! this file is created to check the operations of
spark.


?, and how can we apply functions on that RDD partitions?. All
this will be done through spark programming which is done with
the help of scala language support…
```

- ***Note –*** In above code, map() function map each line of the file with its length.

3.2. flatMap()

With the help of **flatMap()** function, to each input element, we have many elements in an output RDD. The most simple use of flatMap() is to split each input string into words.
Map and flatMap are similar in the way that they take a line from input RDD and apply a function on that line. The key **difference between map() and flatMap()** is map() returns only one element, while flatMap() can return a list of elements.
flatMap() example:
[php]val data = spark.read.textFile(“spark_test.txt”).rdd
val flatmapFile = data.flatMap(lines => lines.split(” “))
flatmapFile.foreach(println)[/php]

- ***Note –*** In above code, flatMap() function splits each line when space occurs.

3.3. filter(func)

Spark RDD **filter()** function returns a new RDD, containing only the elements that meet a predicate. It is a *narrow operation* because it does not shuffle data from one partition to many partitions.
For example, Suppose RDD contains first five natural numbers (1, 2, 3, 4, and 5) and the predicate is check for an even number. The resulting RDD after the filter will contain only the even numbers i.e., 2 and 4.

Filter() example:
[php]val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")
println(mapFile.count())[/php]

- ***Note*** – In above code, flatMap function map line into words and then count the word "Spark" using count() Action after filtering lines containing "Spark" from mapFile.

Read: Apache Spark RDD vs DataFrame vs DataSet

3.4. mapPartitions(func)

The **MapPartition** converts each *partition* of the source RDD into many elements of the result (possibly none). In mapPartition(), the map() function is applied on each partitions simultaneously. MapPartition is like a map, but the difference is it runs separately on each partition(block) of the RDD.

3.5. mapPartitionWithIndex()

It is like mapPartition; Besides mapPartition it provides *func* with an integer value representing the index of the partition, and the map() is applied on partition index wise one after the other.
Learn: Spark Shell Commands to Interact with Spark-Scala

3.6. union(dataset)

With the **union()** function, we get the elements of both the RDD in new RDD. The key rule of this function is that the two RDDs should be of the same type.

For example, the elements of **RDD1** are (Spark, Spark, Hadoop, Flink) and that of **RDD2** are (Big data, Spark, Flink) so the resultant ***rdd1.union(rdd2)*** will have elements (Spark, Spark, Spark, Hadoop, Flink, Flink, Big data).

Union() example:

```
[php]val rdd1 =
spark.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",
2014)))
val rdd2 =
spark.sparkContext.parallelize(Seq((5,"dec",2014),(17,"sep",2015)))
val rdd3 =
spark.sparkContext.parallelize(Seq((6,"dec",2011),(16,"may",2015)))
val rddUnion = rdd1.union(rdd2).union(rdd3)
rddUnion.foreach(Println)[/php]
```

- ***Note –*** In above code union() operation will return a new dataset that contains the union of the elements in the source dataset (rdd1) and the argument (rdd2 & rdd3).

3.7. intersection(other-dataset)

With the **intersection()** function, we get only the common element of both the RDD in new RDD. The key rule of this function is that the two RDDs should be of the same type.

Consider an example, the elements of **RDD1** are (Spark, Spark, Hadoop, Flink) and that of **RDD2** are (Big data, Spark, Flink) so the resultant ***rdd1.intersection(rdd2)*** will have elements (spark).

Intersection() example:

```
[php]val rdd1 =
spark.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014,
(16,"feb",2014)))
val rdd2 =
spark.sparkContext.parallelize(Seq((5,"dec",2014),(1,"jan",2016)))
val comman = rdd1.intersection(rdd2)
comman.foreach(Println)[/php]
```

- ***Note –*** The intersection() operation return a new RDD. It contains the intersection of elements in the rdd1 & rdd2.

Learn to Install Spark on Ubuntu

3.8. distinct()

It returns a new dataset that contains the **distinct** elements of the source dataset. It is helpful to remove duplicate data.
For example, if RDD has elements (Spark, Spark, Hadoop, Flink), then ***rdd.distinct()*** will give elements (Spark, Hadoop, Flink).
Distinct() example:

```
[php]val rdd1 =
park.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014),(3,"nov",2014)))
val result = rdd1.distinct()
println(result.collect().mkString(", "))[/php]
```

- ***Note –*** In the above example, the distinct function will remove the duplicate record i.e. (3,'"nov",2014).

3.9. groupByKey()

When we use **groupByKey()** on a dataset of (K, V) pairs, the data is shuffled according to the key value K in another RDD. In this transformation, lots of unnecessary data get to transfer over the network.
Spark provides the provision to save data to disk when there is more data shuffled onto a single executor machine than can fit in memory. Follow this link to **learn about RDD Caching and Persistence mechanism** in detail.
groupByKey() example:

```
[php]val data =
spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
val group = data.groupByKey().collect()
group.foreach(println)[/php]
```

- ***Note –*** The groupByKey() will group the integers on the basis of same key(alphabet). After that *collect()* action will return all the elements of the dataset as an Array.

3.10. reduceByKey(func, [numTasks])

When we use **reduceByKey** on a dataset (K, V), the pairs on the same machine with the same key are combined, before the data is shuffled.
reduceByKey() example:
[php]val words =
Array("one","two","two","four","five","six","six","eight","nine","ten")
val data = spark.sparkContext.parallelize(words).map(w =>
(w,1)).reduceByKey(_+_)
data.foreach(println)[/php]

- ***Note –*** The above code will parallelize the Array of String. It will then map each word with count 1, then reduceByKey will merge the count of values having the similar key.

Read: Various Features of RDD

3.11. sortByKey()

When we apply the **sortByKey() function** on a dataset of (K, V) pairs, the data is sorted according to the key K in another RDD.
sortByKey() example:
[php] val data =
spark.sparkContext.parallelize(Seq(("maths",52), ("english",75), ("science", 82), ("computer",65), ("maths",85)))
val sorted = data.sortByKey()
sorted.foreach(println)[/php]

- ***Note –*** In above code, sortByKey() transformation sort the data RDD into Ascending order of the Key(String).

Read: Limitations of RDD

3.12. join()

The **Join** is database term. It combines the fields from two table using common values. join() operation in Spark is defined on pair-wise RDD. Pair-wise RDDs are RDD in which each element is in the form of tuples. Where the first element is key and the second element is the value.
The boon of using keyed data is that we can combine the data together. The join() operation combines two data sets on the basis of the key.

Join() example:

[php]val data = spark.sparkContext.parallelize(Array((‘A’,1),(‘b’,2),(‘c’,3)))
val data2
=spark.sparkContext.parallelize(Array((‘A’,4),(‘A’,6),(‘b’,7),(‘c’,3),(‘c’,8)))
val result = data.join(data2)
println(result.collect().mkString(“,”))[/php]

- ***Note*** – The join() transformation will join two different RDDs on the basis of Key.

Read: RDD lineage in Spark: ToDebugString Method

3.13. coalesce()

To avoid full shuffling of data we use coalesce() function. In **coalesce()** we use existing partition so that less data is shuffled. Using this we can cut the number of the partition. Suppose, we have four nodes and we want only two nodes. Then the data of extra nodes will be kept onto nodes which we kept.

Coalesce() example:

[php]val rdd1 =
spark.sparkContext.parallelize(Array(“jan”,”feb”,”mar”,”april”,”may”,”jun”)
,3)
val result = rdd1.coalesce(2)
result.foreach(println)[/php]

- ***Note –*** The coalesce will decrease the number of partitions of the source RDD to numPartitions define in coalesce argument.

4. RDD Action

Transformations create RDDs from each other, but when we want to work with the actual dataset, at that point action is performed. When the action is triggered after the result, new RDD is not formed like transformation. Thus, Actions are Spark RDD operations that give non-RDD values. The values of action are stored to drivers or to the external storage system. It brings laziness of RDD into motion.

An action is one of the ways of sending data from *Executer* to the *driver.* Executors are agents that are responsible for executing a task. While the driver is a JVM process that coordinates workers and execution of the task. Some of the actions of Spark are:

4.1. count()

Action **count()** returns the number of elements in RDD.
For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD
"rdd.count()" will give the result 8.

Count() example:
[php]val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.flatMap(lines => lines.split(" ")).filter(value =>
value=="spark")
println(mapFile.count())[/php]

- ***Note –*** In above code *flatMap()* function maps line into words and count the word "Spark" using *count()* Action after filtering lines containing "Spark" from mapFile.

Learn: Spark Streaming

4.2. collect()

The action **collect()** is the common and simplest operation that returns our entire RDDs content to driver program. The application of collect() is unit testing where the entire RDD is expected to fit in memory. As a result, it makes easy to compare the result of RDD with the expected result.
Action Collect() had a constraint that all the data should fit in the machine, and copies to the driver.
Collect() example:
[php]val data = spark.sparkContext.parallelize(Array(('A',1),('b',2),('c',3)))
val data2
=spark.sparkContext.parallelize(Array(('A',4),('A',6),('b',7),('c',3),('c',8)))
val result = data.join(data2)
println(result.collect().mkString(","))[/php]

- ***Note –*** *join()* transformation in above code will join two RDDs on the basis of same key(alphabet). After that *collect()* action will return all the elements to the dataset as an Array.

4.3. take(n)

The action **take(n)** returns n number of elements from RDD. It tries to cut the number of partition it accesses, so it represents a biased collection. We cannot presume the order of the elements.
For example, consider RDD {1, 2, 2, 3, 4, 5, 5, 6} in this RDD "take (4)" will give result { 2, 2, 3, 4}

Take() example:
[php]val data =
spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)

val group = data.groupByKey().collect()

val twoRec = result.take(2)

twoRec.foreach(println)[/php]

- ***Note*** – The *take(2)* Action will return an array with the first *n* elements of the data set defined in the taking argument.

Learn: Apache Spark DStream (Discretized Streams)

4.4. top()

If ordering is present in our RDD, then we can extract top elements from our RDD using **top()**. Action *top()* use default ordering of data.
Top() example:
[php]val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.map(line => (line,line.length))
val res = mapFile.top(3)
res.foreach(println)[/php]

- ***Note*** – *map()* operation will map each line with its length. And top(3) will return 3 records from mapFile with default ordering.

4.5. countByValue()

The **countByValue()** returns, many times each element occur in RDD.
For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD "rdd.countByValue()" will give the result {(1,1), (2,2), (3,1), (4,1), (5,2), (6,1)}

countByValue() example:
[php]val data = spark.read.textFile("spark_test.txt").rdd
val result= data.map(line => (line,line.length)).countByValue()
result.foreach(println)[/php]

- ***Note –*** The *countByValue()* action will return a hashmap of (K, Int) pairs with the count of each key.

Learn: Apache Spark Streaming Transformation Operations

4.6. reduce()

The **reduce()** function takes the two elements as input from the RDD and then produces the output of the same type as that of the input elements. The simple forms of such function are an addition. We can add the elements of RDD, count the number of words. It accepts commutative and associative operations as an argument.
Reduce() example:
[php]val rdd1 = spark.sparkContext.parallelize(List(20,32,45,62,8,5))
val sum = rdd1.reduce(_+_)
println(sum)[/php]

- ***Note*** – The *reduce()* action in above code will add the elements of the source RDD.

4.7. fold()

The signature of the **fold()** is like *reduce().* Besides, it takes "zero value" as input, which is used for the initial call on each partition. But, the **condition with zero value** is that it should be the **identity element of that operation**. The key difference between *fold()* and *reduce()* is that, *reduce()* throws an exception for empty collection, but *fold()* is defined for empty collection. For example, zero is an identity for addition; one is identity element for multiplication. The return type of *fold()* is same as that of the element of RDD we are operating on.
For example, rdd.fold(0)((x, y) => x + y).
Fold() example:
[php]val rdd1 = spark.sparkContext.parallelize(List(("maths", 80),("science", 90)))
val additionalMarks = ("extra", 4)

val sum = rdd1.fold(additionalMarks){ (acc, marks) => val add = acc._2 + marks._2
("total", add)
}
println(sum)[/php]

- ***Note –*** In above code *additionalMarks* is an initial value. This value will be added to the int value of each record in the source RDD.

Learn: Spark Streaming Checkpoint in Apache Spark

4.8. aggregate()

It gives us the flexibility to get data type different from the input type. The **aggregate()** takes two functions to get the final result. Through one function we combine the element from our RDD with the accumulator, and the second, to combine the accumulator. Hence, in aggregate, we supply the initial zero value of the type which we want to return.

4.9. foreach()

When we have a situation where we want to apply operation on each element of RDD, but it should not return value to the *driver*. In this case, **foreach()** function is useful. For example, inserting a record into the database.

Foreach() example:

[php]val data = spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
val group = data.groupByKey().collect()
group.foreach(println)[/php]

- ***Note –*** *The foreach()* action run a function *(println)* on each element of the dataset group.

Apache Flink – A Big Data Processing Platform

The Big Data came into limelight when **Google published a paper** (long ago in 2004) about MapReduce a programming paradigm it is using to handle the huge volume of contents for indexing the web(Learn more from here about History of big data).

It has been accepted by each and every domain (Telecom, Retail, Finance, Healthcare, Banking etc.) that Big Data is a must to handle the growing data and analytics requirements. Overall we can say Big Data is a must for each and every business to survive or grow.

After Hadoop we keep getting tons of technologies to handle Big Data, like MapReduce – Batch processing engine, Apache Storm – Stream processing engine, Apache Tez – Batch and interactive engine, Apache Giraph – Graph processing engine, Apache Hive – SQL engine.

Every framework is a specialized engine which solves some specific problems. But to solve real-world problems we need to combine multiple frameworks. Integration of multiple frameworks is powerful but making them work on the same platform is non-trivial, costly and complex.

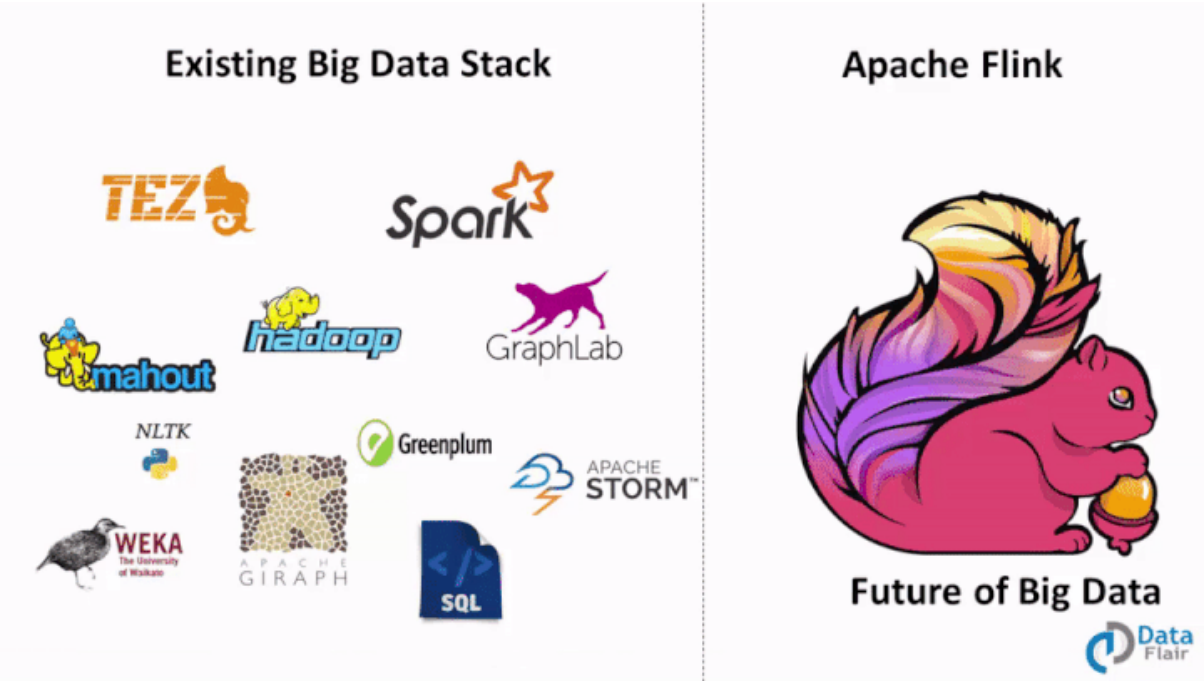
What is the solution??

The industry needs a generalized platform which alone can handle diverse workloads like:

- Batch Processing
- Interactive Processing
- Real-time (stream) Processing
- Graph Processing
- Iterative processing
- In-memory processing

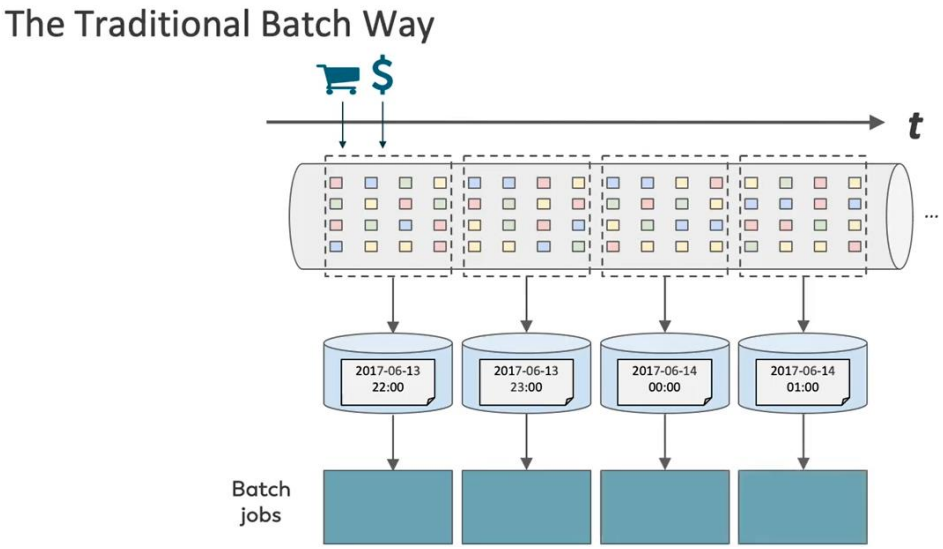
Thus. the platform should also provide distributed computing, fault tolerance, high availability, ease of use and lightning fast speed.

Apache Flink – The Unified Platform



Apache Spark has started the new trend by offering a diverse platform to solve different problems but is limited due to its underlying batch processing engine

---



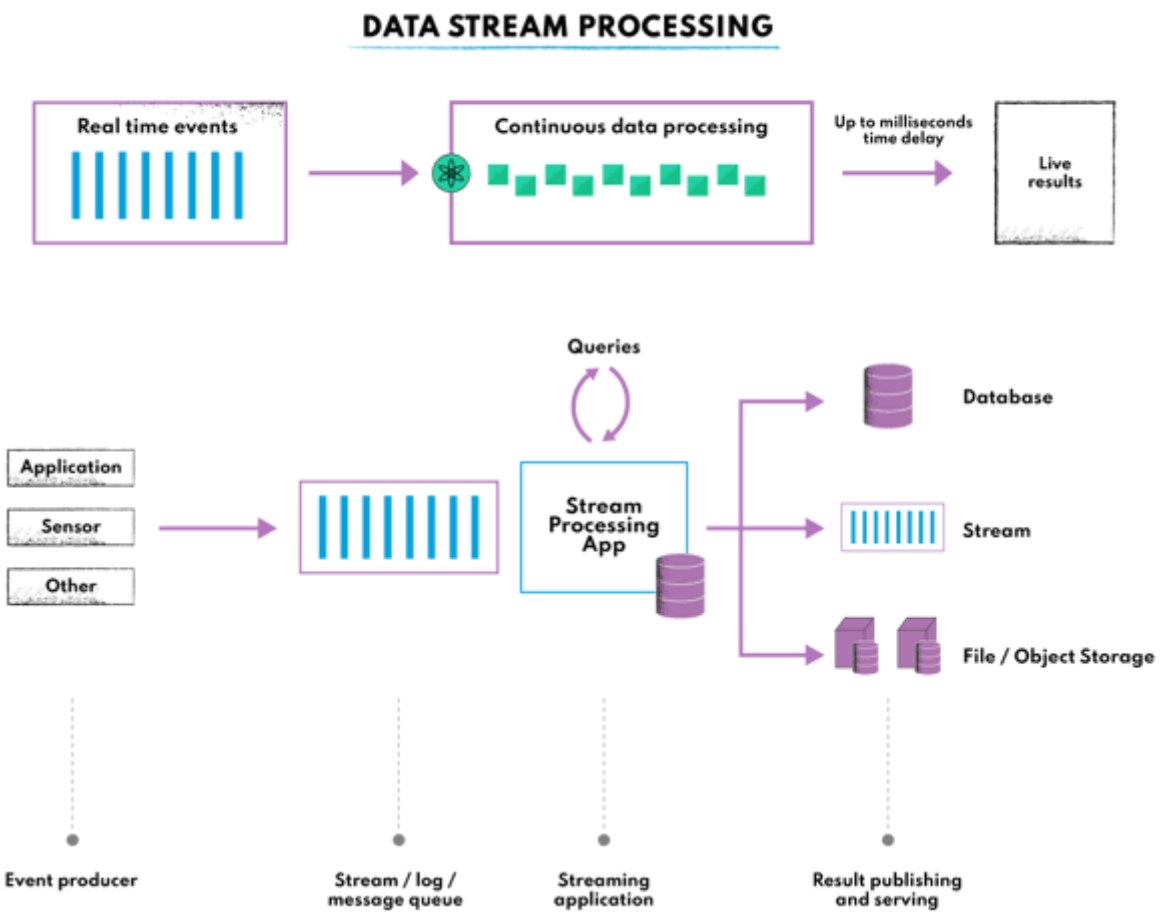
Processing based on the data collected over time is called Batch Processing. For example, a bank manager wants to process past one-month data (collected over time) to know the number of cheques that got cancelled in the past 1 month.

Processing based on immediate data for instant result is called Real-time Processing. For example, a bank manager getting a fraud alert immediately after a fraud transaction (instant result) has occurred.

What is Stream Processing?

Before we get into Apache Flink, it's essential to understand stream processing. Stream processing is a type of data processing that deals with **continuous, real-time data streams.**

Batch processing can be thought of as dealing with "data at rest," while stream processing deals with "data in motion."



stream processing has several benefits over batch processing:

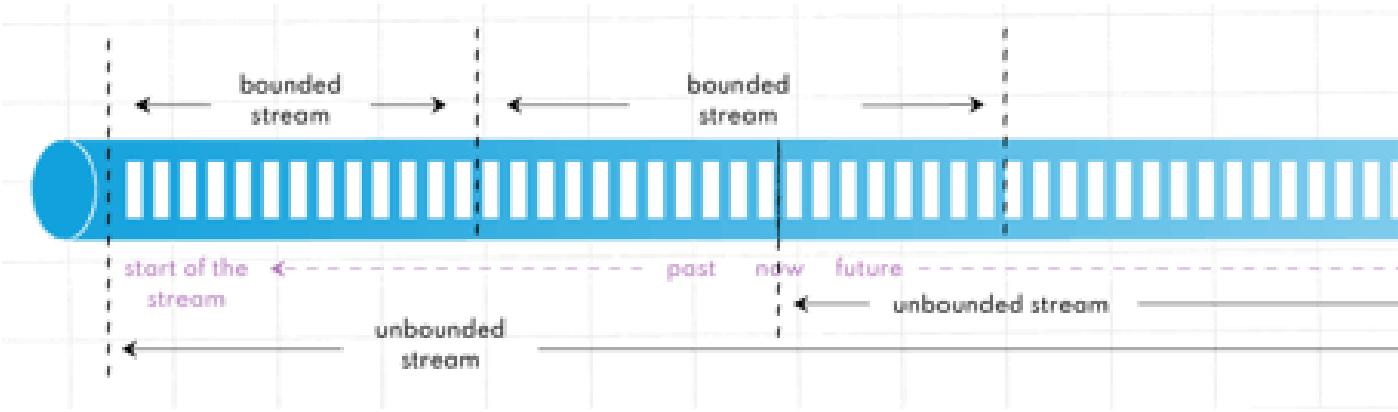
- **Lower latency**: Since stream processors deal with data in **near-real-time**, the overall latency is lower and offers the opportunity for multiple specific use cases that need in-motion checks.
- **Flexibility:** Stream process transaction data is generally more flexible than batch, as a wider variety of end applications, data types, and formats can easily be handled. It can also accommodate changes to the data sources (e.g., adding a new sensor to an IoT application).

- **Less expensive**: Since stream processors can handle a continuous data flow, the overall cost is lower (lack of a need to store data before processing it).

Apache Flink is a general purpose cluster computing tool, which can handle batch processing, interactive processing, Stream processing, Iterative processing, in-memory processing, graph processing.

Process Unbounded and Bounded Data Streams

Apache Flink allows for both bounded and unbounded data stream processing. Bounded data streams are finite, while unbounded streams are infinite.



Bounded and unbounded streams

Bounded Data Streams

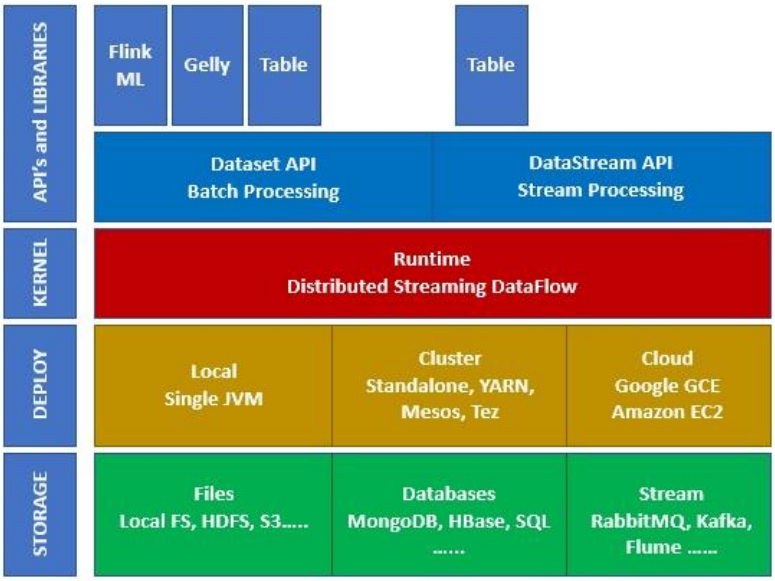
Bounded data streams have a **defined beginning and end**; they can be processed in one batch job or multiple parallel jobs. Apache Flink's **DataSet API** is used to process bounded data sets, consisting of individual elements over which the user iterates. This type of system is often used for batch-like processing of data that is already present and known ahead of time - such as a customer database or log files.

Unbounded Streams

Unbounded data streams, on the other hand, have no start or end point; they continuously receive new elements that need to be processed right away. This type of processing requires a system that is always running and ready to accept incoming elements as soon as they arrive. To accomplish this, Apache Flink offers a **DataStream API** for real-time processing of the streaming data, allowing users to write applications that process unbounded streams of data.

Ecosystem on Apache Flink

The diagram given below shows the different layers of Apache Flink Ecosystem –



Storage

Apache Flink has multiple options from where it can Read/Write data. Below is a basic storage list −

- HDFS (Hadoop Distributed File System)
- Local File System
- S3
- RDBMS (MySQL, Oracle, MS SQL etc.)
- MongoDB
- HBase
- Apache Kafka
- Apache Flume

Deploy

You can deploy Apache Fink in local mode, cluster mode or on cloud. Cluster mode can be standalone, YARN, MESOS.

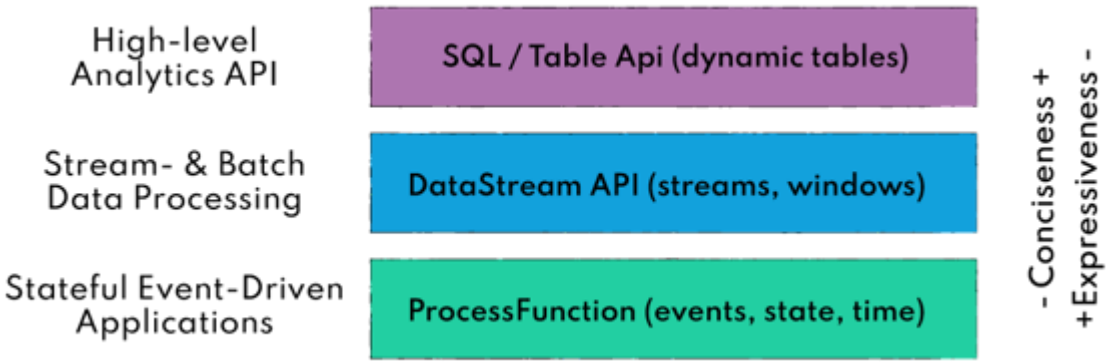
On cloud, Flink can be deployed on AWS or GCP.

Kernel

This is the runtime layer, which provides distributed processing, fault tolerance, reliability, native iterative processing capability and more.

APIs & Libraries

This is the top layer and most important layer of Apache Flink. It has Dataset API, which takes care of batch processing, and Datastream API, which takes care of stream processing. There are other libraries like Flink ML (for machine learning), Gelly (for graph processing), Tables for SQL. This layer provides diverse capabilities to Apache Flink.



Complex Event Processing (CEP)

Flink's Complex Event Processing library allows users to specify patterns of events using a regular expression or state machine.

SQL & Table API

The Flink ecosystem also includes APIs for relational queries - SQL and Table APIs.

Gelly

Gelly is a versatile graph processing and analysis library that runs on top of the DataSet API

FlinkML

FlinkML is a library of distributed machine learning algorithms that run on top of the DataSet API.

Key Use Cases for Flink

Apache Flink is a powerful tool for handling big data and streaming applications. It supports both bounded and unbounded data streams, making it an ideal platform for a variety of use cases, such as:

- **Event-driven applications:**

fraud detection, anomaly detection, rule-based alerting, business process monitoring, financial and credit card transactions systems, social networks, and other message-driven systems

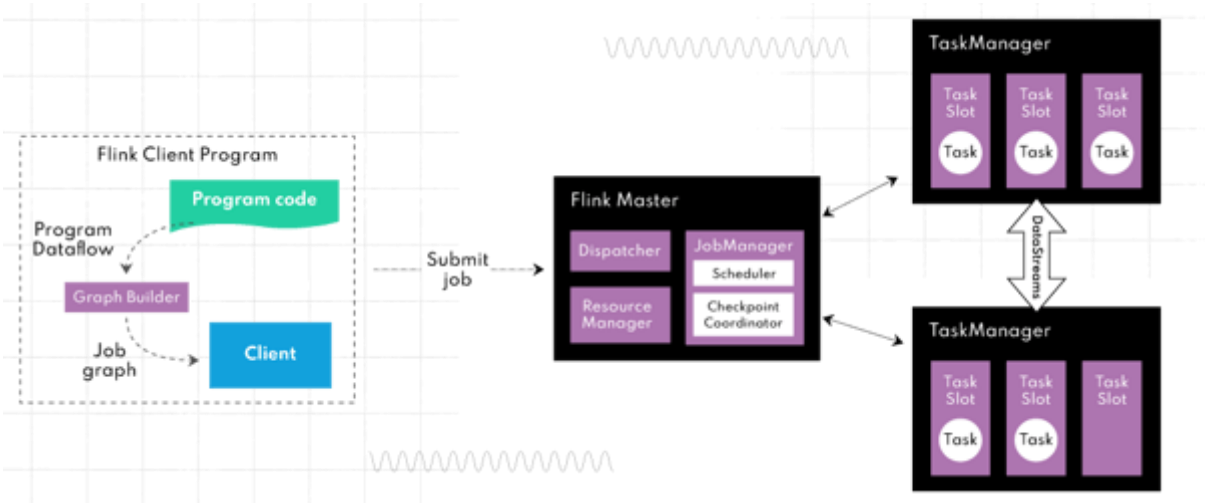
Continuous data pipelines:

Real-time data analytics:

ad-hoc analysis of live data in various industries, customer experience monitoring, large-scale graph analysis, and network intrusion detection.

master/slave architecture

Flink uses a master/slave architecture with **JobManager** and **TaskManagers**. The Job Manager is responsible for scheduling and managing the jobs submitted to Flink and orchestrating the execution plan by allocating resources for tasks. The Task Managers are accountable for executing user-defined functions on allocated resources across multiple nodes in a cluster.



WORD COUNT EXAMPLE

all core classes of the Java DataStream API can be found in org.apache.flink.streaming.api .

the Flink cluster manager will execute your main method and `getExecutionEnvironment()` will return an execution environment for executing your program on a cluster.

For specifying data sources the execution environment has several methods to read from files using various methods: you can just read them line by line, as CSV files, or using any of the other provided sources.

```
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

DataStream<String> text = env.readTextFile("file:///path/to/file");
```

This will create a new DataStream by converting every String in the original collection to an Integer.

You apply transformations by calling methods on DataStream with a transformation functions

```
DataStream<String> input = ...;

DataStream<Integer> parsed = input.map(new MapFunction<String, Integer>() {
    @Override
    public Integer map(String value) {
        return Integer.parseInt(value);
    }
});
```

The `execute()` method will wait for the job to finish and then return a `JobExecutionResult`

Once you have a DataStream containing your final results, you can write it to an outside system by creating a sink. These are just some example methods for creating a sink:

```
writeAsText(String path);

print();
```

```
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.util.Collector;

public class WindowWordCount {

    public static void main(String[] args) throws Exception {

        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStream<Tuple2<String, Integer>> dataStream = env
                .socketTextStream("localhost", 9999)
                .flatMap(new Splitter())
                .keyBy(value -> value.f0)
                .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
                .sum(1);
```

```
        dataStream.print();

        env.execute("Window WordCount");
    }

    public static class Splitter implements FlatMapFunction<String, Tuple2<String,
Integer>> {
        @Override
        public void flatMap(String sentence, Collector<Tuple2<String, Integer>>
out) throws Exception {
            for (String word: sentence.split(" ")) {
                out.collect(new Tuple2<String, Integer>(word, 1));
            }
        }
    }

}
```

Reference

https://nightlies.apache.org/flink/flink-docs-release-1.1/apis/batch/examples.html#word-count

Apache Zookeeper

- The ZooKeeper framework was originally built at “Yahoo!” for accessing their applications in an easy and robust manner.
- Later, Apache ZooKeeper became a standard for organized service used by **Hadoop, HBase**, and other distributed frameworks.
- For example, Apache HBase uses ZooKeeper to **track the status** of distributed data.
- ZooKeeper is a distributed co-ordination service to manage large set of hosts.
- Co-ordinating and managing a service in a distributed environment is a **complicated process**.
- ZooKeeper solves this issue with its **simple architecture and API**.
- ZooKeeper allows developers to focus on **core application logic** without worrying about the distributed nature of the application.

- **Cluster synchronization**: Apache ZooKeeper is a service used by a **cluster** (group of nodes) to coordinate between themselves and maintain **shared data** with **robust synchronization** techniques.
- ZooKeeper is itself a distributed application providing services for writing a distributed application.

Common Services

- **Naming service** − Identifying the nodes in a cluster by name. It is similar to DNS, but for nodes.
- **Configuration management** − Latest and up-to-date configuration information of the system for a joining node.
- **Cluster management** − Joining / leaving of a node in a cluster and node status at real time.
- **Leader election** − Electing a node as leader for coordination purpose.
- **Locking and synchronization service** − Locking the data while modifying it. This mechanism helps you in automatic fail recovery while connecting other distributed applications like Apache HBase.
- **Highly reliable data registry** − Availability of data even when one or a few nodes are down.

Distributed applications vs zookeeper

- Distributed applications offer a lot of benefits, but they throw a few complex and hard-to-crack challenges as well.

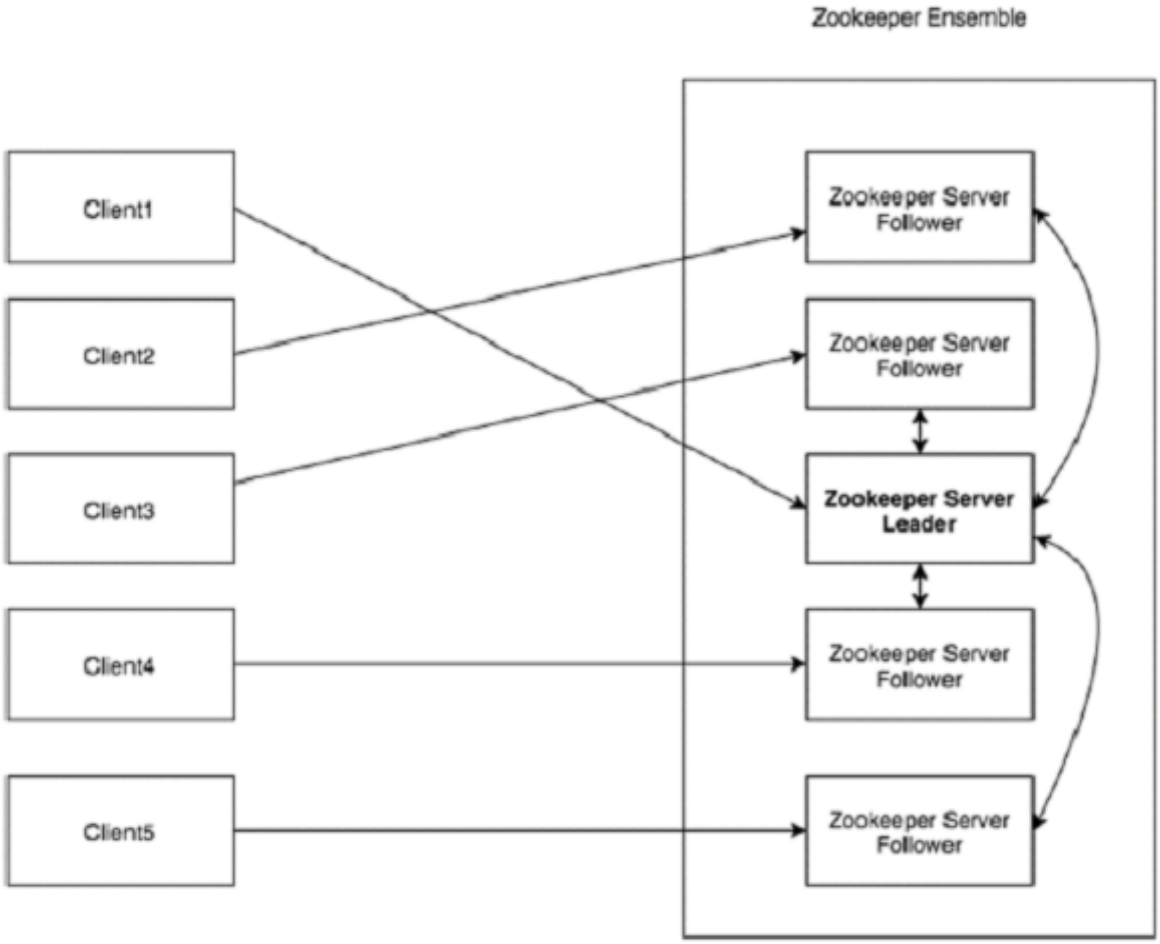
  (eg : distributed appliation problems: race condition, deadlock, inconsistency)
- ZooKeeper framework provides a complete mechanism to overcome all the challenges.
- Race condition and deadlock are handled using **fail-safe synchronization approach**.

- Another main drawback is inconsistency of data, which ZooKeeper resolves with **atomicity**.

BENEFITS OF ZOOKEEPER

- **Simple distributed coordination process**
- **Synchronization** − Mutual exclusion and co-operation between server processes. This process helps in Apache HBase for configuration management.
- **Ordered Messages**
- **Serialization** − Encode the data according to specific rules. Ensure your application runs consistently. This approach can be used in MapReduce to coordinate queue to execute running threads.
- **Reliability**
- **Atomicity** − Data transfer either succeed or fail completely, but no transaction is partial.

Zookeeper Architecture and Data Model



ach one of the components that is a part of the ZooKeeper architecture has been explained in the following table.

Part	Description
Client	Clients, one of the nodes in our distributed application cluster, access information from the server. For a particular time interval, every client sends a message to the server to let the sever know that the client is alive. Similarly, the server sends an acknowledgement when a client connects. If there is no response from the connected server, the client automatically redirects the message to another server.
Server	Server, one of the nodes in our ZooKeeper ensemble, provides all the services to clients. Gives acknowledgement to client to inform that the server is alive.
Ensemble	Group of ZooKeeper servers. The minimum number of nodes that is required to form an ensemble is 3.
Leader	Server node which performs automatic recovery if any of the connected node failed. Leaders are elected on service startup.
Follower	Server node which follows leader instruction.

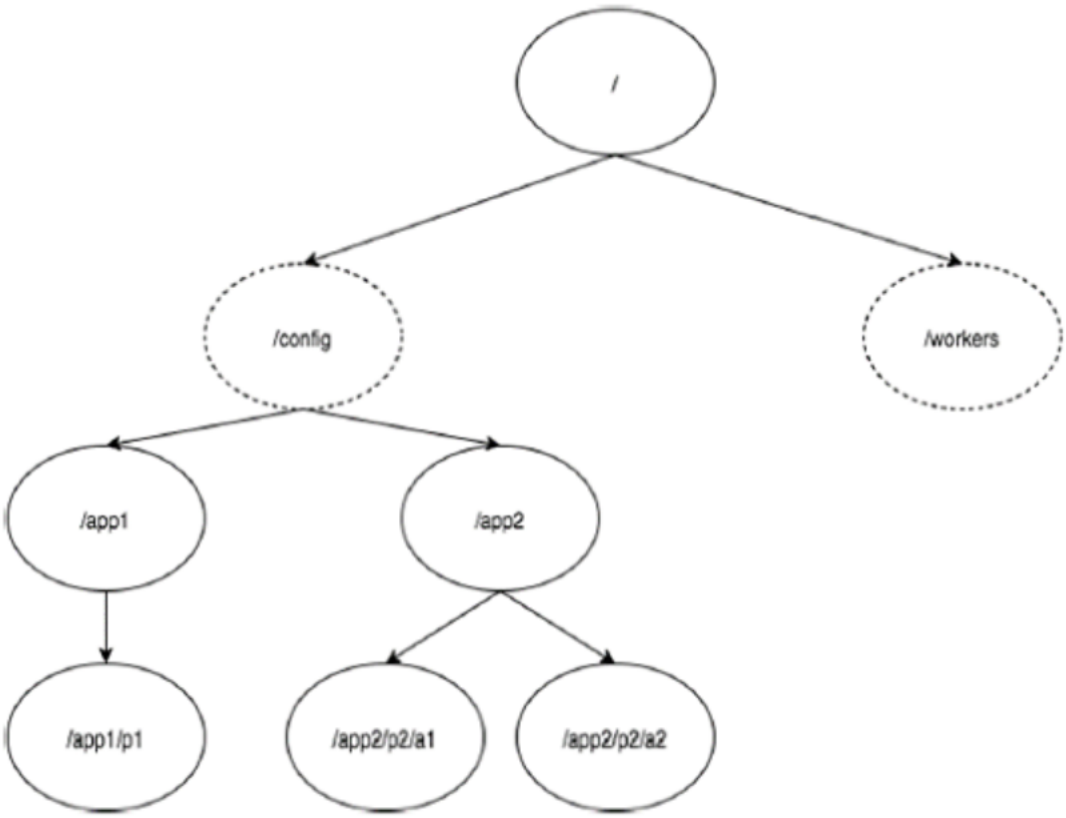
Hierarchical Namespace

The following diagram depicts the tree structure of ZooKeeper file system used for memory representation. ZooKeeper node is referred as **znode**. Every znode is identified by a name and separated by a sequence of path (/).

- In the diagram, first you have a root **znode** separated by "/". Under root, you have two logical namespaces **config** and **workers**.
- The **config** namespace is used for centralized configuration management and the **workers** namespace is used for naming.
- Under **config** namespace, each znode can store upto 1MB of data. This is similar to UNIX file system except that the parent znode can store data as well. The main purpose of this structure is to store synchronized data and describe the metadata of the znode. This structure is called as **ZooKeeper Data Model**.

Every znode in the ZooKeeper data model maintains a **stat** structure. A stat simply provides the **metadata** of a znode. It consists of *Version number, Action control list (ACL), Timestamp, and Data length.*

- **Version number** – Every znode has a version number, which means every time the data associated with the znode changes, its corresponding version number would also increased. The use of version number is important when multiple zookeeper clients are trying to perform operations over the same znode.
- **Action Control List (ACL)** – ACL is basically an authentication mechanism for accessing the znode. It governs all the znode read and write operations.
- **Timestamp** – Timestamp represents time elapsed from znode creation and modification. It is usually represented in milliseconds. ZooKeeper identifies every change to the znodes from "Transaction ID" (zxid). **Zxid** is unique and maintains time for each transaction so that you can easily identify the time elapsed from one request to another request.
- **Data length** – Total amount of the data stored in a znode is the data length. You can store a maximum of 1MB of data.



Types of znode

- Persistent
- Ephemeral
- Sequential

- **Persistence znode** – Persistence znode is alive even after the client, which created that particular znode, is disconnected. By default, all znodes are persistent unless otherwise specified.
- **Ephemeral znode** – Ephemeral znodes are active until the client is alive. When a client gets disconnected from the ZooKeeper ensemble, then the ephemeral znodes get deleted automatically. For this reason, only ephemeral znodes are not allowed to have a children further. If an ephemeral znode is deleted, then the next suitable node will fill its position. Ephemeral znodes play an important role in Leader election.
- **Sequential znode** – Sequential znodes can be either persistent or ephemeral. When a new znode is created as a sequential znode, then ZooKeeper sets the path of the znode by attaching a 10 digit sequence number to the original name. For example, if a znode with path **/myapp** is created as a sequential znode, ZooKeeper will change the path to **/myapp0000000001** and set the next sequence number as 0000000002. If two sequential znodes are created concurrently, then ZooKeeper never uses the same number for each znode. Sequential znodes play an important role in Locking and Synchronization.

Zookeeper Sessions

- Time unit of a series of operation
- Requests - FIFO
- Sessionid
- Heartbeats – client to ensemble
- Timeouts
- Sessions are very important for the operation of ZooKeeper. Requests in a session are executed in FIFO order. Once a client connects to a server, the session will be established and a **session id** is assigned to the client.
- The client sends **heartbeats** at a particular time interval to keep the session valid. If the ZooKeeper ensemble does not receive heartbeats from a client for more than the period (session timeout) specified at the starting of the service, it decides that the client died.
- Session timeouts are usually represented in milliseconds. When a session ends for any reason, the ephemeral znodes created during that session also get deleted.

Zookeeper Watches

- Simple mechanism for the client to get notifications about the changes in the ZooKeeper ensemble
- Clients can set watches while reading a particular znode
- Watches send a notification to the registered client for any of the znode (on which client registers) changes.
    - Data or children
- Triggered once
- Deleted at session termination
- Watches are a simple mechanism for the client to get notifications about the changes in the ZooKeeper ensemble. Clients can set watches while reading a particular znode. Watches send a notification to the registered client for any of the znode (on which client registers) changes.
- Znode changes are modification of data associated with the znode or changes in the znode's children. Watches are triggered only once. If a client wants a notification again, it must be done through another read operation. When a connection session is expired, the client will be disconnected from the server and the associated watches are also removed.

Zookeeper Command Line Interface Commands (CLI):

ZooKeeper Command Line Interface (CLI) is used to interact with the ZooKeeper ensemble for development purpose. It is useful for debugging and working around with different options.

To perform ZooKeeper CLI operations, first turn on your ZooKeeper server ("*bin/zkServer.sh start*") and then, ZooKeeper client ("*bin/zkCli.sh*"). Once the client starts, you can perform the following operation –

- Create znodes
- Get data
- Watch znode for changes
- Set data
- Create children of a znode
- List children of a znode
- Check Status
- Remove / Delete a znode

Now let us see above command one by one with an example.

Create Znodes

Create a znode with the given path. The **flag** argument specifies whether the created znode will be ephemeral, persistent, or sequential. By default, all znodes are persistent.

- **Ephemeral znodes** (flag: e) will be automatically deleted when a session expires or when the client disconnects.
- **Sequential znodes** guaranty that the znode path will be unique.
- ZooKeeper ensemble will add sequence number along with 10 digit padding to the znode path. For example, the znode path */myapp will be converted to /myapp0000000001* and the next sequence number will be */myapp0000000002*. If no flags are specified, then the znode is considered as **persistent**.

Syntax

```
create /path /data
```

Sample

```
create /FirstZnode "Myfirstzookeeper-app"
```

Output

```
[zk: localhost:2181(CONNECTED) 0] create /FirstZnode "Myfirstzookeeper-app"
Created /FirstZnode
```

To create a **Sequential znode**, add **-s flag** as shown below.

Syntax

```
create -s /path /data
```

Sample

```
create -s /FirstZnode second-data
```

Output

```
[zk: localhost:2181(CONNECTED) 2] create -s /FirstZnode "second-data"
Created /FirstZnode0000000023
```

To create an **Ephemeral Znode**, add **-e flag** as shown below.

Syntax

```
create -e /path /data
```

Sample

```
create -e /SecondZnode "Ephemeral-data"
```

Output

```
[zk: localhost:2181(CONNECTED) 2] create -e /SecondZnode "Ephemeral-data"
Created /SecondZnode
```

Remember when a client connection is lost, the ephemeral znode will be deleted. You can try it by quitting the ZooKeeper CLI and then re-opening the CLI.

Get Data

It returns the associated data of the znode and metadata of the specified znode. You will get information such as when the data was last modified, where it was modified, and information about the data. This CLI is also used to assign watches to show notification about the data.

Syntax

```
get /path
```

Sample

```
get /FirstZnode
```

Output

```
[zk: localhost:2181(CONNECTED) 1] get /FirstZnode
“Myfirstzookeeper-app”
cZxid = 0x7f
ctime = Tue Sep 29 16:15:47 IST 2015
mZxid = 0x7f
mtime = Tue Sep 29 16:15:47 IST 2015
pZxid = 0x7f
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 22
numChildren = 0
```

To access a sequential znode, you must enter the full path of the znode.

Sample

```
get /FirstZnode0000000023
```

Output

```
[zk: localhost:2181(CONNECTED) 1] get /FirstZnode0000000023
“Second-data”
cZxid = 0x80
ctime = Tue Sep 29 16:25:47 IST 2015
mZxid = 0x80
mtime = Tue Sep 29 16:25:47 IST 2015
pZxid = 0x80
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 13
numChildren = 0
```

Watch

Watches show a notification when the specified znode or znode's children data changes. You can set a **watch** only in **get** command.

Syntax

```
get /path [watch] 1
```

Sample

```
get /FirstZnode 1
```

Output

```
[zk: localhost:2181(CONNECTED) 1] get /FirstZnode 1
"Myfirstzookeeper-app"
cZxid = 0x7f
ctime = Tue Sep 29 16:15:47 IST 2015
mZxid = 0x7f
mtime = Tue Sep 29 16:15:47 IST 2015
pZxid = 0x7f
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 22
numChildren = 0
```

The output is similar to normal **get** command, but it will wait for znode changes in the background. <Start here>

Set Data

Set the data of the specified znode. Once you finish this set operation, you can check the data using the **get** CLI command.

Syntax

```
set /path /data
```

Sample

```
set /SecondZnode Data-updated
```

Output

```
[zk: localhost:2181(CONNECTED) 1] get /SecondZnode "Data-updated"
cZxid = 0x82
ctime = Tue Sep 29 16:29:50 IST 2015
mZxid = 0x83
mtime = Tue Sep 29 16:29:50 IST 2015
pZxid = 0x82
cversion = 0
```

```
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x15018b47db00000
dataLength = 14
numChildren = 0
```

If you assigned **watch** option in **get** command (as in previous command), then the output will be similar as shown below –

Output

```
[zk: localhost:2181(CONNECTED) 1] get /FirstZnode "Mysecondzookeeper-app"

WATCHER: :

WatchedEvent state:SyncConnected type:NodeDataChanged path:/FirstZnode
cZxid = 0x7f
ctime = Tue Sep 29 16:15:47 IST 2015
mZxid = 0x84
mtime = Tue Sep 29 17:14:47 IST 2015
pZxid = 0x7f
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 23
numChildren = 0
```

Create Children / Sub-znode

Creating children is similar to creating new znodes. The only difference is that the path of the child znode will have the parent path as well.

Syntax

```
create /parent/path/subnode/path /data
```

Sample

```
create /FirstZnode/Child1 firstchildren
```

Output

```
[zk: localhost:2181(CONNECTED) 16] create /FirstZnode/Child1 "firstchildren"
created /FirstZnode/Child1
[zk: localhost:2181(CONNECTED) 17] create /FirstZnode/Child2 "secondchildren"
created /FirstZnode/Child2
```

List Children

This command is used to list and display the **children** of a znode.

Syntax

```
ls /path
```

Sample

```
ls /MyFirstZnode
```

Output

```
[zk: localhost:2181(CONNECTED) 2] ls /MyFirstZnode
[mysecondsubnode, myfirstsubnode]
```

Check Status

Status describes the metadata of a specified znode. It contains details such as Timestamp, Version number, ACL, Data length, and Children znode.

Syntax

```
stat /path
```

Sample

```
stat /FirstZnode
```

Output

```
[zk: localhost:2181(CONNECTED) 1] stat /FirstZnode
cZxid = 0x7f
ctime = Tue Sep 29 16:15:47 IST 2015
mZxid = 0x7f
mtime = Tue Sep 29 17:14:24 IST 2015
pZxid = 0x7f
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 23
numChildren = 0
```

Remove a Znode

Removes a specified znode and recursively all its children. This would happen only if such a znode is available.

Syntax

```
rmr /path
```

Sample

```
rmr /FirstZnode
```

Output

```
[zk: localhost:2181(CONNECTED) 10] rmr /FirstZnode
[zk: localhost:2181(CONNECTED) 11] get /FirstZnode
Node does not exist: /FirstZnode
```

Delete **(delete /path)** command is similar to **remove** command, except the fact that it works only on znodes with no children.

Zookeeper four letter commands:

ZooKeeper Commands: The Four Letter Words

ZooKeeper responds to a small set of commands. Each command is composed of four letters. You issue the commands to ZooKeeper via telnet or nc, at the client port.

Three of the more interesting commands: "stat" gives some general information about the server and connected clients, while "srvr" and "cons" give extended details on server and connections respectively.

conf

> **New in 3.3.0:** Print details about serving configuration.

cons

> **New in 3.3.0:** List full connection/session details for all clients connected to this server. Includes information on numbers of packets received/sent, session id, operation latencies, last operation performed, etc...

crst

> **New in 3.3.0:** Reset connection/session statistics for all connections.

dump

> Lists the outstanding sessions and ephemeral nodes. This only works on the leader.

envi

> Print details about serving environment

ruok

> Tests if server is running in a non-error state. The server will respond with imok if it is running. Otherwise it will not respond at all.
>
> A response of "imok" does not necessarily indicate that the server has joined the quorum, just that the server process is active and bound to the specified client port. Use "stat" for details on state wrt quorum and client connection information.

srst

> Reset server statistics.

srvr

> **New in 3.3.0:** Lists full details for the server.

stat

Lists brief details for the server and connected clients.

wchs

New in 3.3.0: Lists brief information on watches for the server.

wchc

New in 3.3.0: Lists detailed information on watches for the server, by session. This outputs a list of sessions(connections) with associated watches (paths). Note, depending on the number of watches this operation may be expensive (ie impact server performance), use it carefully.

wchp

New in 3.3.0: Lists detailed information on watches for the server, by path. This outputs a list of paths (znodes) with associated sessions. Note, depending on the number of watches this operation may be expensive (ie impact server performance), use it carefully.

mntr

New in 3.4.0: Outputs a list of variables that could be used for monitoring the health of the cluster.

```
$ echo mntr | nc localhost 2185

zk_version  3.4.0
zk_avg_latency  0
zk_max_latency  0
zk_min_latency  0
zk_packets_received 70
zk_packets_sent 69
zk_outstanding_requests 0
zk_server_state leader
zk_znode_count   4
zk_watch_count  0
zk_ephemerals_count 0
zk_approximate_data_size    27
zk_followers    4                   - only exposed by the Leader
zk_synced_followers 4               - only exposed by the Leader
zk_pending_syncs    0               - only exposed by the Leader
zk_open_file_descriptor_count 23    - only available on Unix
platforms
zk_max_file_descriptor_count 1024   - only available on Unix
platforms
```

The output is compatible with java properties format and the content may change over time (new keys added). Your scripts should expect changes.

ATTENTION: Some of the keys are platform specific and some of the keys are only exported by the Leader.

The output contains multiple lines with the following format:

```
key \t value
```

Here's an example of the **ruok** command:

```
$ echo ruok | nc 127.0.0.1 5111
```

```
imok
```

Apache Oozie

Oozie is a workflow scheduler system to manage Apache Hadoop jobs.

Oozie Workflow jobs are Directed Acyclical Graphs (DAGs) of actions.

Oozie Coordinator jobs are recurrent Oozie Workflow jobs triggered by time (frequency) and data availability.

Oozie is integrated with the rest of the Hadoop stack supporting several types of Hadoop jobs out of the box (such as Java map-reduce, Streaming map-reduce, Pig, Hive, Sqoop and Distcp) as well as system specific jobs (such as Java programs and shell scripts).

Oozie is a scalable, reliable and extensible system.

Oozie is a server based *Workflow Engine* specialized in running workflow jobs with actions that run Hadoop Map/Reduce and Pig jobs.

Oozie is a Java Web-Application that runs in a Java servlet-container.

For the purposes of Oozie, a workflow is a collection of actions (i.e. Hadoop Map/Reduce jobs, Pig jobs) arranged in a control dependency DAG (Directed Acyclic Graph). “control dependency” from one action to another means that the second action can’t run until the first action has completed.

Oozie workflows definitions are written in hPDL (a XML Process Definition Language similar to JBOSS JBPM jPDL).

Oozie workflow actions start jobs in remote systems (i.e. Hadoop, Pig). Upon action completion, the remote systems callback Oozie to notify the action completion, at this point Oozie proceeds to the next action in the workflow.

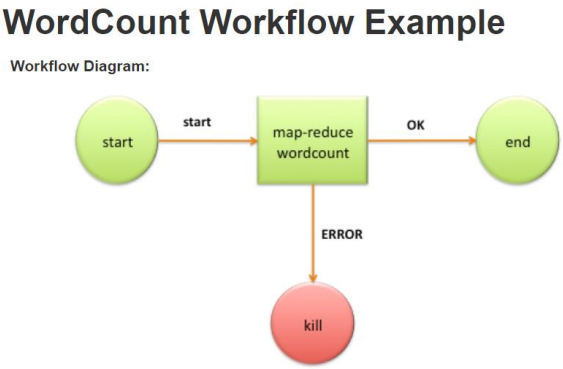
Oozie uses a custom SecurityManager inside it’s launcher to catch exit() calls from the user code. Make sure to delegate checkExit() calls to Oozie’s SecurityManager if the user code uses its own SecurityManager. The Launcher also grants java.security.AllPermission by default to the user code.

Oozie workflows contain control flow nodes and action nodes.

Control flow nodes define the beginning and the end of a workflow (`start`, `end` and `fail` nodes) and provide a mechanism to control the workflow execution path (`decision`, `fork` and `join` nodes).

Action nodes are the mechanism by which a workflow triggers the execution of a computation/processing task. Oozie provides support for different types of actions: Hadoop map-reduce, Hadoop file system, Pig, SSH, HTTP, eMail and Oozie sub-workflow. Oozie can be extended to support additional type of actions.

Oozie workflows can be parameterized (using variables like `${inputDir}` within the workflow definition). When submitting a workflow job values for the parameters must be provided. If properly parameterized (i.e. using different output directories) several identical workflow jobs can concurrently.



Apache Sqoop

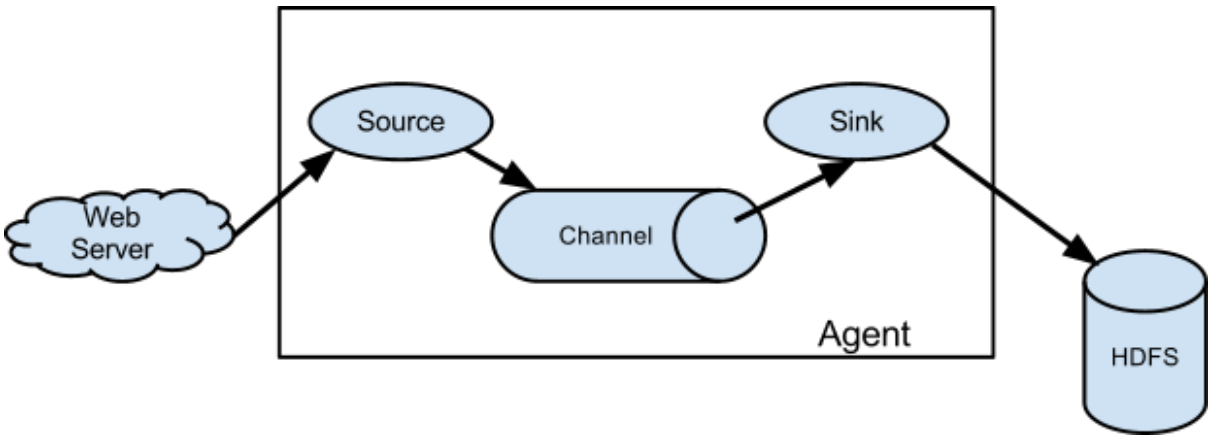
Apache Sqoop – Hadoop The Definitive Guide by Tom White , Chapter 15 -Pages 401 to 407

Apache Flume

Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data. It has a simple and flexible architecture based on streaming data flows. It is robust and fault tolerant with tunable reliability mechanisms and many failover and recovery mechanisms. It uses a simple extensible data model that allows for online analytic application.

Data flow model

A Flume event is defined as a unit of data flow having a byte payload and an optional set of string attributes. A Flume agent is a (JVM) process that hosts the components through which events flow from an external source to the next destination (hop).



A Flume source consumes events delivered to it by an external source like a web server. The external source sends events to Flume in a format that is recognized by the target Flume source. For example, an Avro Flume source can be used to receive Avro events from Avro clients or other Flume agents in the flow that send events from an Avro sink. A similar flow can be defined using a Thrift Flume Source to receive events from a Thrift Sink or a Flume Thrift Rpc Client or Thrift clients written in any language generated from the Flume thrift protocol.When a Flume source receives an event, it stores it into one or more channels. The channel is a passive store that keeps the event until it's consumed by a Flume sink. The file channel is one example – it is backed by the local filesystem. The sink removes the event from the channel and puts it into an external repository like HDFS (via Flume HDFS sink) or forwards it to the Flume source of the next Flume agent (next hop) in the flow. The source and sink within the given agent run asynchronously with the events staged in the channel.

Complex flows

Flume allows a user to build multi-hop flows where events travel through multiple agents before reaching the final destination. It also allows fan-in and fan-out flows, contextual routing and backup routes (fail-over) for failed hops.

Reliability

The events are staged in a channel on each agent. The events are then delivered to the next agent or terminal repository (like HDFS) in the flow. The events are removed from a channel only after they are stored in the channel of next agent or in the terminal repository. This is a how the single-hop message delivery semantics in Flume provide end-to-end reliability of the flow.

Flume uses a transactional approach to guarantee the reliable delivery of the events. The sources and sinks encapsulate in a transaction the storage/retrieval, respectively, of the events placed in or provided by a transaction provided by the channel. This ensures that the set of events are reliably passed from point to point in the flow. In the case of a multi-hop flow, the sink from the previous hop and the source from the next hop both have their transactions running to ensure that the data is safely stored in the channel of the next hop.

Recoverability

The events are staged in the channel, which manages recovery from failure. Flume supports a durable file channel which is backed by the local file system. There's also a memory channel which simply stores the events in an in-memory queue, which is faster but any events still left in the memory channel when an agent process dies can't be recovered.

Flume's *KafkaChannel* uses Apache Kafka to stage events. Using a replicated Kafka topic as a channel helps avoiding event loss in case of a disk failure.

Apache Flume – Hadoop The Definitive Guide by Tom White , Chapter 14 -Pages 381 to 390

Cassandra

What is Apache Cassandra?

- Cassandra is a NoSQL database which is distributed and scalable. It is provided by Apache.
- Apache Cassandra is highly scalable, high performance, distributed NoSQL database. Cassandra is designed to handle huge amount of data across many commodity servers, providing high availability without a single point of failure.
- It is an open source, distributed and decentralized/distributed storage system (database).

History of Cassandra

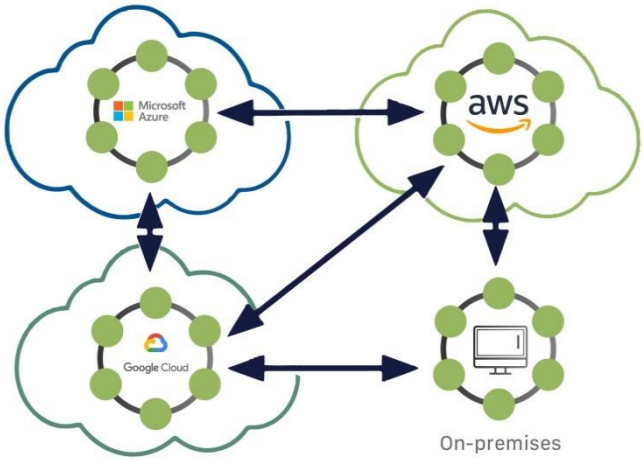
- Cassandra was initially developed at Facebook by two Indians Avinash Lakshman (one of the authors of Amazon's Dynamo) and Prashant Malik.

It was developed to power the Facebook inbox search feature. It was open sourced by Facebook in July 2008.

- It was accepted by Apache Incubator in March 2009.

Important Points of Cassandra

- Cassandra is a column-oriented database.
- Cassandra is scalable, consistent, and fault-tolerant.
- Cassandra is being used by some of the biggest companies like Facebook, Twitter, Cisco, Rackspace, ebay, Twitter, Netflix, and more.
- Cassandra is deployment agnostic. It doesn't care where you put it – on prem, a cloud provider, multiple cloud providers.



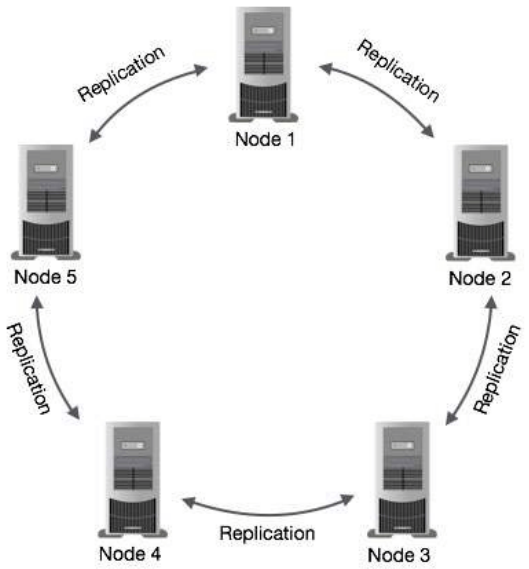
Cassandra Architecture

Cassandra was designed to handle big data workloads across multiple nodes without a single point of failure. It has a peer-to-peer distributed system across its nodes, and data is distributed among all the nodes in a cluster.

- In Cassandra, each node is independent and at the same time interconnected to other nodes. All the nodes in a cluster play the same role.
- Every node in a cluster can accept read and write requests, regardless of where the data is actually located in the cluster.
- In the case of failure of one node, Read/Write requests can be served from other nodes in the network.

Data Replication in Cassandra

- In Cassandra, nodes in a cluster act as replicas for a given piece of data.
- Cassandra will return the most recent value to the client. After returning the most recent value, Cassandra performs a read repair in the background to update the stale values.



Components of Cassandra

- **Node:** A Cassandra node is a place where data is stored.
- **Data center:** Data center is a collection of related nodes.
- **Cluster:** A cluster is a component which contains one or more data centers.
- **Commit log:** In Cassandra, the commit log is a crash-recovery mechanism. Every write operation is written to the commit log.
- **Mem-table:** A mem-table is a memory-resident data structure. After commit log, the data will be written to the mem-table. Sometimes, for a single-column family, there will be multiple mem-tables.
- **SSTable:** It is a disk file to which the data is flushed from the mem-table when its contents reach a threshold value.
- **Bloom filter:** These are nothing but quick, nondeterministic, algorithms for testing whether an element is a member of a set. It is a special kind of cache. Bloom filters are accessed after every query.

NoSQL is an umbrella term to describe any alternative system to traditional SQL databases. NoSQL databases are all quite different from SQL databases. They use a data model that has a different structure than the traditional row-and-column table model used with relational database management systems (RDBMS).But NoSQL databases are all quite different from each other, as well. There are four main types of NoSQL databases. NoSQL ("non SQL" or "not only SQL") databases were developed in the late 2000s with a focus on scaling, fast queries, allowing for frequent application changes, and making programming simpler for developers. Relational databases accessed with SQL (Structured Query Language) were developed in the 1970s with a focus on reducing data duplication as storage was much more costly than developer time. SQL databases tend to have rigid, complex, tabular schemas and typically require expensive vertical scaling.

What NoSQL databases have in common

As explained in When to Use NoSQL Databases, NoSQL databases were developed during the internet era in response to the inability of SQL databases to address the needs of web-scale applications that handled huge amounts of data and traffic.

Companies are finding that they can apply NoSQL technology to a growing list of use cases while saving money in comparison to operating a relational database. NoSQL databases invariably incorporate a flexible schema model and are designed to scale horizontally across many servers, which makes them appealing for large data volumes or application loads that exceed the capacity of a single server.

The popularity of NoSQL has been driven by the following reasons:

- The pace of development with NoSQL databases can be much faster than with a SQL database
- The structure of many different forms of data is more easily handled and evolved with a NoSQL database
- The amount of data in many applications cannot be served affordably by a SQL database
- The scale of traffic and need for zero downtime cannot be handled by SQL
- New application paradigms can be more easily supported

NoSQL databases deliver these benefits in different ways.

Understanding differences in the four types of NoSQL databases

Here are the four main types of NoSQL databases:

- Document databases
- Key-value stores
- Column-oriented databases
- Graph databases

Document databases

A document database stores data in JSON, BSON, or XML documents (not Word documents or Google Docs, of course). In a document database, documents can be nested. Particular elements can be indexed for faster querying.

Documents can be stored and retrieved in a form that is much closer to the data objects used in applications, which means less translation is required to use the data in an application. SQL data must often be assembled and disassembled when moving back and forth between applications and storage.

Document databases are popular with developers because they have the flexibility to rework their document structures as needed to suit their application, shaping their data structures as their application requirements change over time. This flexibility speeds development because, in effect, data becomes like code and is under the control of developers. In SQL databases, intervention by database administrators may be required to change the structure of a database.

The most widely adopted document databases are usually implemented with a scale-out architecture, providing a clear path to scalability of both data volumes and traffic.

Use cases include ecommerce platforms, trading platforms, and mobile app development across industries.

Comparing MongoDB vs. PostgreSQL offers a detailed analysis of MongoDB, the leading NoSQL database, and PostgreSQL, one of the most popular SQL databases.

Key-value stores

The simplest type of NoSQL database is a key-value store. Every data element in the database is stored as a key value pair consisting of an attribute name (or "key") and a value. In a sense, a key-value store is like a relational database with only two columns: the key or attribute name (such as "state") and the value (such as "Alaska").

Use cases include shopping carts, user preferences, and user profiles.

Column-oriented databases

While a relational database stores data in rows and reads data row by row, a column store is organized as a set of columns. This means that when you want to run analytics on a small number of columns, you can read those columns directly without consuming memory with the unwanted data. Columns are often of the same type and benefit from more efficient compression, making reads even faster. Columnar databases can quickly aggregate the value of a given column (adding up the total sales for the year, for example). Use cases include analytics.

Unfortunately, there is no free lunch, which means that while columnar databases are great for analytics, the way in which they write data makes it very difficult for them to be strongly consistent as writes of all the columns require multiple write events on disk. Relational databases don't suffer from this problem as row data is written contiguously to disk.

Graph databases

A graph database focuses on the relationship between data elements. Each element is stored as a node (such as a person in a social media graph). The connections between elements are called links or relationships. In a graph database, connections are first-class elements of the database, stored directly. In relational databases, links are implied, using data to express the relationships.

A graph database is optimized to capture and search the connections between data elements, overcoming the overhead associated with JOINing multiple tables in SQL.

Very few real-world business systems can survive solely on graph queries. As a result graph databases are usually run alongside other more traditional databases.

Use cases include fraud detection, social networks, and knowledge graphs.

As you can see, despite a common umbrella, NoSQL databases are diverse in their data structures and their applications.

Differences between SQL and NoSQL

The table below summarizes the main differences between SQL and NoSQL databases.

	SQL Databases	NoSQL Databases
Data Storage Model	Tables with fixed rows and columns	Document: JSON documents, Key-value: key-value pairs, Wide-column: tables with rows and dynamic columns, Graph: nodes and edges
Development History	Developed in the 1970s with a focus on reducing data duplication	Developed in the late 2000s with a focus on scaling and allowing for rapid application change driven by agile and DevOps practices.
Examples	Oracle, MySQL, Microsoft SQL Server, and PostgreSQL	Document: MongoDB and CouchDB, Key-value: Redis and DynamoDB, Wide-column: Cassandra and HBase, Graph: Neo4j and Amazon Neptune

	SQL Databases	NoSQL Databases
Primary Purpose	General purpose	Document: general purpose, Key-value: large amounts of data with simple lookup queries, Wide-column: large amounts of data with predictable query patterns, Graph: analyzing and traversing relationships between connected data
Schemas	Rigid	Flexible
Scaling	Vertical (scale-up with a larger server)	Horizontal (scale-out across commodity servers)
Multi-Record ACID Transactions	Supported	Most do not support multi-record ACID transactions. However, some — like MongoDB — do.
Joins	Typically required	Typically not required
Data to Object Mapping	Requires ORM (object-relational mapping)	Many do not require ORMs. MongoDB documents map directly to data structures in most popular programming languages.

What are the benefits of NoSQL databases?

NoSQL databases offer many benefits over relational databases. NoSQL databases have flexible data models, scale horizontally, have incredibly fast queries, and are easy for developers to work with.

- Flexible data models

  NoSQL databases typically have very flexible schemas. A flexible schema allows you to easily make changes to your database as requirements change. You can iterate quickly and continuously integrate new application features to provide value to your users faster.

- Horizontal scaling

  Most SQL databases require you to scale-up vertically (migrate to a larger, more expensive server) when you exceed the capacity requirements of your current server. Conversely, most NoSQL databases allow you to scale-out horizontally, meaning you can add cheaper commodity servers whenever you need to.

- Fast queries

  Queries in NoSQL databases can be faster than SQL databases. Why? Data in SQL databases is typically normalized, so queries for a single object or entity require you to join data from multiple tables. As your tables grow in size, the joins can become expensive. However, data in NoSQL databases is typically stored in a way that is optimized for queries. The rule of thumb when you use MongoDB is data that is accessed together should be stored together. Queries typically do not require joins, so the queries are very fast.

- Easy for developers

  Some NoSQL databases like MongoDB map their data structures to those of popular programming languages. This mapping allows developers to store their data in the same way that they use it in their application code. While it may seem like a trivial advantage, this mapping can allow developers to write less code, leading to faster development time and fewer bugs.

What are the drawbacks of NoSQL databases?

One of the most frequently cited drawbacks of NoSQL databases is that they don't support ACID (atomicity, consistency, isolation, durability) transactions across multiple documents. With appropriate schema design, single-record atomicity is acceptable for lots of applications. However, there are still many applications that require ACID across multiple records.

To address these use cases, MongoDB added support for multi-document ACID transactions in the 4.0 release, and extended them in 4.2 to span sharded clusters.

Since data models in NoSQL databases are typically optimized for queries and not for reducing data duplication, NoSQL databases can be larger than SQL databases. Storage is currently so cheap that most consider this a minor drawback, and some NoSQL databases also support compression to reduce the storage footprint.

Depending on the NoSQL database type you select, you may not be able to achieve all of your use cases in a single database. For example, graph databases are excellent for analyzing relationships in your data but may not provide what you need for everyday retrieval of the data such as range queries. When selecting a NoSQL database, consider what your use cases will be and if a general purpose database like MongoDB would be a better option.

MongoDB Commands:

```
C:\Program Files\MongoDB\Server\4.2\bin>mongod.exe --dbpath "C:\data"

C:\Program Files\MongoDB\Server\4.2\bin>mongo.exe



use classdb

db

show dbs

db.createCollection("books")

show collections

db.createCollection("players", {capped : true,  size : 6142800, max : 10000 } )

db.movies.insert({"name" : "chamber of secrets"})

show collections

db.books.insert({

... _id : ObjectId("507f191e810c19729de860ea"),

... title: "MongoDB Overview",

... description: "MongoDB is no sql database",

... by: "packt publishing",

... url: "http://www.packtpubl.com",

... tags: ['mongodb', 'database', 'NoSQL'],

... downloads: 100

... })


db.books.insert([

{
```

```
title: "Mastering Apache Storm",
by: "oreily",
url: "http://www.oreily.com",
tags: ['storm', 'streaming', 'engine'],
downloads: 100,
comments:[{
user:"user1",
message: "My first comment",
dateCreated: new Date(2013,11,10,2,35),
like: 0
}]
},
{
title: "Spark Overview",
description: "Spark Introduction Book",
by: "packt publishing",
url: "http://www.packtpubl.com",
tags: ['spark', 'dstreams', 'RDD'],
downloads: 100000
}
])

db.books.find()
db.books.find().pretty()
db.books.findOne()

db.books.find({$and:[{"by":"packt publishing"},{"title": "Spark Overview"}]}).pretty()

db.books.find({$or:[{"by":"packt publishing"},{"title": "Spark Overview"}]}).pretty()

db.books.insert([
```

{"title":"D3JS Overview"},

{"title":"Python Overview"},

{"title":"Lua Overview"}

])

db.books.remove({'title':'MongoDB Overview'})

db.books.update({'title':'Spark Overview'}, {$set:{'title':'New Spark Tutorial'}},{multi:true})

Data Modeling Introduction

The key challenge in data modeling is balancing the needs of the application, the performance characteristics of the database engine, and the data retrieval patterns. When designing data models, always consider the application usage of the data (i.e. queries, updates, and processing of the data) as well as the inherent structure of the data itself.

Flexible Schema

Unlike SQL databases, where you must determine and declare a table's schema before inserting data, MongoDB's collections, by default, do not require their documents to have the same schema. That is:

- The documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.
- To change the structure of the documents in a collection, such as add new fields, remove existing fields, or change the field values to a new type, update the documents to the new structure.

This flexibility facilitates the mapping of documents to an entity or an object. Each document can match the data fields of the represented entity, even if the document has substantial variation from other documents in the collection.

In practice, however, the documents in a collection share a similar structure, and you can enforce document validation rules for a collection during update and insert operations. See Schema Validation for details.

Document Structure

The key decision in designing data models for MongoDB applications revolves around the structure of documents and how the application represents relationships between data. MongoDB allows related data to be embedded within a single document.

Embedded Data

Embedded documents capture relationships between data by storing related data in a single document structure. MongoDB documents make it possible to embed document structures in a field or array within a document. These *denormalized* data models allow applications to retrieve and manipulate related data in a single database operation.

For many use cases in MongoDB, the denormalized data model is optimal.

See Embedded Data Models for the strengths and weaknesses of embedding documents.

References

References store the relationships between data by including links or *references* from one document to another. Applications can resolve these references to access the related data. Broadly, these are *normalized* data models.

click to enlarge

See Normalized Data Models for the strengths and weaknesses of using references.

Atomicity of Write Operations

Single Document Atomicity

In MongoDB, a write operation is atomic on the level of a single document, even if the operation modifies multiple embedded documents *within* a single document.

A denormalized data model with embedded data combines all related data in a single document instead of normalizing across multiple documents and collections. This data model facilitates atomic operations.

For details regarding transactions in MongoDB, see the Transactions page.

Multi-Document Transactions

When a single write operation (e.g. `db.collection.updateMany()`) modifies multiple documents, the modification of each document is atomic, but the operation as a whole is not atomic.

When performing multi-document write operations, whether through a single write operation or multiple write operations, other operations may interleave.

For situations that require atomicity of reads and writes to multiple documents (in a single or multiple collections), MongoDB supports multi-document transactions:

Data in MongoDB has a flexible schema.documents in the same collection. They do not need to have the same set of fields or structure Common fields in a collection's documents may hold different types of data.

Data Model Design

MongoDB provides two types of data models: — Embedded data model and Normalized data model. Based on the requirement, you can use either of the models while preparing your document.

Embedded Data Model

In this model, you can have (embed) all the related data in a single document, it is also known as de-normalized data model.

For example, assume we are getting the details of employees in three different documents namely, Personal_details, Contact and, Address, you can embed all the three documents in a single one as shown below –

```
{
        _id: ,
        Emp_ID: "10025AE336"
        Personal_details:{
                First_Name: "Radhika",
                Last_Name: "Sharma",
                Date_Of_Birth: "1995-09-26"
        },
        Contact: {
                e-mail: "radhika_sharma.123@gmail.com",
                phone: "9848022338"
        },
        Address: {
                city: "Hyderabad",
                Area: "Madapur",
                State: "Telangana"
        }
}
```

Normalized Data Model

In this model, you can refer the sub documents in the original document, using references. For example, you can re-write the above document in the normalized model as:

Employee:

```
{
        _id: <ObjectId101>,
        Emp_ID: "10025AE336"
}
```

Personal_details:

```
{
        _id: <ObjectId102>,
        empDocID: " ObjectId101",
        First_Name: "Radhika",
        Last_Name: "Sharma",
        Date_Of_Birth: "1995-09-26"
}
```

Contact:

```
{
        _id: <ObjectId103>,
        empDocID: " ObjectId101",
        e-mail: "radhika_sharma.123@gmail.com",
```

```
        phone: "9848022338"
}
```

Address:

```
{
        _id: <ObjectId104>,
        empDocID: " ObjectId101",
        city: "Hyderabad",
        Area: "Madapur",
        State: "Telangana"
}
```

Considerations while designing Schema in MongoDB

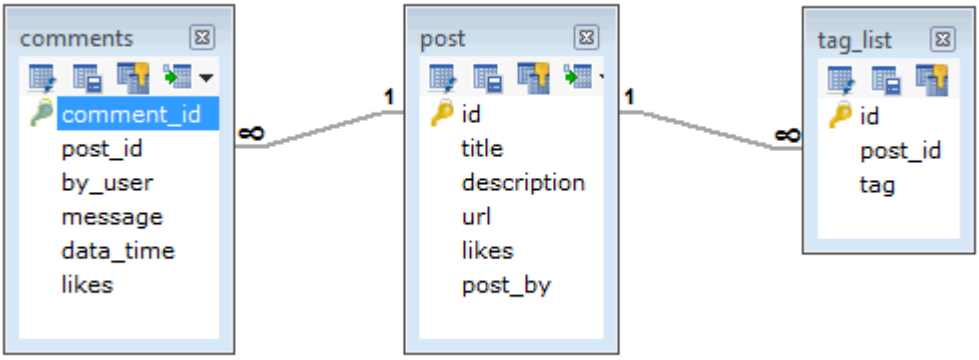
- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

Example

Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.

- Every post has the unique title, description and url.
- Every post can have one or more tags.
- Every post has the name of its publisher and total number of likes.
- Every post has comments given by users along with their name, message, data-time and likes.
- On each post, there can be zero or more comments.

In RDBMS schema, design for above requirements will have minimum three tables.



While in MongoDB schema, design will have one collection post and the following structure –

```
{
  _id: POST_ID
  title: TITLE_OF_POST,
  description: POST_DESCRIPTION,
  by: POST_BY,
  url: URL_OF_POST,
  tags: [TAG1, TAG2, TAG3],
  likes: TOTAL_LIKES,
  comments: [
    {
      user:'COMMENT_BY',
      message: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    },
    {
      user:'COMMENT_BY',
      message: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    }
  ]
}
```

So while showing the data, in RDBMS you need to join three tables and in MongoDB, data will be shown from one collection only.

Data Types

MongoDB supports many datatypes. Some of them are –

- **String** – This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.

- **Integer** – This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** – This type is used to store a boolean (true/ false) value.
- **Double** – This type is used to store floating point values.
- **Min/ Max keys** – This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** – This type is used to store arrays or list or multiple values into one key.
- **Timestamp** – ctimestamp. This can be handy for recording when a document has been modified or added.
- **Object** – This datatype is used for embedded documents.
- **Null** – This type is used to store a Null value.
- **Symbol** – This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date** – This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** – This datatype is used to store the document's ID.
- **Binary data** – This datatype is used to store binary data.
- **Code** – This datatype is used to store JavaScript code into the document.
- **Regular expression** – This datatype is used to store regular expression.