IJCSE International Journal of Computer Sciences and Engineering Open Access
Review Paper Volume-5, Issue-12 E-ISSN: 2347-2693

The Real Time Big Data Processing Framework: Advantages and Limitations

Vairaprakash Gurusamy[1], S. Kannan[2*], K. Nandhini[3]

[1]Department of Computer Applications, School of IT, Madurai Kamaraj University, Madurai, India
[2*] Department of Computer Applications, School of IT, Madurai Kamaraj University, Madurai, India
[3]Technical Support Engineer, Concentrix India Pvt Ltd, Chennai, India

[*]*Corresponding Author: skannanmku@gmail.com*

Available online at: www.ijcseonline.org

Received: 11/Nov/2017, Revised: 24/Nov/2017, Accepted: 16/Dec/2017, Published: 31/Dec/2017

Abstract ---Big data is a blanket term for the non-traditional strategies and technologies needed to gather, organize, process, and gather insights from large datasets. While the problem of working with data that exceeds the computing power or storage of a single computer is not new, the pervasiveness, scale, and value of this type of computing have greatly expanded in recent years. In this paper, we will take a look at one of the essential components of a big data system: processing frameworks. Processing frameworks compute over the data in the system, either by reading from non-volatile storage or as it is ingested into the system. Computing over data is the process of extracting information and insight from large quantities of individual data points.

Keywords- Big Data, Hadoop, HDFS, Spark, Storm, Flink, Samza

I. INTRODUCTION

Processing frameworks and processing engines are responsible for computing over data in a data system. While there is no authoritative definition setting apart "engines" from "frameworks", it is sometimes useful to define the former as the actual component responsible for operating on data and the latter as a set of components designed to do the same. For instance, Apache Hadoop can be considered a *processing framework* with MapReduce as its default *processing engine*. Engines and frameworks can often be swapped out or used in tandem. For instance, Apache Spark, another framework, can hook into Hadoop to replace MapReduce. This interoperability between components is one reason that big data systems have great flexibility. While the systems which handle this stage of the data life cycle can be complex, the goals on a broad level are very similar: operate over data in order to increase understanding, surface patterns, and gain insight into complex interactions. To simplify the discussion of these components, we will group these processing frameworks by the state of the data they are designed to handle. Some systems handle data in batches, while others process data in a continuous stream as it flows into the system. Still, others can handle data in either of these ways. We will introduce each type of processing as a concept before diving into the specifics and consequences of various implementations. In the following section, We will cover the following frameworks:

1. Batch-only frameworks:
    i. Apache Hadoop
2. Stream-only frameworks:
    i. Apache Storm
    ii. Apache Samza
3. Hybrid frameworks:
    i. Apache Spark
    ii. Apache Flink

II. BATCH PROCESSING SYSTEMS

Batch processing has a long history of the big data world. Batch processing involves operating over a large, static dataset and returning the result at a later time when the computation is complete. The datasets in batch processing are

1. bounded: batch datasets represent a finite collection of data
2. persistent: data is almost always backed by some type of permanent storage
3. large: batch operations are often the only option for processing extremely large sets of data

Batch processing is well-suited for calculations where access to a complete set of records is required. For instance, when calculating totals and averages, datasets must be treated holistically instead of as a collection of individual records. These operations require that state is maintained for the duration of the calculations.

Tasks that require very large volumes of data are often best handled by batch operations. Whether the datasets are

processed directly from permanent storage or loaded into memory, batch systems are built with large quantities in mind and have the resources to handle them. Because batch processing excels at handling large volumes of persistent data, it frequently is used with historical data.
The trade-off for handling large quantities of data is longer computation time. Because of this, batch processing is not appropriate in situations where processing time is especially significant.

Apache Hadoop
Apache Hadoop is a processing framework that exclusively provides batch processing. Hadoop was the first big data framework to gain significant traction in the open-source community. Based on several papers and presentations by Google about how they were dealing with tremendous amounts of data at the time, Hadoop reimplemented the algorithms and component stack to make large-scale batch processing more accessible.

Modern versions of Hadoop are composed of several components or layers, that work together to process batch data:

- **HDFS**: HDFS is the distributed file system layer that coordinates storage and replication across the cluster nodes. HDFS ensures that data remains available in spite of inevitable host failures. It is used as the source of data, to store intermediate processing results, and to persist the final calculated results.
- **YARN**: YARN, which stands for Yet Another Resource Negotiator, is the cluster coordinating component of the Hadoop stack. It is responsible for coordinating and managing the underlying resources and scheduling jobs to be run. YARN makes it possible to run much more diverse workloads on a Hadoop cluster than was possible in earlier iterations by acting as an interface to the cluster resources.
- **MapReduce**: MapReduce is Hadoop's native batch processing engine.

Batch Processing Model
The processing functionality of Hadoop comes from the MapReduce engine. MapReduce's processing technique follows the map, shuffle, reduce algorithm using key-value pairs.

The basic procedure involves:

- Reading the dataset from the HDFS file system
- Dividing the dataset into chunks and distributed among the available nodes
- Applying the computation on each node to the subset of data (the intermediate results are written back to HDFS)
- Redistributing the intermediate results to group by key
- "Reducing" the value of each key by summarizing and combining the results calculated by the individual nodes
- Write the calculated final results back to HDFS

Advantages and Limitations
Because this methodology heavily leverages permanent storage, reading and writing multiple times per task, it tends to be fairly slow. On the other hand, since disk space is typically one of the most abundant server resources, it means that MapReduce can handle enormous datasets. This also means that Hadoop's MapReduce can typically run on less expensive hardware than some alternatives since it does not attempt to store everything in memory. MapReduce has incredible scalability potential and has been used in production on tens of thousands of nodes.

As a target for development, MapReduce is known for having a rather steep learning curve. Other additions to the Hadoop ecosystem can reduce the impact of this to varying degrees, but it can still be a factor in quickly implementing an idea on a Hadoop cluster.

Hadoop has an extensive ecosystem, with the Hadoop cluster itself frequently used as a building block for other software. Many other processing frameworks and engines have Hadoop integrations to utilize HDFS and the YARN resource manager.

Summary
Apache Hadoop and its MapReduce processing engine offer a well-tested batch processing model that is best suited for handling very large datasets where time is not a significant factor. The low cost of components necessary for a well-functioning Hadoop cluster makes this processing inexpensive and effective for many use cases. Compatibility and integration with other frameworks and engines mean that Hadoop can often serve as the foundation for multiple processing workloads using diverse technology.

III. STREAM PROCESSING SYSTEMS

Stream processing systems compute over data as it enters the system. This requires a different processing model than the batch paradigm. Instead of defining operations to apply to an entire dataset, stream processors define operations that will be applied to each individual data item as it passes through the system.

The datasets in stream processing are considered "unbounded". This has a few important implications:

- The *total* dataset is only defined as the amount of data that has entered the system so far.

- The *working* dataset is perhaps more relevant and is limited to a single item at a time.
- Processing is event-based and does not "end" until explicitly stopped. Results are immediately available and will be continually updated as new data arrives.

Stream processing systems can handle a nearly unlimited amount of data, but they only process one (true stream processing) or very few (micro-batch processing) items at a time, with the minimal state being maintained in between records. While most systems provide methods of maintaining some state, stream processing is highly optimized for more **functional processing** with few side effects.

Functional operations focus on discrete steps that have limited state or side-effects. Performing the same operation on the same piece of data will produce the same output independent of other factors. This kind of processing fits well with streams because state between items is usually some combination of difficult, limited, and sometimes undesirable. So while some type of state management is usually possible, these frameworks are much simpler and more efficient in their absence.

This type of processing lends itself to certain types of workloads. Processing with near real-time requirements is well served by the streaming model. Analytics, server or application error logging, and other time-based metrics are a natural fit because reacting to changes in these areas can be critical to business functions. Stream processing is a good fit for data where you must respond to changes or spikes and where you're interested in trends over time.

Apache Storm

Apache Storm is a stream processing framework that focuses on extremely low latency and is perhaps the best option for workloads that require near real-time processing. It can handle very large quantities of data with and deliver results with less latency than other solutions.

Stream Processing Model

Stream processing works by orchestrating DAGs (Directed Acyclic Graphs) in a framework it calls **topologies**. These topologies describe the various transformations or steps that will be taken on each incoming piece of data as it enters the system.

The topologies are composed of:

- **Streams**: Conventional data streams. This is unbounded data that is continuously arriving at the system.
- **Spouts**: Sources of data streams at the edge of the topology. These can be APIs, queues, etc. that produce data to be operated on.
- **Bolts**: Bolts represent a processing step that consumes streams, applies an operation to them, and outputs the result as a stream. Bolts are connected to each of the spouts and then connect to each other to arrange all of the necessary processing. At the end of the topology, final bolt output may be used as an input for a connected system.

The idea behind Storm is to define small, discrete operations using the above components and then compose them into a topology. By default, Storm offers at-least-once processing guarantees, meaning that it can guarantee that each message is processed at least once, but there may be duplicates in some failure scenarios. Storm does not guarantee that messages will be processed in order.

In order to achieve exactly-once, stateful processing, an abstraction called **Trident** is also available. To be explicit, Storm without Trident is often referred to as **Core Storm**. Trident significantly alters the processing dynamics of Storm, increasing latency, adding a state to the processing, and implementing a micro-batching model instead of an item-by-item pure streaming system.

Storm users typically recommend using Core Storm whenever possible to avoid those penalties. With that in mind, Trident's guarantee to processes items exactly once is useful in cases where the system cannot intelligently handle duplicate messages. Trident is also the only choice within Storm when you need to maintain state between items, like when counting how many users click a link within an hour. Trident gives Storm flexibility, even though it does not play to the framework's natural strengths.

Trident topologies are composed of:

- **Stream batches**: These are micro-batches of stream data that are chunked in order to provide batch processing semantics.
- **Operations**: These are batch procedures that can be performed on the data.

Advantages and Limitations

The storm is probably the best solution currently available for near real-time processing. It is able to handle data with extremely low latency for workloads that must be processed with minimal delay. The storm is often a good choice when processing time directly affects user experience, for example when feedback from the processing is fed directly back to a visitor's page on a website.

Storm with Trident gives you the option to use micro-batches instead of pure stream processing. While this gives users greater flexibility to shape the tool for an intended use, it also tends to negate some of the software's biggest advantages

over other solutions. That being said, having a choice for the stream processing style is still helpful.

Core Storm does not offer to order guarantees of messages. Core Storm offers at-least-once processing guarantees, meaning that processing of each message can be guaranteed but duplicates may occur. Trident offers exactly-once guarantees and can offer to order between batches, but not within.

In terms of interoperability, Storm can integrate with Hadoop's YARN resource negotiator, making it easy to hook up to an existing Hadoop deployment. More than most processing frameworks, Storm has very wide language support, giving users many options for defining topologies.

Summary
For pure stream processing workloads with very strict latency requirements, Storm is probably the best mature option. It can guarantee message processing and can be used with a large number of programming languages. Because Storm does not do batch processing, you will have to use additional software if you require those capabilities. If you have a strong need for exactly-once processing guarantees, Trident can provide that. However, other stream processing frameworks might also be a better fit at that point.

Apache Samza
Apache Samza is a stream processing framework that is tightly tied to the Apache Kafka messaging system. While Kafka can be used by many stream processing systems, Samza is designed specifically to take advantage of Kafka's unique architecture and guarantees. It uses Kafka to provide fault tolerance, buffering, and state storage.

Samza uses YARN for resource negotiation. This means that by default, a Hadoop cluster is required (at least HDFS and YARN), but it also means that Samza can rely on the rich features built into YARN.

Stream Processing Model
Samza relies on Kafka's semantics to define the way that streams are handled. Kafka uses the following concepts when dealing with data:

- **Topics**: Each stream of data entering a Kafka system is called a topic. A topic is basically a stream of related information that consumers can subscribe to.
- **Partitions**: In order to distribute a topic among nodes, Kafka divides the incoming messages into partitions. The partition divisions are based on a key such that each message with the same key is guaranteed to be sent to the same partition. Partitions have guaranteed to order.
- **Brokers**: The individual nodes that make up a Kafka cluster are called brokers.
- **Producer**: Any component writing to a Kafka topic is called a producer. The producer provides the key that is used to partition a topic.
- **Consumers**: Consumers are any component that reads from a Kafka topic. Consumers are responsible for maintaining information about their own offset, so that they are aware of which records have been processed if a failure occurs.

Because Kafka is represented an immutable log, Samza deals with immutable streams. This means that any transformations create new streams that are consumed by other components without affecting the initial stream.

Advantages and Limitations
Samza's reliance on a Kafka-like queuing system at first glance might seem restrictive. However, it affords the system some unique guarantees and features not common in other stream processing systems.

For example, Kafka already offers replicated storage of data that can be accessed with low latency. It also provides a very easy and inexpensive multi-subscriber model to each individual data partition. All output, including intermediate results, is also written to Kafka and can be independently consumed by downstream stages.

In many ways, this tight reliance on Kafka mirrors the way that the MapReduce engine frequently references HDFS. While referencing HDFS between each calculation leads to some serious performance issues when batch processing, it solves a number of problems when stream processing.

Samza's strong relationship to Kafka allows the processing steps themselves to be very loosely tied together. An arbitrary number of subscribers can be added to the output of any step without prior coordination. This can be very useful for organizations where multiple teams might need to access similar data. Teams can all subscribe to the topic of data entering the system, or can easily subscribe to topics created by other teams that have undergone some processing. This can be done without adding additional stress on load-sensitive infrastructure like databases.

Writing straight to Kafka also eliminates the problems of **backpressure**. Backpressure is when load spikes cause an influx of data at a rate greater than components can process in real time, leading to processing stalls and potentially data loss. Kafka is designed to hold data for very long periods of time, which means that components can process at their convenience and can be restarted without consequence.

Samza is able to store state, using a fault-tolerant checkpointing system implemented as a local key-value store. This allows Samza to offer an at-least-once delivery

guarantee, but it does not provide accurate recovery of aggregated state (like counts) in the event of a failure since data might be delivered more than once.

Samza offers high-level abstractions that are in many ways easier to work with than the primitives provided by systems like Storm. Samza only supports JVM languages at this time, meaning that it does not have the same language flexibility as Storm.

Summary
Apache Samza is a good choice for streaming workloads where Hadoop and Kafka are either already available or sensible to implement. Samza itself is a good fit for organizations with multiple teams using (but not necessarily tightly coordinating around) data streams at various stages of processing. Samza greatly simplifies many parts of stream processing and offers low latency performance. It might not be a good fit if the deployment requirements aren't compatible with your current system, if you need extremely low latency processing, or if you have strong needs for exactly-once semantics.

IV. HYBRID PROCESSING SYSTEMS: BATCH AND STREAM PROCESSORS

Some processing frameworks can handle both batch and stream workloads. These frameworks simplify diverse processing requirements by allowing the same or related components and APIs to be used for both types of data.

As you will see, the way that this is achieved varies significantly between Spark and Flink, the two frameworks we will discuss. This is a largely a function of how the two processing paradigms are brought together and what assumptions are made about the relationship between fixed and unfixed datasets.

While projects focused on one processing type may be a close fit for specific use-cases, the hybrid frameworks attempt to offer a general solution for data processing. They not only provide methods for processing over data, they have their own integrations, libraries, and tools for doing things like graph analysis, machine learning, and interactive querying.

A. Apache Spark
Apache Spark is a next generation batch processing framework with stream processing capabilities. Built using many of the same principles of Hadoop's MapReduce engine, Spark focuses primarily on speeding up batch processing workloads by offering full in-memory computation and processing optimization.

Spark can be deployed as a standalone cluster (if paired with a capable storage layer) or can hook into Hadoop as an alternative to the MapReduce engine.

Batch Processing Model
Unlike MapReduce, Spark processes all data in-memory, only interacting with the storage layer to initially load the data into memory and at the end to persist the final results. All intermediate results are managed in memory.

While in-memory processing contributes substantially to speed, Spark is also faster on disk-related tasks because of holistic optimization that can be achieved by analyzing the complete set of tasks ahead of time. It achieves this by creating Directed Acyclic Graphs, or **DAGs** which represent all of the operations that must be performed, the data to be operated on, as well as the relationships between them, giving the processor a greater ability to intelligently coordinate work.
To implement an in-memory batch computation, Spark uses a model called Resilient Distributed Datasets, or **RDDs**, to work with data. These are immutable structures that exist within memory that represent collections of data. Operations on RDDs produce new RDDs. Each RDD can trace its lineage back through its parent RDDs and ultimately to the data on disk. Essentially, RDDs are a way for Spark to maintain fault tolerance without needing to write back to disk after each operation.

Stream Processing Model
Stream processing capabilities are supplied by Spark Streaming. Spark itself is designed with batch-oriented workloads in mind. To deal with the disparity between the engine design and the characteristics of streaming workloads, Spark implements a concept called *micro-batches**. This strategy is designed to treat streams of data as a series of very small batches that can be handled using the native semantics of the batch engine.

Spark Streaming works by buffering the stream in sub-second increments. These are sent as small fixed datasets for batch processing. In practice, this works fairly well, but it does lead to a different performance profile than true stream processing frameworks.

Advantages and Limitations
The obvious reason to use Spark over Hadoop MapReduce is speed. Spark can process the same datasets significantly faster due to its in-memory computation strategy and its advanced DAG scheduling.

Another of Spark's major advantages is its versatility. It can be deployed as a standalone cluster or integrated with an existing Hadoop cluster. It can perform both batch and stream

processing, letting you operate a single cluster to handle multiple processing styles.

Beyond the capabilities of the engine itself, Spark also has an ecosystem of libraries that can be used for machine learning, interactive queries, etc. Spark tasks are almost universally acknowledged to be easier to write than MapReduce, which can have significant implications for productivity.

Adapting the batch methodology for stream processing involves buffering the data as it enters the system. The buffer allows it to handle a high volume of incoming data, increasing overall throughput, but waiting to flush the buffer also leads to a significant increase in latency. This means that Spark Streaming might not be appropriate for processing where low latency is imperative.

Since RAM is generally more expensive than disk space, Spark can cost more to run than disk-based systems. However, the increased processing speed means that tasks can complete much faster, which may completely offset the costs when operating in an environment where you pay for resources hourly.

One other consequence of the in-memory design of Spark is that resource scarcity can be an issue when deployed on shared clusters. In comparison to Hadoop's MapReduce, Spark uses significantly more resources, which can interfere with other tasks that might be trying to use the cluster at the time. In essence, Spark might be a less considerate neighbor than other components that can operate on the Hadoop stack.

Summary
A spark is a great option for those with diverse processing workloads. Spark batch processing offers incredible speed advantages, trading off high memory usage. Spark Streaming is a good stream processing solution for workloads that value throughput over latency.

B. Apache Flink
Apache Flink is a stream processing framework that can also handle batch tasks. It considers batches to simply be data streams with finite boundaries, and thus treats batch processing as a subset of stream processing. This stream-first approach to all processing has a number of interesting side effects.

This stream-first approach has been called the **Kappa architecture**, in contrast to the more widely known Lambda architecture (where batching is used as the primary processing method with streams used to supplement and provide early but unrefined results). Kappa architecture, where streams are used for everything, simplifies the model and has only recently become possible as stream processing engines have grown more sophisticated.

Stream Processing Model
Flink's stream processing model handles incoming data on an item-by-item basis as a true stream. Flink provides its DataStream API to work with unbounded streams of data. The basic components that Flink works with are:

- **Streams** are immutable, unbounded datasets that flow through the system
- **Operators** are functions that operate on data streams to produce other streams
- **Sources** are the entry point for streams entering the system
- **Sinks** are the place where streams flow out of the Flink system. They might represent a database or a connector to another system

Stream processing tasks take snapshots at set points during their computation to use for recovery in case of problems. For storing state, Flink can work with a number of state backends depending on varying levels of complexity and persistence.
Additionally, Flink's stream processing is able to understand the concept of "event time", meaning the time that the event actually occurred, and can handle sessions as well. This means that it can guarantee ordering and grouping in some interesting ways.

Batch Processing Model
Flink's batch processing model in many ways is just an extension of the stream processing model. Instead of reading from a continuous stream, it reads a bounded dataset off of persistent storage as a stream. Flink uses the exact same runtime for both of these processing models.

Flink offers some optimizations for batch workloads. For instance, since batch operations are backed by persistent storage, Flink removes snapshotting from batch loads. Data is still recoverable, but normal processing completes faster.
Another optimization involves breaking up batch tasks so that stages and components are only involved when needed. This helps Flink play well with other users of the cluster. Preemptive analysis of the tasks gives Flink the ability to also optimize by seeing the entire set of operations, the size of the dataset, and the requirements of steps coming down the line.

Advantages and Limitations
Flink is currently a unique option in the processing framework world. While Spark performs batch and stream processing, its streaming is not appropriate for many use cases because of its micro-batch architecture. Flink's stream-first approach offers low latency, high throughput, and real entry-by-entry processing.

Flink manages many things by itself. Somewhat unconventionally, it manages its own memory instead of relying on the native Java garbage collection mechanisms for

performance reasons. Unlike Spark, Flink does not require manual optimization and adjustment when the characteristics of the data it processes change. It handles data partitioning and caching automatically as well.

Flink analyzes its work and optimizes tasks in a number of ways. Part of this analysis is similar to what SQL query planners do within relationship databases, mapping out the most effective way to implement a given task. It is able to parallelize stages that can be completed in parallel while bringing data together for blocking tasks. For iterative tasks, Flink attempts to do computation on the nodes where the data is stored for performance reasons. It can also do "delta iteration", or iteration on only the portions of data that have changed.

In terms of user tooling, Flink offers a web-based scheduling view to easily manage tasks and view the system. Users can also display the optimization plan for submitted tasks to see how it will actually be implemented on the cluster. For analysis tasks, Flink offers SQL-style querying, graph processing and machine learning libraries, and in-memory computation.

Flink operates well with other components. It is written to be a good neighbor if used within a Hadoop stack, taking up only the necessary resources at any given time. It integrates with YARN, HDFS, and Kafka easily. Flink can run tasks written for other processing frameworks like Hadoop and Storm with compatibility packages.

One of the largest drawbacks of Flink at the moment is that it is still a very young project. Large-scale deployments in the wild are still not as common as other processing frameworks and there hasn't been much research into Flink's scaling limitations. With the rapid development cycle and features like the compatibility packages, there may begin to be more Flink deployments as organizations get the chance to experiment with it.

Summary

Flink offers both low latency stream processing with support for traditional batch tasks. Flink is probably best suited for organizations that have heavy stream processing requirements and some batch-oriented tasks. Its compatibility with native Storm and Hadoop programs, and its ability to run on a YARN-managed cluster can make it easy to evaluate. Its rapid development makes it worth keeping an eye on.

V. CONCLUSION

There are plenty of options for processing within a big data system.

For batch-only workloads that are not time-sensitive, Hadoop is a good choice that is likely less expensive to implement than some other solutions.

For stream-only workloads, Storm has wide language support and can deliver very low latency processing, but can deliver duplicates and cannot guarantee to order in its default configuration. Samza integrates tightly with YARN and Kafka in order to provide flexibility, easy multi-team usage, and straightforward replication and state management.

For mixed workloads, Spark provides high-speed batch processing and micro-batch processing for streaming. It has wide support, integrated libraries and tooling, and flexible integrations. Flink provides true stream processing with batch processing support. It is heavily optimized, can run tasks written for other platforms, and provides low latency processing, but is still in the early days of adoption.

The best fit for your situation will depend heavily upon the state of the data to process, how time-bound your requirements are, and what kind of results you are interested in. There are trade-offs between implementing an all-in-one solution and working with tightly focused projects, and there are similar considerations when evaluating new and innovative solutions over their mature and well-tested counterparts.

REFERENCES

[1] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinlander, M. J. Sax, S. Schelter, M. Hoger, K. Tzoumas, and D. Warneke. The stratosphere platform for big data analytics. The VLDB Journal, 23(6):939-964, 2014.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *IEEE MSST*, 2010.

[3] S. Aridhi and E. M. Nguifo. Big graph mining: Frameworks and techniques. Big Data Research, 6:1-10, 2016.

[4] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. The hadoop approach to large-scale iterative data analysis. The VLDB Journal, 21(2):169-190, Apr. 2012.

[5] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache inkTM: Stream and batch processing in a single engine. IEEE Data Eng. Bull., 38(4):28-38, 2015.

[6] J. Dean and S. Ghemawat. MapReduce: simpli_ed data processing on large clusters. Commun. ACM, 51(1):107-113, 2008.

[7] D. Eadline. Hadoop 2 Quick-Start Guide: Learn the Essentials of Big Data Computing in the Apache Hadoop 2 Ecosystem. Addison-Wesley Professional, 1st edition, 2015.

[8] B. Elser and A. Montresor. An evaluation study of bigdata frameworks for graph processing. In IEEE International Conference on Big Data, pages 60-67, 2013.

[9] A. Gandomi and M. Haider. Beyond the hype: Big data concepts, methods, and analytics. International Journal of Information Management, 35(2):137-144, 2015.

[10] R. Li, H. Hu, H. Li, Y. Wu, and J. Yang. Mapreduce parallel programming model: A state-of-the-art survey. International Journal of Parallel Programming, pages 1-35, 2015.

[11] X. Liu, N. Iftikhar, and X. Xie. Survey of real-time processing systems for big data. In Proceedings of the 18th International Database Engineering & Applications Symposium, pages 356-361. ACM, 2014.

[12] D. Singh and C. K. Reddy. A survey on platforms for big data analytics. Journal of Big Data, 2(1):8, 2014.

[13] M. Tatineni, X. Lu, D. Choi, A. Majumdar, and D. K. D. Panda. Experiences and bene_ts of running rdma hadoop and spark on sdsc comet. In Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale, XSEDE16, pages 23:1-23:5, New York, NY, USA, 2016. ACM.

[14] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In First International Workshop on Graph Data Management Experiences and Systems, GRADES '13, pages 2:1-2:6, New York, NY, USA, 2013. ACM.

Authors Profile

Dr. S.kannan is an Associate Professor in Department of Computer Applications, School of Information Technology, Madurai Kamaraj University, Madurai. He is having more than 25 years of Teaching experience and 15 years of Research experience. His core area of research is Data Mining, Image Processing, Soft Computing, Natural Language Processing and Data Analytics. He published more than 50 Research articles in reputed journals, act as a Reviewer in more standard periodicals and serves as a chair for more Conferences, Workshops, and Viva-voce.

Mr. Vairaprakash Gurusamy pursed MCA from Bharathidasan University, Trichy in 2010. He is currently pursuing Ph.D. from Madurai Kamaraj University, Madurai, India. He has published more than 10 research papers in reputed international journals like Scopus Indexed, UGC approved, SCI Indexed, Web of Science, Thomson Reuters etc. His main research work focuses on Bid Data Analytics, Distributed System, Artificial Intelligence, NLP, Cloud Computing, Data Mining and IOT. He has 3 years of Industry Experience and 4 years of Research Experience.

Ms. K. Nandhini pursed B.Tech (ECE) from Kalasalingam University, Krishnan Kovil, India in 2014. She has published more than 10 research papers in reputed international journals like Scopus Indexed, UGC approved, SCI Indexed, Web of Science, Thomson Reuters etc. His main research work focuses on Bid Data Analytics, Distributed System, Artificial Intelligence, NLP, Cloud Computing, Data Mining and IOT. She has 3 years of Industry Experience.

