Contents

List of Figures

List of Tables

CHAPTER 1

Scalable Query and Analysis for Social Networks: An Integrated High-Level Dataflow System with Pig and Harp

Tak-Lon (Stephen) Wu
Indiana University

Bingjing Zhang
Indiana University

Clayton Davis
Indiana University

Emilio Ferrara
Indiana University

Alessandro Flammini
Indiana University

Filippo Menczer
Indiana University

Judy Qiu
Indiana University

CONTENTS

Every day, vast amounts of data are being collected from social network (e.g., Twitter) applications, and in response there is a growing need for analysis methods that can handle this terabyte-size input. To provide an effective and advanced data processing environment for various types of social data analysis such as political discourses, trending topics, evolution of user behavior, social bots detection and orchestrated campaigns, we need to support both query and complex analysis efficiently. Use of high-level scripting languages to solve Big Data problems has become a mainstream approach for sophisticated data mining and analysis. In particular, high-level interfaces such as Pig, Hive, and Spark SQL are being used on top of the Hadoop framework. This simplifies coding of complex tasks in MapReduce-style systems while improving the flexibility of database systems through user-defined aggregations. In this chapter we will compare different approaches of building high-level Dataflow Systems and propose an integrated solution with Pig and Harp (a plugin to Hadoop) along with giving extensive benchmarks. The results show that Pig and Harp integration for sophisticated iterative applications runs at a factor of 2 to

10 times faster than Pig or Hive implementation executed on Hadoop.

1.1 INTRODUCTION

Social media represents a precious data source providing tremendous amounts of streaming information for analytics and research applications. Many research projects are involved in performing intensive analysis on such data, and the outcome of this analysis is drawing the attention of various applications, including market sales analysts, societal studies (including political polarization [10], congressional elections [14, 13], protest events [12, 11], and the spread of misinformation [38, 37]) and information diffusion [24]. Compared to other problems in computing, social media analysis is "special"; it normally focuses on a subset of data related to a target social event within a specific time frame. To further investigate the inter-relationship of such subsets of data, various sophisticated algorithms and complex data transformations may be applied into a series of stages [19]. Therefore, developing a programmable solution for social media data must include features like expressiveness, ability for data extraction, reusability and interoperability with different computation runtimes. Apache high-level languages and Apache Hadoop [1] ecosystem are some of the existing building block solutions that match the requirements for social network analysis.

The use of high-level language platforms is not just limited to social media data. Other fields of research such as workflow provenance [7], network traffic analysis [26, 23], and geographic data analysis [6] have proved the adaptation of these solutions boosts and scales up their historical data analysis. However, the complex workflows characterizing existing platforms makes it difficult for users to decide what language and low level runtimes best match their needs. Motivated by these challenges, our goal is to provide a comprehensive survey of these high-level abstractions involving experiments with real social media data examples and common query and analysis applications.

The rest of the paper is organized as follows. Section 1.2 gives an overview of Apache high-level languages, especially Pig [22], Hive [40] and Spark SQL [2, 44]. The first two build on Hadoop while Harp [47] and Spark [46] are Apache iterative MapReduce frameworks offering support to complex parallel data systems. Section 1.3 provides a comparison of these languages' features especially the important user-defined functions that make MapReduce a simplified and scalable solution. Sections 1.4 and 1.5 introduce applications that are used for benchmarking later in the chapter.; Section 1.4 introduces the *Truthy project* and the types of queries that it needs to run on top of Twitter data, while Section 1.5 discusses three data analytics use-cases and how to express them in high-level languages. Section 1.6 presents the performance evaluation of the applications presented in Sections 1.4 and 1.5, and the technologies of Section 1.2. Section 1.7 draws our conclusions.

1.2 APACHE HIGH-LEVEL LANGUAGE, SYNTAX AND ITS COMMON FEATURES

Programming languages have been developed for more than 50 years. Each language has its own compiler/interpreter and executes a physical plan on top of the low level (operating) system. Apache high-level languages share the common features of traditional programming languages; in many cases, a compiler built for such a language supports several fundamental functions and operations: a syntax parser, type and compile time semantic checking, logical plan generator and optimizer, and physical plan generator and executor. Here ANTLR (ANother Tool for Language Recognition) [34] is the general syntax parser for Pig, Hive, and Spark SQL. Each language has its own types and plan generator and optimizer, but all of them use YARN [42] as their resource management tool. The next sections will discuss details of Apache Pig, Apache Hive and Apache Spark SQL.

1.2.1 Pig

Pig is a high-level dataflow system that yields simple data transformations in pipeline for large amounts of semi-structured data stored in Hadoop compatible file storage. Applications such as massive system log analysis and traditional Extract, Transform, and Load (ETL) data processing are performed regularly. Pig was first introduced by Yahoo!, and became one of the most popular Hadoop ecosystem projects in the Apache open source community. It uses its built-in procedural language, *Pig Latin* [32], designed for large-scale data analysis with Hadoop MapReduce. The syntax is straightforward so long as the developer is familiar with UNIX bash scripting. Pig hides complicated MapReduce programs with simple notations for a dataflow program. Internally, Pig scripts are compiled into sequences of MapReduce jobs,

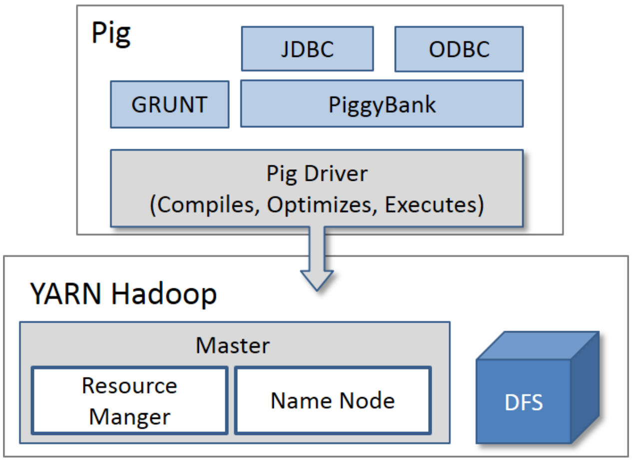

Figure 1.1: Pig's Architecture

which automate parallelization and make the code easy to maintain. Pig also provides an interactive shell interface named GRUNT that generates MapReduce jobs which depend on the type-in lines. Figure 1.1 depicts an overall architecture of Pig. As shown, Pig is standalone and can run as a Java client on any worker node.

When a user submits their Pig scripts in a batch mode or enters line-by-line data transformation commands in an interactive mode, a default compiler handles the overall execution flows. This compiler translates the entered Pig scripts into operators and forms top-down Abstract Syntax Trees (AST) in different stages. It then visits the last compiled AST from the MapReduce operators plan compiler and constructs MapReduce jobs in order. Figure 1.2 shows the dataflow and lists all major steps. Similar to any programming language, Pig checks syntax by parsing the user-submitted script into a parser written in ANTLR. It then generates a logical LOP (Logical Operator Plan) for further optimization. Generally a logical rules-based optimization is performed without looking at the real data (this is different from traditional SQL or SQL-like technologies that take data schema as part of the rules-based optimization). Pig's main driver program converts each MapReduce operator from Map-Reduce Operator Plan (MROperPlan) objects into Hadoop JobControl objects with detailed descriptions, input/output linkages, and other parameters, which are then passed along to each worker node with a configuration in xml format. These translations generate Java jar files that contain the Pig default Map and Reduce classes, including the user-defined functions. The packages

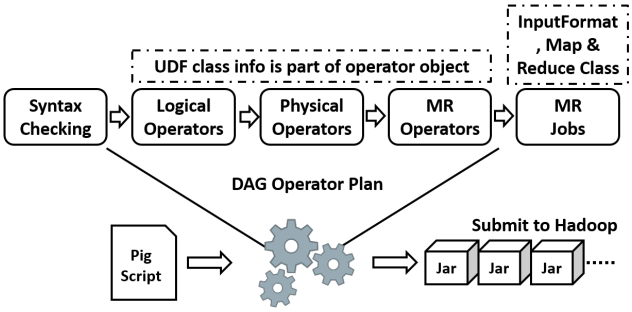

Figure 1.2: Pig High-Level Dataflow

of jar files are submitted to Hadoop Job Manager, and job progress is monitored until completion of the tasks.

An example of a WordCount program written in Pig Latin [32] is provided in Figure 1.3. In a Pig dataflow, each line of code has only one data transformation, which can be nested. The WordCount program consists of seven lines of code, and the syntax is straightforward and easy to understand. Generally, data is loaded as records in a relation/outer bag, and each field in a record is defined according to Pig's default data types: bag, tuple, and field. A bag is a set of unordered columnar tuples. A tuple is a set of fields, where tuples in a bag can contain flexible length of fields, and fields in the same column can have different data types. Lastly, a field is the basic type of a piece of data. Then, based on the supported data types, a developer applies the desired data transformation and generates their results.

In our example, the first line defines an outer bag input and loads a text document from HDFS. Each line of this text file is declared as string (chararray in Pig Latin). The second and third line further converts each line into English words and creates for each individual word a single tuple by using the built-in function TOKENIZE and relational statement FILTER. The fourth line aggregates instances of the exact same word together and constructs a two-cells tuple for each word. Here, the first cell of this tuple stores the text of this word, while the second cell stores a list of the same word. The List length is the total number of occurrences of this word. The fifth line counts the amount of word items in the list and emits a word count pair for each word <word, occurrences>. Line six uses the built-in order statement and reorders the WordCount result with descending order. Finally, line seven stores the ordered result into default file storage. Other than the syntax shown in this paper, Pig provides operations and syntax patterns for various data transformations, although the current version of Pig does not support optimized storage structures such as indices and column groups.

```
1 input    = LOAD 'input.txt'
             AS (line:chararray);
2 words    = FOREACH input GENERATE
             FLATTEN(TOKENIZE(line)) AS word;
3 filWords = FILTER words BY word MATCHES '\\w+';
4 wdGroups = GROUP filWords BY word;
5 wdCount  = FOREACH wdGroups GENERATE
             group AS word,
             COUNT(filWords) AS count;
6 ordWdCnt = ORDER wdCount BY count DESC;
7 STORE ordWdCnt INTO 'result';
```

Figure 1.3: WordCount written in Pig [5]

```
1. CREATE TABLE doc (line STRING);
2. LOAD DATA INPATH '$documentsPath'
   OVERWRITE INTO TABLE doc;
3. INSERT INTO OVERWRITE DIRECTORY '$outputPath'
   SELECT word, count(1) AS count FROM
   (SELECT explode(split(line, '\s'))
    AS word FROM doc) words
   GROUP BY word
   ORDER BY word;
```

Figure 1.4: WordCount written in Hive

Pig performs well for ETL applications, but it does not directly support iterative computations. This implies that Pig can execute simple one-pass algorithms but not complex functions that need to apply a computation repeatedly (e.g., for loop) which exist in graph, linear algebra, and expectation-maximization computations. To write such general data analysis applications using Pig, the control flow should be similar to what is shown in Figure 1.5: an external wrapper script is required because Pig syntax does not provide control flow statements. This causes extra overhead of job startup and cleanup time when a program runs in several rounds of MapReduce jobs. Furthermore, inputs of iterative applications are normally unchanged and cacheable between iterations, whereas Pig has a DAG framework that does not cache those inputs in memory and reuses data efficiently.

To generalize the usage of Pig for scientific applications, we need to enable loop-awareness computation and in-memory caching; our re-

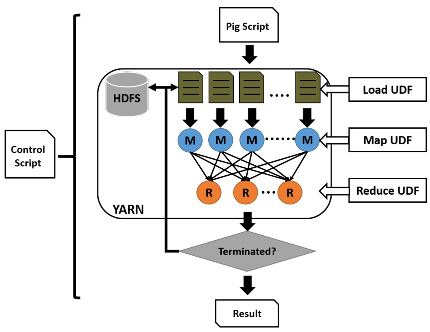

Figure 1.5: Iterative applications with Pig

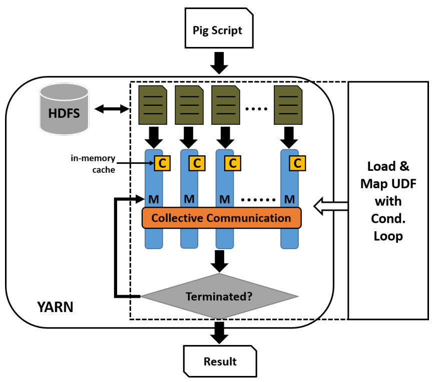

Figure 1.6: Iterative applications with Pig+Harp

search project yielded a version of Pig for scientific applications based on the DAG computation model. There are several iterative MapReduce frameworks available as candidates to integrate with Pig, including Twister [15], Spark [46], HaLoop [9], and Harp [47]. We chose Harp as it is a plug-in to Hadoop that supports our required iteration features, the result being referred to from here on as Pig+Harp [43]. With Harp integration, we replace the Hadoop Mapper interface with Harp's MapCollective, a long-running mapper to support conditional loops. Subsequently, iterative applications implemented in Pig+Harp can cache reusable data and replace the default GROUP BY operation with Harp's collective communication interface featuring high-performance data movement. Figure 1.6 shows a dataflow that can be applied to iterative applications

1.2.2 Hive

Hive is a data warehouse solution for ad-hoc queries, from simple data summarization to business intelligence applications and high-latency queries for extremely large structured data sets stored on top of Hadoop related file storage. Initially developed by the Facebook data infrastructure team, it is used for filtering and summarization of information from their massive amount of stored social network data and support products associated with the collected data. Thousands of Hive jobs were submitted daily since 2010 [8]. Hive uses a SQL-like language named HiveQL which is very attractive for the traditional SQL community. Similar to Pig, HiveQL queries are compiled into MapReduce jobs and executed on top of Hadoop. Hive reintroduces a RDBMS technique - Metastore- that stores data schemas and statistics as a service of an in-memory system catalog to facilitate Hive's compiler and data scanning. Figure 1.7 shows the architecture of Hive.

When a user submits HiveSQL statements via any supported APIs, Hive initially checks the syntax by an ANLTR parser, then cooperates with Metastore for further type checking and semantic analysis, and lastly generates an initial AST as a logical plan. This plan is then optimized through a rule-based optimizer involving the schema and indices metadata obtained from Metastore. Optimizations such as column pruning, pushdown, partition pruning, mapside joins, and join reordering are also performed. Finally, a physical plan is generated from the optimized logical plan and submits a sequence of MapReduce jobs to Hadoop cluster.

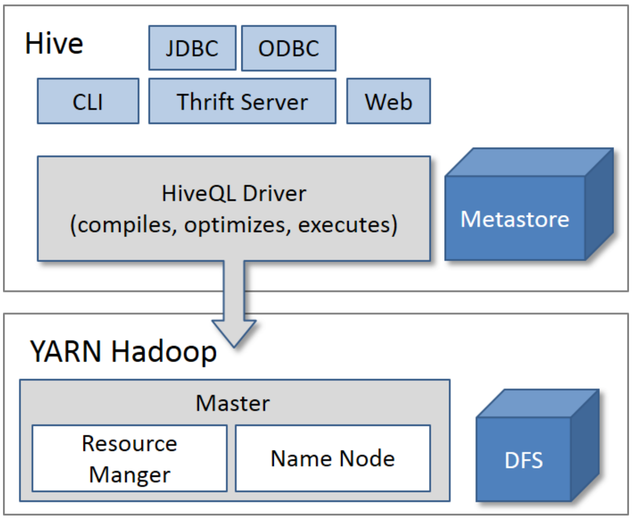

Figure 1.7: Hive Architecture

Hive supports nested statements, and each statement represents a single data transformation. In Figure 1.4, we see a WordCount program written in HiveQL. Hive supports nested statements, and each statement represents a single data transformation. The first line declares a table named doc with only a string column line. The second line reads files from the given path and overwrites the table doc. The third line is a nested statement that splits all words of each record of lines in table “doc”, then groups all emitted (word, 1) pairs from the temporary table “words” in decreasing order with their occurrence. As shown below, the overall syntax is very SQL friendly.

By default, Hive is compatible with local file systems HDFS [39] and HBase [4]. A user is required to provide data schema by creating tables before accessing the files in storage. For instance, prior to reading existing tables in HBase, users need an additional step to make tables in Metastore and link the schema of Hive to the HBase tables, such as row key and column families of the reading tables.

1.2.3 Spark SQL/Shark

Spark SQL [2] and Shark [44] are other open source projects directly inspired by Hive. Both use Spark runtime and RDD [45] as the core engine to execute their physical plan on top of YARN. Spark SQL is the latest release replacing Shark, now merged as a branch project under the Spark ecosystem.

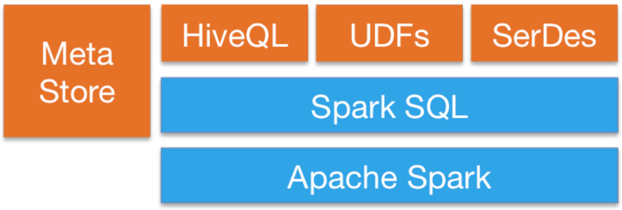

Figure 1.8: Spark SQL Architecture [2]

Spark SQL reuses Hive's query parser to generate a logical operator plan. With this compatibility support, general Hive queries can run on Spark SQL without any changes to the execution script. Spark SQL has its own rule-based logical operator plan optimizer for matching the physical operators that run on Spark. As claimed by Spark runtime developers, this allows Spark SQL queries to run better on RDD operations and best match the Spark execution model, rather than tuning Spark low-level execution to support Hive's Hadoop implementation. The architecture of Spark SQL appears in Figure 1.8. Spark SQL can support most of the HiveQL statements with several limitations; e.g., bucketed tables in Hive are not currently supported in Spark SQL.

1.3 PIG, HIVE AND SPARK SQL COMPARISON

Before comparing the differences between Pig, Hive and Spark SQL, we need to look into two fundamental terms: dataflow system and data warehouse system.

Dataflow system is a type of data processing system where data is transformed from one format to another via different processing units in a directed path. Data can be structured (with a predefined schema) or unstructured (e.g., logs); it therefore requires customized data selection and operations in order to extract meaningful information. Pig falls into this category. Data warehouse is a system that handles cleaned, structured and cataloged data in organized hierarchical data storage units. Data is available to observers for conducting data analysis. Hive and Spark SQL are designed for the data warehouse community.

Table 1.8 gives a comprehensive comparison between Pig, Hive and Spark SQL. Even though they are designed for different systems and applications, these three tools share many common features, operations and functions.

Pig can be used for unstructured raw data batch processing and simple statistical analytics, especially with massive logs for text mining. It is more like an alternative to Hadoop MapReduce applications in high-level abstraction with an extensible subset of general data operations. Data is stored in HDFS or HBase with high-latency data scanning operations. Pig scans data entirely (i.e., it must scan all data for filtering fields with numeric type less than 10) without the help of data indices. As such, it is considered "slower" in supporting ad-hoc queries than Hive.

Meanwhile, Hive and Spark SQL are SQL-like distributed systems that run high-latency queries for data sets stored on top of MapReduce (HDFS) file system. Hive still scans data from disk or HDFS directly for assigned map/reduce tasks. Metastore provides the data schema and indices while scanning the data. Spark SQL uses Hive's query parser and Metastore generates the operator plan, but it then uses Catalyst as its logical plan generator and optimizer (Shark uses Hive's query planner), executes with Spark, and stores the processed/queried data into DataFrame (columnar RDD in Shark) instead of files on HDFS. Here rule-based optimizations of Hive/Shark and Spark SQL are expandable. The use of RDD provides in-memory reusable access to the scanning data. It saves significant disk I/O and job restart overhead if the data is hit frequently, especially when the cases of mixing ad-hoc queries and further sophisticated applications are involved within the lines of the same submitted program. Spark SQL is still a newly released ongoing project, so some query plan optimizations of Hive/Shark are not included in Spark SQL such as block level bitmap indices and virtual columns. Catalyst is the core difference between Shark and Spark SQL. DataFrame is a special type of row objects RDD that has associated data schema such as column field name and data type as a collection of named and typed tuples. It can then support operations from the submitted relational queries in line. In addition, with the help of the known type of the row objects, DataFrame can be cached with better compression than general RDD objects.

All of Pig, Hive and Spark SQL introduce User-Defined Functions (UDF) for advanced tuple/record-based data transformation, which enables the possibility to implement special computation and sophisticated algorithms in addition to the basic queries.

1.4 AD-HOC QUERIES: TRUTHY AND TWITTER DATA

Ad-hoc queries are the most common benchmark for ETL applications. This also applies to Apache high-level languages, which are mainly designed to support ETL operations. Here we use Truthy project and Twitter-generated social media data to evaluate ad-hoc query performance among these runtimes.

Truthy [30, 21] is a public social media observatory developed as a research project at Indiana University. It analyzes and visualizes information diffusion on Twitter. Truthy monitors and collects Twitter data in real-time directly through the Twitter public steaming API. The overall size of compressed historical raw data from 2010 till April 2015 is about 3.2TB. IndexedHBase [19, 20] is used to store, load, and index this data as tables into HBase on a private large-storage, high-performance, and large-memory cluster called MOE. As of today, the overall data on HBase including the raw data tables and index tables (as well as the standard 3 replicas) occupies nearly 133TB of disk space. We expect to continue storing more historical data, as the Truthy team aims to perform innovative and large-scale social network research and analysis to understand how information propagates in complex socio-technical systems. Many researchers [14, 13, 12, 11, 38, 37] have built their prototypes, models and analyses based upon this complex infrastructure, which shows the capability to capture the spread of information, from political discourses to trending topics [17], the evolution of user behavior [41], and even the presence of social bots and orchestrated campaigns [16].

The data collected from the Twitter streaming API consists of tweets containing various attributes. The most common attributes used for intensive analysis include hashtags, user metadata, text and media content, retweets information, user mentions, and specific time intervals. Truthy identifies this information and utilizes the concept of "meme" [25] (a piece of data that corresponds to specific topics, communication channels, or shared elements by people in a social network) to construct a set of temporal queries for extracting tweets' information for further data intensive analysis. These queries can be classified into two categories [20]: ad-hoc queries for simple tweet retrieval with the help of index tables, and combination of tweet retrieval with extra data transformation. We only discuss the ad-hoc queries here because it is the best practice for matching the common features of high-level languages. Table 1.1 shows four different queries that firstly search the related index table and then redirect the obtained tweets back to the user.

Previous research [10, 19] utilized the above queries on top of further data mining techniques, such as eigenvector modularity [31] and label propagation [36], on two datasets about political discussion collected during the six weeks leading up to the 2010 U.S. congressional midterm elections and 2012 U.S. presidential elections. The results shown in [10, 19] prove that the retweet networks exhibited a highly-segregated partisan structure; users of those tweets are mainly split into two homogeneous communities corresponding to the political left and right leanings.

Table 1.1: TRUTHY'S AD-HOC QUERIES FOR SIMPLE TWEETS RETRIEVAL

Queries	**Description**	**Index Table Name**
get-tweets-with-meme	Search tweets with given memes such as hashtags, user-mentions, and URLs	memeIndexTable
get-tweets-with-text	Search tweets with given keywords	textIndexTable
get-tweets-with-user	Search tweets with given user information, e.g., user ID and screen name	userTweetsIndexTable
get-retweets	Search retweets with given tweet Ids	retweetIndexTable

1.5 ITERATIVE SCIENTIFIC APPLICATIONS

Many domain scientists who work on scientific applications use programming languages such as Python, Matlab and R to perform standard data-analysis tasks. Oftentimes, such analysis involves sophisticated data mining and machine learning techniques that must run in several rounds of computation to complete a full task. With the built-in mathematics and statistical operations in Pig and Hive, these two runtimes could be candidate tools to support scientific applications run with very large-scale data. Due to the fact that Pig and Hive do not support iterative applications directly, we need to extend them with an external wrapper script/program to handle the loop control and link the program inputs from HDFS between iterations. On the other hand, Pig+Harp integrated Harp's collective communication API to support iterative application as well as single pass in common Hadoop jobs for both high-level language tools.

```
1 raw     = LOAD $hdfsInputDir USING
            PigKmeans('$centroids',
            '$numOfCentroids') AS (datapoints
2 dptsBag = FOREACH raw GENERATE
            FLATTEN(datapoints) AS dptInStr;
3 dpts    = FOREACH dptsBag GENERATE
            STRSPLIT(dptInStr, ',', 5)
            AS splitedDP;
4 grouped = GROUP dpts BY splitedDP.$0;
5 newCens = FOREACH grouped GENERATE
            CalculateNewCentroids($1);
6 STORE newCens INTO 'output';
```

Figure 1.9: Pig K-means for a single iteration

```
1 CREATE EXTERNAL TABLE IF NOT EXISTS
  $INPUTTABLE (filename String)
  LOCATION '$hdfsInputDir ';
2 DROP TABLE IF EXISTS interKmeansTable;
3 CREATE TABLE interKmeansTable(x double, y double,
  z double, beta double)
  LOCATION '$hdfsOutputDir;
4 INSERT OVERWRITE TABLE interKmeansTable
  SELECT SUM(KmeansTable.ret.x)/SUM(KmeansTable.ret.count),
  SUM(KmeansTable.ret.y)/SUM(KmeansTable.ret.count),
  SUM(KmeansTable.ret.z)/SUM(KmeansTable.ret.count),
  0.0
  FROM
  (SELECT explode(Kmeans(INPUT__FILE__NAME,
  '$initCentroidOnHDFS', '$centroidSize')) AS ret
  FROM $INPUTTABLE T) KmeansTable
  GROUP BY KmeansTable.ret.assignedcentroid;
```

Figure 1.10: Hive K-means for a single iteration

1.5.1 K-means Clustering and PageRank

We present two popular algorithms, K-means clustering and PageRank computing, as our standard benchmarks. K-means clustering [29] is one of the standard clustering algorithms that has been widely used for finding distance and similarity among a set of objects with multidimensional vectors. In addition to various applications in social media analysis [18], studies involving large-scale image classification [27, 35] have used this technique on high-dimensional (over a hundred dimensions) SIFT descriptors [28]. This allowed for creating a training dataset of visual vocabulary to automatically determine specific characteristics of a given photo, e.g., whether it was taken in an urban or rural environment. Meanwhile, the PageRank [33] webpage-ordering algorithm was introduced by Google co-founders, Larry Page and Sergey Brin while they were researching the next generation of search engine in 1996. Eventually, this ranking algorithm became the key technology of the Google search engine, and it was applied to several general-purpose graph or network problems in bibliometrics, social network analysis, and recommendation systems. We show the difference in implementations below by using Pig and Hive.

Pig K-means consists of three components: a Python control-flow script, a Pig data-transform script for a single iteration, and two K-means user-defined functions written with a Pig provided Java interface. A single iteration of K-means written in Pig Latin is included in Figure 1.9. Our customized Loader yields the aggregated centroids into memory as vector objects loaded from the distributed cache on disk before computing the Euclidean distances for all data points at each iteration of the algorithm. Every loader outputs the assigned centroids and data points as fields in a single bag; each field in a bag is defined as string data type which further splits into tuples to match

```
1 centds = LOAD $hdfsInputDir USING
           HarpKmeans('$initCentroidOnHDFS',
           '$numOfCentroids', '$numOfMappers',
           '$iteration', '$jobID', '$Comm')
           AS (result);
2 STORE centroids INTO '$hdfsOutputDir';
```

Figure 1.11: Pig+Harp K-means

Pig's GROUP operation and collect partial centroid vectors from mappers. It then takes the average of all partitions, emits to a final centroids file and saves it to HDFS.

Hive K-means is written in SQL-like syntax and uses a UNIX bash script as the loop conditions wrapper for supporting iterations. Figure 1.10 depicts a single iteration of K-means written in HiveQL. Data points and centroids are originally stored on HDFS during each iteration, where they overwrite the intermediate centroids to HDFS. The default `INPUT__FILE__NAME` field provided by Hive is used, and our K-means UDF directly loads entire files for computation instead of using Hadoop InputFormat with a series of input splits. General data aggregations such as GROUP BY and SUM are executed, while Euclidean distance computation is handled by the K-means UDF.

Pig+Harp K-means script in Figure 1.11 illustrates a similar idea using R. Users only provide the parameters, such as number of mappers, total amount of iterations, and communication patterns used for global data synchronization. In the case of executing Pig+Harp K-means, a customized Loader in each Mapper first loads the initial centroids and data points from HDFS to memory and caches the data points for all iterations. Then the UDF computes Euclidean distances and emits partial centroids locally. The Harp communication layer then exchanges these partial centroids in each mapper. By default, Pig+Harp K-means UDF uses AllReduce to synchronize among all partitions. The program reuses the same set of mapper processes until exit conditions have been reached.

For *Pig PageRank*, we use a model with fewer UDF functions by leveraging Pig built-in operators. Figure 1.12 has a single iteration of the PageRank algorithm, which is created and iteratively invoked by a Java wrapper. The script involves the following steps: a) Load the given input file using the custom loader into variable raw; b) Extract the outgoing URLs and emit both outgoing URL and partial page rank from the source URL; c) CO-GROUP above two aliases to calculate new page rank and store it in an alias newPgRank; d) Store new page

```
1 raw       = LOAD '$InputDir' USING
              CmLoader('$noOfURLs','$itrs')
              AS (source,pagerank, out:bag);
2 prePgRank = FOREACH raw GENERATE
              FLATTEN(out) AS source,
              pagerank/SIZE(out) AS pagerank;
3 newPgRank = FOREACH (COGROUP raw BY source,
              prePgRank BY source OUTER)
              GENERATE
              group AS source,
              (1-$dpFactor) +
              $dpFactor*(SUM(prePgRank.pagerank)
              IS NULL?0:SUM(prePgRank.pagerank))
              AS pagerank,
              FLATTEN(raw.out) AS out;
4 STORE newPgRank INTO '$outputFile';
```

Figure 1.12: Pig PageRank for a single iteration

```
1 CREATE EXTERNAL TABLE pageRankInput(line String)
  LOCATION '$INPUTDIR';
2 CREATE TABLE PageRankComputeTable(pagerankCell
  struct<source:int,pagerank:double,outLinks:array<int>>)
  CLUSTERED BY(pagerankCell)
  INTO $MAP_SIZE BUCKETS
  LOCATION '$tmpPageRankResult';
3 INSERT OVERWRITE TABLE PageRankComputeTable
  SELECT InitialPageRank(line, '$numOfUrls') AS ret
  FROM pageRankInput;
4 INSERT OVERWRITE TABLE PageRankComputeTable
  SELECT named_struct('source', T1.pagerankCell.source,
  'pagerank', PageRank(T2.pagerank, $dpFactor, $noOfURLs),
  'outLinks', T1.pagerankCell.outLinks) AS cell
  FROM
  PageRankComputeTable T1
  LEFT OUTER JOIN
  (SELECT outlink,
   SUM(pagerankCell.pagerank/size(pagerankCell.outlinks))
   AS pagerank
   FROM PageRankComputeTable
   LATERAL VIEW
   explode(pagerankCell.outlinks) outLinkTable AS outlink
   Group by outlink) T2
  ON (T1.pagerankCell.source = T2.outlink);
```

Figure 1.13: Hive PageRank for a single iteration

```
1 pagerank = LOAD '$InputDir' USING
             HarpPageRank('$totalUrls',
             '$numMaps', '$itrs', '$jobID')
             AS (result);
2 STORE pagerank INTO '$output';
```

Figure 1.14: Pig+Harp PageRank

rank in a HDFS temp file, which will be the input file for the next iteration. One drawback of this program is that the default Pig runtime optimizer creates extra mappers for the final STORE step when it calls the raw and prePgRank variables for CO-GROUP operators, which utilizes extra computing and memory resources.

Hive PageRank follows a similar logic as Pig PageRank, but the HiveQL script uses tables as data abstraction and nested queries for computation, as well as OUTER LEFT JOIN seen in Figure 1.13.

In *Pig+Harp PageRank* implementation, we provide a new data loader UDF to calculate the probabilities for each web page. For the first iteration, data is loaded in a graph data structure where vertices are partitioned across all worker nodes. Each vertex receives all in-edges information by calling regroupEdges collective communication, and the number of out-edges is sent to all vertices by calling an AllMsgToAllVtx operation. The vertex and edge information is cached in memory for all iterations. Subsequently the PageRank values of each vertex are updated during every iteration and distributed by an AllGather communication until the program satisfies break conditions, e.g., the end of iterations. The script shown in Figure 1.14 is similar to that of Pig+Harp K-means.

1.6 BENCHMARKS

We have performed a set of extensive ad-hoc queries against Twitter's social network data using these high-level languages to illustrate their overheads and performance differences. We compare two scientific applications, K-means and PageRank, to evaluate the language expressiveness and performance in support of generic scientific algorithms in regards to high-level data abstractions, operations and execution flows. Currently we are not able to perform Spark SQL tests as the existing Spark SQL (latest version 1.3.1 as of April 2015) only supports a subset for HiveQL query and is best compatible with Hive 0.13.1. This limited compatibility causes our tests to fail. For example, when Spark SQL scans data from HBase, although the high-level abstraction StringType is used, Spark SQL in low level execution retrieves HBase's record as String instead of LazyString in Hive, which causes data loss to our ad-hoc queries test cases [3].

Our experiments run on MOE, a large-storage, large-memory and high-performance private cluster at Indiana University devoted to the Truthy project [30, 38, 21]. It consists of 3 login nodes and 10 compute nodes, where each login node is set up with two Intel(R) Xeon(R) CPU E5-2620 v2 CPUs, 64 GB memory, and each compute node has five Intel(R) Xeon(R) CPU E5-2660 v2 CPUs, 128 GB memory, 48TB HDD and 120GB SSD. All nodes are interconnected with a 10Gb Ethernet. Table 1.2 shows the specifications of MOE.

Table 1.2: HARDWARE SPECIFICATION OF MOE

	CPU	**RAM**	**DISK**	**NETWORK**
Login Node	2 x Intel(R) Xeon(R) CPU E5-2620 v2	64GB	120 SSD	10Gb Ethernet
Computer Node	5 x Intel(R) Xeon(R) CPU E5-2660 v2	128GB	48TB HDD + 120GB SSD	10Gb Ethernet

YARN Hadoop cluster on MOE is configured with a master on an independent login node. Meanwhile HBase uses another login as the master node and runs a ZooKeeper on each login node. YARN's NodeManager, HDFS's DataNode and HBase's RegionServer run on

Table 1.3: RUNTIME SOFTWARE SPECIFICATION OF MOE

	Version	**Memory**	**Disk**
YARN	2.5.1	66GB per node	48TB per node
Pig	0.14.0	2GB per worker	Data on HDFS
Hive	1.0.0	2GB per worker	Data on HDFS
Pig+Harp	0.14, 0.1.0	2GB per worker	Data on HDFS
HBase	0.94.23	30GB per node	Data on HDFS
IndexedHBase	0.94 branch	2GB per worker	Data on HBase

individual nodes and memory is shared among these processes. Table 1.3 specifies the software and runtime settings of MOE.

1.6.1 Performance of Ad-hoc Queries

We present the performance of running Truthy's queries on Twitter data using IndexedHBase in Figure 1.15. The query get-tweets-with-X initiates two steps; first it searches for tweet IDs from the related index table on HBase by given keys such as meme, text or user ID under a specified time interval. The second step reuses the obtained tweet IDs to scan related tweets from the raw tweets table in HBase and stores the retrieved tweets on HDFS. The overall performance is dominated by the total amount of retrieved tweet IDs. Table 1.4 displays the number of records obtained from each query; we use hashtag "#ff" as meme, keyword "NBA" as text, and randomly choose a user ID to search tables in the December 2012 dataset.

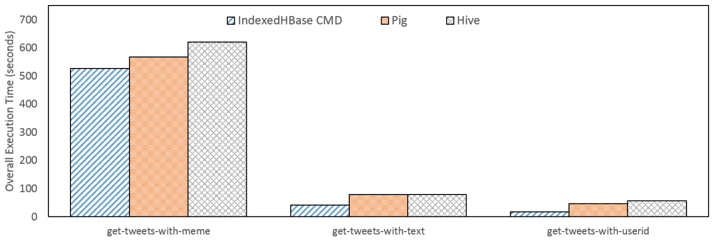

Figure 1.15: Truthy's get-tweets-with-X queries on Twitter data

The results in Figure 1.15 show that the IndexedHBase API command-line script outperforms the other solutions. This is because it calls an optimized Hadoop MapReduce job directly, and even the "search for tweet IDs" step is run as a local process. The Pig and Hive solutions execute these two steps in MapReduce jobs, therefore their performance is comparable. Hive requires more time for setting up the Table schema (including DROP and CREATE statements) in Metastore and Hive-related parameters in script for each query. As a result Hive performs the slowest in all of our tests. Table 1.5 lists the lines of code in script and the amount of submitted Hadoop jobs for each runtime based on query "get-tweets-with-X".

Table 1.4: SIZE OF RECORDS OBTAINED BY "get-tweets-with-X"

	get-tweets-with-meme	**get-tweets-with-text**	**get-tweets-with-userid**
# of Records	1570261	202076	22

Table 1.5: SCRIPT AND EXECUTION COMPARISON OF "get-tweets-with-X"

get-tweets-with-X	**IndexedHBase**	**Pig**	**Hive**
Lines of code in script	1	11	17
Hadoop job(s)	1	2	2
Map(s)/Reduce(s)	24/0	1/0, 24/0	1/24, 24/0

1.6.2 Performance of Data Analysis

We use K-means and PageRank to evaluate the difference in performance for our solutions. Both algorithms are implemented in the same dataflow logic but using different syntax in Pig, Hive and Pig+Harp implementations.

The tests for K-means algorithm are shown in Table 1.6, where we compute 10 iterations for two data sets: a 100 million 3-dimensional data points against 500 centroids, and a 100 million 3-dimensional data points against 5000 centroids. The dataset is split into 128 partitions running 128 mappers and 8 reducers. Each mapper or reducer runs on 1 CPU core with 2GB memory.

Table 1.6: SCRIPT AND EXECUTION COMPARISON OF K-MEANS

K-means	**Pig+Harp**	**Pig**	**Hive**
Lines of code in script	3	11	13
Hadoop job(s) per iteration	1	1	1
Map(s)/Reduce(s)	128/0	128/8	128/8

Figure 1.16 illustrates the total execution time for the K-means algorithm with each runtime. Pig+Harp outperforms the other two runtimes due to in-memory objects cache for the loaded data points

Table 1.7: SCRIPT AND EXECUTION COMPARISON OF PAGERANK

K-means	**Pig+Harp**	**Pig**	**Hive**
Lines of code in script	3	8	16
Hadoop job(s) per iteration	1	1	4
Map(s)/Reduce(s)	64/0	128/64	64/64, 64/64, 128/64, 64/64

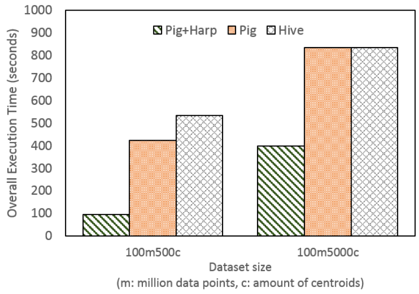

Figure 1.16: K-means Result

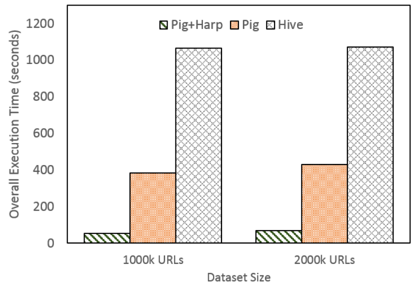

Figure 1.17: PageRank Result

and centroids, fast network I/O for data aggregation, and reduced overheads of the job restart between iterations. In contrast, the Pig and Hive implementations have a huge cost due to reloading, at each iteration, of the intermediate data points and centroids from HDFS. The execution plans for Pig and Pig+Harp are similar. Pig K-means generates 1 Hadoop job per iteration and Pig+Harp K-means generates a single job for all iterations.

In the PageRank test, we compute 10 iterations for two data sets: 1 million numeric URLs and a 2 million numeric URLs. The data is split into 64 partitions running 64 mappers and 64 reducers. Each mapper or reducer has 1 CPU core and 2GB memory.

Figure 1.17 presents the execution time of PageRank algorithm where Pig+Harp performs the best by storing the adjacency matrices as objects in memory, exchanging partial PageRank values via network I/O, and using long-running tasks.

As shown in Table 1.7, a maximum of 128 mappers (rather than our expected 64 mappers) are invoked for the partitions. This is due to the use of LEFT OUTER JOIN both in Pig and Hive implementation, and each partition is separately loaded in an extra mapper and prepared for the JOIN operations. In the case of Hive PageRank, although the HiveQL logic is the same as Pig's, Hive's physical plan executor generates a total of 4 Hadoop jobs per iteration, which results in a dramatic performance loss.

1.7 CONCLUSION

In this chapter, we investigated the Apache high-level languages and several runtimes of Hadoop ecosystems by conducting tests on real world applications with social media data. Terabytes of data streams are collected every day and stored on different large scale storage systems such as HDFS and HBase. Pig, Hive, and Spark SQL have been widely adopted by developers and domain scientists for rapidly building their prototypes and performing daily analysis tasks on both new and historical data. Although Pig and Hive have provided desirable features and performance for ad-hoc queries, these high-level abstractions lack support for interactive applications that require in-memory caches and fast job restart between iterations for sophisticated post-query data analysis. This chapter compares different approaches of building high-level Dataflow Systems and ultimately demonstrates an integrated solution with Pig and Harp (called Pig+Harp) that outperforms both Pig and Hive by a factor of 2 to 10 in overall metrics. Our experimental results show that these high-level languages and their integrations make it easier for users to perform data analysis, and improve the flexibility of database systems through user-defined aggregations.

Table 1.8: CROSS COMPARISON FOR PIG, HIVE AND SPARK SQL

	Pig	**Hive**	**Spark SQL**
Target System	Dataflow	Data warehouse	Data warehouse, then data analytic applications
Syntax	Pig Latin	HiveQL (SQL-like)	HiveQL (SQL-like)
Script Parser	ANTLR	ANTLR	ANTLR
Logical Plan Compiler	Script → AST → Operator Trees	Script → AST → Operator Trees (DML DDL by tables)	Catalyst
Logical Plan Optimizer	Operators Trees (Rules based)	Operator Trees (Rules based)	Operator Trees (Rules based)
Physical / MR Compiler	Operators Trees → MR jobs	Operators Trees → MR jobs	Operators Trees → Spark jobs on YARN
Structured or Unstructured Data	Unstructured, structured, nested Structured raw data	Structured tabular data	Structured tabular data
Catalog Services	HCatalog (Optional)	Metastore and HCatalog	Metastore and HCatalog
Primitives Data Type	INT, LONG, FLOAT, DOUBLE, CHARARRAY, etc.	TINYINT, SMALLINT, INT, BIGINT, FLOAT, DOUBLE, etc.	ByteType, ShortType, IntegerType, LongType, FloatType, DoubleType , etc. And most of Hive's Primitives DataType
Non-Primitives Data Type	map, tuple, bag	maps, arrays, structs, union	ArrayType, MapType, StructType

Relational Statements	GROUP, DEFINE, FILTER, FOREACH, JOIN, UNION, ORDER BY, SAMPLE, etc.	SELECT, GROUP BY, ORDER BY, CLUSTER BY, DISTRIBUTE BY, JOIN, UNION, TABLESAMPLE, etc.	SELECT, GROUP BY, ORDER BY, CLUSTER BY, JOIN, UNION, TABLESAMPLE, etc.
Math Operators	ADDITION, SUBTRACTION, MULTIPLICATION, DIVISION, MODULO, etc.	ADDITION, SUBTRACTION, MULTIPLICATION, DIVISION, MODULO, etc.	ADDITION, SUBTRACTION, MULTIPLICATION, DIVISION, MODULO, etc.
Logical Operators	AND, OR, IN, NOT, EQUAL, NOT EQUAL, LESS THAN, GREATER THAN, PATTERN MATCHING	AND, OR, NOT, IN, EQUAL, NOT EQUAL, LESS THAN, GREATER THAN, PATTERN MATCHING, EXISTS, IF, COALESCE, CASE	AND, OR, NOT, IN, EQUAL, NOT EQUAL, LESS THAN, GREATER THAN, PATTERN MATCHING, EXISTS, IF, COALESCE, CASE
Collection and Aggregate Functions	AVG, SUM, COUNT, CONCAT, MAX, MIN, SIZE, SUBSTRACT, etc.	AVG, SUM, COUNT, CONCAT, MAX, MIN, SIZE, SUBSTRACT, etc.	AVG, SUM, COUNT, CONCAT, MAX, MIN, SIZE, SUBSTRACT, etc.
String Functions	Yes	Yes	Yes
DateTime Functions	Yes	Yes	Yes
UDF Support	Yes	Yes	Yes (partially Hive UDF)
JDBC/Thrift Support	Partial (No Thrift API)	Yes	Yes
Index Table	No	Yes	Yes

Storage Layer	Local Disk, HDFS, HBase	Local Disk, HDFS, HBase (Optional)	Local Disk, HDFS, HBase (Optional)
Applications	Data filtering, ETL, log analysis, general statistic applications, text processing	Ad-hoc queries, ODBC/JDBC applications, high-latency queries	Ad-hoc queries, ODBC/JDBC applications, low-latency queries

Bibliography

[1] Apache hadoop. http://hadoop.apache.org/core/.

[2] Apache spark sql. https://spark.apache.org/sql/.

[3] Test cases for scalable query and analysis for social networks. https://github.com/taklwu/apache-high-level-languages-survey.

[4] Hbase implementation of bigtable on hadoop file system. http://hbase.apache.org/, 2010.

[5] Pig programming tools. http://en.wikipedia.org/wiki/Pig_(programming_tool), 2014.

[6] Ablimit Aji, Xiling Sun, Hoang Vo, Qioaling Liu, Rubao Lee, Xiaodong Zhang, Joel Saltz, and Fusheng Wang. Demonstration of hadoop-gis: A spatial data warehousing system over mapreduce. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 528–531. ACM, 2013.

[7] Yael Amsterdamer, Susan B Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. *Proceedings of the VLDB Endowment*, 5(4):346–357, 2011.

[8] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, et al. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1071–1080. ACM, 2011.

[9] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[10] Michael Conover, Jacob Ratkiewicz, Matthew Francisco, Bruno Gonffalves, Filippo Menczer, and Alessandro Flammini. Political polarization on twitter. In *ICWSM*, 2011.

[11] Michael D Conover, Clayton Davis, Emilio Ferrara, Karissa McKelvey, Filippo Menczer, and Alessandro Flammini. The geospatial characteristics of a social movement communication network. *PloS one*, 8(3):e55957, 2013.

[12] Michael D Conover, Emilio Ferrara, Filippo Menczer, and Alessandro Flammini. The digital evolution of occupy wall street. *PloS one*, 8(5):e64679, 2013.

[13] Michael D Conover, Bruno Gonffalves, Alessandro Flammini, and Filippo Menczer. Partisan asymmetries in online political activity. *EPJ Data Science*, 1(1):1–19, 2012.

[14] Joseph DiGrazia, Karissa McKelvey, Johan Bollen, and Fabio Rojas. More tweets, more votes: Social media as a quantitative indicator of political behavior. *PloS one*, 8(11):e79449, 2013.

[15] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[16] Emilio Ferrara, Onur Varol, Clayton Davis, Filippo Menczer, and Alessandro Flammini. The rise of social bots. *arXiv preprint arXiv:1407.5225*, 2014.

[17] Emilio Ferrara, Onur Varol, Filippo Menczer, and Alessandro Flammini. Traveling trends: social butterflies or frequent fliers? In *Proceedings of the first ACM conference on Online social networks*, pages 213–222. ACM, 2013.

[18] Xiaoming Gao, Emilio Ferrara, and Judy Qiu. Parallel clustering of high-dimensional social media data streams. *arXiv preprint arXiv:1502.00316*, 2015.

[19] Xiaoming Gao and Judy Qiu. Social media data analysis with indexedhbase and iterative mapreduce. In *Proc. Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS 2013) at Super Computing*, 2013.

[20] Xiaoming Gao and Judy Qiu. Supporting end-to-end social media data analysis with the indexedhbase platform. In *Invited talk at 6th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS) SC13*, 2013.

[21] Xiaoming Gao, Evan Roth, Karissa McKelvey, Clayton Davis, Andrew Younge, Emilio Ferrara, Filippo Menczer, and Judy Qiu. Supporting a social media observatory with customizable index structures: Architecture and performance. In *Cloud Computing for Data-Intensive Applications*, pages 401–427. Springer, 2014.

[22] Alan F Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, 2009.

[23] Tim Hegeman, Bogdan Ghit, Mihai Capota, Jan Hidders, Dick Epema, and Alexandru Iosup. The btworld use case for big data analytics: Description, mapreduce logical workflow, and empirical evaluation. In *Big Data, 2013 IEEE International Conference on*, pages 622–630. IEEE, 2013.

[24] CJ Hutto and Eric Gilbert. Vader: A parsimonious rule-based model for sentiment analysis of social media text. In *Eighth International AAAI Conference on Weblogs and Social Media*, 2014.

[25] Mohsen JafariAsbagh, Emilio Ferrara, Onur Varol, Filippo Menczer, and Alessandro Flammini. Clustering memes in social media streams. *Social Network Analysis and Mining*, 4(1):1–13, 2014.

[26] Yeonhee Lee and Youngseok Lee. Toward scalable internet traffic measurement and analysis with hadoop. *ACM SIGCOMM Computer Communication Review*, 43(1):5–13, 2013.

[27] Yunpeng Li, David J Crandall, and Daniel P Huttenlocher. Landmark classification in large-scale image collections. In *Computer*

vision, 2009 IEEE 12th international conference on, pages 1957–1964. IEEE, 2009.

[28] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[29] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[30] Karissa McKelvey and Filippo Menczer. Design and prototyping of a social media observatory. In *Proceedings of the 22nd international conference on World Wide Web companion*, pages 1351–1358. International World Wide Web Conferences Steering Committee, 2013.

[31] Mark EJ Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical review E*, 74(3):036104, 2006.

[32] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[33] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[34] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[35] Judy Qiu and Bingjing Zhang. Mammoth data in the cloud: Clustering social images. *Clouds, Grids and Big Data*, 2013.

[36] Usha Nandini Raghavan, Reka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[37] Jacob Ratkiewicz, Michael Conover, Mark Meiss, Bruno Gonffalves, Alessandro Flammini, and Filippo Menczer. Detecting and tracking political abuse in social media. In *ICWSM*, 2011.

[38] Jacob Ratkiewicz, Michael Conover, Mark Meiss, Bruno Gonffalves, Snehal Patil, Alessandro Flammini, and Filippo Menczer. Truthy: mapping the spread of astroturf in microblog streams. In *Proceedings of the 20th international conference companion on World wide web*, pages 249–252. ACM, 2011.

[39] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[40] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[41] Onur Varol, Emilio Ferrara, Christine L Ogan, Filippo Menczer, and Alessandro Flammini. Evolution of online user behavior during a social upheaval. In *Proceedings of the 2014 ACM conference on Web science*, pages 81–90. ACM, 2014.

[42] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[43] Tak-Lon Wu, Abhilash Koppula, and Judy Qiu. Integrating pig with harp to support iterative applications with fast cache and customized communication. In *Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds*, pages 33–39. IEEE Press, 2014.

[44] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, pages 13–24. ACM, 2013.

[45] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[46] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[47] Bingjing Zhang, Yang Ruan, and Judy Qiu. Harp: Collective communication on hadoop. In *IEEE International Conference on Cloud Engineering (IC2E) conference.*