Big Data Processing Systems: State-of-the-art and Open Challenges

Fuad Bajaber [1], Sherif Sakr [2], Omar Batarfi [1], Abdulrahman Altalhi [1], Radwa Elshawi[3] , Ahmed Barnawi [1]
[1]King Abdulaziz University, Saudi Arabia, ffbajaber,obatarfi,ahaltalhi,ambarnawig@kau.edu.sa
[2]University of New South Wales, Australia, ssakr@cse.unsw.edu.au
[3]Princess Nourah Bint Abdulrahman University, Saudi Arabia, rmelshawi@pnu.edu.sa

Abstract— The growing demand for large-scale data processing and data analysis applications spurred the development of novel solutions from both the industry and academia. In the last decade, the MapReduce framework has emerged as a highly successful framework that has created a lot of momentum in both the research and industrial communities such that it has become the defacto standard of big data processing platforms. In particular, the MapReduce framework has been introduced to provide a simple but powerful programming model and runtime environment that eases the job of developing scalable parallel applications to process vast amounts of data on large clusters of commodity machines. However, recently, academia and industry have started to recognize the limitations of the Hadoop framework in several application domains such as large scale processing of structured data, graph data and streaming data. Thus, in recent years, we have witnessed an unprecedented interest to tackle these challenges which constitutes a new wave of domain-specific optimized big data processing platforms. To better understand the latest ongoing developments in the world of big data processing systems, in this paper, we provide a detailed overview and analysis of the state-of-the-art in this domain. In addition, we identify a set of the current open research challenges and discuss some promising directions for future research.

I. Introduction

According to IBM, we are currently creating 2.5 quintillion bytes of data, everyday[1]. In practice, this data is generated from many different sources and in different formats including digital pictures, videos, posts to social media sites, intelligent sensors, purchase transaction records and cell phone GPS signals. This situation represent a new scale of *Big Data* which has been attracting a lot of interest from both the research and industrial communities with the aim of creating the best means to process and analyze this data in order to make the best use of it. In the last decade, the MapReduce framework [12] has represented the defacto standard of big data technologies and has been widely utilized as a popular mechanism to harness the power of large clusters of computers. In general, the fundamental principle of the MapReduce framework is to move analysis to the data, rather than moving the data to a system that can analyze it. It allows programmers to think in a data-centric fashion where they can focus on applying transformations to sets of data records while the details of distributed execution and fault tolerance are transparently managed by the MapReduce framework. The Hadoop project2, the open source realization of the

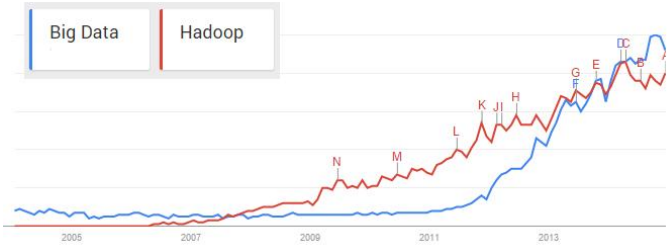

Fig. 1: Google's Web search trends for the two search items: Big Data and Hadoop (created by Google trends)

MapReduce framework, has emerged to be a highly successful framework and created a lot of momentum in both the research and industrial communities. For example, due to its success, it has been supported by many big players in their big data commercial platforms such as IBM3, Oracle4 and Microsoft5. In addition, several successful startups such as Cloudera6 , MapR7, Trifcata8 and Platfora9 have built their solutions and services based on the Hadoop project. Furthermore, at some point and for a while, Hadoop had a dominant presence in the big data world and technologies such that the words Hadoop and big data came to be used interchangeably. Figure 1 illustrates Google's web search trends for the two search items: Big Data and Hadoop, according to the Google trend analysis tool10. In principle, Figure 1 shows that the search item Hadoop has overtaken the search item Big Data and have since dominated the Web users' search requests during the period between 2008 and 2012 while since 2013, the two search items have started to go side by side. In particular, in the last few years, both the research and industrial communities have recognized some main limitations in the MapReduce/Hadoop framework [28] and it has been acknowledged that it cannot be the *one-size-*

[1] http://www-01.ibm.com/software/data/bigdata/ what-is-big-data.html
[2] http://hadoop.apache.org/
[3] http://www-01.ibm.com/software/data/infosphere/ biginsights/
[4] http://www.oracle.com/us/products/middleware/ data-integration/hadoop/overview/index.html
[5] http://azure.microsoft.com/en-us/services/ hdinsight/
[6] http://www.cloudera.com/
[7] https://www.mapr.com/
[8] http://www.trifacta.com/
[9] https://www.platfora.com/
[10] http://www.google.com/trends/

978-1-4673-6618-2/15/$31.00 ©2015 IEEE

fits-all solution for all big data processing problems. For example, in processing large-scale structured data, several studies reported on the significant inefficiency of the Hadoop framework. In particular, the studies claim that Hadoop is the wrong choice for interactive queries that have a target response time of a few seconds or milliseconds [25]. Therefore, Google has created the *Dremel* system [24], commercialized under the name of *BigQuery*[11], to support interactive analysis of nested data. Other projects which have been designed to tackle these challenges include Apache *HIVE*[12], *Cloudera Impala*[13], *IBM Big SQL*[14] and *Pivotal HAWQ*[15]. The Hadoop framework has also been shown to be inefficient within the domain of large-scale graph processing [28]. There-fore, in 2010, Google introduced the Pregel system [23], open-sourced by *Apache Giraph*[16], that uses a bulk synchronous parallel (BSP) based programming model for efficient and scalable processing of massive graphs on distributed clusters of commodity machines. Several other projects (e.g., *GraphLab*[17] and *Trinity*[18]) were introduced to tackle the problem of large-scale graph processing. In large-scale processing of streaming data, Hadoop had shown to be an inadequate platform as well [28]. Twitter announced the release of the *Storm*[19] system that fills this gap by providing a distributed and fault-tolerant platform for implementing continuous and real-time processing applications of streamed data. Other systems in this domain include *IBM InfoSphere Streams*[20] and *Apache S4* [21]All of these systems are just representatives of a new wave of domain-specific systems which constitute the new generation of big data systems.

This paper aims to provide an overview of the state-of-the-art of the new generation of big data processing systems. In addition, we identify a set of the current open research challenges and highlight some promising directions for future research and development. The reminder of this paper is organized as follows. Section II provides an overview of the set of platforms that are focused on providing scalable performance for processing structured and tabular data, collectively labeled as *Big SQL* systems. Section III covers the set of systems which have been introduced to target the problem of large scale graph processing. Section IV focuses on covering the set of systems which have been introduced to deal with the challenge of processing large scale streaming data. A discussion of open research challenges and future research directions are presented in Section V before we conclude the paper in Section VI.

II. BIG SQL PROCESSING SYSTEMS

For programmers, one of the key appealing features in the MapReduce framework is that there are only two high-level declarative primitives (*map* and *reduce*) that can be written in any programming language of choice and without worrying about the details of their parallel execution. However, with such a rigid one-input and two-stage data flow, performing tasks that have a different data flow (e.g. joins or *n* stages) would require the need to devise inelegant workarounds. In addition, many programmers could be unfamiliar with the MapReduce framework and they would prefer to use SQL (in which they are more proficient) as a high level declarative language to express their task while leaving all of the execution optimization details to the backend engine. Furthermore, it is beyond doubt that high level language abstractions enable the underlying system to perform automatic optimization [28]. On the other side, several studies(e.g. [25]) have reported that Hadoop is the wrong choice for interactive queries that have a target response time of a few seconds or milliseconds. In particular, Hadoop has shown significant inefficiency for processing large scale structured data and traditional parallel database systems have been doing much better in this domain.

In order to tackle the above challenges, several systems have been introduced to support the SQL flavor on top of the Hadoop infrastructure and provide competing and scalable performance on processing large scale structured data. For example, *Hive* [31] is considered to be the first system which has been introduced to support SQL-on-Hadoop with familiar relational database concepts such as tables, columns, and partitions. Hive has been widely used in many reputable organizations to manage and process large volumes of data, such as Facebook, eBay, LinkedIn and Yahoo! [19]. Hive supports all of the major primitive types (for example, integers, floats, and strings) and complex types (for example, maps, lists, and structs). It also supports queries that are expressed in an SQL-like declarative language, Hive Query Language (HiveQL[22]), which represents a subset of SQL92, and therefore can be easily understood by anyone who is familiar with SQL. These queries automatically compile into MapReduce jobs that are run by using Hadoop. HiveQL enables users to plug custom MapReduce scripts into queries as well. Recently, Huai et al. [19] have reported about the major technical advancements that have been implemented into the HIVE project by its development community. These advancements include a new file format, Optimized Record Columnar File (ORC File), which is designed to provide high storage and data access efficiency with low overhead. In addition, the query planning component has been updated to provide more sophisticated optimizations for complex queries and significantly reduce unnecessary operations in the executed query plans.

[11] https://developers.google.com/bigquery/

[12] https://hive.apache.org/

[13] http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html

[14] http://www-01.ibm.com/software/data/what-is/ big-sql.html

[15] http://www.pivotal.io/big-data/pivotal-hd

[16] https://giraph.apache.org/

[17] http://graphlab.com

[18] http://research.microsoft.com/en-us/projects/ trinity

[19] https://storm.apache.org/

[20] http://www-03.ibm.com/software/products/en/infosphere-streams

[21] http://incubator.apache.org/s4/

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

Impala is another open source project, built by Cloudera, to provide a massively parallel processing SQL query engine that runs natively in Apache Hadoop. Impala does not use Hadoop to run the queries. Instead, it relies on its own set of daemons, which are installed alongside the data nodes and are tuned to optimize the local processing to avoid bottlenecks. In principle, Impala is part of the Hadoop ecosystem and shares the same infrastructure (for example, metadata, Apache Hive). Therefore, by using Impala, the user can query data which is stored in Hadoop Distributed File System (HDFS)[23]. It also uses the same metadata, SQL syntax (HiveQL), that Apache Hive uses. One of the main limitations of Impala is that it relies on an in-memory join implementation. Therefore, queries can fail if the joined tables can't fit into memory.

HAdoop With Query (*HAWQ*) [8] is a closed-source project that is offered by EMC Pivotal[24] as a massively parallel processing database. It breaks complex queries into small tasks and distributes them to query processing units for execution. HAWQ relies on both the Postgres database and the HDFS storage as its backend storage mechanism. Because it uses Postgres, HAWQ can support the full SQL syntax. The SQL access for the HDFS data is managed by using external tables.

Big *SQL* is the SQL interface for the IBM big data processing platform, InfoSphere BigInsights, which is built on the Apache Hadoop framework. In particular, it provides SQL access to data that is stored in InfoSphere BigInsights and uses the Hadoop framework for complex data sets and direct access for smaller queries. In the initial implementation of Big SQL, the engine was designed to decompose the SQL query into a series of Hadoop jobs. For interactive queries, Big SQL relied on a built-in query optimizer that rewrites the input query as a local job to help minimize latencies by using Hadoop dynamic scheduling mechanisms. The query optimizer also takes care of traditional query optimization such as optimal order, in which tables are accessed in the order where the most efficient join strategy is implemented for the query. The design of the recent version of the Big SQL engine has been implemented by adopting a shared-nothing parallel database architecture, in which it replaces the underlying Hadoop framework with a massively parallel processing SQL engine that is deployed directly on the physical Hadoop Distributed File System (HDFS). Therefore, the data can be accessed by all other tools of the Hadoop ecosystem, such as Pig and Hive. The system infrastructure provides a logical view of the data through\ the storage and management of metadata information. In particular, a table is simply a view that is defined over the stored data in the underlying HDFS. In addition, the Big SQL engine uses the Apache Hive database catalog facility for storing the information about table definitions, location and storage format.

The *HadoopDB* project [1], commercialized by *Hadapt Inc*.[25] and recently acquired by Teradata, is a hybrid sys-tem that combines the scalability advantages of MapReduce with the performance and efficiency advantages of parallel databases. The basic idea behind HadoopDB is to connect multiple single node database systems (PostgreSQL) using Hadoop as the task coordinator and network communication layer. Queries are expressed in SQL but their execution are parallelized across nodes using the MapReduce framework, however, as much of the single node query work as possible is pushed inside of the corresponding node databases. Thus, HadoopDB tries to achieve fault tolerance and the ability to operate in heterogeneous environments by inheriting the scheduling and job tracking implementation from Hadoop. Parallely, it tries to achieve the performance of parallel databases by doing most of the query processing inside the database engine.

Facebook has released *Presto*[26] as an open source distributed SQL query engine for running interactive analytic queries against large scale structured data sources of sizes up to gigabytes to petabytes. In particular, it targets analytic operations where expected response times range from sub-second to minutes. Presto allows querying data where it lives, including Hive, NoSQL databases (e.g., Cassandra[27]), relational databases or even proprietary data stores. Therefore, a single Presto query can combine data from multiple sources.

Microsoft has developed the Polybase[28] project which allows SQL Server Parallel Datawarehouse (PDW) users to execute queries against data stored in Hadoop, specifically the Hadoop distributed file system (HDFS). Polybase provides users with the ability to move data in parallel between nodes of the Hadoop and PDW clusters. In addition, users can create external tables over HDFS-resident data. This allows queries to reference data stored in HDFS as if it were loaded into a relational table. In addition, users can seamlessly perform joins between tables in PDW and data in HDFS. When optimizing a SQL query that references data stored in HDFS, the Polybase query optimizer makes a cost-based decision (using statistics on the HDFS file stored in the PDW catalog) on whether or not it should transform relational operators over HDFS-resident data into MapReduce jobs for execution on the Hadoop cluster [13]. In addition, Polybase is capable of fully leveraging the larger compute and I/O power of the Hadoop cluster by moving work to Hadoop for processing even for queries that only reference PDW-resident data [15].

Google has introduced the Dremel project [24], open sourced as Apache Drill[29] and commercialized as BigQuery, as a scalable, interactive ad-hoc query system for the analysis of read-only nested data. It combines multi-level execution trees and columnar data layout over thousands of CPUs and petabytes of data. Dremel provides a high-level, SQL-like language to express ad-hoc queries which are executed natively without translating them into Hadoop jobs. Dremel.

[23] http://hadoop.apache.org/docs/r1.2.1/hdfs_design. html

[24] http://www.pivotal.io/

[25] http://hadapt.com/

[26] http://prestodb.io/

[27] http://cassandra.apache.org

[28] http://gsl.azurewebsites.net/Projects/Polybase. aspx

[29] http://drill.apache.org/

III. Big Graph Processing Systems

Nowadays, graphs with millions and billions of nodes and edges have become very common. For example, in 2012, Facebook has reported that its social network graph contains more than a billion users[30] (nodes) and more than 140 billion friendship relationships (edges). The enormous growth in graph sizes requires huge amounts of computational power to analyze [26], [27]. In general, graph processing algorithms are iterative and need to traverse the graph in some way. In practice, graph algorithms can be written as a series of chained MapReduce invocations that requires passing the entire state of the graph from one stage to the next. However, this approach is ill-suited for graph processing and leads to inefficient performance due to the additional communication and associated serialization overhead in addition to the need of coordinating the steps of a chained MapReduce. Several approaches have proposed Hadoop extensions (e.g., *HaLoop* [7], *Twister* [14], *iMapReduce* [33]) to optimize the iterative support of the MapReduce framework, however, these approaches remain inefficient for the graph processing case because the efficiency of graph computations depends heavily on inter-processor bandwidth as graph structures are sent over the network after each iteration. While much data might be unchanged from iteration to iteration, the data must be reloaded and reprocessed at each iteration, resulting in the unnecessary wastage of I/O, network bandwidth, and processor resources. In addition, the termination condition might involve the detection of when a fix point is reached. The condition itself might require an extra MapReduce task on each iteration, again increasing the resource usage in terms of scheduling extra tasks, reading extra data from disk, and moving data across the network.

To solve this inherent performance problem of the MapReduce framework, several specialized platforms which are de-signed to serve the unique processing requirements of large-scale graph processing have recently emerged. These systems provide programmatic abstractions for performing iterative parallel analysis of large graphs on clustered systems. In particular, in 2010, Google has pioneered this area by introducing the *Pregel* [23] system as a scalable platform for implementing graph algorithms. Pregel uses a bulk synchronous parallel programming model (BSP) that uses a message passing interface (MPI) to address the scalability challenge of parallelizing jobs across multiple nodes [32]. In principle, BSP represents a "*think like a vertex*" programming model where the computation on vertices are represented as a sequence of *supersteps* with synchronization between the nodes participating at superstep barriers (Figure 2) and each vertex can be active or inactive at each iteration (superstep). Such a programming model can be seen as a graph extension of the actors programming model [9] where each vertex represents an actor and edges represent the communication channel between actors. In such a model, users can focus on specifying the computation on the graph nodes and the communication among them without worrying about the specifics of the underlying organization

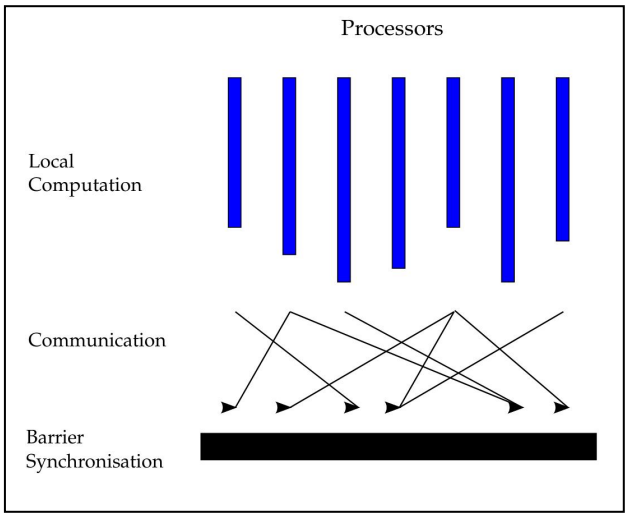

Fig. 2: BSP Programming Model

or resource allocation of the graph data. To avoid communication overheads, Pregel preserves data locality by ensuring computation is performed on locally stored data. In particular, Pregel distributes the graph vertices to the different machines of the cluster where each vertex associated with the set of its neighbors are placed to the same node.

The introduction of Google's Pregel has triggered much interest in the field of large-scale graph data processing and inspired the development of several Pregel-based systems which have been attempting to exploit different optimization opportunities. For example, Apache Giraph is an open source project that clones the ideas and implementation of Pregel specification in Java on top of the infrastructure of the Hadoop framework. In principle, the relationship between the Pregel system and Giraph project is similar to the relationship between the MapReduce framework and the Hadoop project. Giraph has been initially implemented by Yahoo!. Later, Facebook built its Graph Search services using Giraph[31]. Giraph runs graph processing jobs as map-only jobs on Hadoop and uses HDFS for data input and output. Apache Hama[32] is another BSP-based implementation project which is designed to run on top of the Hadoop infrastructure, like Giraph. However, it focuses on general BSP computations and not only for graph processing. For example, it includes algorithms for matrix inversion and linear algebra. In addition, several other systems have been introduced to further optimize the Pregel system [6], [20], [29]. For example, *GPS*[33] is an open source Java implementation of Google's Pregel which comes from Stanford InfoLab [29]. GPS offers the Large Adjacency List Partitioning (LALP) mechanism as an optional performance optimization for algorithms that send the same message to all of its neighbours. In particular, LALP works by partitioning the adjacency lists of high-degree vertices across different workers. For each partition of the adjacency list of a

[30] http://www.insidefacebook.com/2012/10/04/ facebook-reaches-billion-user-milestone/

[31] https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges 10151617006153920

[32] https://hama.apache.org/

[33] http://infolab.stanford.edu/gps/

high-degree vertex, a mirror of the vertex is created in the worker that keeps the partition. When a high-degree vertex broadcasts a message to its neighbors, at most one message is sent to its mirror at each machine. Then, the message is forwarded to all its neighbors in the partition of the adjacency list of the high-degree vertex. Furthermore, GPS applies a dynamic repartitioning strategy based on the graph processing workload in order to balance the workload among all workers and reduce the number of exchanged messages over the network. In particular, GPS exchanges vertices between workers based on the amount of data sent by each vertex. Similar to GPS, *Mizan* is an open-source project developed in C++ by KAUST, in collaboration with IBM Research [20]. Mizan's dynamic repartitioning strategy is based on monitoring the runtime characteristics of the graph vertices (e.g., their execution time, and incoming and outgoing messages) and uses this information, at the end of every superstep, to construct a migration plan with the aims of minimizing the variations across workers by identifying which vertices to migrate and where to migrate them to. *Pregel+*[34] is another Pregel-based project implemented in C/C++ with the aim of reducing the number of exchanged messages between the worker nodes using a mirroring mechanism. In particular, Pregel+ selects the vertices for mirroring based on a cost model that analyzes the tradeoff between mirroring and message combining. *Pregelix*[35] has the optimized the Pregel system by treating the messages and vertex states in the graph computation as relational tuples with a well-defined schema and uses relational database-style query evaluation techniques to execute the graph computation [6]. For example, Pregelix treats message exchange as a join operation followed by a group-by operation that embeds functions which capture the semantics of the graph computation program. Therefore, Pregelix generates a set of alternative physical evaluation strategies for each graph computation program and uses a cost model to select the target execution plan among them.

GraphLab [22] is another open-source large scale graph processing project, implemented in C++, which started at CMU and is currently supported by GraphLab Inc[36]. GraphLab relies on the shared memory abstraction and the GAS (Gather, Apply, Scatter) processing model which is similar to but also fundamentally different from the BSP model that is employed by Pregel. In the GAS model, a vertex collects information about its neighbourhood in the Gather phase, performs the computations in the *Apply* phase, and updates its adjacent vertices and edges in the *Scatter* phase. As a result, in GraphLab, graph vertices can directly *pull* their neighbours' data (via Gather) without the need to explicitly receive messages from those neighbours. In contrast, in the BSP model of Pregel, a vertex can learn its neighbours' values only via the messages that its neighbours *push* to it. The P*owerGraph* system [17] has extended the GraphLab system by introducing a partitioning scheme that cut the vertex set in a way that the edges of a high-degree vertex are handled by multiple workers. Therefore, as a tradeoff, vertices are replicated across workers, and communication among workers

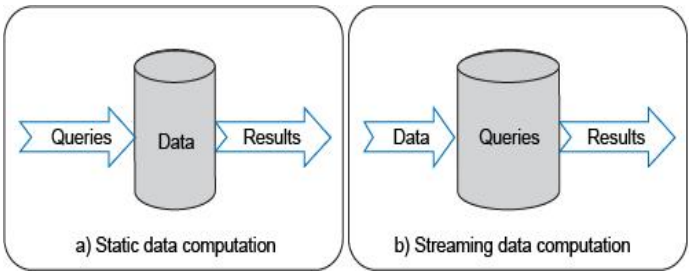

Fig. 3: Static data computation versus streaming data computation

are required to guarantee that the vertex value on each replica remains consistent. Trinity[37] is a memory-based distributed system which focuses on optimizing memory and communication cost under the assumption that the whole graph is partitioned across a memory cloud [30]. Trinity is designed to support fast graph exploration as well as efficient parallel graph computations. In particular, Trinity organizes the memory of multiple machines into a globally addressable, distributed memory address space (a memory cloud) to support large graphs. In addition, Trinity leverages graph access patterns in both online and offline computation to optimize memory and communication for best performance.

IV. BIG STREAM PROCESSING SYSTEMS

Stream computing is a new paradigm necessitated by new data-generating scenarios, such as the ubiquity of mobile de-vices, location services, and sensor pervasiveness. In general, stream processing systems support a large class of applications in which data are generated from multiple sources and are pushed asynchronously to servers which are responsible for processing. Therefore, stream processing applications are usually deployed as continuous jobs that run from the time of their submission until their cancellation. Figure 3 illustrates the difference between the computations performed on static data and those performed on streaming data. In particular, in static data computation, questions are asked of static data. In streaming data computation, data is continuously evaluated by static questions.

In principle, the basic architecture of the MapReduce framework requires that the entire output of each map and reduce task to be materialized into a local file before it can be consumed by the next stage. This materialization step allows for the implementation of a simple and elegant checkpoint/restart fault tolerance mechanism. The *MapReduce Online approach* [10], [11] have been proposed as a modified architecture of the MapReduce framework in which intermediate data is *pipelined* between operators while preserving the programming interfaces and fault tolerance models of previous MapReduce frameworks. In this approach, the Hadoop imple-mentation is modified so that each reduce task contacts every map task upon initiation of the job and opens a TCP socket which will be used to pipeline the output of the map function. As each map output record is produced, the mapper determines which partition (reduce task) the record should be sent to, and immediately sends it via the appropriate socket. A reduce task accepts the pipelined data it

[34] http://www.cse.cuhk.edu.hk/pregelplus/
[35] http://pregelix.ics.uci.edu/
[36] http://graphlab.org/
[37] http://research.microsoft.com/en-us/projects/ trinity/

receives from each map task and stores it in an in-memory buffer. Once the reduce task learns that every map task has completed, it performs a final merge of all the sorted runs. In addition, the reduce tasks of one job can optionally pipeline their output directly to the map tasks of the next job, sidestepping the need for expensive fault-tolerant storage in HDFS for storing what is essentially a temporary file. However, the computation of the reduce function from the previous job and the map function of the next job cannot be overlapped as the final result of the reduce step cannot be produced until all map tasks have completed, which prevents effective pipelining.

The M^3 system [2] has been proposed to support the answer-ing of continuous queries over streams of data by bypassing the HDFS so that data gets processed only through a main-memory-only data-path that totally avoids any disk access. In this approach, Mappers and Reducers never terminate where there is only one MapReduce job per query operator that is continuously executing. In M^3, query processing is incremental where only the new input is processed, and the change in the query answer is represented by three sets of inserted (*+ve*), deleted (*-ve*) and updated (*u*) tuples. The query issuer receives as output a stream that represents the deltas (incremental changes) to the answer. Whenever an input tuple is received, it is transformed into a modify operation *(+ve, -ve* or u) that is propagated in the query execution pipeline, producing the corresponding set of modify operations in the answer. Supporting incremental query evaluation requires that some intermediate state be kept at the various operators of the query execution pipeline. Therefore, Mappers and Reducers run continuously without termination, and hence can maintain main-memory state throughout the execution.

The *Stormy system* [21] has been presented as a distributed stream processing service for continuous data processing that relies on techniques from existing Cloud storage systems that are adapted to efficiently execute streaming workloads. It uses distributed hash tables (DHT) [3] to distribute queries across all nodes, and route events from query to query according to the query graph. To achieve high availability, Stormy uses a replication mechanism where queries are replicated on several nodes, and events are concurrently executed on all replicas. As Stormy's architecture is by design decentralized, there is no single point-of-failure. However, Stormy uses the assumption that a query can be completely executed on one node. Thus, there is an upper limit on the number of incoming events of a stream.

The *Storm* system has been presented by Twitter as a distributed and fault-tolerant stream processing system that instantiates the fundamental principles of Actor theory [9]. The core abstraction in Storm is the *stream*. A *stream* is an unbounded sequence of *tuples*. Storm provides the primitives for transforming a stream into a new stream in a distributed and reliable way. The basic primitives Storm provides for performing stream transformations are *spouts* and *bolts*. A spout is a source of streams. A *bolt* consumes any number of input streams, carries out some processing, and possibly emits

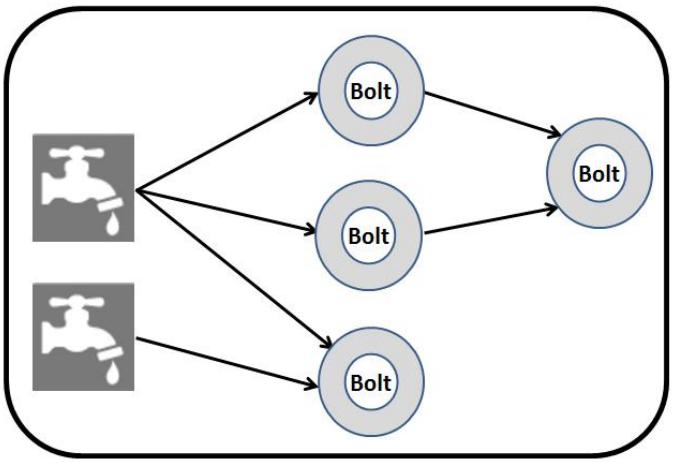

Fig. 4: Sample Storm Topology

new streams. Complex stream transformations, such as the computation of a stream of trending topics from a stream of tweets, require multiple steps and thus multiple bolts. A *topology* is a graph of stream transformations where each node is a spout or bolt. Edges in the graph indicate which bolts are subscribing to which streams. When a spout or bolt emits a tuple to a stream, it sends the tuple to every bolt that subscribed to that stream. Links between nodes in a topology indicate how tuples should be passed around. Each node in a Storm topology executes in parallel. In any topology, we can specify how much parallelism is required for each node, and then Storm will spawn that number of *threads* across the cluster to perform the execution. Figure 4 depicts a sample Storm topology.

The Storm system relies on the notion of *stream grouping* to specify how tuples are sent between processing components. In other words, it defines how that stream should be partitioned among the bolt's tasks. In particular, Storm supports different types of stream groupings such as: 1) *Shuffle grouping* where stream tuples are randomly distributed such that each bolt is guaranteed to get an equal number of tuples. 2) *Fields grouping* where the tuples are partitioned by the fields specified in the grouping. 3) All *grouping* where the stream tuples are replicated across all the bolts. 4) *Global grouping* where the entire stream goes to a single bolt. In addition to the supported built-in stream grouping mechanisms, the Storm system allows its users to define their own custom grouping mechanisms.

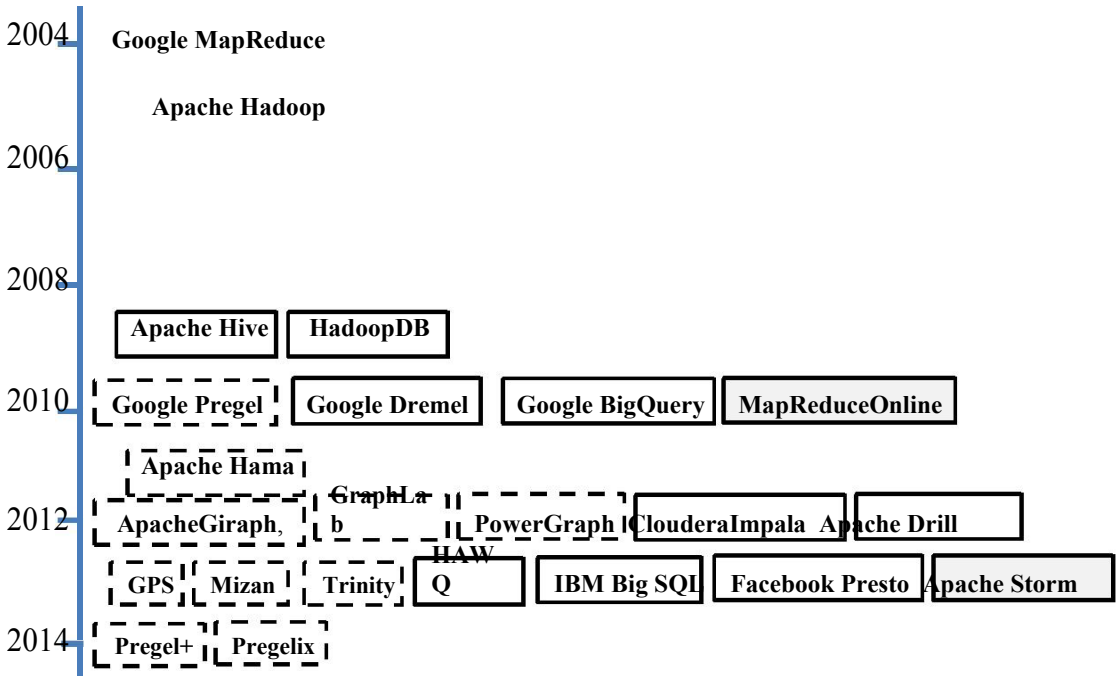

Fig. 5: Timeline representation of big data processing platforms. Solid rectangles denote the Big SQL processing platform, dashed-lined rectangles denote large scale graph processing platforms, grey-filled rectangles denote large scale stream processing systems.

V. OPEN CHALLENGES

Figure 5 provides a timeline representation of the big data processing platforms. As we can notice, there has been increasing interest from both industry and academia (especially starting from 2010) on developing a new generation of domain-specific and optimized big data processing systems. In this section, we discuss some of the research challenges which, we believe, need to be addressed in order to ensure that the vision of designing and implementing successful and scalable new generation of big data processing platform can be achieved and can significantly contribute to the goal of conducting efficient and effective large scale data analytics in different application domains. For example, in the early days of the Hadoop framework, the lack of declarative languages to express the large scale data processing tasks has limited its practicality and the wide acceptance and usage of the framework [28]. Therefore, several systems (e.g., Hive) have been introduced to the Hadoop stack to fill this gap and provide higher-level languages for expressing large scale data analysis tasks on Hadoop. In practice, these languages have seen wide adoption in the industry and research communities. Currently the systems/stacks of large scale stream or graph processing platforms are suffering from the same challenge. They are still relying on low level APIs which might limit their adoption by large numbers of users. Therefore, we believe that it is beyond doubt that high level language abstractions that ease the user's job for expressing their stream or graph analytics jobs and enable the underlying systems/stack to perform automatic optimization are crucially required and represent an important research direction to enrich this domain.

In practice, one of the common scenarios is that users need to be able to break down the barriers between data silos such that they are able to design computations or analytics that combine different types of data (e.g., structured, unstructured, stream, graph). Currently, most of the large scale data processing platforms have the limitation that they are not able to seamlessly integrate with the vast ecosystem of other analytics systems. In addition, there is, yet, no clear vision or right abstractions or frameworks that can imagine how such integration could be achieved. Therefore, we believe that further research and development is still required to tackle this important challenge.

With the emergence of big data processing platforms, several studies have attempted to assess the different performance characteristics of these systems using different experimental settings or frameworks [4], [5], [16], [18], [25]. Unfortunately, there is still a clear lack of expertise and in-depth understanding of the performance characteristics for most of these systems. In addition, most of the reported benchmarking studies have been self-designed and there is a clear lack of *standard* benchmarks that can be employed for building the required depth and common global understanding. This is a clear gap that requires more attention from the research community in order to guide and improve the significance of the outcomes of such evaluation and benchmarking studies.

VI. CONCLUSIONS

Although the MapReduce framework has been considered at some stage as the one-size-fits-all solution for big data processing problems, recently, the framework has shown to have limited ability to implement scalable analytic computations over large-scale graph or stream processing data. In addition, the framework has been shown to be inefficient for processing large scale structured data. As a result, a new wave of domain-specific optimized systems has been introduced to tackle these challenges resulting in the creation of the new generation of big data systems. To this end, we presented a detailed overview of the state-of-the-art of the emerging platforms in the big data processing world. In addition, we identified and presented a set of the current open research challenges and also presented some of the promising directions for future research. In general, we believe that there are still many opportunities for new innovations and optimizations in the domain of large scale graph processing. Hence, we consider this article as an important step on helping researchers to understand the domain and guiding them towards the right direction to improve the state-of-the-art.

ACKNOWLEDGEMENT

This work is supported by King Abdulaziz City for Science and Technology (KACST), project 11-INF1992-03.

REFERENCES

[1] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. PVLDB, 2(1):922–933, 2009.

[2] Ahmed M. Aly, Asmaa Sallam, Bala M. Gnanasekaran, Long-Van Nguyen-Dinh, Walid G. Aref, Mourad Ouzzaniy, and Arif Ghafoor. M^3: Stream Processing on Main-Memory MapReduce. In ICDE, 2012.

[3] Hari Balakrishnan, M. Frans Kaashoek, David R. Karger, Robert Morris, and Ion Stoica. Looking up data in p2p systems. Commun. ACM, 46(2):43–48, 2003.

[4] Ahmed Barnawi, Omar Batarfi, Seyed Behteshi, Radwa Elshawi, Ayman Fayoumi, Reza Nouri, and Sherif Sakr. On Characterizing the Perfor-mance of Distributed Graph Computation Platforms. In TPCTC, 2014.

[5] Ivan Bedini, Sherif Sakr, Bart Theeten, Alessandra Sala, and Peter Cogan. Modeling performance of a parallel streaming engine: bridging theory and costs. In ICPE, pages 173–184, 2013.

[6] Yingyi Bu, Vinayak R. Borkar, Jianfeng Jia, Michael J. Carey, and Tyson Condie. Pregelix: Big(ger) Graph Analytics on a Dataflow Engine. PVLDB, 8(2):161–172, 2014.

[7] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. The HaLoop approach to large-scale iterative data analysis. VLDB J., 21(2):169–190, 2012.

[8] Lei Chang, Zhanwei Wang, Tao Ma, Lirong Jian, Lili Ma, Alon Goldshuv, Luke Lonergan, Jeffrey Cohen, Caleb Welton, Gavin Sherry, and Milind Bhandarkar. HAWQ: a massively parallel processing SQL engine in hadoop. In SIGMOD, pages 1223–1234, 2014.

[9] William D Clinger. Foundations of Actor Semantics. Technical report, Cambridge, MA, USA, 1981.

[10] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In NSDI, pages 313–328, 2010.

[11] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. Online aggregation and continuous query support in MapReduce. In SIGMOD Conference, pages 1115–1118, 2010.

[12] Jeffrey Dean and Sanjay Ghemawa. MapReduce: Simplified Data Processing on Large Clusters. In OSDI, pages 137–150, 2004.

[13] David J. DeWitt, Alan Halverson, Rimma V. Nehme, Srinath Shankar, Josep Aguilar-Saborit, Artin Avanes, Miro Flasza, and Jim Gramling. Split query processing in polybase. In SIGMOD, pages 1255–1266, 2013.

[14] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative MapReduce. In HPDC, pages 810–818, 2010.

[15] Vinitha Reddy Gankidi, Nikhil Teletia, Jignesh M. Patel, Alan Halver-son, and David J. DeWitt. Indexing HDFS Data in PDW: Splitting the data from the index. PVLDB, 7(13):1520–1528, 2014.

[16] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. BigBench: towards an industry standard benchmark for big data analytics. In SIGMOD, pages 1197–1208, 2013.

[17] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In OSDI, pages 17–30, 2012.

[18] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Ozsu, Xingfang Wang, and Tianqi Jin. An Experimental Comparison of Pregel-like Graph Processing Systems. PVLDB, 7(12):1047–1058, 2014.

[19] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther¨ Hagleitner, Eric N. Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. Major technical advancements in Apache Hive. In SIGMOD, pages 1235–1246, 2014.

[20] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In EuroSys, pages 169–182, 2013.

[21] Simon Loesing, Martin Hentschel, Tim Kraska, and Donald Kossmann. Stormy: an elastic and highly available streaming service in the cloud. In EDBT/ICDT Workshops, pages 55–60, 2012.

[22] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed GraphLab: A Frame-work for Machine Learning in the Cloud. PVLDB, 5(8):716–727, 2012.

[23] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In SIGMOD Conference, pages 135–146, 2010.

[24] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. PVLDB, 3(1):330–339, 2010.

[25] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In SIGMOD, pages 165–178, 2009.

[26] Sherif Sakr. GraphREL: A Decomposition-Based and Selectivity-Aware Relational Framework for Processing Sub-graph Queries. In DASFAA, 2009.

[27] Sherif Sakr, Sameh Elnikety, and Yuxiong He. G-SPARQL: a hybrid engine for querying large attributed graphs. In CIKM, 2012.

[28] Sherif Sakr, Anna Liu, and Ayman G. Fayoumi. The family of mapreduce and large-scale data processing systems. ACM Computing Surveys, 46(1):11, 2013.

[29] Semih Salihoglu and Jennifer Widom. GPS: a graph processing system. In SSDBM, page 22, 2013.

[30] Bin Shao, Haixun Wang, and Yatao Li. Trinity: a distributed graph engine on a memory cloud. In SIGMOD, pages 505–516, 2013.

[31] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In SIGMOD, pages 1013–1020, 2010.

[32] Leslie G. Valiant. A Bridging Model for Parallel Computation. Commun. ACM, 33(8):103–111, 1990.

[33] Yanfeng Zhang, Qinxin Gao, Lixin Gao, and Cuirong Wang. iMapRe-duce: A Distributed Computing Framework for Iterative Computation. J. Grid Comput., 10(1):47–68, 2012.