GOA COLLEGE OF ENGINEERING

"Bhausaheb Bandodkar Technical Education Complex"

Assignment No: 1

1. What is a Distributed Computing Framework?

A. Big Data volume, velocity, and veracity characteristics are both advantageous and disadvantageous during handling large amounts of data. It is really difficult to process, store, and analyze data using traditional approaches as such. To process data in a very small span of time, we require a modified or new technology which can extract those values from the data which are obsolete with time. The distributed computing frameworks come into the picture when it is not possible to analyze a huge volume of data in a short timeframe by a single system. Distributed programming frameworks or distributed computing frameworks that play crucial roles in big data processing are covered here. These frameworks provide predefined building blocks to developers to have faster application development. The framework will have execution flow pre-defined including task scheduling, fault tolerance and other aspects. The frameworks also provide programming models that can help in writing algorithms for data analytics.

2. Write short reports on each of the following distributed computing frameworks:

A.

1. Apache Spark: Apache Spark is an open-source, distributed processing system used for big data workloads. It utilizes in-memory caching, and optimized query execution for fast analytic queries against data of any size. It provides development APIs in Java, Scala, Python and R, and supports code reuse across multiple workloads—batch processing, interactive queries, real-time analytics, machine learning, and graph processing. Apache Spark started in 2009 as a research project at UC Berkeley's AMPLab, a collaboration involving students, researchers, and faculty, focused on data-intensive application domains. The goal of Spark was to create a new framework, optimized for fast iterative processing like machine learning, and interactive data analysis, while retaining the scalability, and fault tolerance of Hadoop MapReduce.

Hadoop MapReduce is a programming model for processing big data sets with a parallel, distributed algorithm. Developers can write massively parallelized operators, without having to worry about work distribution, and fault tolerance. However, a challenge to MapReduce is the sequential multi-step process it takes to run a job. With each step, MapReduce reads data from the cluster, performs operations, and writes the results back to HDFS. Because each step requires a disk read, and write, MapReduce jobs are slower due to the latency of disk I/O.

Spark was created to address the limitations to MapReduce, by doing processing in-memory, reducing the number of steps in a job, and by reusing data across multiple parallel operations. With Spark, only one-step is needed where data is read into memory, operations performed, and the results written back—resulting in a much faster execution. Spark also reuses data by using an in-memory cache to greatly speed up machine learning algorithms that repeatedly call a function on the same dataset. Data re-use is accomplished through the creation of DataFrames, an abstraction over Resilient Distributed Dataset

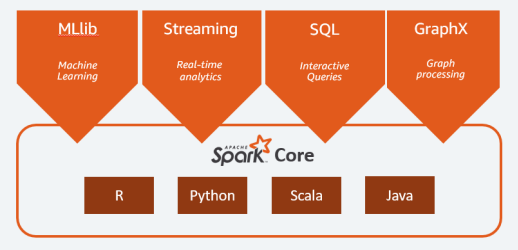
(RDD), which is a collection of objects that are cached in memory, and reused in multiple Spark operations. This dramatically lowers the latency making Spark multiple times faster than MapReduce, especially when doing machine learning, and interactive analytics.

There are many benefits of Apache Spark to make it one of the most active projects in the Hadoop ecosystem. These include:

1. Fast: Through in-memory caching, and optimized query execution, Spark can run fast analytic queries against data of any size.
2. Developer friendly: Apache Spark natively supports Java, Scala, R, and Python, giving you a variety of languages for building your applications. These APIs make it easy for your developers, because they hide the complexity of distributed processing behind simple, high-level operators that dramatically lowers the amount of code required.
3. Multiple workloads: Apache Spark comes with the ability to run multiple workloads, including interactive queries, real-time analytics, machine learning, and graph processing. One application can combine multiple workloads seamlessly.

FRAMEWORK: The Spark framework includes:

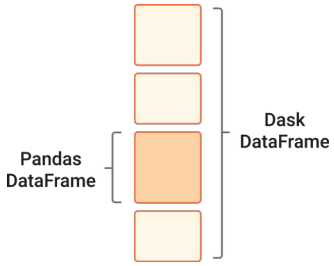
- Spark Core as the foundation for the platform
- Spark SQL for interactive queries
- Spark Streaming for real-time analytics
- Spark MLlib for machine learning
- Spark GraphX for graph processing



- **Spark Core:** Spark Core is the foundation of the platform. It is responsible for memory management, fault recovery, scheduling, distributing & monitoring jobs, and interacting with storage systems. Spark Core is exposed through an application programming interface (APIs) built for Java, Scala, Python and R. These APIs hide the complexity of distributed processing behind simple, high-level operators.

- **MLlib:** Spark includes MLlib, a library of algorithms to do machine learning on data at scale. Machine Learning models can be trained by data scientists with R or Python on any Hadoop data source, saved using MLlib, and imported into a Java or Scala-based pipeline. Spark was designed for fast, interactive computation that runs in memory, enabling machine learning to run quickly. The algorithms include the ability to do classification, regression, clustering, collaborative filtering, and pattern mining.
- **Spark Streaming Real-time:** Spark Streaming is a real-time solution that leverages Spark Core's fast scheduling capability to do streaming analytics. It ingests data in mini-batches, and enables analytics on that data with the same application code written for batch analytics. This improves developer productivity, because they can use the same code for batch processing, and for real-time streaming applications. Spark Streaming supports data from Twitter, Kafka, Flume, HDFS, and ZeroMQ, and many others found from the Spark Packages ecosystem.
- **Spark SQL Interactive Queries:** Spark SQL is a distributed query engine that provides low-latency, interactive queries up to 100x faster than MapReduce. It includes a cost-based optimizer, columnar storage, and code generation for fast queries, while scaling to thousands of nodes. Business analysts can use standard SQL or the Hive Query Language for querying data. Developers can use APIs, available in Scala, Java, Python, and R. It supports various data sources out-of-the-box including JDBC, ODBC, JSON, HDFS, Hive, ORC, and Parquet. Other popular stores—Amazon Redshift, Amazon S3, Couchbase, Cassandra, MongoDB, Salesforce.com, Elasticsearch, and many others can be found from the Spark Packages ecosystem.
- **GraphX Graph Processing:** Spark GraphX is a distributed graph processing framework built on top of Spark. GraphX provides ETL, exploratory analysis, and iterative graph computation to enable users to interactively build, and transform a graph data structure at scale. It comes with a highly flexible API, and a selection of

distributed Graph algorithms.



2. **Dask:** A Dask DataFrame is a large parallel DataFrame composed of many smaller Pandas DataFrames, split along the index. These Pandas DataFrames may live on disk for larger-than-memory computing on a single machine, or on many different machines in a cluster. One Dask DataFrame operation triggers many operations on the constituent Pandas DataFrames. Dask DataFrames coordinate many Pandas DataFrames/Series arranged along the index. A Dask DataFrame is partitioned *row-wise*, grouping rows by index value for efficiency. These Pandas objects may live on disk or on other machines. Dask DataFrame is used in situations where Pandas is commonly needed, usually when Pandas fails due to data size or computation speed:

- Manipulating large datasets, even when those datasets don't fit in memory
- Accelerating long computations by using many cores
- Distributed computing on large datasets with standard Pandas operations like groupby, join, and time series computations

Dask DataFrame may not be the best choice in the following situations:

- If your dataset fits comfortably into RAM on your laptop, then you may be better off just using Pandas. There may be simpler ways to improve performance than through parallelism
- If your dataset doesn't fit neatly into the Pandas tabular model, then you might find more use in dask.bag or dask.array
- If you need functions that are not implemented in Dask DataFrame, then you might want to look at dask.delayed which offers more flexibility
- If you need a proper database with all that databases offer you might prefer something like Postgres

Dask DataFrame covers a well-used portion of the Pandas API. The following class of computations works well:

- Trivially parallelizable operations (fast):

- Element-wise operations: df.x + df.y, df * df
- Row-wise selections: df[df.x > 0]
- Loc: df.loc[4.0:10.5]
- Common aggregations: df.x.max(), df.max()
- Is in: df[df.x.isin([1, 2, 3])]
- Date time/string accessors: df.timestamp.month

- Cleverly parallelizable operations (fast):
  - groupby-aggregate (with common aggregations): df.groupby(df.x).y.max(), df.groupby('x').max()
  - groupby-apply on index: df.groupby(['idx', 'x']).apply(myfunc), where idx is the index level name
  - value_counts: df.x.value_counts()
  - Drop duplicates: df.x.drop_duplicates()
  - Join on index: dd.merge(df1, df2, left_index=True, right_index=True) or dd.merge(df1, df2, on=['idx', 'x']) where idx is the index name for both df1 and df2
  - Join with Pandas DataFrames: dd.merge(df1, df2, on='id')
  - Element-wise operations with different partitions / divisions: df1.x + df2.y
  - Date time resampling: df.resample(...)
  - Rolling averages: df.rolling(...)
  - Pearson's correlation: df[['col1', 'col2']].corr()
- Operations requiring a shuffle (slow-ish, unless on index)
  - Set index: df.set_index(df.x)
  - groupby-apply not on index (with anything): df.groupby(df.x).apply(myfunc)
  - Join not on the index: dd.merge(df1, df2, on='name')

However, Dask DataFrame does not implement the entire Pandas interface. Users expecting this will be disappointed. Notably, Dask DataFrame has the following limitations:

- Setting a new index from an unsorted column is expensive
- Many operations like groupby-apply and join on unsorted columns require setting the index, which as mentioned above, is expensive
- The Pandas API is very large. Dask DataFrame does not attempt to implement

many Pandas features or any of the more exotic data structures like NDFrames

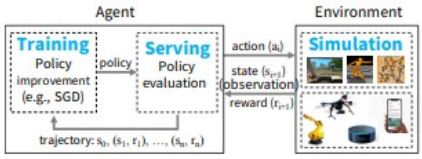
- Operations that were slow on Pandas, like iterating through row-by-row, remain slow on Dask DataFrame

By default, Dask DataFrame uses the multi-threaded scheduler. This exposes some parallelism when Pandas or the underlying NumPy operations release the global interpreter lock (GIL). Generally, Pandas is more GIL bound than NumPy, so multi-core speed-ups are not as pronounced for Dask DataFrame as they are for Dask Array. This is changing, and the Pandas development team is actively working on releasing the GIL. When dealing with text data, you may see speedups by switching to the distributed scheduler either on a cluster or single machine

3. **Ray :** Ray is an open-source project developed at UC Berkeley RISE Lab. As a general-purpose and universal distributed compute framework, you can flexibly run any compute-intensive Python workload — from distributed training or hyperparameter tuning to deep reinforcement learning and production model serving. Ray implements a unified interface that can express both task-parallel and actor-based computations, supported by a single dynamic execution engine. To meet the performance requirements, Ray employs a distributed scheduler and a distributed and fault-tolerant store to manage the system's control state
    - Ray Core provides a simple, universal API for building distributed applications.
    - Ray's native libraries and tools enable you to run complex ML applications with Ray.
    - You can deploy these applications on any of the major cloud providers, including AWS, GCP, and Azure, or run them on your own servers.
    - Ray also has a growing ecosystem of community integrations, including Dask, MARS, Modin, Horovod, Hugging Face, Scikit-learn, and others. The following figure gives you an overview of the Ray ecosystem.

Ray, a general-purpose cluster-computing framework that enables simulation, training, and serving for RL applications. The requirements of these workloads range from lightweight and stateless computations, such as for simulation, to long running and stateful computations, such as for training. To satisfy these requirements, Ray implements a unified interface that can express both task-parallel and actor based computations. Tasks enable Ray to efficiently and dynamically load balance simulations, process large inputs and state spaces (e.g., images, video), and recover

from failures. In contrast, actors enable Ray to efficiently support stateful computations, such as model training, and expose shared mutable state to clients, (e.g., a parameter server). Ray implements the actor and the task abstractions on top of a single dynamic execution engine that is highly scalable and fault tolerant.



To meet the performance requirements, Ray distributes two components that are typically centralized in existing frameworks [64, 28, 40]: (1) the task scheduler and (2) a metadata store which maintains the computation lineage and a directory for data objects. This allows Ray to schedule millions of tasks per second with millisecond-level latencies. Furthermore, Ray provides lineage-based fault tolerance for tasks and actors, and replication-based fault tolerance for the metadata store.

While Ray supports serving, training, and simulation in the context of RL applications, this does not mean that it should be viewed as a replacement for systems that provide solutions for these workloads in other contexts. In particular, Ray does not aim to substitute for serving systems like Clipper [19] and TensorFlow Serving [6], as these systems address a broader set of challenges in deploying models, including model management, testing, and model composition. Similarly, despite its flexibility, Ray is not a substitute for generic data-parallel frameworks, such as Spark [64], as it currently lacks the rich functionality and APIs (e.g., straggler mitigation, query optimization) that these frameworks provide.

Pros and Cons

	PROS	**CONS**
SPARK	a. Established and Mature Technology b. Plenty of companies providing commercial support / services. c. Ideal for data engineering / ETL type of tasks against large datasets. d. Provides higher-level SQL	a. A steep learning curve involving a new execution model and API b. Debugging can be challenging. c. Complex architecture, which is difficult to maintain by IT alone as proper maintenance requires

	abstractions (Spark SQL).	understanding of the computation paradigms and inner workings of Spark (e.g. memory allocation). d. Lack of a rich data visualization ecosystem. e. No built-in GPU acceleration. Needs RAPIDS Accelerator for accessing GPU resources.
DASK	a. Pure Python framework - very easy to ramp up. b. Out-of-the-box support for Pandas DataFrames and NumPy arrays. c. Out-of-the-box support for Pandas DataFrames and NumPy arrays. d. Easy exploratory data analysis against billions of rows via Datashader. e. Provides *Dask Bags* - a Pythonic version of the PySpark RDD, with functions like *map*, *filter*, *groupby,* etc. f. Dask can lead to impressive performance improvements.	a. Not much commercial support is available b. No built-in GPU support. Relies on RAPIDS for GPU acceleration.
RAY	a. Minimal cluster configuration b. Best suited for computation-heavy workloads. c. Because Ray is being used more and more to scale different ML libraries, you can use all of them together in a scalable, parallelised fashion d. Unique actor-based abstractions, where multiple tasks can work on the same cluster asynchronously	a. Relatively new b. Not really tailored to distributed data processing. c. Ray has no built-in primitives for partitioned data. d. GPU support is restricted to scheduling and reservations. It is up to the remote function to actually make use of the GPUf

	leading to better utilization	