Pyspark

Sunil Kumar Miriyala

Table of Contents

[Environment 3](#_Toc69219400)

[GIT 3](#_Toc69219401)

[IDE (Scala): 3](#_Toc69219402)

[Spark 3](#_Toc69219403)

[Python 3](#_Toc69219404)

[Spark Databricks: 3](#_Toc69219405)

[Spark Standalone: 3](#_Toc69219406)

[Spark 4](#_Toc69219407)

[Overview: 4](#_Toc69219408)

[Features 4](#_Toc69219409)

[Spark 1.X 4](#_Toc69219410)

[Spark 2.X 4](#_Toc69219411)

[Spark 3.X 5](#_Toc69219412)

[Caching: 5](#_Toc69219413)

[Security: 5](#_Toc69219414)

[YARN: 6](#_Toc69219415)

[AES: 6](#_Toc69219416)

[RDD: 6](#_Toc69219417)

[Stack 7](#_Toc69219418)

[Deployment 7](#_Toc69219419)

[Execution Architecture 7](#_Toc69219420)

[Spark Session: 9](#_Toc69219421)

[YARN: 9](#_Toc69219422)

[Driver: 9](#_Toc69219423)

[Executors: 10](#_Toc69219424)

[Streaming: 10](#_Toc69219425)

[DStream: 10](#_Toc69219426)

[Window Operations: 11](#_Toc69219427)

[Direct Streaming 11](#_Toc69219428)

[Structured Streaming: 12](#_Toc69219429)

[RDD (Resilient Distributed Dataset) 13](#_Toc69219430)

[Shuffle 13](#_Toc69219431)

[RDD Cache & Persistence: 13](#_Toc69219432)

[Processing Semantics: 13](#_Toc69219433)

[Spark-SQL: 14](#_Toc69219434)

[Data Formats: 14](#_Toc69219435)

[AVRO 14](#_Toc69219436)

[ORC 14](#_Toc69219437)

[PARQUET 15](#_Toc69219438)

[Kafka 15](#_Toc69219439)

[Overview: 15](#_Toc69219440)

[Topic: 15](#_Toc69219441)

[Partition: 15](#_Toc69219442)

[Retention Period: 15](#_Toc69219443)

[Topic and Partition: 16](#_Toc69219444)

[Offset Positions Per Consumer: 16](#_Toc69219445)

[Kafka Cluster: Multi-Node Kafka Servers. 17](#_Toc69219446)

[Kafka Services in HDP: 17](#_Toc69219447)

[Check: 18](#_Toc69219448)

[Create Kafka Topic: 18](#_Toc69219449)

[Produce Data into Topic: 19](#_Toc69219450)

[Consume Data from Topic: 19](#_Toc69219451)

[Topic Offsets: 19](#_Toc69219452)

[Consume Group Offsets: 20](#_Toc69219453)

# Environment

## GIT

<https://github.com/sunmiri/pyspark>

git clone https://github.com/sunmiri/pyspark.git

## IDE (Scala):

* PyCharm
* [https://www.jetbrains.com/pycharm/download/](https://www.jetbrains.com/pycharm/download/#section=windows)

## Spark

* <https://spark.apache.org/downloads.html>
* Version:
  + Spark Release: 3.0.0 or later
  + Hadoop: 3.2

## Python

* Windows:
  + PyCharm/Visual Studio Code/Spyder
  + Linux Sub System
    - <https://docs.microsoft.com/en-us/windows/wsl/install-win10>
* Mac
  + Visual Studio Code/PyCharm/Spyder
  + Python3
  + <https://pypi.org/project/pyspark/>
  + <https://spark.apache.org/docs/latest/api/python/index.html>
* Create New Virtual Environment
* <https://docs.python.org/3/library/venv.html>

### Spark Databricks:

* https://community.cloud.databricks.com/

### Spark Standalone:

* Windows Instructions:
  + https://phoenixnap.com/kb/install-spark-on-windows-10
* IDE: <https://www.jetbrains.com/pycharm/>
* Spark:
  + https://spark.apache.org/docs/latest/spark-standalone.html
  + <https://archive.apache.org/dist/spark/>
  + <https://ftp.wayne.edu/apache/spark/spark-3.0.1/spark-3.0.1-bin-hadoop3.2.tgz>
* Python Interpreter
  + Graphical user interface, application, Teams

    Description automatically generated
* Download WinUtils:
  + <https://github.com/cdarlint/winutils>

# Spark

## Overview:

A high-performance large-scale data processing analytical engine that supports both batch and streaming data. It uses DAG schedulers, query optimizers and physical execution engines to achieve this performance. Opensource analytical processing engine for large scale powerful distributed data processing and machine learning applications

It supports running on Hadoop, Kubernetes, Standalone and Cloud.

## Features

* In-memory computation
* Distributed processing using parallelize (RDD)
* Support for Standalone, YARN, Mesos, etc.
* Fault-tolerant
* Immutable
* Lazy evaluation
* Cache & persistence
* DataFrame/Dataset
* Supports ANSI SQL.
* Supports reading data from HDFS, S3, GCS, Azure Blob, Databricks File System and more.
* Supports both Streaming and Batch executions.
* Supports ML and Graph libraries.

## Spark 1.X

With Spark 1.x, important interface was RDD (Resilient Distributed Dataset).

## Spark 2.X

Starting Spark 2.0, RDD’s are replaced with Datasets. RDD is still supported, however Dataset provides much easier to work with and better performant compared to RDD.

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.

* Native CSV data source, based on Databricks’ [spark-csv module](https://github.com/databricks/spark-csv)
* Off-heap memory management for both caching and runtime execution
* Hive style bucketing support
* Approximate summary statistics using sketches, including approximate quantile, Bloom filter, and count-min sketch.

## Spark 3.X

* [Project Hydrogen] Accelerator-aware Scheduler ([SPARK-24615](https://issues.apache.org/jira/browse/SPARK-24615))
* Adaptive Query Execution ([SPARK-31412](https://issues.apache.org/jira/browse/SPARK-31412))
* Dynamic Partition Pruning ([SPARK-11150](https://issues.apache.org/jira/browse/SPARK-11150))
* Redesigned pandas UDF API with type hints ([SPARK-28264](https://issues.apache.org/jira/browse/SPARK-28264))
* Structured Streaming UI ([SPARK-29543](https://issues.apache.org/jira/browse/SPARK-29543))
* Catalog plugin API ([SPARK-31121](https://issues.apache.org/jira/browse/SPARK-31121))
* Java 11 support ([SPARK-24417](https://issues.apache.org/jira/browse/SPARK-24417))
* Hadoop 3 support ([SPARK-23534](https://issues.apache.org/jira/browse/SPARK-23534))
* Better ANSI SQL compatibility

## Caching:

Spark’s cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it. Spark also supports pulling data sets into a cluster-wide in-memory cache. This is very useful when data is accessed repeatedly. By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also *persist* an RDD in memory using the persist (or cache) method, in which case Spark will keep the elements around on the cluster for much faster access the next time you query it. There is also support for persisting RDDs on disk or replicated across multiple nodes. When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset.

When we use cache() method, all the RDD stores in-memory. When RDD stores the value in memory, the data that does not fit in memory is either recalculated or the excess data is sent to disk. Whenever we want RDD, it can be extracted without going to disk. This reduces the space-time complexity and overhead of disk storage.

When we use persist() method the RDDs can also be stored in-memory, we can use it across parallel operations. The difference between cache() and persist() is that using cache() the default storage level is MEMORY\_ONLY while using persist() we can use various storage levels.

The available storage levels in Python *include MEMORY\_ONLY, MEMORY\_ONLY\_2, MEMORY\_AND\_DISK, MEMORY\_AND\_DISK\_2, DISK\_ONLY, and DISK\_ONLY\_2*

## Security:

Spark Security is OFF by default. Kerberos is commonly enabled in the spark deployments.

spark.authenticate: Spark currently supports authentication for RPC channels using a shared secret.

spark.authenticate.secret: Secret config option. Alternatively supports secret file config.

### YARN:

When submitting spark jobs on Yarn, Spark will automatically handle generating and distributing the shared secret. Each submit will have a unique shared secret.

### AES:

Spark support AES based encryption for RPC connections. SASL based encryption is also supported. AES uses apache common crypto.

### RDD:

A collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. Gets created when we load a file from HDFS file system. Support caching and fault-tolerant. Spark tries to set the number of partitions automatically based on your cluster.

Spark creates one partition for each block of the file (blocks being 128MB by default in HDFS), but you can also ask for a higher number of partitions by passing a larger value.

RDD Operations

* *transformations*, which create a new dataset from an existing one.
  + map is a transformation that passes each dataset element through a function and returns a new RDD representing the results
* *actions*, which return a value to the driver program after running a computation on the dataset.
  + reduce is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program.

A fundamental data structure of Spark and it is the primary data abstraction in Apache Spark and the Spark Core.

RDDs are fault-tolerant, immutable distributed collections of objects, which means once you create an RDD you cannot change it.

Each dataset in RDD is divided into logical partitions, which can be computed on different nodes of the cluster.

* In-Memory Processing
* Immutability
* Fault Tolerance
* Lazy Evolution
* Partitioning
* Parallelize

Shared Variable:

Spark runs a function in parallel as a set of tasks on different nodes, it ships a **copy** of each variable used in the function to each task. Pass-By-Value. If a variable need to be shared across tasks, or between tasks and the driver program, spark supports 2 types of variables.

* *broadcast variables*, which can be used to cache a value in memory on all nodes.
  + Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. Example, a slow changing lookup table.
  + Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost
  + Spark automatically broadcasts the common data needed by tasks within each stage.
* *accumulators*, which are variables that are only “added” to, such as counters and sums.
  + Accumulators are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel.
  + Implement counters.

Modules:

* Spark Core
  + Task dispatching, scheduling, I/O etc
* Spark Session
  + Entry point
  + Used to create RDD, Dataset and DataFrame.
* Spark Context
* Spark SQL
* Spark Streaming
* Spark MLlib
* Spark GraphX

Initialize:

* Create SparkConf
  + Information about your application.
  + Configuration for a Spark application. Used to set various Spark parameters as key-value pairs.
  + Loads properties with spark.\* defined in the property file
* Create SparkContext
  + Tells spark on how to access the cluster.
  + Main entry point for Spark functionality. A SparkContext represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.
  + Only one SparkContext should be active per JVM. You must stop() the active SparkContext before creating a new one.
* appName: name of your application. Must be unique.
* master: local, yarn, mesos etc.

## Stack

A picture containing diagram

Description automatically generated

## Deployment

* Standalone
* Hadoop YARN

## Execution Architecture

Master – Slave Architecture.

Master = Driver

Slave = Worker



* Cluster Managers:
  + Allocates resources for the application.
  + Local/Standalone
  + YARN
  + Mesos
  + Kubernetes(K8)
* Executors:
  + Processes that run the actual computations and data.
  + Spark breaks up the processing of RDD operations into tasks, each of which is executed by an executor.
* Driver programs create SparkContext.
* Sends application code to the executors.
* SparkContext send tasks to the executors to run
* Isolates one application from another sharing the same set of executors. When application is created, context, it gets a set of executor processes. It stays for length of the execution and runs tasks in multiple threads.
* Driver program controls schedule of tasks on the cluster. It should coexist with the worker for better efficiency.
* Spark-Submit
* Spark breaks up the processing of RDD operations into tasks, each of which is executed by an executor
* Spark computes the task’s **closure,** variables and methods which must be visible for the executor to perform its computations on the RDD (in this case foreach()).
* This closure is serialized and sent to each executor.
* The variables within the closure sent to each executor are now copies and thus, when **counter** is referenced within the foreach function, it’s no longer the **counter** on the driver node (counter on driver is not visible to executors)
  + Consider using Accumulators in Spark. They are used specifically to provide a mechanism for safely updating a variable when execution is split up across worker nodes in a cluster.
* To print all elements on the driver, one can use the collect() method to first bring the RDD to the driver node thus: rdd.collect().foreach(println).
  + This can cause the driver to run out of memory, though, because collect() fetches the entire RDD to a single machine.
* Shuffle:
  + Distributed “shuffle” operations, such as grouping or aggregating the elements by a key.
  + The shuffle is Spark’s mechanism for re-distributing data so that it’s grouped differently across partitions.
  + This typically involves copying data across executors and machines, making the shuffle a complex and costly operation.
  + The Shuffle is an expensive operation since it involves disk I/O, data serialization, and network I/O.
  + Spark generates sets of tasks - *map* tasks to organize the data, and a set of *reduce* tasks to aggregate it.
  + Can consume significant amounts of heap memory since they employ in-memory data structures to organize records before or after transferring them
  + A single task will operate on a single partition - thus, to organize all the data for a single reduceByKey reduce task to execute, Spark needs to perform an all-to-all operation. It must read from all partitions to find all the values for all keys, and then bring together values across partitions to compute the final result for each key - this is called the **shuffle**.
    - Repartition:
    - Coalesce:
    - groupByKey
    - reduceByKey
    - join
    - cogroup.
* The Spark RDD API also exposes asynchronous versions of some actions, like foreachAsync for foreach, which immediately return a FutureAction to the caller instead of blocking on completion of the action

## 

## Spark Session:

* SparkSession in Spark 2.0 provides builtin support for Hive features including the ability to write queries using HiveQL, access to Hive UDFs, and the ability to read data from Hive tables

## YARN:

* Acts as resource allocator and Cluster Manager.
* Alternatives: Kubernetes, Apache Mesos.
* YARN-Cluster: Spark driver runs inside an application master process which is managed by yarn on the cluster. Client is released once app is initialized.
* YARN-Client: Drives runs in the client process and the application master is only used for requesting resources from yarn.
* YARN UI: For application monitoring and debugging performance issues:
* <https://databricks.com/blog/2015/07/08/new-visualizations-for-understanding-apache-spark-streaming-applications.html>

## Driver:

* Runs main function and executes various *parallel operations* on a cluster.
* Creates various parallel operations to perform on executors
* All aggregations runs on driver nodes like: collect(), take()

## Executors:

* All Parallel executions happens on executors
* Execution unit inside each function runs here.

## Streaming:

* An extension that supports scalable, high-throughput, fault-tolerant stream processing engine.





## DStream:

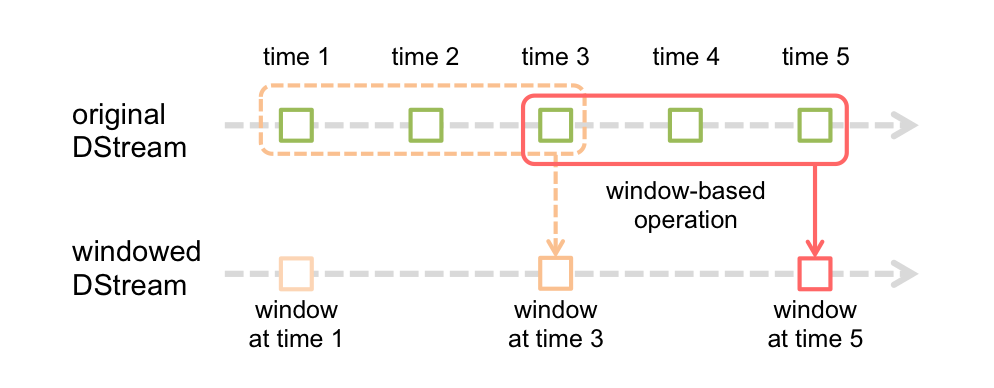
* Discrete Streams, a continuous stream of data. Also referred as sequence of RDDs.

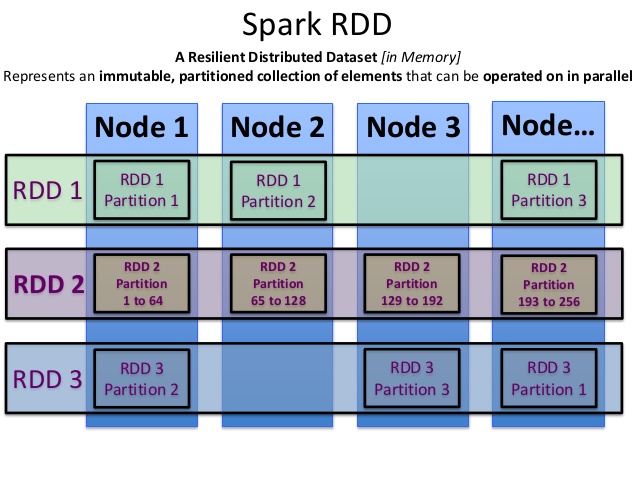


* Input DStreams are DStreams representing the stream of input data received from streaming sources like Kafka.
* Each RDD in a DStream contains data from a certain interval
* Any operation applied on a DStream translates to operations on the underlying RDDs, like split etc.
* Each RDD pushed into the queue will be treated as a batch of data in the DStream, and processed like a stream.
* Input Sources 🡪 DStream 🡪 Receiver 🡪 Spark In-Memory

## Window Operations:

* Every time the window slides over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream

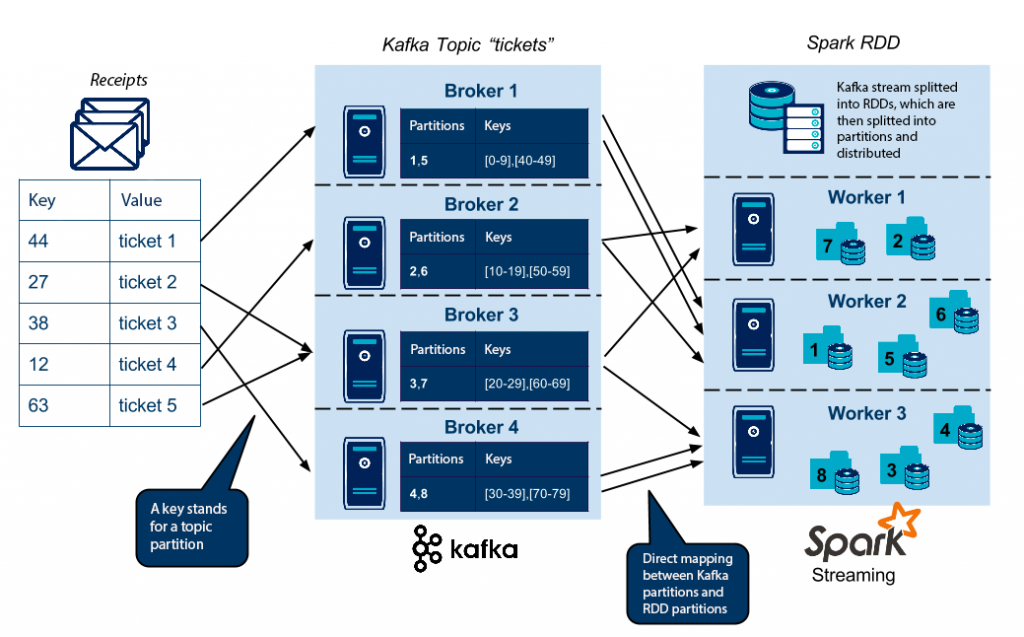




Direct Streaming:

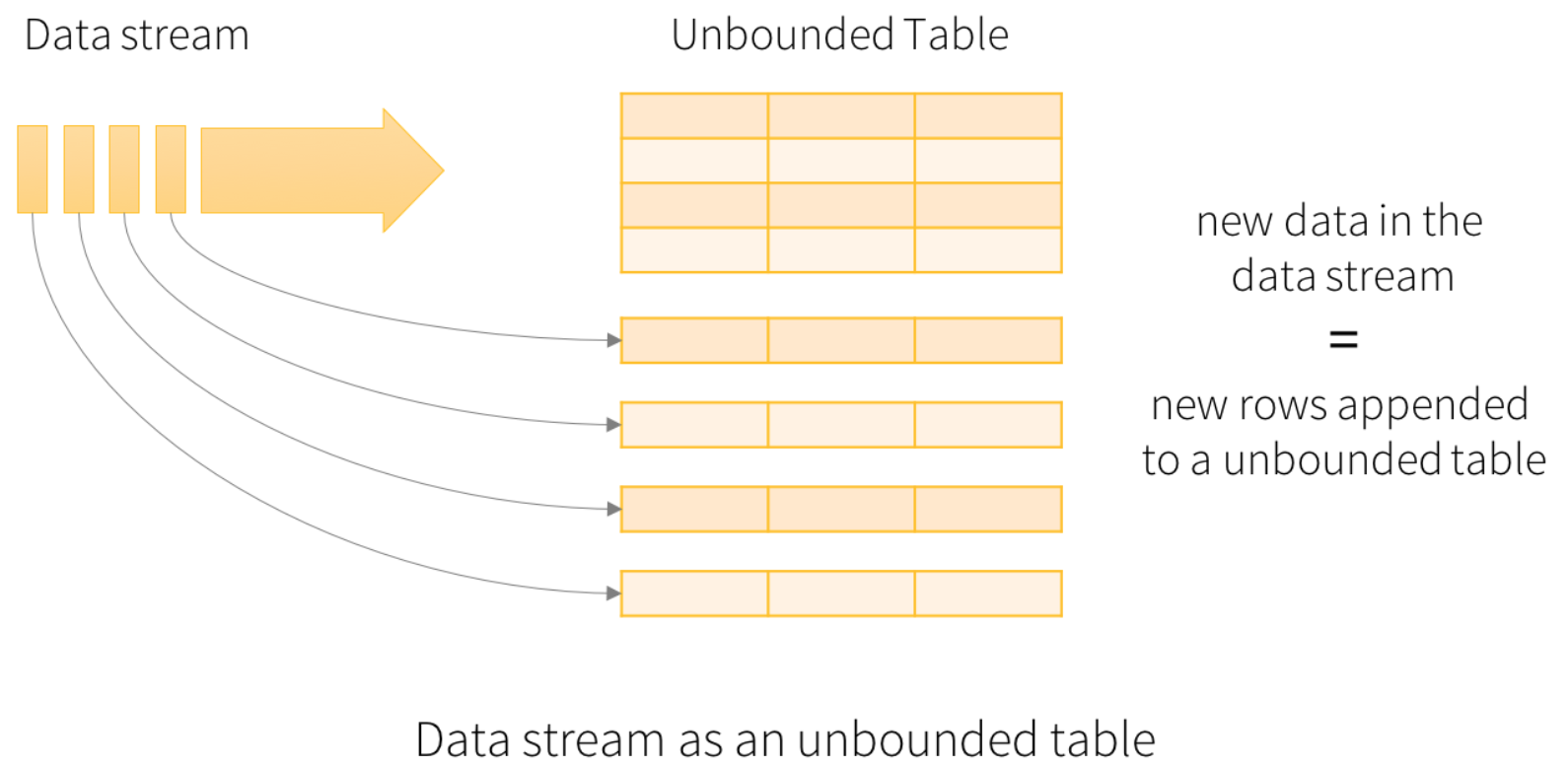
A screenshot of a cell phone

Description automatically generated



## Structured Streaming:

Structured Streaming treats a live data stream as a table that is being continuously appended.



## RDD (Resilient Distributed Dataset)

* An RDD is a fault-tolerant collection of elements that can be operated on in parallel.
* A primary data abstraction in Apache Spark and the Spark Core.
* You can create them parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat
* Each dataset in RDD is divided into logical partitions, which can be computed on different nodes of the cluster
* RDDs are fault-tolerant, immutable distributed collections of objects, which means once you create an RDD you cannot change it
* Supports: Transformations (flatmap, map, filter, ..) and Actions (count, collect, reduce, foreach, ..).

## Shuffle

* Shuffling is a mechanism Spark uses to redistribute the data across different executors and even across machines. Spark shuffling triggers when we perform certain transformation operations like gropByKey(), reduceByKey(), join() on RDD
* Spark shuffle is an expensive operation involving disk I/O, network I/O, data serialization and deserialization.

## RDD Cache & Persistence:

* Using cache() and persist() methods, Spark provides an optimization mechanism to store the intermediate computation of an RDD so they can be reused in subsequent actions.
* When you persist or cache an RDD, each worker node stores its partitioned data in memory or disk and reuses them in other actions on that RDD.
* Spark’s persisted data on nodes are fault-tolerant meaning if any partition is lost, it will automatically be recomputed using the original transformations that created it.
* Cache: MEMORY\_ONLY
* Persistence: MEMORY\_ONLY, MEMORY\_AND\_DISK, …

## Processing Semantics:

* At most once: Each record will be either processed once or not processed at all.
* At least once: Each record will be processed one or more times. This is stronger than at-most once as it ensures that no data will be lost. But there may be duplicates.
* Exactly once: Each record will be processed exactly once - no data will be lost, and no data will be processed multiple times. This is obviously the strongest guarantee of the three

## Spark-SQL:

* Spark SQL is a Spark module for structured data processing.
* Useful for querying structured data using DataFrame API or SQL syntax. Supports filtering, joins, grouping and more. Support HiveQL syntax, UDF’s and direct access to hive data stores.
* Uses internally Cost Based Optimization for executing queries. It uses columnar storage and code generation that assist in running performant queries. It will launch spark jobs and use the power of cluster and scale to high volume data queries.
* A Dataset is a distributed collection of data.
* Python does not have the support for the Dataset API, however Python native supports lot of Dataset APIs.
* A DataFrame is a *Dataset* organized into named columns.
  + Conceptually equivalent to a table in a relational database or a data frame in R/Python.
* Global Temp View:
  + Temporary view that is shared among all sessions and keep alive until the Spark application terminates, you can create a global temporary view.
* Local Temp View.
* Datasets are similar to RDDs, however, instead of using Java serialization or Kryo they use a specialized [Encoder](https://spark.apache.org/docs/latest/api/scala/org/apache/spark/sql/Encoder.html) to serialize the objects for processing or transmitting over the network
* Spark SQL can convert an RDD of Row objects to a DataFrame, inferring the datatypes.

A picture containing drawing, woman

Description automatically generated

## Data Formats:

### AVRO

* Avro, as a component, supports a rich set of primitive data types including: numeric, binary data and strings; and a number of complex types including arrays, maps, enumerations and records. A sort order can also be defined for the data.

### ORC

* Fastest & Smallest columnar storage system.
* Supports ACID Transactions and snapshot isolations.
* Supports built-in indexes
* Support for Hive and other storage systems.
* Native support in Spark, Hadoop MR and other applications.

### PARQUET

* Another columnar storage system.
* Compressed and Encoded.
* Supports complex nested data structures
* Record shredding
* Row group: A logical horizontal partitioning of the data into rows. There is no physical structure that is guaranteed for a row group. A row group consists of a column chunk for each column in the dataset.
* Column chunk: A chunk of the data for a particular column. These live in a particular row group and is guaranteed to be contiguous in the file.
* Page: Column chunks are divided up into pages. A page is conceptually an indivisible unit (in terms of compression and encoding). There can be multiple page types which is interleaved in a column chunk.

# Kafka

### Overview:

A streaming platform has three key capabilities:

* Publish and subscribe to streams of records, like message queues.
* Store streams of records in a fault-tolerant durable way.
* Process streams of records as they occur

Kafka stores streams of records in categories called topics. Each record consists of a key, a value, and a timestamp.

### Topic:

A topic is a category or feed name to which records are published. Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it.

### Partition:

Each partition is an ordered, immutable sequence of records that is continually appended. Each record in the partitions are each assigned a sequential id number called the offset that uniquely identifies each record within the partition.

### Retention Period:

If the retention policy is set to two days, then for the two days after a record is published, it is available for consumption, after which it will be discarded to free up space.

### Topic and Partition:

A screenshot of a cell phone

Description automatically generated

### Offset Positions Per Consumer:

A picture containing bird

Description automatically generated

### Kafka Cluster: Multi-Node Kafka Servers.

A picture containing drawing, table

Description automatically generated

A picture containing bird

Description automatically generated

### Kafka Services in HDP:

Wait for some time for all the services to start. Check the status here:

A screenshot of a social media post

Description automatically generated

Started Fine:

A screenshot of a cell phone

Description automatically generated

Ambari: <http://192.168.5.137:8080/#/main/dashboard/metrics>

### Check:

* Cluster is up
* Zookeeper is up
* Bootstrap server is up
* Kafka service is up

### Create Kafka Topic:

#### Sandbox:

> /usr/hdp/3.0.1.0-187/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 2 --topic tp\_in\_profileupdat

Es

#### GCP:

>sh /usr/hdp/3.1.4.0-315/kafka/bin/kafka-topics.sh --create --zookeeper instance-1.us-east1-b.c.wired-effort-272101.internal:2181 --replication-factor 1 --partitions 2 --topic tp\_in\_profileupdates

Created topic "tp\_in\_profileupdates".

>sh /usr/hdp/3.1.4.0-315/kafka/bin/kafka-topics.sh --create --zookeeper instance-1.us-east1-b.c.wired-effort-272101.internal:2181 --replication-factor 1 --partitions 2 --topic SourceTopic

Created topic "SourceTopic".

>sh /usr/hdp/3.1.4.0-315/kafka/bin/kafka-topics.sh --create --zookeeper instance-1.us-east1-b.c.wired-effort-272101.internal:2181 --replication-factor 1 --partitions 2 --topic SinkTopic

Created topic "SinkTopic".

Partitions: 2

Replication Factor: 1

#### List Topics:

Sandbox

>/usr/hdp/3.0.1.0-187/kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181

#### Describe Topic:

Sandbox

>/usr/hdp/3.0.1.0-187/kafka/bin/kafka-topics.sh --zookeeper localhost:2181 --describe --topic tp\_in\_profileupdates

GCP Cluster:

>/usr/hdp/3.1.4.0-315/kafka/bin/kafka-topics.sh --zookeeper instance-1.us-east1-b.c.wired-effort-272101.internal:2181 --describe --topic tp\_in\_profileupdates

Topic:tp\_in\_profileupdates PartitionCount:2 ReplicationFactor:1 Configs:

Topic: tp\_in\_profileupdates Partition: 0 Leader: 1001 Replicas: 1001 Isr: 1001

Topic: tp\_in\_profileupdates Partition: 1 Leader: 1001 Replicas: 1001 Isr: 1001

### Produce Data into Topic:

Sandbox:

>/usr/hdp/3.0.1.0-187/kafka/bin/kafka-console-producer.sh --broker-list sandbox-hdp.hortonworks.com:6667 --topic tp\_in\_profileupdates

>{"user\_fname":"sunil", "user\_lname":"miriyala", "user\_id":"sunil.miriyala", "updated\_at":"2020-03-22 14:15:16.000", "modified\_fields":[{"addr

ess":"new value"}]}

>

GCP Cluster:

>cd /usr/hdp/3.1.4.0-315/kafka/bin/

>/usr/hdp/3.1.4.0-315/kafka/bin//kafka-console-producer.sh --broker-list localhost:6667 --topic tp\_in\_profileupdates

### Consume Data from Topic:

Using Bootstrap:

>/usr/hdp/3.0.1.0-187/kafka/bin/kafka-console-consumer.sh --bootstrap-server sandbox-hdp.hortonworks.com:6667 --topic tp\_in\_profileupdates --group tp\_in\_userprof\_grp

{"user\_fname":"sunil", "user\_lname":"miriyala", "user\_id":"sunil.miriyala", "updated\_at":"2020-03-22 14:15:16.000", "modified\_fields":[{"addre

ss":"new value"}]}

### Topic Offsets:

>/usr/hdp/3.0.1.0-187/kafka/bin/kafka-topics.sh --zookeeper localhost:2181 --describe --topic tp\_in\_profileupdates

Topic:tp\_in\_profileupdates PartitionCount:2 ReplicationFactor:1 Configs:

Topic: tp\_in\_profileupdates Partition: 0 Leader: 1001 Replicas: 1001 Isr: 1001

Topic: tp\_in\_profileupdates Partition: 1 Leader: 1001 Replicas: 1001 Isr: 1001

### Consume Group Offsets:

>/usr/hdp/3.0.1.0-187/kafka/bin/kafka-consumer-groups.sh --bootstrap-server sandbox-hdp.hortonworks.com:6667 --describe --group tp\_in\_userprof\_grp

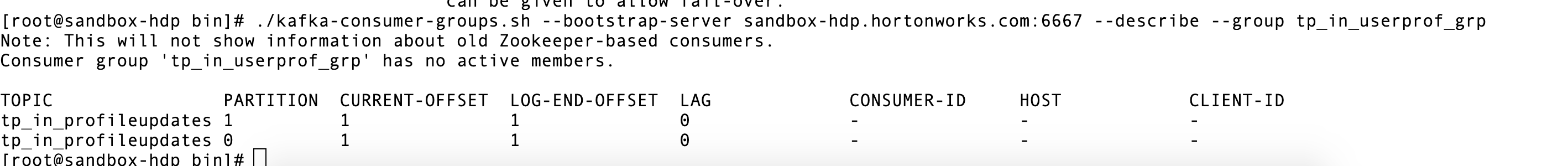
GCP Cluster:

>/usr/hdp/3.0.1.0-187/kafka/bin/kafka-consumer-groups.sh --bootstrap-server instance-1.us-east1-b.c.wired-effort-272101.internal:6667 --describe –group nptest1\_

TOPIC PARTITION CURRENT-OFFSET LOG-END-OFFSET LAG CONSUMER-ID HOST CLIENT-ID

tp\_in\_profileupdates 1 1 1 0 - - -

tp\_in\_profileupdates 0 1 1 0 - - -



A screenshot of a cell phone

Description automatically generated

A close up of a device

Description automatically generated

Diagram

Description automatically generated

Users

|  |  |  |
| --- | --- | --- |
| User\_Id | Name | Display Name |
| 1001 | Sunil | Sunil Miriyala |
| 2001 | Manish | Manish K |

User Role

|  |  |  |
| --- | --- | --- |
| Role\_Id | Role\_Name | User\_Id |
| 100 | Application Engineer | 1001 |
| 101 | Systems Engineer | 2001 |

User Role Report

|  |  |  |  |
| --- | --- | --- | --- |
| User Name | Role Name |  |  |
| Sunil Miriyala | Applications Engineer |  |  |
| Manish K | Systems Engineer |  |  |