IJCSN - International Journal of Computer Science and Network, Volume 6, Issue 6, December 2017
ISSN (Online) : 2277-5420
www.IJCSN.org
Impact Factor: 1.5

Data Processing Models for Distributed Computing and it's Ecosystem: A Survey

[1] Harish Mamilla; [2] Sai Pradeep; [3] Dr. Suresh C Gupta

[1] Department of Computer Science and Engineering, Indian Institute of Technology Delhi
New Delhi, India

[2] Department of Computer Science and Engineering, Indian Institute of Technology Delhi
New Delhi, India

[3] Department of Computer Science and Engineering, Indian Institute of Technology Delhi
New Delhi, India

Abstract - As the data volumes are growing rapidly, most of the processing needs to be distributed to several machines. Whereas the data processing over the distributed machines accompanies a few difficulties such as parallelism, scalability, machine failures and large data sizes. To face these challenges, much work has been carried out in this Big data domain. As a result, an extensive list of processing models and its co-existent technologies has been proposed for distributed cluster computing. However, there is a lack of comprehensive and comparative study to evaluate and choose from the number of options available. While many distributed computing technologies have emerged recently it can also be tough to understand how these technologies are related. We comprehend and briefly discussed the underlined architecture of distributed computing for a better understanding of the relations among various Big data technologies. This work also surveys features, strengths, and limitations of existing processing models by comparing them. Most of the existing data processing models are generally specific to particular application domain. In this work, we are also supporting the generic processing model that works for a variety of workloads as well as new application domains. Few of the key characteristics mentioned here can potentially guide in making an educated decision about the right combination of distributed processing technologies.

Keywords - *Distributed Computing, Big data, Data processing models, Hadoop, MapReduce, Spark, Flink*

1. Introduction

We organized this work as a glance at one place for entire distributed processing ecosystem. By the end of this study, we will be introduced to Big Data processing vocabulary. Big data is complex in nature which demands powerful technologies, a variety of techniques and advanced algorithms. Data analytics requires complex procedural model which may involve batch processing, real-time processing, machine learning, graph processing or a mixture of them. For example, very low latency processing of data demands real-time applications. As per user preferences, Big data has to support declarative queries for its data processing. It needs support from the underlined cluster resource management methodology for better load balancing of physical machines. To speed up the execution it also demands the availability of advanced commodity hardware.

The efficiency of distributed Big data processing over large clusters depends on many system dependent parameters such as data I/O performance, processing chipset architecture, availability of main memory, memory hierarchy optimization, commodity hardware cost. It also depends on application dependent parameters such as processing model design, type of workload supported, data size, fault tolerance, scalability. Especially on a large distributed system, latency and data locality plays a key role to deliver high-performance by taking processing to the data instead of moving the data around the cluster. Nevertheless it is the requirements which drive us to wisely choose between the key factors from the above-listed parameters. For example, one can compromise on the accuracy of the result to benefit from the speed of execution.

Most of the engines can only solve a specific type of workloads such as graph processing or machine learning etc., they are not capable enough to handle the variety of processing requirements which is most likely in modern scenarios. In this survey, we touched few parameters and we grouped our analysis based on the vital parameter "type of workload supported". Some of the mentioned frameworks in this survey build on

top of a general-purpose processing model which is independent of the type of workload. We always in need of a general-purpose processing framework which can save implementation effort and operational effort. There are several works that illustrates the below-discussed technologies but to the best of our knowledge, there is no existing work which compares data processing models and also briefs the full-stack technology ecosystem of distributed computing.

2. Distributed Computing Ecosystem - Architecture

Distributed cluster computing over large data often requires well-designed architectures that typically includes a combination of tools and techniques for data collection, data storage, data processing, data analysis and rendering of data results in dashboards. A common data pipeline architecture begins with raw data ingestion, followed by ETL (Extract-Transform-Load) to a data store then the core data processing, followed by high-level abstractions and finally rendering of extracted data results in the user interface dashboards. In Fig 1, proposed layered full-stack technology architecture illustrates the interrelationships among the different layers of a large-scale distributed processing ecosystem.

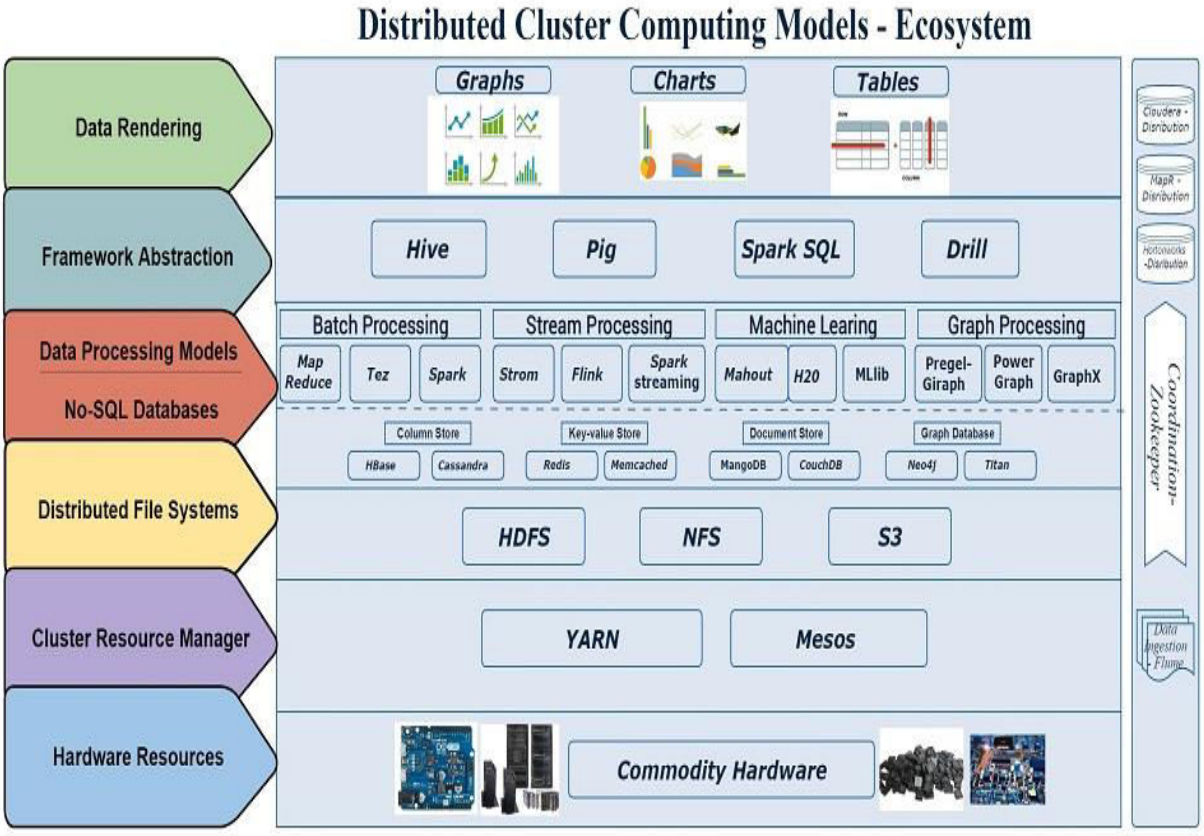

Fig 1: Distributed Computing Models- Ecosystem –Architecture

3. Cluster Resource Management

Commodity hardware is being vastly used for modern day computing as it is cost-efficient. Distributed fault-tolerant frameworks are designed to run on top of these commodity hardware clusters. It is the resource management layer that manages these large data clusters. It provides a resource-efficient platforms for executing the upper layer recommended jobs. Apache YARN [1] and Apache Mesos [2] are gained popularity as resource managers for distributed fault-tolerant computing clusters.

3.1 YARN

From Hadoop [3] version v2.0, a resource management layer was introduced for efficient sharing of cluster resources. YARN (Yet Another Resource Negotiator) is a software rewrite that decouples resource management, scheduling techniques from the MapReduce [3] data processing component.

YARN architecture Design proposed and developed by Horton Works [4]. HDFS [3] combined with YARN, facilitates an ideal data platform and processing solutions for Apache Hadoop. YARN is optimized for scheduling Hadoop jobs faster. Its recent versions are designed to accommodate support multiple frameworks along with Hadoop. Its ResourceManager is known as the master daemon and NodeManager is known as the worker daemon. NodeManager has a number of dynamically created resource containers of different sizes. A short-lived version of MapReduce JobTracker known as ApplicationMaster. ResourceManager tracks the number of live nodes and resources available on the cluster and then arbitrates the available cluster resources. When a client submits an application, an instance of ApplicationMaster is started to coordinate all task's execution within that application. Therefore, ApplicationMaster becomes responsible for monitoring tasks, restarting failed tasks etc. which were earlier handled by the single JobTracker in MapReduce framework.

3.2 MESOS

Few requirements may demand that multiple computational frameworks should run alongside other. Mesos was built to be a scalable global resource manager for the entire datacenter which typically supports different types of processing frameworks. Similar frameworks can intelligently co-exist, scheduled and run by Mesos on the same cluster. It is based on the concepts of Linux OS kernel, only at a different level of abstraction.

Mesos Design proposed and developed by the University of California, Berkeley [5]. Mesos has a master process, set of slave processes which run on the cluster nodes. On the slave nodes, different computational frameworks are being deployed and individual tasks run on these. Mesos master process uses resource allocation policies and available free resources for allocating resources to the computational framework. The master process has the responsibility of resource sharing using "resource offers" to the frameworks. A framework rejects resource offers which do not meet requirements, can wait for the satisfied ones and Frameworks can also set filters based on preferences. Each of Mesos framework is expected to implement an application scheduler and an executor process. Each framework can choose its algorithm for the scheduling of its jobs.

3.3 Discussion

Difference between Mesos and YARN is about their design priorities and how they approach scheduling jobs.

3.3.1 Scope

YARN was initially created out of the necessity to scale the Hadoop. YARN is best suitable for Hadoop environment. But its stable design has recently grown to accommodate heterogeneous frameworks too. Mesos initially designed as a general purpose scheduler for datacenter and cloud environments.

3.3.2 Implementation

YARN was written in Java, Its core uses lightweight UNIX processes, It has inbuilt support for processing frameworks. Mesos Written in C++, Its core uses Linux container groups. Mesos is lighter but demands more effort to customize it for processing framework.

3.3.3 Architecture

YARN architecture is based on the Remote Procedure Calls. Processing framework requests for a container with specifications and locality preferences, i.e Information passed here. Mesos architecture based on the message passing. Processing framework gets resource offers to choose from, i.e Information received here.

3.3.4. Decision Making

YARN decides the suitable resources for a given job. Data locality efficiently handled by YARN and jobs are deployed faster. Whereas in Mesos, it is the framework that makes the decision for the best possible fit.

3.3.5 Security

YARN inherits the Hadoop security and it is not a concern here. For Mesos, we need to deal with the security additionally.

3.3.6 Type of Work Load

YARN suitable for long running batch jobs and stateless batch jobs which can restart easily. Whereas, Mesos supports variety workloads which may support non-Hadoop frameworks also. Both of them have improved the scalability issues in Hadoop clusters but their centralized ResouceManager is still a barrier for extreme scale scenarios of distributed systems. Nowadays every corporation is opting for distributed datacenters for disaster recovery and business continuity. Data depositary is also getting distributed across datacenters which leads to the high availability of applications and data access. This kind of data distribution also helps in load balancing and performance scalability. If the organization has

multiple co-operating datacenters across the geography then the current resource management layer solutions Apache YARN and Mesos do not suffice. This limitation also triggers further research scope. Its centralized resource manager needs to address this kind of datacenters. Apache Myriad [6] enables the co-existence of Apache Hadoop and Apache Mesos on the physical infrastructure which is undergoing incubation.

4. Storage Systems

Storage systems here are typically a combination of distributed file system and a database system. Most of the storage distributions are de-coupled into the standard distributed file system and database system. Few storage distributions have their own distributed file system tightly coupled with the database. Distributions that incorporate their own file systems provide more convenience and faster back-ups. Most of the time, the file system is paired with a non-relational database in a Big data distributed computing.

4.1 Distributed File Systems

The distributed file system provides the basic storage infrastructure in the above architecture. Similar to shared file systems, distributed file system sources can range from NFS [7], S3 [8], or HDFS [9], where HDFS is more prominent for large clusters.

4.1.1 HDFS - Hadoop Distributed File System

HDFS (Hadoop Distributed File System) is an integral module of Apache Hadoop [10] environment. It forms the core components which builds the Hadoop Ecosystem. It provides scalable and reliable data storage, and it was designed to span large clusters over commodity hardware. It is a Java-based distributed file system which primarily provides high-throughput access to application data. HDFS designed as per Master/Slave architecture [10]. HDFS cluster contains NameNode, which is a Master server. NameNode manages the file system namespace and also provides the access control. HDFS cluster has many DataNodes which are responsible for storing the data blocks of files on their disks. For each and every file the NameNode tracks the block mapping to each DataNode. NameNode is made High- Available with its recent versions. HDFS cluster provides reliability, simple administration, and easy maintenance.

4.1.2 NFS – Network File System

NFS (Network File System): A protocol developed for which it allows clients to access files over the network. NFS clients access the data as if the files reside on the local machine, even though they reside on the disk of a networked machine. Its architecture is not highly sophisticated to cater the requirements of the production environment where reliability and High-Availability are not be compromised.

4.1.3 Amazon S3

Amazon Simple Storage Service is storage for the Internet [11]. Amazon S3 is a kind of cloud storage which provides storage through web service interfaces such as REST, SOAP etc. Amazon S3 manages data with an object storage architecture. Its design aims to provide scalability, high availability, and low latency. Amazon S3 is not an open source, mostly it has commercial offerings.

4.1.4 Discussion:

NFS is a file system that is distributed amongst many networked machines whereas HDFS is fault tolerant as it stores multiple replicas of files. Since S3 is in a cloud-native architecture it provides elasticity, scalability, built-in persistence but it incurs additional licensing cost. With limited sizes to store, S3 has lower prices than HDFS data storage but traditionally HDFS commoditized for huge data sizes to store and distribute a large amount of data.

4.2 NoSQL Databases

Databases represent the way of modeling the raw data which is physically present in the underlined file system. Relational database management systems are strict in nature and generally handles structured data. These type of databases do not scale if used with very large datasets. Non-relational databases which are referred as NoSQL, those will be suitable for various workloads as they support structured, semi-structured, and unstructured data. Also, these NoSQL databases are scalable in nature. As we typically deal with large data sets, NoSQL databases emphasized further in this study. The decision to choose the database is primarily based on how the data being stored and retrieved. NoSQL Databases majorly use one of below briefly mentioned types of data models [12].

4.2.1 Column-oriented databases

Data is stored in the form of columns rather than rows. However, by storing data in columns rather than rows, the database can more precisely access the data it needs to answer a query rather than scanning and discarding unwanted data in rows. Query performance is often increased as a result, particularly in very large data sets [13]. HBase [14] and Cassandra [15] are the

examples of majorly discussed column-oriented databases.

HBase is a non-relational column-oriented database management system which runs on top of HDFS in Hadoop environment. Hadoop MapReduce jobs can access the data stored in HBase tables. HBase is written in Java and has a Java Native API. Its Java API provides convenient base classes to the client for database operations. It supports sparse data sets, which are highly relevant in Big data domains.

Cassandra [16] is also a distributed database system which design based on the Column Data Model. Linear scalability and proven fault-tolerance on commodity hardware without compromising the performance make it the right choice for mission-critical data. It additionally offers robust support for clusters spanning across multiple datacenters with asynchronous replication. This asynchronous replication has very low latency to sync the primary database with remote site database which is designed for the business continuity in case disaster at primary datacenter.

4.2.2 Key-value databases

Key-value databases [17] are less complex of the all models and its implementation is based on the most commonly used data structure, a hash table. Each data item in the hash table has a unique key referring to it. They are fast and highly scalable. Redis [18] and Memcached [19] are the examples of majorly discussed key-value stores.

Redis [20] is an in-memory open-source database software project which implements a networked, in-memory key-value store with optional durability. Redis supports different kinds of abstract data structures, such as strings, lists, maps, sets, hyper logs, bitmaps and spatial indexes. Redis was the most popular implementation of a key-value database according to DB-Engines Ranking [12].

Memcached is an open source, high-performance, distributed memory object caching system. It is intended to use in speeding up dynamic web applications by eliminating database load. Memcached is an in-memory key-value store for small chunks of arbitrary data triggered from results of database calls, API calls, or page rendering. Memcached is simple yet powerful and its design promotes quick deployment, ease of development, and solves many problems facing large data caches. Its API is available for the most of popular languages.

4.2.3 Document databases

These type of NoSQL databases can be interpreted as nested key-value stores. Here a key refers to a set of key-value stores rather than simply a value. MongoDB [21] and CouchDB [22] are the examples of majorly discussed Document stores.

A MongoDB Document Data Model deployment hosts a number of databases. A database holds a set of collections. A collection holds a set of documents. A document is a set of key-value pairs. MongoDB can store data in JSON and gives us the rich features of an RDBMS such as indexes, dynamic queries, sorting, updates, and aggregation.It has flexibility and a scaling capability.

CouchDB [23] Document Data Model is implemented in the concurrency-oriented language Erlang [24]. It uses JSON to store data, JavaScript to query over MapReduce, and HTTP protocol for an API. It is designed for creating modern applications over a database model that can provide high availability and scalability.

4.2.4 Graph databases

These NoSQL databases are optimized for graph structures. Graph databases are intended for data that can be designed as a graph and can be used for jobs such as network analysis. Neo4J [25] and Titan [26] are the examples of majorly discussed Graph databases.

Neo4j[27] is a graph database management system developed by Neo Technology, Inc. A graph database targeted at very fast querying of huge graphs. Described by its developers as an ACID-compliant transactional database with native graph storage and processing.

Titan is a scalable graph database optimized for storing and querying graphs containing hundreds of billions of vertices and edges distributed across a multi-machine cluster. It is a transactional database that can support thousands of concurrent users executing complex graph traversals in real time.

5. Core Data Processing

The data to be processed can come from different sources like a distributed file system, structured storage, and network data. Large-scale data processing frameworks are placed above the data storage layer. For most of the frameworks the data processing layer shipped with database capabilities. In those

frameworks data processing layer directly talk to the resource manager and underlined file system.

5.1 HDD Computing

In the earlier processing engines, the only way to share data between computations is to write it to an external persistent storage system such as distributed file system. This causes significant overheads because of data replication, disk I/O [28], and serialization which may dominate application execution.

5.2 In-Memory Computing

As cost of main memory declining it is allowing for computing systems to be equipped with a huge amount of RAM at a relatively low price. This also provides the scope of using main memory as a data store instead of only processing purpose. Especially in a distributed computing environment if we combine space from all main memories then it will suffice to store and process the huge data sizes. Most part of the database can be accommodated by main memory [29], rest will spill out to disk. Disk I/O is the primary bottleneck in processing the data which is stored on the disk. In a distributed cluster computing setup if the network latency time is less than the disk I/O then In-memory computing un-doubtfully yields better results. While running In-Memory computations, to speed up the execution we can use buffers through which we can achieve near zero network latency.

This kind processing shift from disk to main memory also includes few challenges to address. The columnar data store in main memory does not natively support OLTP so we need hybrid storage model if OLTP also considered. Underlined processing engine should distribute the computation steps efficiently over the large cluster, i.e synchronization barrier should happen at main memory level. To make in-memory computing efficient we need to implement concurrency control using lightweight locking and hardware transactional memory. While reading data from the filesystem databases generates indexes for high-speed lookups. Databases in HDD are designed using B+ tree indexing in the view of disk I/O but databases in main memory focus on key value pair which can will perform better over fast tree-based indexing [30]. However, few issues like fault-tolerance and consistency are more difficult to handle with in-memory systems.

5.3 Generic Analysis of Models

MPI (Message Passing Interface) [31] is one of the oldest distributed computing models. MPI design is based on the peer-to-peer networking and master-slave paradigm. Peer-to-peer networks are used to communicate and exchange the data between nodes using broadcasting messages. But in peer-to-peer networks data shuffling between nodes is much more expensive [32]. One of its primary drawbacks is the fault intolerance as MPI has no mechanism to handle faults. MPI is not being widely used anymore in high-available large-scale cluster computing domain.

Distributed cluster computing for large data consists of many open source software projects licensed under the Apache Software Foundation [33]. Apache Hadoop project initiated as a robust framework which provides distributed computing ability. Hadoop first shipped with four core modules such as HDFS (Hadoop Distributed File System), YARN (Resource Manager), MapReduce (Processing Engine) and Hadoop Common (Libraries and Utilities for others) from the Apache Software Foundation. Today, Hadoop and Big-data stack has evolved to a stage where there are many processing frameworks are proposed based different type of workloads, application areas, and business scenarios. Few major models listed and studied in the due course of this survey.

Despite the fact that many frameworks address the specific challenging issues in the cluster environment, they are also having several drawbacks like limited scope, work duplication, inefficient resource sharing and complex administration. These drawbacks are the prominent reasons why we always in need of solutions which are built on top of a general-purpose processing engine. We also listed and discussed various processing models in each type of workload by comparing with generic engines such as Apache Spark [34] and Apache Flink [35]. Spark is one of the most prominent general purpose computing abstractions which is currently available in distributed processing of Big data.

5.4 Understanding Spark RDD

Apache Spark is a fast and general-purpose cluster computing system which is built around the abstraction called Resilient Distributed Datasets (RDD) [36]. Spark has unified processing architecture supported by Resilient Distributed Datasets. Resilient Distributed Datasets (RDDs) are fault-tolerant, in-memory and parallel data structures which provide efficient data sharing across parallel computations. Formally, an RDD is a read-only, partitioned collection of records. RDDs can only be created through deterministic operations on either data in persistent storage or from other RDDs. Examples of transformations include map, filter, and join. These RDDs extend the programming model introduced by MapReduce. It can run batch, interactive, iterative and streaming computations at the same time and provides fault

tolerance, scalability. Spark has an advanced DAG (Directed Acyclic Graph) execution model that supports in-memory processing. Lazy evaluation of RDDs helps for optimization of overall data processing work. Spark also provides an interactive shell and it has rich API for Scala, Java, and Python which are especially great for data science environment.

RDDs let users to explicitly store data on disk or in memory, to control its partitioning, and to manipulate it using a rich set of operators. Spark RDD implements lineage-based fault tolerance with negligible overhead as well as optional dataset replication. Once a failure happens the lineage graph along with the input data can be used to reproduce the lost RDDs. Each RDD remembers the graph of operations used to build it. RDDs provide an interface which is based on coarse-grained transformations that applies the same operation to many data elements. This permits them to provide fault tolerance by logging the operations used to build a dataset rather than the actual data. An RDD has enough information about how it was derived from other datasets. This will help to compute its partitions from data present in stable storage in case of the worst case scenario when all its preceded datasets are failed. This is a more efficient way of providing fault tolerance than duplicating the data over the network.

From the memory perspective, the fundamental difference between RDDs and Distributed Shared Memory (DSM) [37] is that RDDs must be created through coarse-grained transformations, while DSM permits fine-grained reads and writes to each memory location. Besides, only lost partitions of an RDD need to be recomputed upon failure by parallel re-computing on different nodes, without having the checkpoints to roll back the whole changes. Further, RDDs are immutable in nature this lets a system predict slow nodes and resolve them by running backup copies of slow tasks. Spark RDDs runtime can also schedule tasks based on data locality to improve performance.

5.5 Discussion: MapReduce, Spark, and Flink

MapReduce [38] is a fundamental data processing engine used in the Hadoop environment. MapReduce design incorporates the Map function and the Reduce function which handles the data computations. It is parallel data processing framework for which traditionally been used to run the map and reduce jobs. Map function regroups the input data into independent key-value pair partitions. Then, the engine will send all the partitioned key-value pairs into the Mapper which processes each of them exclusively. These parallel mapper tasks are launched throughout the cluster. Then the Mapper outputs one or more intermediate key-value pairs. For each unique key, the Reduce function aggregates the values associated with that key. The final output generated then is written back to HDFS.

MapReduce is the most mature and it executes tasks where data is located. This reduces a significant network communication load between cluster nodes. MapReduce operations are not always optimized for I/O efficiency [39]. MapReduce is inefficient in running iterative algorithms and is not designed for iterative processes. Mappers read the same data again and again from the disk. Whereas, Spark was initially designed to run on top of Hadoop and it is an alternative to the traditional batch MapReduce model. Spark can also be used for the fast interactive, iterative queries and real-time streaming data processing.

Spark stores data In-memory on the other hand MapReduce stores data on disk which incur additional network I/O latency. Hadoop MapReduce uses duplication of data on disk to accomplish fault tolerance whereas Spark uses a new storage abstraction called Resilient Distributed Datasets (RDD) to ensure fault tolerance. Spark uses more RAM and in case of data spill to disk, its disk I/O is relatively fast as compared to MapReduce. But as Spark demands a huge RAM it needs a dedicated high-end physical machine for its data processing. The amount of time and code required for writing a Spark job is significantly less than writing an equivalent MapReduce job. Executing Spark processes does not limit to only Hadoop YARN cluster. Spark has its own standalone cluster resource manager. Moreover, Spark can also run on Mesos clusters.

Apache Flink [40] is an open-source stream processing framework for distributed, high-performing, and accurate data streaming applications. Flink's pipelined runtime system enables the execution of batch and stream processing programs [41]. Flink designed as fault-tolerant and its core is written in Java and Scala. Flink also supports the execution of iterative algorithms natively [42]. Flink is among the young projects and currently proving in a production environment. It has a growing group of committers and it does not offer stable solution libraries for graph processing and machine learning. Gelly [43] is a Graph API for Flink which contains a set of methods and utilities to simplify the development of graph analysis applications in Flink. Flink-ML [44], a machine learning library is in development, it needs more research into its viability.

Flink and Spark are both general-purpose data processing platforms and top-level projects of the Apache Software Foundation (ASF). They have a wide range of applications when dealing with Big-data scenarios. However, the way they are particularly

specialized may differ. However, Flink is also a strong tool for batch processing. Spark, on the other hand, is based on resilient distributed datasets (RDDs). This in-memory data structure gives the power to Spark's functional programming paradigm. It is capable of big batch calculations by pinning memory. In fact, the use-cases of Spark and Flink overlap a bit. However, the technology used is quite different. Flink shares a lot of similarities with relational Database Management Systems. Data is serialized in byte buffers and processed a lot in binary representation. This also allows for fine-grained memory control. Flink uses a pipelined processing model and it has a query optimizer that selects execution strategies and avoids expensive partitioning and sorting steps.

Flink is a true streaming engine instead of micro-batch processing in spark, it processes the data streams as true streams, i.e., data elements are immediately "pipelined" through a streaming program as soon as they arrive there. This allows it to perform more flexible window operations on streams. While Spark's strategy of micro batching the streams may have a little lag associated with it in capturing results. Flink designed to compatible with YARN cluster so that it allows already existing jobs to run directly on it and it has better integration with other projects. Flink avoids memory spikes which were typically seen in a Spark by better managing its own memory resources. Flink enables iterative processing to occur on the same nodes instead of having the cluster to run each iteration independently.

MapReduce is getting outdated in the cluster computing environment and it is not suggested for the mixed variety of applications due to its high latency and lack of support for iterative algorithms. But in the case where we may not afford enough RAM for cluster then MapReduce supports larger datasets on disk. Nonetheless, several improvements are proposed and these are in-progress for Hadoop MapReduce upcoming releases. Modern MapReduce designed to optimize the iterative support of the MapReduce framework.

If real-time solutions are of importance, one may wish to consider Flink since they offer genuine stream processing. Flink offers the best with a combination of batch and true stream processing. On the other hand, Most of the real world streaming issues could be solved adequately with micro-batch type streaming that Spark offers. Spark's maturity, established community and support for in-memory fault tolerant data make it a better choice for nearly every type of workload. There are also new projects are growing up to challenge both Spark and Flink in the context of general purpose computing.

6. Batch Processing - MapReduce, Tez, and Spark

Batch processing is a type of processing model where it executes processing logic in the form batches and each batch contains a collection of jobs. Batch processing excels at processing large amounts of stored and stable data. Based on the size of the data being processed and the computational power of the system, execution result may differ. However, it generally incurs a relatively high-latency and is unsuitable to process real-time incoming data. These batch processing models are just the general case, rather than a special type of processing. Batch processes will run its program only once for the set of related jobs. Processing engine expresses a computation as a data flow graph and executes them. Directed Acyclic Graph (DAG) is the generalization of this type model. Most of the models differ the way how it expresses these Directed Acyclic Graphs (DAG).

6.1 MapReduce

As discussed earlier MapReduce is the Batch Processing System of Hadoop ecosystem. It emphasizes on the volume of the data which is typically stored on disks. The DAG expressed in MapReduce often results in multiple MapReduce jobs which can potentially harm latency for short queries. For large-scale queries, MapReduce has much overhead for materializing intermediate job results.

6.2 Tez

Apache Tez, an open-source framework designed to build data-flow driven processing runtimes [45]. With Tez, we introduce a processing engine which can express a complex DAG of tasks for building an application. It is currently built atop Apache Hadoop YARN cluster. Tez intentionally designed as a raw and expressive tool which is meant to be controlled by compiler developers. The main focus of Tez so far has been providing a faster interactive analytics engine for Hadoop's traditional data-processing languages such as Apache Hive [46] and Apache Pig [47]. Tez allows interactive batch execution and its DAGs are highly customizable by upper-level querying frameworks.

6.3 Spark

Spark library is a batch processing system at its core. Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing [48]. Spark SQL is Spark's module for Batch processing over structured data [49]. Spark has a

batch execution engine of clean user API with a rich set of operators and support for high-level querying language such as Spark SQL.

6.4 Discussion:

Spark and Tez are both DAG frameworks are more flexible than MapReduce with less overhead. Tez built on YARN cluster with the MapReduce experience. Tez is best suited as a model to build abstraction frameworks over it, instead of building applications directly using its API. Spark has these both capabilities. One major difference is that Spark can run as a standalone or on top of Hadoop YARN or on Mesos while Tez can only run on top of YARN. While Tez is an assembly language at its very core. Pig and Hive already have compilers into the Tez backend. Spark also claim to support Pig and Hive.

In a nutshell, Spark is a general purpose engine with APIs for mainstream developers, while Tez is a framework for purpose-built tools such as Hive and Pig. If Tez comes with matching the version of Hive or Pig, we can use it as the backend execution engine over MapReduce. If we are planning to directly use the APIs, whether to write a data-transformation job, implement a distributed machine learning algorithm, or write your own higher-level data processing language, it is efficient to use General purpose Spark. In terms of performance Spark expected to have up to 100× better performance than Hadoop MapReduce [48]. While Hive which runs over compatible Tez version has significant performance improvements than Spark.

7. Streaming - Strom, Flink, and Spark streaming

Stream processing is a real-time execution engine where it processes data as it arrives [50]. Stream processing can be described as a continuous collection of the data and processing of the data instantly in real time. Processing engine applies the business logic to each transaction that is being arrived while program execution itself, rather than store all the events and process them later. It emphasizes the Velocity of the data. Stream processing is the best fit choice when sub-second latencies matters in real time. Most of streaming styles will enable state-full processing, which comes in handy in facilitating fault recovery. This also allows it to replay any lost data.

In stream processing, “message delivery” defines on how messages in a stream are sent and received while. Message delivery semantics means that for each message handed to the mechanism, whether the message is delivered zero or more times. These are categorized as At-least-once delivery, At-most-once delivery, and Exactly-once delivery semantics. Based on the requirements we choose either there is importance on processing *every* single record or is some nominal amount of data loss acceptable.

7.1 Storm

Apache Storm is a special-purpose, distributed, real-time computation engine [51]. The core Storm engine follows stream processing model. Apache Storm is a data stream processor without batch capabilities. Strom is a task parallel continuous computational engine. The way it was modeled differently from conventional MapReduce and other course-grained designs. Its fine-grained transformations are giving flexibility while designing processing topologies.

Storm designed based on concepts of tuples, spouts, and bolts. A tuple is basically what your data is and it is the core data structure in the storm. Spouts are the sources for these tuples to arrive. Spouts communicate with incoming data tuples, API calls, and other ingestion systems.
Bolts are used for the execution of the business logic over the tuples. Storm defines its workflows in Directed Acyclic Graphs (DAG) topologies which are directed graphs of spouts and bolts. Storm does not natively run on top of typical Hadoop clusters but it can still consume files from HDFS and write files to HDFS.

7.1.1 Storm Trident [52] is an extension of Storm project. Trident additionally provides a high-level abstraction for doing real-time state-full stream processing on top of Storm. Trident provides high-level data operators like joins, filters, aggregators, grouping, and functions. The core data in Trident is referred as "Stream", each stream is processed as a series of batches. Since trident processes, messages in batches throughput time could be longer.

7.2 Flink:

Flink executes programs as pipelined fault-tolerant dataflow [54]. It is a true streaming engine, which process data in exact sense real time instead of near real time. The core of Apache Flink is a distributed streaming dataflow engine.

Flink Streaming designed based on concepts of data source, transformations, and data sink. The data source is the incoming data that Flink processes, Transformations are the processing step and Data sink is where it sends data after processing.

Flink's DataStream API contains programs that implement transformations to parallel data streams. Flink is optimized for cyclic or iterative processes by using iterative transformations on collections. Moreover, Flink features a special kind of iterations called delta-iterations that can significantly reduce the number of computations as iterations go on.

7.3 Spark Streaming

Spark Streaming makes it easy to build scalable fault-tolerant streaming applications [53]. Spark Streaming built on top of Spark core for performing the micro-batch streaming analysis.

Spark Streaming designed based on concepts of receivers, Dstreams, and transformations. The receiver is the sources for the data to arrive. The received data is then placed into a Spark RDD. Discretized Stream (DStream) is a stream of these data and they are implemented as a sequence of RDDs. The transformations are the functional and computational abilities of Spark core. Spark Streaming operates by scheduling data streams into micro-batches. It collects all data that arrives from the receivers within a specific time and runs a regular batch program on the gathered data. While the batch program is running, the data for the following mini-batch is collected. Micro-batching is a special case of batch processing where the batch sizes are smaller. Micro-batching incurs a cost of latency due to windowing time to accumulate the data.

7.4 Additional Tools

Apache Kafka [55] is a real-time, fault-tolerant, a scalable messaging system for moving data in real time. It is a better choice for use cases like capturing user activity on websites, logs, stocks, and instrumentation data.

Apache Kafka was originally developed by LinkedIn [56] to serve as the foundation for their activity stream and the data processing pipeline.

Apache Samoa is a platform for mining Big data streams [57]. SAMOA has a focus on giving users the necessary tools to so that users can create his or her own implementations of algorithms on several stream processing engines.

7.5 Discussion

All these frameworks provide real-time processing of data. Apache Spark is a matured project whereas Apache Storm, Flink is currently evolving. Apache Storm is a task parallel continuous computational engine. Apache Spark is a data parallel general purpose batch processing engine. Flink's pipelined engine internally looks a bit similar to Storm, i.e., the interfaces of Flink's parallel tasks are similar to Storm's bolts. One kind is a true stream processing engine that can also do micro-batching, the other is a batch processing engine which micro-batches but cannot perform streaming in the strictest sense. One processes the data in real time and another in near real time.

If sub-second latency is the focused parameter, micro-batching will not be sufficient. Then Storm, Flink work much more easily and with fewer restrictions than Spark Streaming. On the other hand, micro-batching guarantees give state-full computation, making windowing an easy task. Storm and Flink have in common that they aim for low latency stream processing by pipelined data transfers. However, Flink offers a more high-level API compared to Storm.

Core Storm offers At-most-once message delivery [58]. On the other hand, it will be difficult to achieve Exactly-once processing in the case of Storm. Hence Trident will be useful for those use-cases where you require exactly once processing [59] and it makes stateful processing easier. But trident adds complexity to a Storm topology, lowers performance.

Apache Flink has also become a viable option to consider, as it is a streaming-first processing engine with star performance with exactly-once processing models. Spark trivially yields perfect for Exactly-once message delivery. In Spark, we can replay RDD data while replicating over the persistent file systems. If we are in need of strict, stateful processing, we may go for Spark Streaming for their exactly-once semantics.

Storm and Kafka combination is promising for true stream processing, and they are already in use at a number of high-profile companies. When paired together, you get the stream, you get it in real time, and you get it at linear scale.

8. Machine Learning - Mahout, H20, and MLlib

Learning may be described as the process of improving one's knowledge to perform a task efficiently. Machine learning is a field of computer science that gives computers the ability to learn without being explicitly programmed [60]. Machine learning is a discipline of artificial intelligence focused on pattern recognition and computational learning theory. In modern days, machine learning has been deployed in a wide range of applications, where designing efficient

algorithms and programs becomes rather difficult. Machine learning can be done over batch processing or stream processing and it is commonly used to make high-end predictions on data based on previous outcomes. Machine learning is prominently used in application fields such as email filtering, search engine improvement, digital image processing, data mining etc.

There is a range of open source machine learning processing models available which enable engineers to build, implement and maintain machine learning systems. We can also build new projects to create impactful machine learning systems.

8.1 Apache Mahout

Apache Mahout [61] is an open source project of the Apache Software Foundation to build distributed or scalable machine learning algorithms. Apache Mahout is deployed on top of Hadoop using the MapReduce engine. Mahout provides Java libraries and Java collections for most of the complex mathematical operations. Once large data is stored on the Hadoop Distributed File System (HDFS), Mahout provides the data science algorithms to find meaningful patterns, hidden useful information form these Big data sets.

Apache Mahout does not restrict its contributions to only Hadoop based implementations. It offers more. It is primarily focused on algorithms such as collaborative filtering, clustering, and classification and has been shown to scale well with the increases in the size of the data [62].

Collaborative filtering: These Mahout algorithms mine the user behavior and build product recommendations. Mahout provides tools for building a fast and flexible engine recommendation engine.

Clustering: Inbuilt algorithms organizes items into naturally occurring groups such that items belonging to the same class are similar to each other. Mahout Support prominent clustering algorithms such as k-Means, Mean-Shift, Dirichlet etc [63].

Classification: Mahout Categorises content from existing. It learns and then assigns unclassified items to the best category, by using the simple Map-Reduce-enabled naïve Bayes classifier [63].

8.2 Spark MLlib

Apache Spark MLlib [64] is regarded as a distributed and scalable machine learning machine learning library on top of the Spark Core. Spark Mlib can be invoked from both Scala, Java, and Python [65]. Common machine learning and statistical algorithms have been implemented in MLlib such as summary statistics, correlations, hypothesis testing, random data generation, etc. which are listed at Spark webpage [66]. Remaining models can be learned offline and applied online to new streaming data. It comprises common learning algorithms and utilities as well as lower-level optimisation primitives and higher-level pipeline APIs.

Classification and Regression: Supports vector machines, logistic regression, linear regression, naïve Bayes classification [66]. Collaborative filtering techniques including Alternating Least Squares (ALS). Cluster analysis methods including k-means and Latent Dirichlet Allocation (LDA). Optimisation algorithms such as stochastic gradient descent and limited-memory BGGS. Streaming execution of Logistic Regression, Linear Regression, and k-Means Clustering are included in spark.

Recently proposed machine learning algorithms which are to be listed in addition to existing with upcoming versions of MLlib.

8.2.1 ML pipelines[67] is an MLlib core package designed on top of Data Frames that has tools for dataset transformations to handle learning process for extracting features. It represents a pipeline as a sequence of dataset transformations.

8.2.2 MLbase [68] is a layer which wraps Spark MLlib and other projects which makes machine learning on data sets of all sizes available to a wide range of users.

8.3 H20:

H2O is an open source, in-memory, distributed, fast, and scalable machine learning and analytics platform on Big data and provides easy customization to an enterprise environment [69]. H2O allows users to fit thousands of potential models as part of discovering patterns in data [70]. It is a Java library API and Programming in H2O is possible with R, Python, Scala and JSON.

It supports Multi-node clusters and in-memory computation. As we discussed In-memory computations facilitates faster speed and accuracy. H2O designed on top of the data frames which are a collection of vectors. Its columns are distributed across nodes. Each node able to see the entire dataset. It is implemented with a key-value store which is a classic peer to peer distributed hash table [71].

H2O is a web-based model and Its Graphical User Interface is compatible with all leading browsers. It Supports a wide range of algorithms and is extensible to add and model new machine learning algorithms. Users with minimal programming knowledge can still utilize this framework using the web-based UI.

The most promising features of this are that it provides a handset of tools for Deep Neural Networks. Deep Learning is a modern research area of machine learning, making it a promising feature of H2O [72]. There are many offerings for deep learning but it is targeted towards business instead of research, whereas H2O targets both. While H2O community offers an enterprise edition with support, many of their offerings are available to open source so that can be used without purchasing a license.

8.4 Flink-ML

FlinkML [73] is a machine learning library currently in development for the Flink platform.

8.5 Oryx

Oryx [74] does not offer a broad selection of algorithms, but still covers the major areas of classification, clustering, and collaborative filtering for real-time large-scale ML. Oryx is also stand-alone, special-purpose machine learning engines.

8.5 Additional Tools

Samsara is a Linear Algebra library for Mahout which includes statistical operations, and data structures [75]. The objective of the Mahout-Samsara project is to enable users to build their own distributed algorithms, instead of a library of pre-written implementations.

8.6 Discussion

These frameworks enable machine learning analysis in a distributed environment. These can be built on top of a general-purpose framework, such as Spark MLlib, or as a special-purpose framework, such as H2O. Both H2O and Spark use distributed data frames.

MLlib is a Spark subproject that uses Apache Spark as the underlying framework. Mahout uses more common Hadoop MapReduce as the underlying framework. MLlib is implemented using Spark's iterative batch and streaming approaches. MLlib based on in-memory computation which enables jobs to run significantly faster than those using Mahout [76]. Overall MLlib will be faster than Mahout as it is built on Apache Spark, but Mahout is more stable for Apache Hadoop machine learning environment. Mahout includes the most options for the recommendation and has more maturity than the others. Mahout knew for having a wide selection of robust algorithms whereas it has inefficient runtimes due to the slowness of disk-based MapReduce engine. Mahout was falling out of favor but this may change due to design modifications made in the latest versions of MapReduce. MLlib strength lies in its commercial backing, programming languages and matured community but its algorithm implementations are young and recent.

MLlib and H2O are available options for speed and scalability. Both have a reasonable choice of algorithms, but H2O has additional solutions for deep learning. In terms of ease of use, both have APIs for programming in multiple languages, and H2O also offers a GUI. They also incorporate APIs for development in Scala, Java, and Python. Since H2O comes as a bundle with most of the configurations already tuned, set up is easy and requiring lesser learning curve than other open source alternatives. While H2O maintains their own processing engine, they additionally offer integrations which allow the use of its models on Spark and Storm.

9. Graph Processing - Pregel, PowerGraph, and GraphX

Graphs become the powerful abstraction mechanism for representing many relationships types of complex data. Graphs are standard for depicting complex relationships, interactions, and interdependencies. Internet, Maps, Logistics chains, Social networks, and other many entities have been extensively connected themselves as large interconnected graphs.

Graphs are everywhere and continuously growing in size but to processing them efficiently remains challenging. Analysing this immense useful information using classical graph algorithms can be difficult due to their huge sizes. Recently, a bunch of distributed graph processing frameworks has emerged. In general, distributed graph processing algorithms are iterative and the way of traversing the graphs may differ.

9.1 Pregel

Pregel [77] is the first vertex-centric large-scale distributed graph processing framework. The vertex-centric computing engine makes the design and execution as scalable. Pregel vertex-centric computations adopt Valiant's Bulk Synchronous Parallel Model [78]. Bulk Synchronous Parallel (BSP)

is a parallel programming model that uses a message passing interface (MPI) for parallelizing jobs across multiple machines [79].

Its built-in library divides a graph into partitions, each partition consisting of a set of vertices and all of those related outgoing edges. In a large cluster, Pregel distributes the vertices to different nodes in such a way that each vertex in a graph receives messages from its incoming neighbors, performs the user specified processing and then sends messages to its outgoing neighbors. The partitioning function is just a hash of the vertex with a number of partitions however users can also customize this function. Persistent information is stored as files on a distributed storage system, whereas temporary data is placed on the local disk.

9.1.1 Giraph

Giraph is an open source Hadoop implementation of Pregel. Giraph library runs on top of Hadoop, is used by the Graph Search Service which can process graphs ranging trillion edges. Further improved the performance of Giraph by recent optimizations in Multi-threading, Memory optimization. Giraphx: An extension of Giraph that additionally brings serializability and direct memory reads to Giraph.

9.2 PowerGraph

PowerGraph: [80] is an Edge-Centric Distributed Graph-Parallel Computation model for Natural Graphs and its operations design also based on Gather, Apply and Scatter (GAS). PowerGraph Model derived from real-world natural graphs in which typically many real-world graphs have vertices that follow power-law degree distributions. Graphs with Power-law degree contains a small subset of the vertices connects to a large part of the graph. It emphasis to address the imbalanced workload due to high degree vertices in power-law graphs.

PowerGraph introduces a vertex-cut graph partitioning scheme to handle natural graphs. By partitioning out the edges, the total data and network overhead of each vertex will spread across but it requires additional memory for storing the vertex mirrors, which is large especially when the vertex value is of huge size. This partitioning scheme divides the vertex set in a way such that the edges of a high-degree vertex are handled by multiple workers. PowerGraph eliminated the degree dependence of the vertex by GAS decomposition of vertex over edges.

9.3 GraphX

GraphX [81] is a distributed graph engine built on top of Spark which aims at processing of graphs and graph-parallel computations. At a much higher level, GraphX extends Sparks Resilient Distributed Dataset (RDD) by introducing a new Graph abstraction the Resilient Distributed Graph (RDG) [82]. This new Graph abstraction is a directed multigraph with properties attached to each vertex and edge. GraphX design has a vertex collection containing the vertex properties uniquely represented by the vertex identifier. It includes an edge collection containing the edge properties keyed by the source and destination vertex identifiers. GraphX new property graphs can be designed by combining different vertex and edge properties.

GraphX provides a collection of powerful computational operations. For example, these operators can be subgraph, join vertices and aggregate messages, additional optimizations. GraphX performs very sparse join operations against datasets by keeping them in a hash table. This hash table containing multiple graph records within each RDD record. This design will enable fast lookups of specific vertices or edges when joining with new data.

GraphX implements the Gather Apply Scatter (GAS) Model decomposition, this enables vertex-cut partitioning, enhanced work balance, and minimal data movement. GraphX includes a growing collection of graph algorithms and functions to simplify graph analytics tasks and it also exploits the graph structure to minimize network and storage overhead.

9.3.1 Titan

GraphX has no real persistence layer, it can persist to HDFS files, but it cannot persist to a distributed data store in a common schema. A graph has an only way to exist in GraphX when it is loaded off into memory from raw data and interpreted as Graph RDDs, Titan provides a way to store the graph permanently. By default, GraphX solves queries via distributed processing on many nodes in parallel where possible as opposed to Titan processing pipelines on a single node. Titan can also take advantage of parallel processing via Faunus/HDFS if necessary. They are both exceptionally powerful in their own right and open up graph processing to other data stores such as Cassandra stored data.

They have two different interpretations of how to process graphs at scale and are fantastic query layers for NoSQL systems.

9.4 Flink Gelly

Gelly is a Graph API for Flink to process large graphs by distributed processing [83]. In Gelly, graphs will be updated and transformed using high-level functions. Gelly provides methods to create, transform and update graphs along with the library of graph algorithms. Graph Transformations are implemented with the methods such as map, filter, reverse, union, and difference. In Gelly, each Graph is designed by a DataSet of vertices and a DataSet of edges. The Graph nodes are represented by type Vertex. Each Vertex is given by a unique ID and a value. Vertex IDs will implement the Comparable interfaces.

9.5 Discussion

Initial Graph processing frameworks just enabled graph processing capabilities on Hadoop. Later they built on top of a general-purpose framework or as a stand-alone or a special-purpose framework.

MapReduce is a general purpose engine for handling Big data. But it is not reasonable to process large graphs because of its high iteration costs, excessive disk I/O and complex representation for graph algorithms. A framework that specially designed only for graph processing will increases performance and usability compared to general frameworks since general-purpose distributed computing tools are not tailored for graph processing. Most of these kind graph frameworks adopt Pregel's vertex-centric computing model, while different techniques have been proposed to match the limitations in the Pregel framework. For example, Pregel's implements data-pushing model and message passing paradigm technique while PowerGraph implemented Gather, Apply, Scatter (GAS) data-pulling model using shared memory abstraction. Giraph generally has the poorer execution performance since it does not utilize any particular technique for handling the skewed workload. PowerGraph provides better performance for natural graphs. Apache Flink Gelly, it allows us to pre-process graph data, process graphs and transform result graphs using one system and one API. In general vertex-centric programming interfaces is a better fit for graph problems than the general-purpose interfaces.

A distributed graph computation framework on Spark can emulate both Pregel and PowerGraph. Spark is significantly different from the other graph processing frameworks in which it attempts to bridge the gap between distributed data flow frameworks like Hadoop and graph only processing abstractions like Pregel.

Pregel accomplishes fault tolerance by deploying checkpointing, PowerGraph also takes snapshots of the data graph. As specified before, GraphX is built on Spark, which implements lineage-based fault tolerance with negligible overhead as well as optional dataset replication where Flink's Gelly is young and evolving fast. In comparison with other large-scale graph processing systems, data replication and re-computation in GraphX are relatively faster than completely restarting after a failure. This is primarily because of the performance overhead in conjunction with the little incentive to implement recovery due to small mean times to failure of modern hardware.

10. Framework Abstraction

This abstraction layer is built on top of all processing models to make the data processing easy and more interactive. These tools allow the user to process data using a higher level abstraction. These wrappers provide a better environment and make the code development simpler since the programmers do not have to deal with the complexities of underlined processing engines while coding. This layer typically contains Scripting languages, Query inspired languages and custom Domain Specific Languages (DSL).

Framework abstractions are highly recommended since we do not have to code all of our jobs using the low-level APIs of underlined processing engines. It saves considerable efforts by not having to implement common processing tasks. Coding directly on the engines API leads to re-write our code if we decided to change the execution engine. While in framework abstractions we can replace processing engine at any given time. We can also continue using our existing visualization and BI reporting tools to analyze and report Big data. These frameworks enable support for the REST kind of APIs where web-based applications can talk directly to data over these APIs. It enhances the range of connectivity options.

Abstraction frameworks also enable User Defined Functions (UDFs) which provides the ability to create custom functions for databases. An efficient framework should have the ability to allow these custom functions and an underlined engine should be

able to parallelize the execution of User Defined Functions across all the nodes.

In case of NoSQL is considered, schemas for data write are not mandatory whereas in data analysis schema on data read is an essential factor. Abstraction frameworks can also be implemented as a familiar Structured Query Languages (SQL). This allows the bigger set of enterprise users to query, analyze and relate the data using traditional SQL Queries. Currently, almost all the SQL-on-BigData solutions around are SQL-Like (Partial SQL capabilities) and are NOT True SQL (ANSI SQL standards capabilities). Various Framework Abstractions are available in cluster computing domain. We can choose suitable abstraction based on the processing engine and the expertise.

10.1 Hive

Apache Hive [84] is a data warehousing software facilitates querying and managing large datasets residing in distributed storage. This was one of the first engines which designed by Facebook [85] to make users familiar with SQL query the data in a Hadoop environment. SQL-like query language called HiveQL [86] that represents data in the form of tables and Hive programming is similar to database programming. While Hive QL execution SQL-like queries are implicitly compiled into map-reduce jobs that are processed as batch jobs. HiveQL additionally enables users to access and manipulate Hadoop-based data stored in HDFS or HBase. MapReduce manages data within HDFS files while in contrast, Hive enables to represent data in a structured database that is well-known for users. Hive lacks full SQL support but it has schema-on-read and supports multiple schemas for different applications. Hive can run on top of MapReduce, Tez, and Spark.

Its Built-in User Defined Functions to manipulate dates, strings, and other data types. Hive supports extending the UDF set to handle use-cases not supported by built-in functions. Hive having built-in indexing to provide query acceleration. Hive can read the data from different storage types such as plain text, RCFile, HBase, and others. It's Metadata storage in an RDBMS which significantly minimizes the time to perform semantic checks during query execution. Hive low-latency operations are not efficient for real-time transactions. Since it built for large-scale processing over Hadoop, the benefit here is that it loads faster but even small jobs may take time.

10.2 Pig

Apache Pig is a platform for analyzing large data sets. Apache Pig consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs [47]. It is an open source framework that generates a high-level scripting textual language called Pig Latin. Apache Pig is a SQL-like environment initially developed at Yahoo, later being used by many organizations. Pig's infrastructure layer consists of a compiler that produces sequences of Map-Reduce programs. Pig Latin [87] is a script based language which is designed in an interactive environment to enable easy development of MapReduce jobs and workflows. It reduces the development time by supporting parallel execution of MapReduce tasks and workflows on top of Hadoop. Pig also permits interaction with external programs such as shell scripts.

Pig Latin is like a programming language with scripts similar to a data flow Directed Acyclic Graph (DAG). In contrast to SQL-like, Pig does not require a schema and it can process semi-structured and unstructured data. Its schema can be optionally defined at runtime. It supports more data formats than HiveQL. Pig can execute on both the local environment in a single JVM and the distributed Hadoop cluster environment.

10.3 Spark SQL

Spark SQL is primarily designed to enable developers to incorporate SQL statements into Spark programs[88]. Spark SQL primarily designed on the concept of Data Frame [89]. Spark RDD abstraction takes files/raw data from the Distributed File System. Data Frame consumes those RDD and forms tables, implements all table operations. These Data Frames are looks like database tables. Spark SQL can also use its Data Frame over other available data sources.

In the Spark SQL, we keep multiple database records in one Spark record to apply optimizations. To store and process relational data efficiently, Spark implemented in-memory columnar storage and compression. This reduced both the data size and the processing time to optimize SQL queries. Spark SQL can re-optimize a running query after running the first few stages of its task DAG, choosing better strategies or the right degree of parallelism based on observed statistics.

10.4 Drill

There are several special-purpose SQL engines aimed at faster SQL which also includes Apache Drill [90]. Drill is a native massively parallel processing query engines on read-only data. Drill make large-scale, ad-hoc querying of data with lower latencies that are especially suitable for data exploration. They make it possible to scan over petabytes of data. Drill is the

open source version of Google's Dremel system [91]. To its advantage Drill uses schema-free document model similar to MongoDB so that it can query non-relational data easily. The Drill is an independent execution engine and is targeted only at business users, analysts, data scientists, and developers. Drill supports a variety of non-relational data stores in addition to Hadoop. Drill can discover metadata dynamically and does not have to use Hive's meta store. However, Drill is comparatively less mature and still lacks some of the analytical functions that others offer.

11. Data Rendering

This Data display layer renders the processed information by using graphs, tables, charts and other data relationship rendering techniques. Data collections from the various execution results are depicted here which primarily provides a single, comprehensive view for decision makers. These management dashboards are easy, efficient, and user-friendly to make an effective decision from the historical data trends. Data analysis, visualizations, and data reporting are typically focused here.

12. Additional Tools

12.1 Data Ingestion

12.1.1 Flume [92] is a distributed software that is designed to collect, aggregate and reliable transfer of data from external machines to distributed file systems such as HDFS. It has an extensible data model, flexible architecture to handle massively distributed data flows. It provides various use full features including fault-tolerance. Flume highly supports Hadoop, however, it is an independent component that can work on other platforms. It is known for dataset sink which uses API to stream the data from various volume sources into data stores such as HDFS and HBase. In addition, it has a query processing mode which can transform data before it is pushed to the mentioned sink.

12.2 Coordination and Workflow

12.2.1 ZooKeeper

ZooKeeper [93] is an effort to an open source service which designed to coordinate applications and clusters in a distributed environment. ZooKeeper focused on building fast and reliable distributed systems. It simplifies the development process for making it agiler and more robust implementations. While deploying the distributed applications in a production environment, different implementations of the services lead to management complexity. In such cases, its centralized service gives us the tools for naming, maintaining configuration information, providing distributed synchronization and providing group services. Several Hadoop projects such as HBase, Storm, Kafka are already using ZooKeeper to coordinate the cluster and provide highly-available distributed services. It has several use cases, for example, Mesos uses ZooKeeper service to coordinate and elect a new master in case of master failure. The ZooKeeper Failover Controller is responsible for HighAvailable monitoring of its service to avoid a single point of failure.

12.2.2 Apache Ambari

Apache Ambari [94] project is aimed at making Hadoop management simpler by developing software for provisioning, managing, and monitoring health Apache Hadoop clusters. Ambari web user interface provides a step-by-step wizard to install and manage Hadoop services. Ambari provides an intuitive dashboard, easy-to-use Hadoop management backed by its RESTful APIs.

12.3 Software Distributions

Distributed computing has a range of offerings which includes a mixture of solutions packaged into the form of distributions such as Hortonworks [95] Cloudera [96] and MapR [97]. These are one stop solutions shipped along with the online support for distributed computations. They has both non-commercial and commercial offerings. Generally commercial offerings built specifically to meet enterprise demands which incurs additional licensing costs for end to end support. Whereas open source offerings are for community and restricted for commercial deployments.

13. Research Scope and Future Work

We provided qualitative details based on our exposure to each project and related works. In this survey, we also provided relative comparisons based on comprehensive literature available over online documentation. Big data by itself is a complex domain and has many constraints which needs further attention. This also throws us the opportunity for future research. Future work will include quantitative comparisons of discussed models based on formally defined criteria. Research scope identified on centralized cluster resource manager that scales geographically across multiple datacenters. The future work also involves contributing stable changes to generic processing model such as Spark. We continue this work further by designing an interface which can

be useful for any generic processing engine to move in a direction for efficient distributed computing model.

14. Conclusion

In this work, we analyzed the distributed data processing complete technology stack along with their ecosystem by putting them into a layered architecture. We discussed recent projects in each of these layers and highlighted some design principles. Major approaches to distributed processing such as batch, iterative batch, and real-time streams were described and related insights were presented and discussed. In this survey, we primarily focused on the comprehensive review of data processing engines that are currently available. Besides, we have focused on the components of the generic processing engine. The intention of this survey is neither to endorse nor to be judgmental for one particular project but rather to compare and explore them briefly. Choosing the models purely based on finding the best balance between the computational requirements. Most of these projects have found ways to co-exist complimenting each other to create a unique open source environment for innovative product development in the Big data distributed computing domain.

Acknowledgements

I must first express my deepest gratitude to my guide, Dr. Suresh C. Gupta, for his excellent guidance, support, attention to detail and for instilling in me confidence whenever needed. I would like to thank him for sharing with me his knowledge and experience for further research. We thank the reviewers for giving us many constructive comments, with which we have significantly improved our paper.

References

[1] Vinod K. Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In SoCC, 2013.

[2] Benjamin Hindman, Andrew Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the datacenter. In NSDI, 2011.

[3] Apache Hadoop. http://hadoop.apache.org/

[4] Murthy, Arun (2012-08-15). "Apache Hadoop YARN – Concepts and Applications". hortonworks.com. Hortonworks. Retrieved 2014-09-30.

[5] Zaharia, Matei. "HUG Meetup August 2010: Mesos: A Flexible Cluster Resource manager - Part 1". youtube.com. Retrieved 13 January 2015.

[6] Myriad Home. https://cwiki.apache.org/confluence/display/MYRIAD/Myriad+Home

[7] NFS. http://en.wikipedia.org/wiki/Network_File_System

[8] S3. http://aws.amazon.com/s3/

[9] HDFS. http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html

[10] Borthakur D. HDFS architecture guide. HADOOP APACHE PROJECT.2008.

[11] Amazon Simple Storage Service. http://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html

[12] DB-Engines. https://db-engines.com/

[13] Column-oriented DBMS. https://en.wikipedia.org/wiki/Column-oriented_DBMS

[14] Apache HBase. http://hbase.apache.org/

[15] Apache Cassandra. http://cassandra.apache.org/

[16] Apache Cassandra. https://en.wikipedia.org/wiki/Apache_Cassandra

[17] Key-value database. https://en.wikipedia.org/wiki/Key-value_database

[18] Redis. http://redis.io/

[19] Memcached. https://www.memcached.org/

[20] Redis. https://en.wikipedia.org/wiki/Redis

[21] MongoDB. https://www.mongodb.org/

[22] Apache CouchDB. http://couchdb.apache.org/

[23] Apache CouchDB. https://en.wikipedia.org/wiki/CouchDB

[24] Erlang. https://en.wikipedia.org/wiki/Erlang_(programming_language)

[25] Neo4j. http://neo4j.com/

[26] Titan Distributed Graph Database. http://thinkaurelius.github.io/titan/

[27] Neo4j. https://en.wikipedia.org/wiki/Neo4j

[28] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Eric Baldeschwieler, Scott Shenker, and Ion Stoica. Tachyon: Memory throughput I/O for cluster computing frameworks. In LADIS, 2013.

[29] J. Ousterhout, P. Agrawal, D. Erickson et al., "The case for ramclouds: Scalable high-performance storage entirely in dram," OSR, 2010.

[30] Hao Zhang, Gang Chen, Member, IEEE, Beng Chin Ooi, Fellow, IEEE, Kian-Lee Tan, Member, IEEE, Meihui Zhang, Member, IEEE. In-Memory Big Data Management and Processing: A Survey

[31] MPI: A Message-Passing Interface Standard-Message Passing Interface Forum- http://mpi-forum.org/docs/

[32] Milojicic DS, Kalogeraki V, Lukose R, Nagaraja K, Pruyne J, Richard B, Rollins S, Xu Z. Peer-to-peer computing. Technical Report HPL-2002-57, HP Labs. 2002.

[33] Apache Software Foundation: https://www.apache.org/

[34] Apache Spark. http://spark.incubator.apache.org
[35] Apache Flink. http://flink.apache.org/
[36] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2011-82, University of California, Berkeley, 2011.
[37] Distributed shared memory. B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. Computer, 24(8):52–60, Aug 1991.
[38] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In OSDI, 2004.
[39] Lee K-H, Lee Y-J, Choi H, Chung YD, Moon B. Parallel data processing with MapReduce: a survey. ACM SIGMOD Record. 2012;40(4):11–20. doi: 10.1145/2094114.2094118.
[40] Apache Flink. https://flink.apache.org/
[41] Ian Pointer (7 May 2015). "Apache Flink: New Hadoop contender squares off against Spark". InfoWorld.
[42] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. 2012. Spinning fast iterative data flows. Proc. VLDB Endow. 5, 11 (July 2012), 1268-1279. DOI/
[43] Flink Gelly. https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/gelly/index.html
[44] Flink-ML. https://cwiki.apache.org/confluence/display/FLINK/FlinkML%3A+Vision+and+Roadmap
[45] Bikas Sahah , Hitesh Shahh , Siddharth Sethh , Gopal Vijayaraghavanh , Arun Murthyh , Carlo Curino Hortonworks, Microsoft. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications.
[46] Apache Hive. https://hive.apache.org/
[47] Apache Pig. https://pig.apache.org/
[48] Apache Spark. https://spark.apache.org/
[49] Spark SQL: https://spark.apache.org/sql/
[50] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B. Zdonik. Scalable distributed stream processing. In CIDR, 2003.
[51] Apache Storm. http://storm-project.net.
[52] Apache Storm Trident. http://storm.apache.org/releases/current/Trident-state.html
[53] Spark Streaming. https://spark.apache.org/streaming/
[54] Apache Flink: Stream and Batch Processing in a Single Engine: P Carbone, S Ewen, S Haridi, A Katsifodimos, V Markl, K Tzoumas
[55] Apache Kafka. http://kafka.apache.org/
[56] Apache Kafka. https://en.wikipedia.org/wiki/Apache_Kafka
[57] Apache Samoa. https://samoa.incubator.apache.org/documentation/Home.html
[58] Magdalena Balazinska, Hari Balakrishnan, Samuel R. Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. ACM Trans. Database Syst., 2008.
[59] Apache Strom. http://storm.apache.org/releases/1.1.0/Guaranteeing-message-processing.html
[60] Supposedly paraphrased from: Samuel, Arthur (1959). "Some Studies in Machine Learning Using the Game of Checkers". IBM Journal of Research and Development. 3 (3). doi:10.1147/rd.33.0210.
[61] Apache Mahout. https://mahout.apache.org/
[62] Seminario CE, Wilson DC. Case Study Evaluation of Mahout as a Recommender Platform. In: 6th ACM conference on recommender engines (RecSys 2012); 2012. pp. 45–50.
[63] List of algorithms. https://mahout.apache.org/users/basics/algorithms.html
[64] Spark machine learning library (MLlib). http://spark.incubator.apache.org/docs/latest/mllib-guide.html.
[65] Spark Mlib. https://spark.apache.org/docs/1.1.0/mllib-guide.html
[66] List of algorithms. http://spark.apache.org/mllib/
[67] ML pipelines. https://spark.apache.org/docs/latest/ml-pipeline.html
[68] Pan X, Sparks ER, Wibisono A. MLbase: Distributed Machine Learning Made Easy. University of California Berkeley Technical Report; 2013.
[69] H2O. http://docs.h2o.ai/h2o/latest-stable/h2o-docs/welcome.html
[70] H2O. https://en.wikipedia.org/wiki/H2O_(software)
[71] High Performance Machine Learning in R with H2O Erin LeDell Ph.D. ISM HPC on R Workshop Tokyo, Japan October 2015.
[72] Najafabadi MM, Villanustre F, Khoshgoftaar TM, Seliya N, Wald R, Muharemagic E. Deep learning applications and challenges in Big data analytics. J Big data. 2015;2(1):1–21.
[73] Flink-ML. https://cwiki.apache.org/confluence/display/FLINK/FlinkML%3A+Vision+and+Roadmap
[74] Oryx. https://github.com/cloudera/oryx
[75] Samsara. http://mahout.apache.org/users/environment/in-core-reference.html
[76] Zheng J, Dagnino A. An initial study of predictive machine learning analytics on large volumes of historical data for power system applications. In: 2014 IEEE International Conference on Big data; 2014. pp. 952–59.
[77] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In SIGMOD Conference, pages 135–146, 2010.
[78] Leslie G Valiant, “A bridging model for parallel computation,” Communications of the ACM, vol. 33, no. 8, pp. 103– 111, 1990.
[79] Wijnand J. Suijlen: BSPonMPI, 2006.
[80] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin, “Powergraph: Distributed graph-parallel computation on natural graphs,” in Presented as

part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), Hollywood, CA, 2012, pp. 17–30, USENIX.

[81] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica, "Graphx: Graph processing in a distributed dataflow framework," in 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), Broomfield, CO, Oct. 2014, pp. 599–613, USENIX Association.

[82] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In First International Workshop on Graph Data Management Experiences and Systems, GRADES '13, 2013.

[83] Gelly. https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/libs/gelly/index.html

[84] Shaw, S., Vermeulen, A.F., Gupta, A., Kjerrumgaard, D., 2016. Hive architecture. In: Practical Hive. Springer, pp. 37–48.

[85] Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R. Hive: a warehousing solution over a map-reduce framework.

[86] HiveQL. https://cwiki.apache.org/confluence/display/Hive/LanguageManual

[87] Olston C, Reed B, Srivastava U, Kumar R, Tomkins A. Proceedings of the ACM SIGMOD international conference on Management of Data. : ACM; 2008. Pig latin: a not-so-foreign language for data processing; pp. 1099–1110.

[88] Spark SQL. https://spark.apache.org/sql/

[89] DataFrame. https://spark.apache.org/docs/latest/sql-programming-guide.html#datasets-and-dataframes

[90] Apache Drill. https://drill.apache.org/

[91] Apache Drill. https://en.wikipedia.org/wiki/Apache_Drill

[92] Apache Flume: Distributed Log Collection for Hadoop S. Hoffman Packt Publishing Ltd. (2015)

[93] Apache ZooKeeper. https://zookeeper.apache.org/

[94] Apache Ambari. https://cwiki.apache.org/confluence/display/AMBARI/Ambari

[95] Why a connected data strategy is critical to the future of your data: a Hortonworks white paper, March 2016

[96] Cloudera. https://www.cloudera.com

[97] MapR: https://mapr.com/products/mapr-distribution-including-apache-hadoop/

First Author Harish Mamilla is a research assistant at the Indian Institute of Technology Delhi. He received his bachelor's degree in computer science from Jawaharlal Nehru Technological University, Kakinada, India. He is currently working as Information Systems Officer at India's Largest Enterprise, Indian Oil Corporation Limited. His research interests include distributed fault-tolerant computing, Big data platforms.

Second Author Sai Pradeep is a research assistant at the Indian Institute of Technology Delhi. He received his bachelor's degree in computer science from M. S. Ramaiah Institute Of Technology, Bangalore, India. His research interests include distributed computing, computing cluster resource management.

Third Author Suresh C Gupta is a visiting faculty in the Department of Computer Science and Engineering at the Indian Institute of Technology Delhi. He worked as Scientist at Computer Group at Tata Institute of Fundamental Research and NCSDCT (Now C-DAC Mumbai). Till recently, he worked as Deputy Director General, Scientist -G at National Informatics Centre, Government of India. His research interests includes Software Engineering, Data Bases, Cloud Computing, Software Defined Storage and Networks.