

Copyright (©) @ Kaustuv Kunal

Copyright (©) @ Kaustuv Kunal

Apache Spark & It's Ecosystem

Copyright (©) @ Kaustuv Kunal

Kaustuv Kunal

<https://github.com/kaustuvkunal/Spark-Ecosystem-Tutorial>

Copyright (©) @ Kaustuv Kunal

Copyright (©) @ Kaustuv Kunal



Contents

- Introduction
- Spark Features
- History & Versions
- Spark Architecture
- Cluster Execution
- Spark APIs
 - RDD
 - Shared Variables
 - DataFrames
 - Datasets
- Caching: Persist & Storage Levels
- Optimizers & Plans
- Installing Spark
- Spark Programming
 - Job Submission
 - Job Monitoring
 - Job Scheduling
 - Job Tuning
 - Job Testing
- Spark High Level Libraries
 - PySpark
 - Spark SQL
 - Spark Streaming
 - Structured Streaming
 - Spark MLlib
 - Spark GraphX
- Competitors & References



Spark Introduction

- A unified computing engine & set of libraries for parallel data processing on computer clusters.
- Originally, designed as advancement of MapReduce for multiple parallel operations such as iterative jobs (Machine learning Algorithm) and as interactive data analysis tools.
- Spark libraries are written in Scala language.

KK

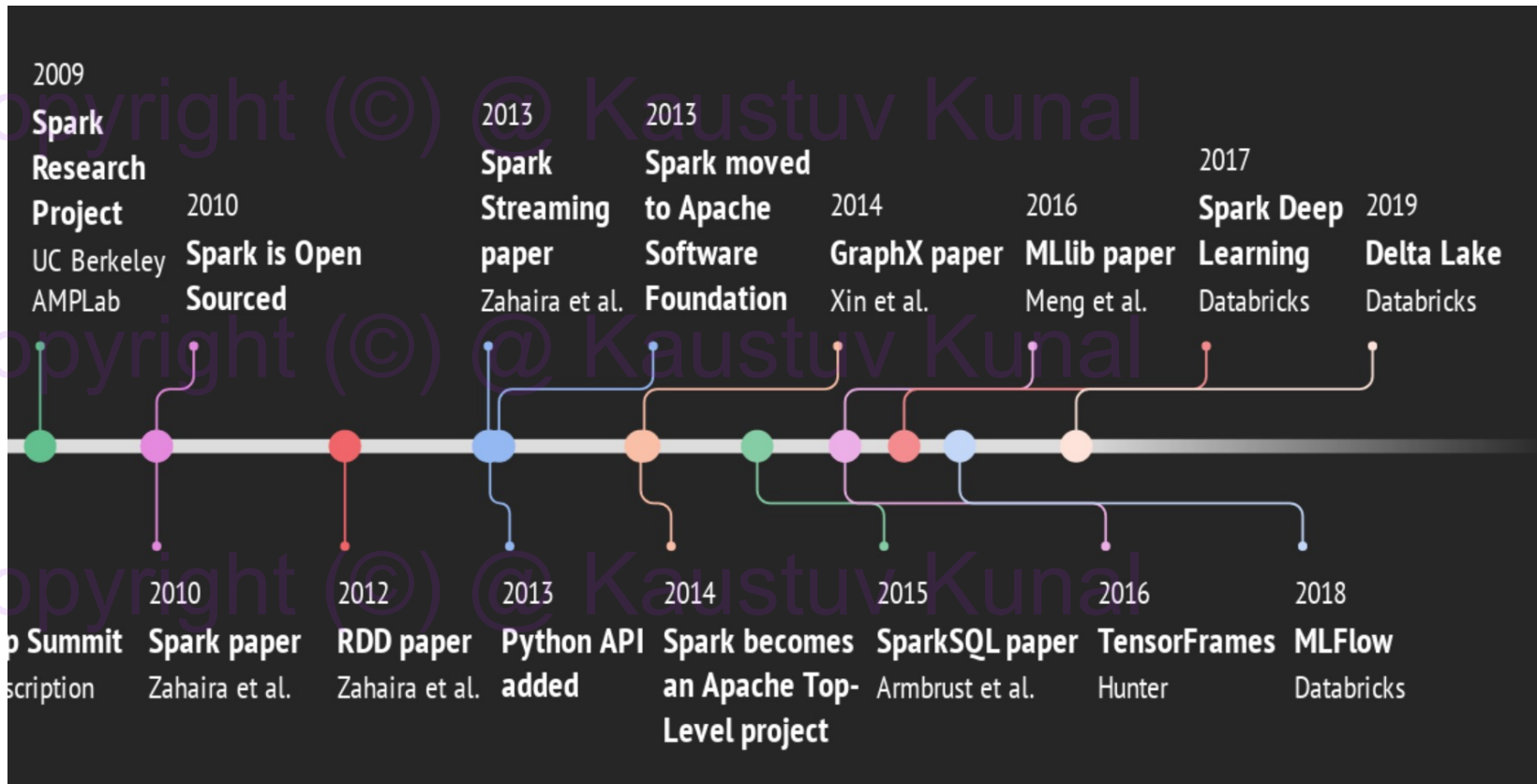


Spark Features

- **Distributed Processing:** Underneath, in spark, all data is RDD(resilient distributed data) a distributed, immutable object.
- **Fault tolerance:** RDD partitions forms DAG (directed acyclic graph). Any lost partitions can be recreated using a mechanisms called lineage.
- **In-memory cluster computing:** Once a data is read into RAM, it can be cached based on the available memory and reused without being read it again from the disk.
- **Multi language support :** Spark jobs can be written in Scala, Python, R, Java and SQL.
- **Multi task support:** It has libraries for SQL, Streaming, machine learning and graph tasks.
- **Scalable:** Manage and coordinate execution of tasks on data across a cluster of computers. This cluster of machine is managed by cluster mangers (spark standalone cluster manger, yarn, Mesos etc.,).



Spark History

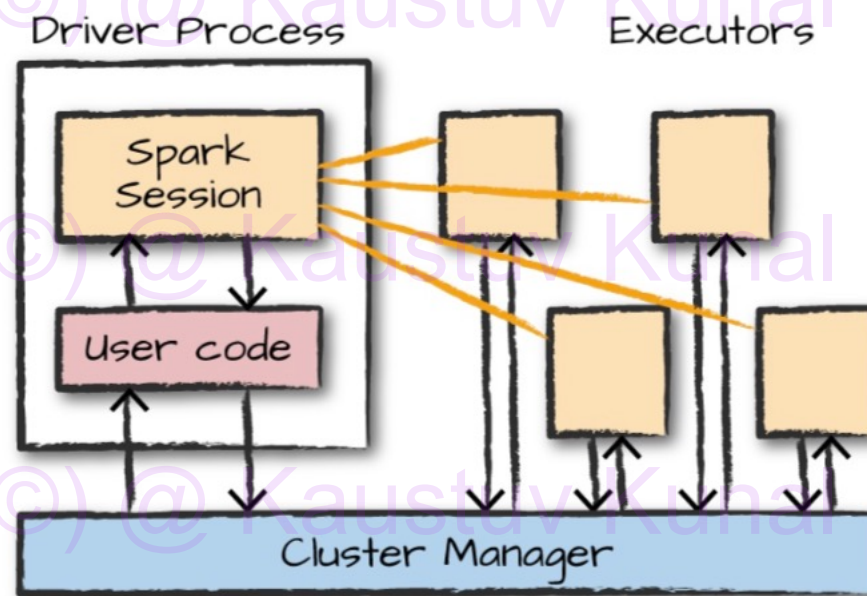


Spark Versions

- Spark 0.X.X (2012)
 - RDD
 - YARN support
- Spark 1.X.X (2014)
 - Spark SQL
- Spark 2.X.X (2016)
 - Unified Dataset and DataFrames APIs
 - SparkSession
 - Structured Streaming
 - Partition pruning
 - Builtin support for Hive features
 - More optimization
- Spark 3.X.X (2020)
 - Python pandas PySpark support
 - Optimization techniques (Auto Broadcast Join & Dynamic Partition Pruning)
 - Adaptive query execution



Spark Architecture



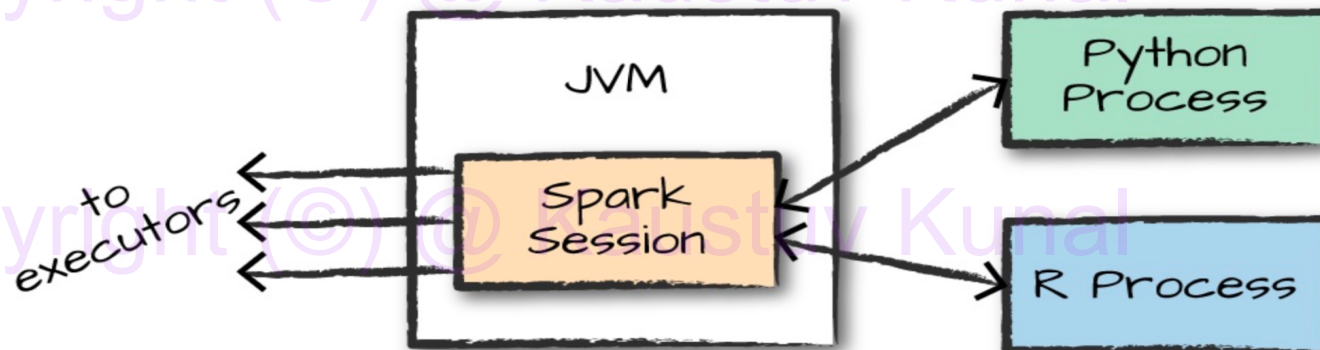
Architecture Components

- Driver Program/Process
 - Runs main function
 - Manage/Maintain information about spark application
 - Respond to user program/input
 - Analyze/schedule/distribute work across executor
- Executor Process
 - Perform tasks/code that driver assigns them
 - Reports execution state back to driver
- Cluster Manager (in cluster mode)
 - Keeps track of available resources



Architecture Salient Points

- User can specify how many executors should fall on each node through configuration.
- Spark session object is entrance point of spark code, python and R jobs translate code to generate spark session object.

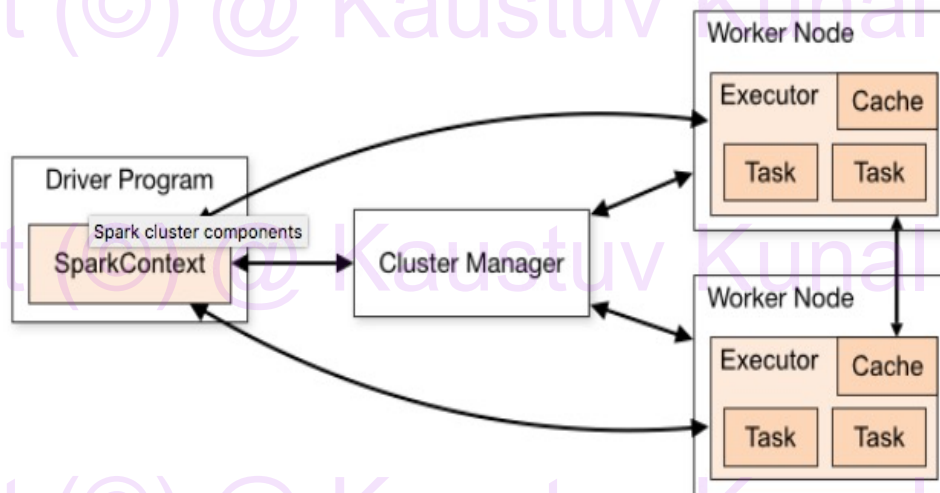


Execution/Deployment Modes

- Spark Jobs can be executed locally (local mode) or on a distributed system. Distributed execution has two deployment modes (client mode & cluster mode)
 - **Local mode** : Spark jobs can be executed locally just as a program using spark libraries. Both driver and executor runs on single JVM.
 - **Client mode** : The driver runs in the client process, and the application master is only used for requesting resources from YARN.
 - **Cluster mode** : Spark driver runs inside an application master process which is managed by respective cluster managers. Driver and executor runs on different machines.



Spark Cluster Execution



Execution Sequence

1. The main Driver program initializes SparkContext object.
2. SparkContext connects to Cluster Managers (in order to execute Spark applications as independent sets of processes on a cluster).
3. Cluster Managers allocate resources across applications.
4. Spark Application acquires Executors on nodes in the cluster.
5. SparkContext sends application code to Executors.
6. SparkContext sends tasks to the Executors to run.



Execution Salient Points

- Each application gets its own executor processes. Thus isolate applications on scheduler side (each driver schedules its own tasks) and executor side (tasks from different applications run in different JVMs). Downside is, data cannot be shared across different Spark applications.
- Driver program must listen for and accept incoming connections from its executors throughout its lifetime.
- Driver should run close to the worker nodes.



Supported Cluster Manager

- Following cluster managers are supported by Spark,
 - **Standalone** : Default cluster manager included with Spark that makes it easy to set up a cluster.
 - **Apache Mesos** : A general cluster manager that can also run Hadoop MapReduce and service applications.
 - **Hadoop Yarn** : The default resource manager of Hadoop .
 - **Kubernetes** : An open-source system for automating deployment, scaling, and management of containerized applications.



Execution Arguments

- Application :
 - Application Jar : Path to a bundled jar including your application and all dependencies
 - Driver Program : Main program location & name
- Cluster manager :
 - Deploy mode : Client/Cluster
 - Worker node : node used by executors
 - Executor : are JVMs that run on Worker nodes.

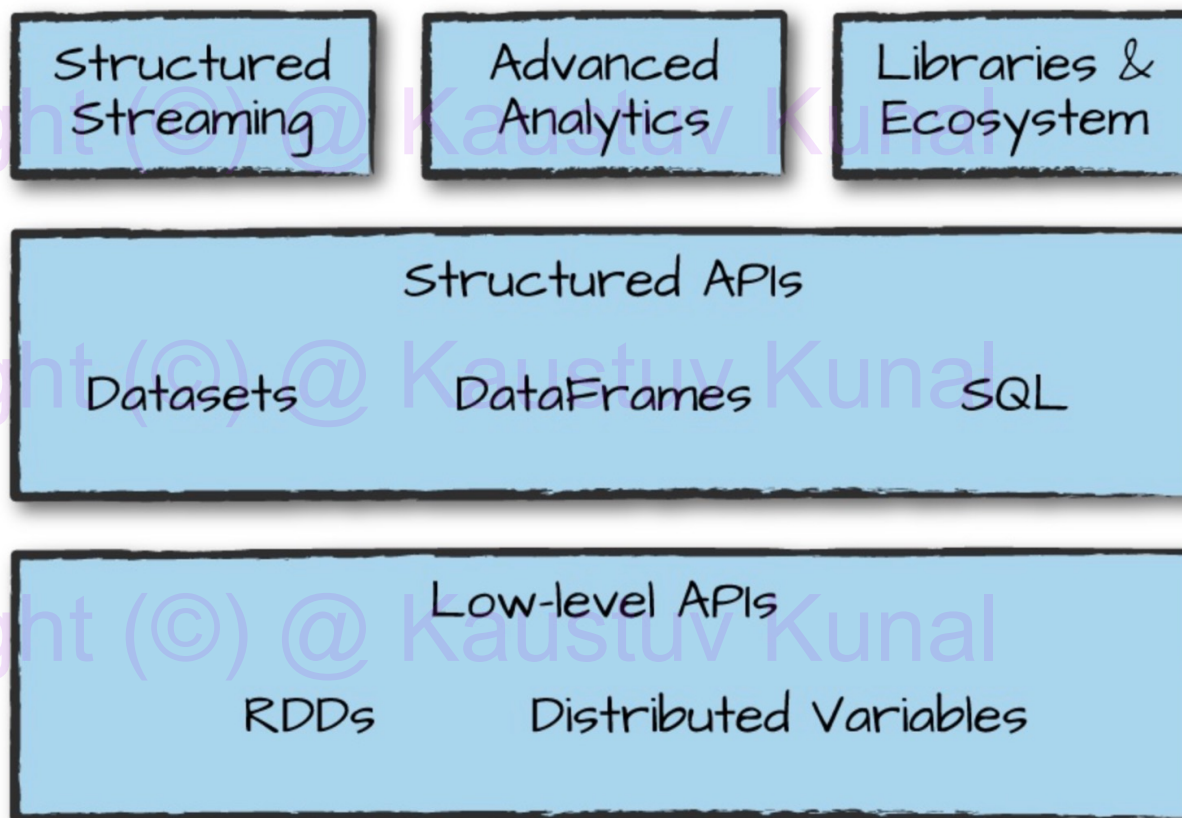


Execution Terms

- **Task** : Task is executed as a single thread in an Executor
- **Partition** : Data is split into Partitions so that each Executor can operate on a single part, enabling parallelization
- **Shuffle** : Refers to an operation where data is re-partitioned(shuffles) across a Cluster, e.g. join and any operation that ends with ByKey() will trigger a Shuffle
- **Stage** : Sequence of Tasks that can all be run together, in parallel, without a shuffle
- **Job** : Job is a sequence of Stages, triggered by an Action such as, count(), foreachRdd(), collect(),

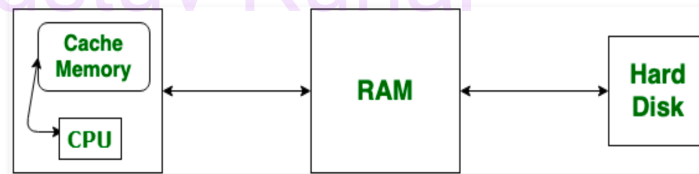


Spark API Ecosystem



RDD (Resilient Distributed dataset)

- An RDD is a read-only (immutable) collection of objects, partitioned across a set of machines that can be rebuilt if a partition is lost.
- Normally, RDD are cached data, however datasets that do not fit in memory are either spilled to disk or recomputed on the fly when needed, as determined by the RDD's storage level
- RDD rebuild lost data on failure using lineage. Spark keeps track of set of dependencies b/w RDD partitions called a lineage graph (DAG)
- It can be Scala, Java or Python object.



RDD Features

- Distributed
- Immutable (read only)
- Resilient (recovered lineage) i.e. ability to recreate
- Compile-time Type Safe i.e. throws compile time error if there is an data type operation error
- Can be Structured or Unstructured (logs, tweets, articles)
- Lazy evolution (materialized at action level only and not transformation level i.e. execution starts only when there is an action)



Creating RDD

- Parallelizing an existing collection in your driver program or referencing a dataset in an external storage system. Spark lets programmers construct RDDs in four ways,
 - From a file : in a shared file system, such as the Hadoop Distributed File System (HDFS)
 - Parallelizing : which means dividing it into a number of slices that will be sent to multiple node
 - By transforming an existing RDD : using an operation called flatMap, which passes each element through a user-provided function of type $A \Rightarrow \text{List}[B]$, expressed using flatMap, including map
 - Persistence : User can alter the persistence of an RDD through two action, cache action, save action



RDD Operations

- **Transformation** : Generates a new RDD from an existing one. Example `map()`, `filter()`. It is computed lazily i.e., only at their first use of Action.
- **Action** : Triggers a computation on a RDD and does something with the results i.e. either returning them to the user, or saving them to external storage. eg., `count()`, `first()`.

Q) How to check if a function is Transformation or Action ?

Ans) See the return type. If its an RDD it is transformation otherwise Action.



RDD Transformations

1. `map(func)`
2. `filter(func)`
3. `flatMap(func)`
4. `mapPartitions(func)`
5. `mapPartitionsWithIndex(func)`
6. `sample(withReplacement, fraction, seed)`
7. `union(otherDataset)`
8. `intersection(otherDataset)`
9. `distinct([numTasks])`
10. `groupByKey([numTasks])`



RDD Transformation

11. `reduceByKey(func, [numTasks])`
12. `aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])`
13. `sortByKey([ascending], [numTasks])`
14. `join(otherDataset, [numTasks])`
15. `cogroup(otherDataset, [numTasks])`
16. `cartesian(otherDataset)`
17. `pipe(command, [envVars])`
18. `coalesce(numPartitions)`
19. `repartition(numPartitions)`
20. `repartitionAndSortWithinPartitions(partitioner)`



RDD Actions

1. `reduce(func)`
2. `collect()`
3. `count()`
4. `first()`
5. `take(n)`
6. `takeSample (withReplacement,num, [seed])`
7. `takeOrdered(n, [ordering])`
8. `saveAsTextFile(path)`
9. `countByKey()`
10. `foreach(func)`
11. `saveAsSequenceFile(path) (Java and Scala)`
12. `saveAsObjectFile(path) (Java and Scala)`



Actions Allowing Parallel Operations

- Reduce
- Collect
- Foreach

KK



Shared Variables

- When Spark runs a function in parallel as a set of tasks on different nodes, if a variable needs to be shared across tasks, or between tasks and the driver program, it ships a copy of shared variable used in the function to each task.
- Spark supports two types of shared variables
 1. Broadcast Variables
 2. Accumulators

KK



Broadcast Variables

- Used to cache a value in memory on all nodes.
- If a large read-only piece of data (e.g., a lookup table) is used in multiple parallel operations, it is preferable to distribute it to the workers only once instead of packaging it with every closure. Spark lets the programmer create a “broadcast variable” object that wraps the value and ensures that it is only copied to each worker once.

KK

```
>>> broadcastVar = sc.broadcast([1, 2, 3])
<pyspark.broadcast.Broadcast object at 0x102789f10>
>>> broadcastVar.value
[1, 2, 3]
```



Accumulators

- The write only global variable that can be shared across tasks. (similar to counters of MapReduce). Updated by each task and the aggregated result is propagated to the driver program.
- These are variables that workers can only “add” to using an associative operation and that only the driver can read.

```
>>> accum = sc.accumulator(0)
>>> accum
Accumulator<id=0, value=0>

>>> sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
...
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s

>>> accum.value
10
```



DataFrame

- Represents a data table with rows and columns
- Uses schema for defining column name and type (metadata)
- Can span across clusters
- Can be constructed from a wide array of sources such as, structured data files, tables in Hive, external databases or existing RDDs.
- Available in Scala, Java, Python, and R
- Introduced in 1.3 release
- Used in Spark-SQL

Note: Do not confuse Spark DataFrame with panda DataFrame. Infact, with spark interface panda DataFrame can be easily converted into spark dataframe.



Dataset

- Type-Safe distributed collection of data type
- Provides object oriented programming interface
- Available for Java & Scala and not available for python and R as they are dynamically typed language
- Introduced in 1.3 release
- On Dataset, 'Collect' call will collect object of declared type only
- DataFrame is a Dataset organized into named columns
- Both Dataset and DataFrame can be used in same application, Dataset after operation can be converted in DataFrame



Dataset Benefits

- Instead of using Java serialization or Kryo, Datasets use a specialized Encoder to serialize the objects for processing or transmitting over the network.
- Allows Spark to perform many operations like filtering, sorting and hashing without deserializing the bytes back into an object.
- Provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine.
- Conceptually Spark Dataset is a DataFrame column with additional type safety.
- It means you get all the benefits of Catalyst and Tungsten optimization. It includes logical and physical plan optimization, vectorized operations and low level memory management.



RDD to Dataset/DataFrame

- An RDD can be converted into Dataset(or DataFrame in python) by two methods,
 1. By Inferring the RDD Schema Using Reflection
 2. By Programmatically Specifying the Schema

KK



Persist

- RDD are recomputed each time we run an 'Action'. To avoid this, store them in memory using persist. `RDD.persist()` .
- Persist on disk is also possible and useful in Big Data.

KK



RDD Storage Levels

- **MEMORY_ONLY** : default level, it stores the RDD as deserialized Java objects in the JVM; If the RDD doesn't fit in memory, some partitions will not be cached and recomputed each time they're needed
- **MEMORY_AND_DISK**: RDD as deserialized Java objects in the JVM; If the RDD doesn't fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed
- **MEMORY_ONLY_SER**: Stores RDD as serialized Java objects; space-efficient
- **MEMORY_AND_DISK_SER** : similar to **MEMORY_ONLY_SER**, but spill partitions that don't fit in memory to disk
- **DISK_ONLY** : stores the RDD partitions only on disk.
- **MEMORY_ONLY_2** : same as **MEMORY_ONLY**, but replicate each partition on two cluster nodes
- **MEMORY_AND_DISK_2** : same as **MEMORY_AND_DISK**, but replicate each partition on two cluster nodes
- **OFF_HEAP** : similar to **MEMORY_ONLY_SER**, but store the data in off-heap memory (Off-heap memory offloads values to a storage area that is not subject to Java GC)



Partitions

- To perform execution in parallel, spark break up data in chunks called partition.
- Collection of rows sits on a physical machine
- Allows executor to run parallel
- No of executor is equal no of parallel tasks
- With DataFrame no need to manipulate partition explicitly, it has to be done by configuration



Spark Execution Steps

1. **Submit Job** : User Submit spark Job (PySpark, Spark SQL etc.,)
2. **Logical Plan Creation** : If valid, spark converts job into a Logical Plan (it uses catalog which is kind of meta-store)
3. **Physical plan Creation** : Transforms logical plan into physical plan (execution strategies i.e. series of RDD transformation) also checks for **optimization** along the way
4. **Physical plan Execution** : Spark executes physical plan(RDD manipulations) on cluster (by generating java byte code)

* Use explain to see the physical plan



Optimization

- Process of converting logical plan to physical plan
- Spark uses following two engines to optimize and run the queries, they are mostly black-boxed for developers
 1. Catalyst
 2. Tungsten

KK



Catalyst

- Optimizes structural queries – expressed in SQL or DataFrame/Dataset APIs, to reduce the runtime of programs and save cost.
- Generates an optimized physical query plan from the logical query plan by applying a series of transformation e.g. predicate pushdown, column pruning, and constant folding.
- Are mostly of two kinds ,
 1. Rule-based optimization: indicates how to execute the query from a set of defined rules
 2. Cost-based optimization : generates multiple execution plans and compares them to choose the lowest cost one



Tungsten

- Improve Spark execution by optimising Spark jobs for CPU and memory efficiency through,
 - Off-Heap Memory Management : using binary in-memory data representation, aka Tungsten row format and managing memory explicitly
 - Cache Locality : is about cache-aware computations with cache-aware layout for high cache hit rates
 - Whole-Stage Code Generation

KK



Spark Installation Steps

1. **Download** : Choose Spark release package type from [Spark official download page](#)
2. **Set environments variables** : SPARK_HOME, HADOOP_HOME(for yarn clusters) , PYSPARK_PYTHON, SPARK_DIST_CLASSPATH has to be set
3. **Start spark**
4. **Validate UI & Master urls**
 - <http://<host>:4040/> (Spark UI - displaying spark jobs)
 - <http://<host>:8080/> (Spark master - It is the resource manager for the Spark Standalone cluster to allocate the resources)



Spark Consoles

- Following interactive spark console are available for users,
 - PySpark : /bin/pyspark
 - Scala : /bin/spark-shell
 - SQL: /bin/spark-SQL
 - For cloud support

KK



Job Submission

- Jobs shall be submitted using 'spark-submit' command

```
./bin/spark-submit \  
  --class <main-class> \  
  --master <master-url> \  
  --deploy-mode <deploy-mode> \  
  --conf <key>=<value> \  
  ... # other options  
  <application-jar> \  
  [application-arguments]
```



Job Submission Arguments

- `--Class` : main driver class
- `--Master` : master URL for the cluster
- `--deploy-mode` : client or cluster mode
- `--conf` : Spark configuration property in key=value format
- `application-jar` : bundled jar including spark application and all dependencies
- `application-arguments` : Arguments passed to the main method of application main class, if any
- `--supervise` : specify for standalone cluster mode; supports restarting application automatically if it has exited with non-zero exit code
- `--executor-memory` : specify amount of memory to use per executor process
- `--total-executor-cores` : the number of cores to use on each executor
- `--num-executors` : specify number of executors
- `--queue` : name of queues application need to be submitted



Job/Application Monitoring

- Each driver program has a web UI
- It can be accessed on port 4040 i.e. `http://<driver-node>:4040`
- If multiple Spark Contexts are running on the same host, they will bind to successive ports beginning with 4040 then 4041, 4042, etc.
- Users can also construct the UI of an application through Spark's history server '`sbin/start-history-server.sh`'
- Default URL of history server is `http://<server-host>:18080`
- Job metrics, are also available as JSON for developers to create new visualizations and monitoring tools for Spark



Application Monitoring

- Information about the application includes,
 - List of scheduler stages and tasks
 - Summary of RDD sizes & memory usage
 - Environmental information
 - Running Executors information

KK



Job Scheduling

- Cluster managers, that Spark runs on, provide scheduling facilities.
- Within each Spark application, multiple “jobs” (Spark actions) may be running concurrently if they were submitted by different threads.
- Spark also includes a fair scheduler to schedule resources within each SparkContext. (fair scheduler aims to maximize overall CPU utilization while also maximizing interactive performance).
- Dynamic Resource Allocation is by default false, to enable set below two properties to true,
 - `spark.dynamicAllocation.enabled`
 - `spark.dynamicAllocation.shuffleTracking.enabled`



Job Tuning

- **Serialization** : Storing RDD in serialized form (Kryo serialization)
 - `conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")`
 - Use the serialized storage levels in the RDD persistence API, such as `MEMORY_ONLY_SER`
- **Parallelism** : Try setting the level of parallelism for each operation high enough
- **Data Structure choices** : Choose data structure with less overhead e.g.,
 - Use arrays of objects, and primitive types, instead of the standard collection classes
 - Avoid nested structures with a lot of small objects



Job Testing

- Test locally outside clusters
- Purpose is to prevent bugs propagation in clusters
- Test the logic and not the Spark
- IDE is preferred over notebook for testing & debugging
- To test, simply create a SparkContext in your test with the master URL set to local, run your operations, and then call `SparkContext.stop()` to tear it down. Make sure you stop the context within a finally block or the test framework's tear Down method, as Spark does not support two contexts running concurrently in the same program.
- For Pyspark shall also use 'unittest.mock' library



PySpark

- Python on Spark
- Python advantages,
 - Fastest growing
 - Interactive
 - Language of machine learning

KK



PySpark Data Types

- `ByteType()`
- `ShortType()`
- `IntegerType()`
- `LongType()`
- `FloatType()`
- `DoubleType()`
- `StringType()`
- `BinaryType()`
- `BooleanType()`
- `TimestampType()`
- `DateType()`
- `ArrayType(elementType,[ContainsNull].)`
- `MapType(keyType, valueType[valueContainNull])`
- `StructType(fields)`
- `StructField(name, dataType,[nullable])`



PySpark Data Sources

- PySpark supports following data source,

- CSV
- JSON
- Parquet
- ORC
- JDBC/ODBC
- Plain-text Files

KK



Spark SQL

- Spark SQL is a Spark module for structured data processing
- Spark SQL extends Spark with a declarative DataFrame API to allow relational processing
- When working with data with schema Spark SQL is preferred. It provides benefits such as,
 - Loading data from variety of sources (JSON, hive, parquet)
 - SQL query interface through standard JDBC/ODBC connector
 - Rich integration b/w SQL and Python/Java/Scala code with ability to join RDD and SQL tables and custom functions
- There are multiple ways to interact with Spark SQL including, SQL interface, DataFrame and the Dataset API



Spark SQL Goals

- Support relational processing both within Spark programs (on native RDDs) and on external data sources using programmer friendly API.
- Provide high performance using established DBMS techniques.
- Easily support new data sources, including semi-structured data and external databases.
- Enable extension with advanced analytics algorithms such as graph processing and machine learning.



Spark SQL Essentials

- **SqlContext** : is entry point of SparkSQL which can be received from sparkContext.
- **Schema RDD** : Used in implementing Spark SQL, are RDD of row object each representing a record, Can be created from external data sources, query results or from regular RDDs.
- **DataFrame Schema** : can be defined explicitly (preferred for production system) or left for data source.
- **Persistent Tables** : DataFrames can also be saved as persistent tables into Hive metastore using the *saveAsTable* command. Bucketing and sorting are applicable only to persistent tables.



SQL (DF) Transformation

- Following SQL operation can be performed in DataFrame tables,
 - Adding rows or columns
 - Removing rows or columns
 - Transforming a row into column
 - Changing the order of rows based on values in columns

KK



Views in Spark SQL

- Temporary views are session-scoped and will disappear if the session that creates it terminates.
- If you want to have a temporary view that is shared among all sessions and keep alive until the Spark application terminates, you can create a global temporary view.

KK



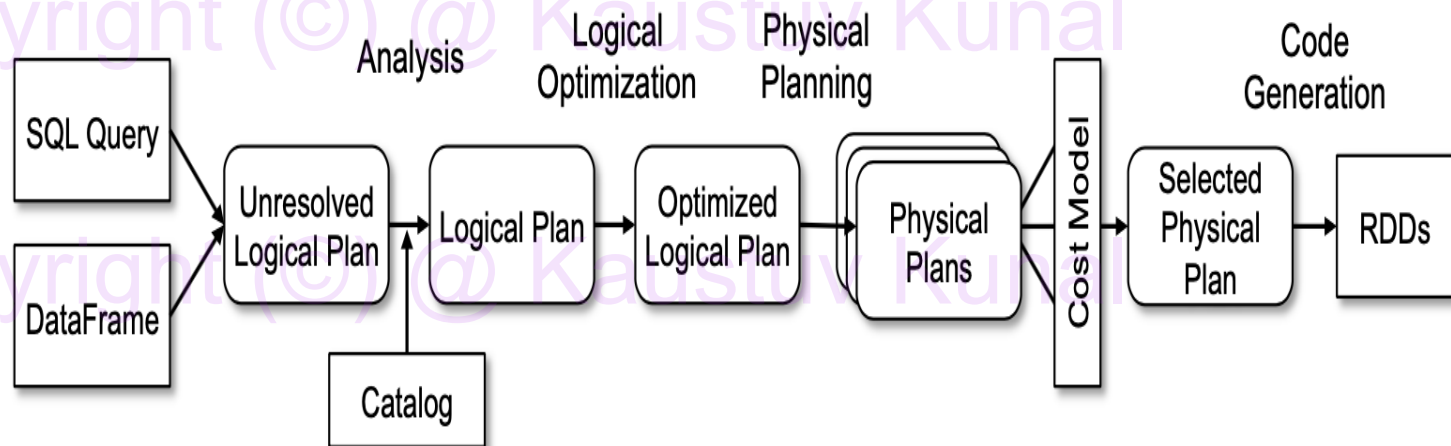
Apache Arrow : Panda DF to Spark DF

- Apache Arrow is an in-memory columnar data format that is used in Spark to efficiently transfer data between JVM and Python processes.
- We can Create a Spark DataFrame from a Pandas DataFrame using Arrow and convert the Spark DataFrame back to a Pandas DataFrame using Arrow.

KK



Query Planning in Spark-SQL



Spark Streaming

- High level library of Spark for processing continuously flowing streaming data
- Uses micro batch Architecture
- Split data stream into batches of as low as 100 milliseconds and exactly-once fault-tolerance guarantees
- Each batch of data is treated as RDD and processed using RDD operations
- Processed results are pushed out in batches
- In Spark 2.3, A new low-latency processing mode called Continuous Processing is introduced, it can achieve end-to-end latencies as low as 1 millisecond with at-least-once guarantees
- The fundamental stream unit is DStream which is basically a series of RDDs to process the real-time data



Spark Streaming

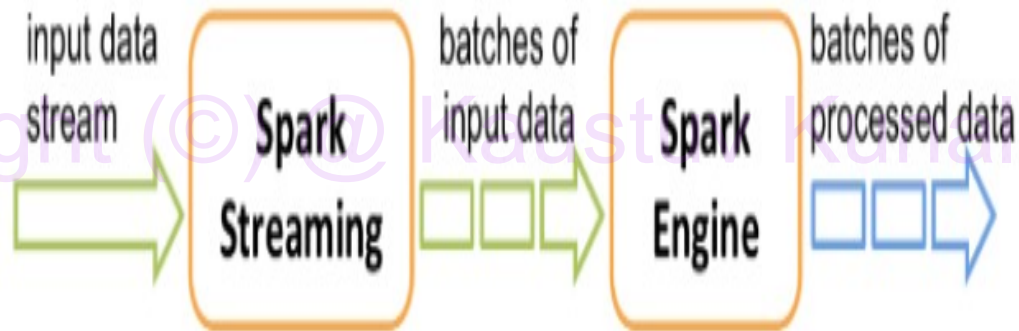


Dstream (Discretized Stream)

- Represents a continuous stream of data
- Can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying operations on other Dstreams
- Stream pipeline is registered with some operations and the Spark polls the source after every batch duration
- Dstream support the following operation,
 - Map, flatMap, filter
 - count
 - reduce
 - countByValue
 - reduceByKey
 - join
 - updateStateByKey



Streaming Flow



Persist in DStream

- DStream generated by window-based operations are automatically persisted
- Call `persist()` to Persist every RDD of that DStream in memory
- For received data over the network it replicate data in two nodes
- Default persistence level of DStream keeps the data serialized in memory

KK



Streaming Checkpointing

- Checkpoint is storing enough information to recover from failures.
- Use the checkpointing to save the progress of a job to be used in case of failure
- Metadata checkpoints includes Configurations, operations, batches
- Data checkpoints includes RDDs of stateful transformations
- To configure use 'Context.checkpoint(checkpoint_directory)'
- Accumulators and Broadcast variables cannot be recovered from checkpoint in Spark Streaming for that create lazily instantiated singleton instances for Accumulators and Broadcast variables so that they can be re-instantiated after the driver restarts on failure



Streaming Performance Tuning

- Set the **right batch size** such that the batches of data can be processed as fast as they are received
- **Reduce the processing time** of each batch by,
 - Parallelizing the data receiving
 - Creating multiple input DStreams
 - Reduce the block interval by configuring 'spark.streaming.blockInterval'

KK



Streaming Memory Tuning

- Enabling Kryo Serialization
- Clearing old data frequently (`streamingContext.remember`)



Streaming ML Operations

- Streaming machine learning algorithms (e.g. Streaming Linear Regression, Streaming KMeans, etc.) can simultaneously learn from the streaming data as well as apply the model on the streaming data.
- Alternatively, for much larger class of machine learning algorithms, you can train a learning model offline (i.e. using historical data) and then apply the model online on streaming data.

KK



Streaming Limitations

- **Susceptible to data loss:** Spark streaming put the data in a batch even if the event is generated early and belonged to the earlier batch which may result in less accurate information as it is equal to the data loss
- Spark Streaming still don't have any **support for python 3**



Structured Streaming

- Streaming Engine built on Spark SQL for Realtime Structured data processing
- Production ready since spark 2.2
- Easily apply any SQL query on streaming data
- Uses DataFrames to process streams of data pouring into the analytics engine (not Dstream)
- As DataFrame API provides a higher level of abstraction it minimizes latency compare to Spark-Streaming



Structured Streaming Processing

- Key idea in Structured Streaming is to treat a live data stream as a table that is being continuously appended
- Polls the data after some duration, based on the trigger interval
- Received data in a trigger is appended to the continuously flowing data stream i.e. DataFrame (or dataset)
- Structured Streaming only reads the latest available data from the streaming data source, processes it incrementally to update the result, and then discards the source data



Structured Streaming Fault Tolerance

- Structured Streaming engine uses checkpointing & write-ahead logs to record the offset range of the data being processed in each trigger.
- Structured streaming has applied below two conditions to recover from any error,
 1. The source must be replayable
 2. The Sinks must support idempotent operations to support reprocessing in case of failures

KK



Structured Streaming Features

- Allow **window-based aggregations** using window keyword
- **Watermarking**, lets the engine automatically track the current event time in the data and attempt to clean up old state accordingly
- Supports **Joining** a streaming Dataset/DataFrame with a static Dataset/DataFrame as well as another streaming Dataset/DataFrame
- **Deduplication**: Eliminate duplicate or redundant information using a unique identifier in the events (with or without watermark)




Structured Streaming Preconditions

- By default schema is should to be pre-specified
- Directories that make up the partitioning scheme must be present when the query starts and must remain static
- Arrival delays : late data within the threshold will be aggregated, but data later than the threshold will start getting dropped. Threshold can be manually defined

KK



Structured Streaming Limitations

- Following Dataset actions, will not work on streaming Datasets,
 - `count()`
 - `foreach()`
 - `show()`
- Following type of operations are not supported on Spark Streaming,
 - Multiple streaming aggregations
 - Limit and take the first N rows 
 - Distinct operations
 - Sorting



Continuous Processing

- A new experimental streaming execution mode introduced in Spark 2.3 enables low (~ 1 ms) end-to-end latency with at-least-once fault-tolerance guarantees
- To run a supported query in continuous processing mode, all is needed is to specify a continuous trigger with the desired checkpoint interval as a parameter (trigger settings of a streaming query defines the timing of streaming data processing)

KK



MLlib

- Spark's machine learning (ML) library makes practical machine learning scalable and easy.
- It provides tools for,
 - ML Algorithms: Classification, Regression, Clustering, and Collaborative filtering
 - Featurization: Feature Extraction, Transformation, Dimensionality Reduction, and Selection
 - Pipelines: Tools for constructing, evaluating, and tuning ML Pipelines
 - Persistence: Saving and load algorithms, models, and Pipelines
 - Utilities: Includes linear algebra, statistics, data handling, etc.
- The primary Machine Learning API for Spark is DataFrame (not RDD)



Spark-ML

- Not an official name but occasionally used to refer to the MLlib DataFrame-based API
- Spark MLlib can also be used with RDD, while Spark-ML refers to DataFrame support only

KK



Statistic Operations Support

- Basic Statistics support for Dataframe API
 - Summary Statistic
 - Correlation : to calculate correlations between series
 - Hypothesis testing : ChiSquareTest
 - Summarizer
- Basic Statistics support for RDD API
 - Stratified Sampling
 - Random Data Generation
 - Kernel Density Estimation



Supported Data Sources

- Support for general data sources

- Parquet

- CSV

- JSON

- JDBC

- Support for specific data sources

- Image

- LIBSVM format

KK



RDD – Supported Data Type

- Local vector
- Labeled point
- Local matrix
- Distributed matrix
 - RowMatrix
 - IndexedRowMatrix
 - CoordinateMatrix
 - BlockMatrix

KK



ML Pipelines

- APIs to create and tune practical machine learning pipelines
- Allow combining Model's transformer, estimators, persistence, cross validation, etc.,
 - **Transformer** : is an algorithm which can transform one DataFrame into another DataFrame
 - **Estimators** : is an algorithm which can be fit on a DataFrame to produce a Transformer
 - **Pipeline** : chains multiple Transformers and Estimators together to specify an ML workflow
 - **ML persistence** : save a model or a pipeline to disk for later use



MLlib Feature Engineering

Feature Extractors DF APIs

- TF-IDF
- Word2Vec
- CountVectorizer
- FeatureHasher

KK



MLlib Feature Engineering

Feature Transformers DF APIs

- Tokenizer
- StopWordsRemover
- n-gram
- Binarizer
- PCA
- PolynomialExpansion
- Discrete Cosine Transform (DCT)
- StringIndexer
- IndexToString
- OneHotEncoder
- VectorIndexer
- Interaction
- Normalizer
- StandardScaler
- RobustScaler
- MinMaxScaler
- MaxAbsScaler
- Bucketizer
- ElementwiseProduct
- SQLTransformer
- VectorAssembler
- VectorSizeHint
- QuantileDiscretizer Imputer



MLlib Feature Engineering

Feature Selectors DF APIs

- VectorSlicer
- RFormula
- ChiSqSelector
- UnivariateFeatureSelector
- VarianceThresholdSelector

KK



Feature Engineering Support For RDDs

- TF-IDF
- Word2Vec
- StandardScaler
- Normalizer
- ChiSqSelector
- ElementwiseProduct
- PCA

KK



MLlib Classification Models

- Logistic regression
 - Binomial logistic regression
 - Multinomial logistic regression
- Decision tree classifier
- Random forest classifier
- Gradient-boosted tree classifier
- Multilayer perceptron classifier
- Linear Support Vector Machine
- One-vs-Rest classifier
- Naive Bayes
- Factorization machines classifier



MLlib Regression Models

- Linear regression
- Generalized linear regression
- Decision tree regression
- Random forest regression
- Gradient-boosted tree regression
- Survival regression
- Isotonic regression
- Factorization machines regressor



MLlib Unsupervised Models

- K-means
- Latent Dirichlet allocation (LDA)
- Bisecting k-means
- Streaming k-means
- Gaussian Mixture Model (GMM)
- Power Iteration Clustering (PIC)
- Collaborative Filtering
- FP (Frequent Pattern)-Growth



MLlib Other APIs Support

- Cross-Validation (Model/ Hyperparameter selection)
- Train-Validation Split



Spark Graph-X

- Spark component for graphs and graph-parallel computation
- Extends the Spark RDD by introducing a new Graph abstraction (a directed multigraph with properties attached to each vertex and edge)
- Support graph computation with operators such as subgraph, joinVertices, and aggregateMessages
- Includes a growing collection of graph algorithms and builders to simplify graph analytics tasks
- No PySpark support yet for GraphX



Spark Competitors

- Apache Apex
- Apache Flink (Streaming)
- H2O (Spark MLlib)
- Hive (Spark-Sql)
- Apache Storm (Streaming)

KK



References

- <https://spark.apache.org/>
- Spark: The definitive guide Bill Chambers & Matei Zaharia
- MLlib: Machine Learning in Apache Spark, Journal of Machine Learning Research 17 (2016) 1-7
- Spark SQL: Relational Data Processing in Spark, Michael Armbrust et.al,.

KK

