PySpark

Date / /
Page L&W

* Apache Spark provides powerful APIs for large-scale data processing. Here's an overview of its core components and how to g

- Apache Spark offers a unified analytics engine for large-scale data processing, with APIs in Scala, Java, Python, and R. These APIs provide high-level abstractions to interact with Spark's various components, enabling tasks like:

- Batch processing: Processing large datasets in a batch mode.
- Stream processing: Real-time processing of data structure
- SQL Queries: Performing SQL-like queries on structured data.
- Machine learning: Building and training ml models.
- Graph processing: Analyzing relationships in graph-structured data.

* Initializing Spark

- Before performing any operations, you need to initialize Spark. This involves setting up the necessary entry points to interact with the Spark cluster.

- SparkSession:- The SparkSession is the unified entry point for all spark functionalities. It allows you to programmatically create Dataframe

Datasets, and interact with various data sources.

- Key Features

a) Unified Entry Point: Replaces older contexts.
b) Configuration: Allows setting spark configurations like application name, master URL, memory allocation, etc.
c) Dataframe and Dataset API: Provides methods to create and manipulate Dataframes and Datasets.
d) SQL Execution: Enables executing SQL queries directly.

* SparkContext.

- The SparkContext is the heart of a spark application. It represents the connection to a Spark cluster and is used to create Resilient Distributed Datasets (RDDs), accumulators, and broadcast variables.

- key feature

* Cluster Connection: connects your application to the spark cluster.
* RDD creation: Provides methods to create RDDs from various data sources (e.g. text files, HDFS).
* Low-level API: Offers a lower-level API for distributed computations.
* Resource Allocation: Manages the allocation of resources on the cluster.

sc = spark.SparkContext

Date / /
Page L&W

* Resilient Distributed Datasets (RDDs)
- Immutable distributed collection of objects
- Fault-tolerant and parallelized across nodes
- created using sc.parallelize() or from external data.
- supports in-memory computation.

* External Datasets
- Data loaded from sources like:
  - HDFS
  - Local File System
  - Amazon S3
  - cassandra, HBase, JDBC, etc.

ex: sc.textfile ('path')

* RDD operation.
- Transformations (lazy): create a new RDD from an existing one; computation is deferred until an action is called.

ex.
- map() - apply function to each element
- filter() - select elements matching condition
- flatMap() - flatten results.
- groupByKey() - group values by key.

- Actions (trigger computation): Execute the th transformations and return a final result or write data to storage

ex.
- collect() - return all elements
- count() - number of elements
- take() - return first N elements

- saveAsTextFile() - write RDD to text file)

* Passing Functions to Spark.
- In PySpark, functions are passed to RDD transformations (like map(), filter()) to define the logic applied to each element.
- These functions are serialized and distributed to worker nodes for execution.
- Types of functions that can be passed:
  - Lambda functions
  - Predefined functions
  - UDFs (UserDefined Functions)

* Working with key-value Pairs
- key-value RDDs are used for operations like aggregation, joins and grouping.
- They are created using transformations like:

```
rdd.map(lambda x: (key, value))
```

- Common operations:
  - reduceByKey() :- Aggregates values by key using a function (e.g., sum). Faster and more efficient than groupByKey().
  - groupByKey() :- group all values with the same key. Can cause high data shuffle; use with care.
  - join() - Joins two RDDs based on matching keys.
  - leftOuterJoin() - Join two RDDs, keeping all keys from the left RDD.

- These operation often involve shuffle, which moves data across the cluster

* Shuffle operations
- Redistribution of data across partitions.
- Triggered by operations like groupBykey(), reduceBykey(), join().
- Expensive due to disk and n/w I/O.

* RDD Persistence
- Persistence allows you to store RDDs in memory (or disk) to reuse them across multiple actions, avoiding recomputation and improving performance.
- Useful when the same RDD is used multiple times in an application.
- Methods:
  - rdd.cache() - Persists the RDD in memory with default storage level (MEMORY_ONLY).
  - rdd.persist(storage_level) - Allows specifying storage levels like MEMORY_AND_DISK, DISK_ONLY, etc.
- Hepls optimize iterative algorithms and complex pipelines

* Removing Data
- Remove unneeded RDDs using unpersist()
- Helps to free memory.

* Shared variables
- In PySpark, shared variables allow communication bet^n the driver and workers nodes in a controlled way.
- They help avoid sending a copy of the variable with every task, improving performance.
- Types:
  - Broadcast variables: Used to efficiently share large read-only data (like lookup tables) with all worker nodes. Created using sc.broadcast(variable)
  - Accumulators: Used for aggregating values (e.g. counters, sums) across tasks. Created using sc.accumulator(0) and updated using add() method.
- These variables are write-only for workers and read-only for the driver (except broadcast, which is read-only everywhere).

* Deploying to a cluster
- PySpark application can run in different deployment modes depending on the environment and scale.
- Deployment mode determines where the driver program runs and how tasks are distributed
- Common cluster modes:
  - Local - Runs spark locally on a single machine (for testing/debugging)

- Standalone - Uses Spark's built-in cluster manager.
- YARN - Integrates with Hadoop's resource manager.
- Mesos - A general-purpose cluster manager.

- Applications are submitted using the spark-submit script, which handles deployment, resource allocation, and execution.

* Spark Dataframes.
- Distributed collection of data organized into named columns.
- Similar to pandas Dataframe but optimized for big data.
- Supports SQL queries via spark.sql()
- Created using spark.read From CSV, JSON, Parquet, etc.

* EDA (Exploratory Data Analysis) using PySpark
- EDA is the process of analyzing datasets to summarize their main characteristics, detect patterns, spot anomalies, and test hypotheses.
- In PySpark, EDA is performed in a distributed fashion using Dataframes, which provide scalable, SQL-like operations for big data.
- Key steps.
- Loading data: Load large datasets from multiple formats like CSV, JSON, Parquet, etc.

eg. spark.read.csv('data.csv')

- Schema inspection:
  - Understand data structure and types using df.printSchema(), df.types

- Summary statistics:
  - Generate descriptive stats using df.describe() or df.summary() to get count, mean, stddev, min, max.

- Handling nulls / missing values:
  - Detect and handle missing data using df.filter(df.colname.isNull())

- Value counts:
  Count occurrences of values using df.
  df.groupBy("column").count()

- Correlation and patterns:
  - Analyze relationships using df.stat.corr(), groupings, or visualize sampled data with libraries like matplotlib.

- Pyspark EDA is essential for preparing data before modeling in big data workflows.

* ETL Jobs using Spark

- ETL (Extract, Transform, Load) is a critical data pipelines used to collect data from various sources, clean and process it

and store it in a target system for analysis or further processing.
- PySpark is widely used for ETL in big data environments due to its scalability, distributed processing, and rich DataFrame APIs.

- Steps in ETL
- Extract: Retrieve data from various structured or unstructured sources like:
  - CSV, JSON File.
  - HDFS
  - Relational databases via JDBC
  - APIs, NoSQL stores
  ex. spark.read.csv('data.csv')

- Transform: cleanse, enrich and format the data for analysis.
  - common operations: filter(), select(), join(), withColumn(), groupBy().agg()
  - Apply UDFs (User Defined Functions) for custom logic
  - Perform schema validation and type casting.

- Load: Store the final transformed data into:
  - Distributed file systems (HDFS, S3)
  - Data warehouses or RDBMS
  - NoSQL databases (e.g. Cassandra, MongoDB)
  ex. df.write.parquet('output/')

* Introduction to Kafka

- Apache kafka is a distributed, fault-tolerant streaming platform for building real-time data pipelines and streaming applications.
- It is used to ingest, store and process, and forward high volumes of data in real times.
- kafka follows a publish-subscribe model, where producers send data to topics and consumers read from them.

- Components:

• Producer:
- Publishes data (message) to kafka topics.
- Can send data continuously (e.g. from sensors, logs, applications).
- Supports load balancing across partitions for scalability.

• Consumer:
- Subscribes to topics and reads data from them.
- Can be part of a consumer group to allow parallel consumption and fault tolerance.
- Each message is delivered to only one consumer in a group.

Date / /
Page L&W

- Broker:
  - A kafka server that stores data and serves clients (producers / consumers).
  - kafka cluster typically have multiple brokers for reliability and load distribution.
  - Brokers manage partitions, replication and storage.

- Topic:
  - A logical category or stream to which records are sent by producers.
  - Topics are split into partition, which allow parallelism.
  - consumers subscribe to topics to receive relevant messages.

- Zookeeper
  - Zookeeper manages brokers (keep a list of them).
  - zookeeper helps in performing leader election for partitions.

Note - kafka cannot work without zookeeper

* Spark streaming
- Spark streaming is a module in Spark for processing real-time data streams.
- Data is divided into micro-batches and processed using Spark's API's.
- Sources: kafka, flume, TCP sockets, Files etc.
- Common transformations: map(), flatmap(), reduceByKey(), window()
- O/p can be stored in files, databases or dashboards.

* Integration of setting spark and kafka.

- Spark can integrate with kafka to consume real-time data streams using Spark structured streaming.
- Spark reads data from kafka topics and processes it in micro-batches or continuous mode.

Steps:

1. Set up a kafka cluster and create topics
2. Use spark's readStream API to read from kafka.
3. Process the stream using Dataframe or SQL APIs
4. Write the o/p to a sink (e.g. console, file, database)

* setting up kafka Producer and consumer.

a] kafka Producer.
- sends (publishes) messages to a kafka topic
- Needs:
  - kafka client library (kafka-python)
  - kafka broker address
  - Target topic name

b] kafka Consumer:
- Listens (subscribes) to one or more topics and processes incoming messages

Date / /
Page L&W

* Kafka Connect API
- KAFKA Connect is a tool for scalable and fault-tolerant streaming of data between kafka and external systems (like databases, files, cloud services).
- Uses connectors:
  - Source connectors: Pull data into kafka from sources like MySQL, HDFS, MongoDB, etc.
  - Sink connectors: push data from kafka to sinks like Elasticsearch, PostgreSQL, or S3.
- Features:
  - Built-in REST API for managing connectors.
  - Distributed and standalone modes.
  - Pre-built connectors available via Confluent Hub.

* Machine learning using Spark's MLlib
- MLlib is spark's machine learning library designed for scalable, distributed ML tasks on large datasets.
- It supports classification, regression, clustering, reccomendation, and model evaluation.
- Built on top of Spark core and DataFrames, allowing seamless integration with big data workflows.
- key features:
  - High Scalability
  - pipeline API to build end-to-end ML Workflows

- Built-in algorithms :
  - classification : logistic Regression, Decision T
  - Regression : Linear Regression
  - clustering : k-Means
  - Recommendation : ALS (Alternating least Squa
- Feature Engineering : Tokenizath, scaling, One-Ho Encoding, etc.
- Model evaluation : cross-val, hyperparamete tuning, accuracy, RMSE, etc

* Workflow.

1. load Data → spark.read.csv()
2. feature Transformation → vectorAssembler, standardScaler
3. split Data → randomSplit()
4. Train Model → model = algo.fit(train_data
5. Evaluate → evaluator.evalator(prediction
6. Predict → model.transform(test_data)

* Deep learning using spark

- Spark does not natively support deep learni but it can be integrated with external deep learning frameworks for distribute training and large-scale data preproces

- libraries / tools for dl with spark.
  - BigDL - DL library for Spark (by Intel).
  - TensorFlow OnSpark - Runs tF on spark cluster

Imp Note :- Basic Spark SQL does not support UPDATE and DELETE unless you use Delta lake, Hive, or other table formats that allow ACID transactions.

- Elephas - Integrates Keras with Spark for distributed deep learning.
- Horovod + Spark - for scalable model training with frameworks like pyTorch and Tensorflow.

* Spark SQL
- Spark SQL is a Spark module for structured data processing using SQL queries.
- It allows querying data using SQL syntax or Dataframe APIs.
- Integrates seamlessly with Hive, JDBC-compliant databases, and structured files files (CSV, Parquet, JSON, etc.)

- use spark.read.format("jdbc") to read data, and df.write.format("jdbc") to write data.

- some queries.
- create a Temporary View.
  df.createOrReplaceTempView('employees')

- Select Query
spark.sql("SELECT * FROM employees").show()
spark.sql('SELECT * From employees where sal>5000').show()