SCALABLE ARCHITECTURE FOR INTEGRATED BATCH AND STREAMING ANALYSIS OF BIG DATA

Xiaoming Gao

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics and Computing
Indiana University
January 2015

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Judy Qiu, Ph.D.

Geoffrey Charles Fox, Ph.D.

Filippo Menczer, Ph.D.

Dirk Van Gucht, Ph.D.

January 21st, 2015

Acknowledgements

First, I would like to thank Dr. Judy Qiu for being such a great advisor. She has been not only insightful and inspiring in guiding me through the research projects, but also supportive and encouraging when I need help in getting more resources or connecting to the right collaborators. Let me also thank Dr. Geoffrey C. Fox for being on my research committee and offering valuable discussions about my research. Every discussion with him is inspiring and pumps the progress of my projects. Thanks to Dr. Fil Menczer for his tremendous help throughout our collaboration and during my preparation for the dissertation proposal and defense, and for putting trust in me to develop the data infrastructure for the Truthy project. Thanks also to Dr. Dirk Van Gucht for having insightful discussions with me and guiding me through all the related literature from the database field, which has been extremely valuable for defining the depth and scope of this dissertation. His classes are the reason I chose database as a minor, which has proven to be a very wise decision.

Next I would like to express my particular appreciation for the collaboration and help from my colleagues in the SALSAHPC group: Bingjing Zhang, Stephen Wu, Yang Ruan, Andrew Younge, Jerome Mitchell, Saliya Ekanayake, Supun Kamburugamuve, Thomas Wiggins, Zhenghao Gu, Jaliya Ekanayake, Thilina Gunarathne, Yuduo Zhou, Fei Teng, Zhenhua Guo, and Tao Huang. Thank you for continuously working with me on every concrete problem and giving me all the great advice on my papers and presentations. Thanks to my early collaborators Marlon Pierce, Yu Ma, Jun Wang, and Robert Granat for their guidance and help in developing the QuikeSim project and writing related research papers. Thanks to the collaborators from the Center for Complex Networks and Systems Research at Indiana University; Emilio Ferrara,

Clayton Davis, Mohsen JafariAsbagh, Onur Varol, Karissa McKelvey, Giovanni L. Ciampaglia, and Alessandro Flammini, for working together on the Truthy project and providing all the help and information that has been necessary for the progress of my research work.

I would also like to recognize Prof. Yuqing Wu for helping me accelerate my progress towards graduation. Thanks to Koji Tanaka, Allan Streib, Rob Henderson, and Gary Miksik for their help in configuring and solving our problems with the computing infrastructures from FutureGrid, Madrid, and Moe. Thanks to Lynne Mikolon, Patty Reyes-Cooksey, Becky Curtis, and Christi Pike for their help on administrative issues related to my working contracts and course registration.

Finally, I am especially grateful to my family and friends who have always supported and understood me, making this entire process a precious and memorable experience in my life.

Xiaoming Gao

Scalable Architecture for Integrated Batch and Streaming Analysis of Big Data

Scientific research has entered an era driven by data, and many modern data intensive applications demonstrate special characteristics. Data exists in the form of both large historical datasets and high-speed real-time streams, while many analysis pipelines require integrated parallel batch processing and stream processing. In most cases, despite the large size of the whole dataset, most analyses tend to focus on specific data subsets according to certain criteria. Faced with all these situations, scalable solutions are essential to achieve optimal performance. Correspondingly, integrated support for efficient queries and post- query analysis is required.

To address the system-level requirements brought by such characteristics, this dissertation proposes a scalable architecture for supporting integrated queries, batch analysis, and streaming analysis of Big Data in the cloud. We verify its effectiveness and efficiency with real use cases from a representative application domain – social media data analysis – and tackle related research challenges emerging from each module of the architecture by integrating and extending multiple state-of-the-art Big Data storage and processing systems.

In the storage layer, we reveal that existing text indexing techniques do not work well for the unique queries of social media data, which involve constraints on both textual and social context such as temporal or network information. To address this issue we leverage the HBase system as the storage substrate and extend it with a flexible indexing framework – IndexedHBase. This allows users to define fully customizable text index structures that are not supported by current state-of-the-art text indexing systems such as distributed Solr. Such index structures can embed the precise social context information that is necessary for efficient evaluation of the queries.

The batch analysis module demonstrates that social media data analysis workflows usually consist of multiple algorithms with varying computation and communication patterns which are suitable for different processing frameworks such as Hadoop, Twister, etc. In order to achieve efficient execution of the whole workflow, we extend IndexedHBase to an integrated analysis stack based on YARN, which can dynamically adopt different processing frameworks to complete analysis tasks. Based on this we develop a set of analysis algorithms that are useful for many research scenarios, and demonstrate the composition and execution of workflows by reproducing the end-to-end analysis processes from published research projects.

Finally, in the streaming analysis module, the high-dimensional data representation of social media streams poses special challenges to sophisticated parallel stream mining problems such as clustering. Due to the sparsity of the high-dimensional data vectors, traditional synchronization that directly broadcasts the centroids becomes too expensive and severely impacts the scalability of the parallel algorithm. Our solution is to extend the Storm stream processing engine by building a separate synchronization channel using a pub-sub messaging system, and design a novel synchronization strategy that broadcasts the incremental changes ("deltas") instead of the whole centroids of the clusters.

We use real applications from the Truthy social media data observatory to evaluate our architecture. Preliminary performance tests show that our solutions for parallel data loading/indexing, query and analysis task execution, and stream clustering all outperform implementations using current state-of-the-art technologies.

____
Judy Qiu, Ph.D.

____
Geoffrey Charles Fox, Ph.D.

____
Filippo Menczer, Ph.D.

____
Dirk Van Gucht, Ph.D.

Table of Contents

List of figures and tables

Chapter 1

Introduction

1.1 Big Data: Emerging Characteristics

Scientific research has entered a “Big Data” era [79]. As data is growing exponentially in every area of science, more and more discoveries are driven by the capability of collecting and processing vast amounts of data. Therefore, in order to boost the progress of scientific research, scalable IT infrastructures are needed to deal with the **high volume**, **high velocity**, and **high variety** of Big Data. This in turn brings up research challenges and opportunities for the distributed data processing architecture running at the backend of IT infrastructures.

As Big Data processing problems evolve, many applications demonstrate special characteristics with regards to their data and analysis process. First of all, besides a large amount of historical data, streaming data plays a more and more important role. For instance, earthquake monitoring and prediction systems detect geological events based on real-time analysis of data streams generated by GPS ground stations [117]; automated trading systems rely on the dynamic stream of stock price values to make smart trading decisions [101], etc. Correspondingly, the data processing architecture needs to provide scalable solutions not only for storing, querying, and analyzing the static historical data, but also for loading and processing the streaming data in a parallel fashion. The loading and analysis of static data and streaming data need to be handled in an integrated way. For example, an integrated general storage substrate should be provided to host both historical data and incremental changes coming from the streams. At the same time, online stream analysis should be able to use the results of batch analysis over static data for bootstrapping or checkpointing purposes.

On the other hand, despite the large size of the whole dataset, most analyses tend to focus on specific data subsets. For example, gene sequence analysis may focus on a certain family of sequences [140], and social data analysis may concentrate on data related to certain global or local events [131]. For such research scenarios, limiting analysis computation to the exact scope of the target subsets is important in terms of both efficiency and better resource utilization. Therefore efficient query mechanisms for quickly locating the relevant data subsets are needed on the data storage and analysis architecture. Furthermore, queries need to be closely integrated with post-query analysis tasks to support efficient end-to-end analysis workflows.

1.2 Social Media Data Analysis

Social media data analysis is one specific application domain that follows the Big Data trend. Motivated by the widespread adoption of social media platforms such as Twitter and Facebook, investigating social activities through analysis of large scale social media datasets has been a popular research topic in recent years. For example, many studies investigate the patterns of information diffusion on social networks by processing historical datasets generated during real-world social events [119][130][162]. By analyzing real-time social media data streams, more sophisticated applications such as online event detection [10][120] and social-bots detection [63][64] can be supported.

Social media data analysis problems also reflect the emerging characteristics of Big Data, which bring special research challenges for developing a scalable architecture. On the one hand, the data source contains not only a large historical dataset at TB or even PB level, but also a high-speed stream at the rate of tens to hundreds of millions of social updates per day generated by people all over the world. On the other hand, most analyses focus on data subsets related to specific social events or special aspects of social activities: congressional elections [42][44], protest events [40][41], social link creation [160], etc. With regards to query patterns, social media data is unique in that it contains not only textual content, but also rich information about the social context including time, geolocation, relationship among users on the social network, etc. Most queries involve selection of data records following constraints over both text elements and the social context such as temporal or geospacial information. The purpose of the queries is to extract social information such as network connections from all the selected data records rather than finding the top-K most relevant data records according to a set of text keywords. As a result, traditional static text indexing techniques [170] designed for information retrieval applications do not work well for queries over social media data. Therefore, a novel indexing component that can help deliver the most efficient queries over social media data is a necessary aspect of a scalable data analysis architecture.

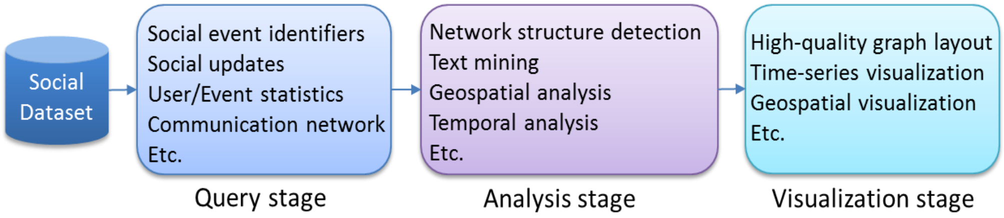


Figure 1-1. Stages in a social media data analysis workflow

Another important feature of social data analysis is that the analysis workflow normally consists of multiple stages, as illustrated in Figure 1-1. The query stage is normally followed by a series of analysis tasks for processing or visualizing the query results. Therefore, integrated support for queries and post-query analysis tasks is required on the analysis architecture.

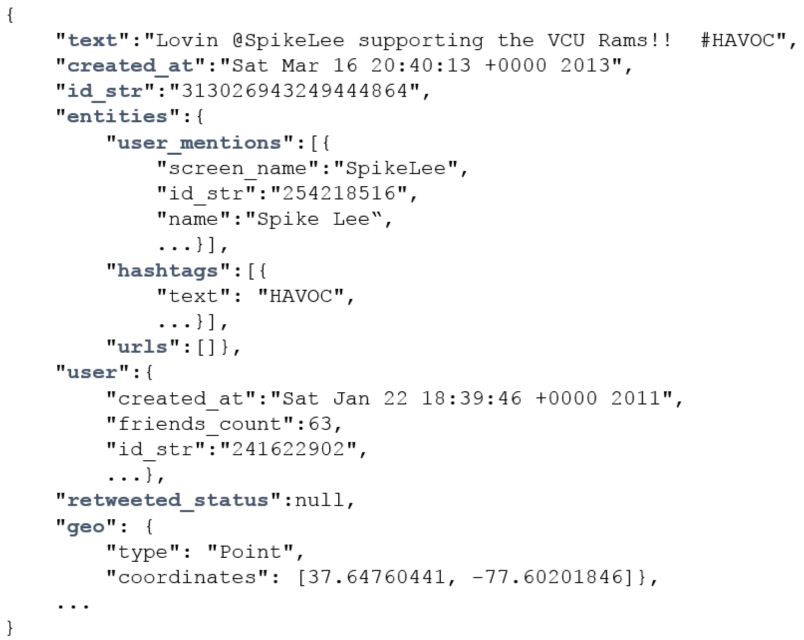
Due to the representativeness of social media data analysis, it provides a good starting point for investigating the general research challenges associated with the emerging characteristics of Big Data problems. The target of this dissertation is to analyze such challenges and address them by proposing a scalable and integrated architecture that is generally applicable to a broad scope of application domains. Specifically, we study the challenges related to the queries, batch analysis, and streaming analysis through representative and published use cases from existing social media data analysis systems. We propose corresponding solutions in different modules of the architecture, and use real analysis workflows and applications from these systems to evaluate the effectiveness and efficiency of our methods. To uncover the underlined research problems, we start from reviewing the characteristics of existing social media data analysis platforms.

1.2.1 Truthy Social Media Observatory

Truthy [102] is a public social media observatory developed by the Center for Complex Networks and Systems Research at Indiana University. It is designed for analysis and visualization of information diffusion on Twitter. Research performed on the data collected by this system covers a broad spectrum of social activities, including political polarization [43][44], congressional elections [42], protest events [40][41], and the spread of misinformation [120]. Truthy has also been instrumental in shedding light on communication dynamics such as user attention allocation [159] and social link creation [160].

Data Characteristics

Truthy has been collecting social media data through the Twitter gardenhose stream [67] since May of 2010, which provides a sample of approximately 10% of their public tweets. The entire dataset consists of two parts: historical data in .json.gz files, and real-time data coming from the Twitter streaming API [148]. Currently, the total size of historical data collected continuously by the system since August 2010 is approximately 20 Terabytes. At the time of this writing, the data rate of the Twitter streaming API is in the range of 45-50 million tweets per day, leading to a growth of approximately 20GB per day in the total data size. Figure 1-2 illustrates a sample data item, which is a structured JSON string containing information about a tweet and the user who posted it. Furthermore, if the tweet is a retweet, the original tweet content is also included in a “retweeted_status” field. For hashtags, user-mentions, and URLs contained in the text of the tweet, an “entities” field is included to give detailed information, such as the ID of the mentioned user and the expanded URLs.



```
{
    "text":"Lovin @SpikeLee supporting the VCU Rams!!  #HAVOC",
    "created_at":"Sat Mar 16 20:40:13 +0000 2013",
    "id_str":"313026943249444864",
    "entities":{
        "user_mentions":[{
            "screen_name":"SpikeLee",
            "id_str":"254218516",
            "name":"Spike Lee“,
            ...}],
        "hashtags":[{
            "text": "HAVOC",
            ...}],
        "urls":[]},
    "user":{
        "created_at":"Sat Jan 22 18:39:46 +0000 2011",
        "friends_count":63,
        "id_str":"241622902",
        ...},
    "retweeted_status":null,
    "geo": {
        "type": "Point",
        "coordinates": [37.64760441, -77.60201846]},
    ...
}
```

Figure 1-2. An example tweet in JSON format

Queries

In social network analysis, the concept of “*meme*” is often used to represent a set of related posts corresponding to a specific discussion topic, communication channel, or information source shared by users on platforms such as Twitter. Memes can be identified through elements contained in the text of tweets, like *keywords*, *hashtags* (e.g., #euro2012), *user-mentions* (e.g., @youtube), and *URLs*. Based on rich experience from previous research projects, Truthy identifies a set of temporal queries that are generally applicable in many research scenarios for extracting and generating various information about tweets, users, and memes. These queries can be categorized into two subsets. The first contains basic queries for getting the ID or content of tweets created during a given time window from their text or user information, including:

get-tweets-with-meme (*memes, time_window*)
get-tweets-with-text (*keywords, time_window*)
get-tweets-with-user (*user_id, time_window*)
get-retweets (*tweet_id, time_window*)

For the parameters, *time_window* is given in the form of a pair of strings marking the start and end points of a time window, e.g., [2012-06-08T00:00:00, 2012-06-23T23:59:59]. The *memes* parameter is given as a list of hashtags, user-mentions, or URLs; *memes* and *keywords* may contain wildcards, e.g., "#occupy*" will match all tweets containing hashtags starting with "#occupy."

The second subset of queries need information extracted from the tweets returned by queries in the first subset. These include:

timestamp-count (*memes, time_window*)
user-post-count (*memes, time_window*)
meme-post-count (*memes, time_window*)
meme-cooccurrence-count (*memes, time_window*)
get-retweet-edges (*memes, time_window*)
get-mention-edges (*memes, time_window*)

Here for example, *user-post-count* returns the number of posts about a given meme by each user. Each "edge" has three components: a "from" user ID, a "to" user ID, and a "weight" indicating how many times the "from" user has retweeted the tweets from the "to" user or mentioned the "to" user in his/her tweets.

The most significant characteristic of these queries is that they all take a time window as a parameter. This originates from the temporal nature of social activities. An obvious brute-force solution is to scan the whole dataset, try to match the content and creation time of each tweet with the query parameters, and generate the results using information contained in the matched tweets. However, due to the drastic difference between the size of the entire dataset and the size of the query result, this strategy is prohibitively expensive. For example, in the time window [2012-06-01, 2012-06-20] there are over 600 million tweets, while the number of tweets containing the most popular meme "@youtube" is less than two million, which is smaller by more than two orders of magnitude. As will be discussed in Section 1.3, proper indexing techniques are needed for efficient evaluation of such queries.

Analysis Workflows and Streaming Applications

Most analysis workflows completed on Truthy follow the multi-stage pattern as illustrated in Figure 1-1. For example, in the workflow for analyzing political polarization [44], the first stage applies the *get-retweet-edges* and *get-mention-edges* queries to retrieve the retweet network and mention network from the tweets selected by a set of hashtags related to politics; the second stage completes community detection analysis over the networks; finally, the third stage uses a graph layout algorithm to visualize the polarization of communities on the social network. In another project that investigates the digital evolution of the "Occupy Wall Street" event, the first stage queries for tweets related to both general politics and the specific "Occupy" event based on a manually selected set of hashtags (e.g. *#p2*, *#tcot*, and *#occupy**), using a 15-month time window that covers most of the event's development; the second stage processes the tweets and measures the evolution of the amount of social network traffic, the degree of user engagement, and the intensity of information diffusion along the time dimension; the final stage visualizes these patterns of evolution as time series plots. Figure 1-3 [41] shows an example plot that illustrates the total number of tweets related to the “Occupy Wall Street” event during a one-year time period.

Besides batch analysis over historical data, applications that complete online analysis of real-time streams are also being developed. In particular, Bot or Not [64] is an online service that can dynamically classify a given user ID as a human user or social bot with a certain confidence level by analyzing a small number of sample tweets retrieved from the Twitter Streaming API [148]. To support more sophisticated application scenarios, the problem of social media stream clustering [85] has also been investigated. The major discovery is that by using a combination of proper data representations and similarity metrics, it is possible to generate high-quality clusters that can effectively group messages with similar social meaning together.

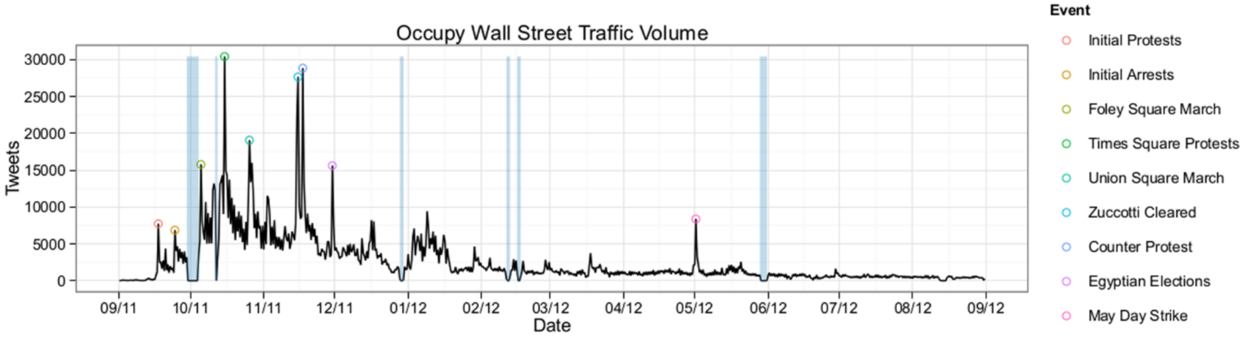


Figure 1-3. Total number of tweets related to Occupy Wall Street between 09/2011 and 09/2012 [41]

Based on the tools and algorithms used in the analysis workflows, Truthy provides a nice web interface [145] for users to retrieve derived data such as social network information and statistics about certain users and memes, as well as visualization generated by some previous research projects.

Despite the richness and scope of research that has been covered by Truthy, most projects complete data processing in a sequential way or only with a limited level of parallelism (e.g. using multi-thread solutions) over the raw data. As a result, the processing speed is unsatisfactory when measured against the volume of the whole dataset. For example, it takes as many as 43 hours for a sequential implementation of the stream clustering algorithm in [85] to process one hour's worth of data collected through the Twitter gardenhose stream [67]. Most data is stored as raw .json.gz files, which are not suitable for random access to individual social messages. A MySQL database is used to maintain important sumarries about certain users and memes, but is obviously not scalable enough to support fine-grained access and efficient queries over the whole dataset. This situation forms a strong and practical motivation for the research work of this dissertation.

1.2.2 Other Similar Systems

To the best of our knowledge, Truthy is the first complete social media observatory in terms of functionality and interface. VisPolitics [155], TwitInfo [146], and Ripples [126] are similar analysis systems that generate visualizations about different aspects of social media network, but do not provide a rich set of statistics and derived data as Truthy does. Meanwhile, many query patterns and analysis components defined by Truthy are generally useful for constructing the functionality of these systems. For example, similar queries can be used to generate the 'repost network' in Ripples, or support searching of topic keywords and URL counting in TwitInfo. Commercial entities such as PeopleBrowsr [116], Datasift [50], and SocialFlow [133] provide consulting services to their customers through analytics over social media data, but they don't expose their raw data or results to the public for research purposes. Padmanabhan et al. presented FluMapper [112], an interactive map-based interface for flu-risk analysis using near real-time processing of social updates collected from the Twitter streaming API. FluMapper applies a set of advanced technologies, including NoSQL database (MongoDB [105]), GPU processing, and flow mapping, to support its data collection, processing, and visualization modules.

Although these systems demonstrate a broad scope of applications involving social media data, none of them has done an in-depth investigation about the fundamental research challenges from the perspective of distributed systems. On the other hand, many Big Data tools have been developed in the past, including storage systems such as Hadoop Distributed File System (HDFS) [132] and HBase [19], and various processing tools as illustrated in Figure 1-4. Specifically, Hadoop [19] provides an easy to use MapReduce [53] programming interface to support single-pass parallel processing of Big Data, and automatically handles issues such as locality-aware task scheduling, failure recovery, and intermediate data transmission at the platform level. Beyond this, frameworks such as Twister [60] and Spark [166] are specially optimized for iterative computation that can be described with a MapReduce model. For iterative algorithms over graph data, frameworks such as Giraph [17] and Harp [169] can directly support data abstraction in the form of nodes and edges in graphs. To enable efficient queries over large-scale datasets, systems such as Power Drill [75], Pig [23], and Hive [20] were developed with original support for high-level query languages. Finally, to support distributed parallel processing of streaming data, stream processing engines such as S4 [108] and Storm [25] have been proposed. Despite the richness and variaty of all these existing systems, it is still unclear what kind of extensions and combinations of them are necessary for handling the new characteristics of Big Data problems, as represented by social media data analysis. This dissertation tries to bridge the gap between these two sides, and we start by studying the specific research challenges.

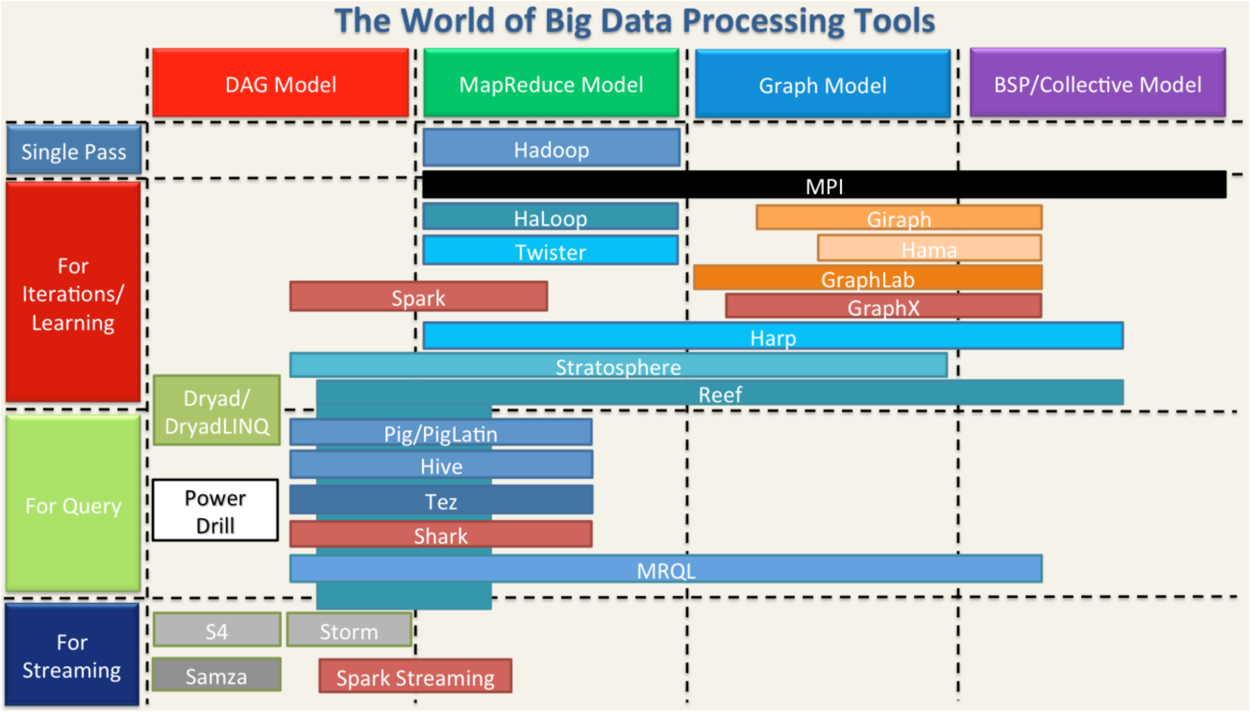


Figure 1-4. Big Data processing tools [169]

1.3 Research Challenges

Due to the special characteristics of social media data, we are facing research challenges related to three major aspects at the distributed system level: indexing, dynamic runtime processing frameworks, and parallel stream processing.

1.3.1 Requirements for Novel Text Indexing Techniques

First of all, as demonstrated in Section 1.2.1, most queries over social media data can be categorized as text queries with constraints about social context. However, traditional text indexing techniques (i.e. inverted indices [170]) supported by many existing distributed storage systems such as distributed Solr [57], DataStax [52], and Riak [125] do not provide the most efficient solution to such queries. One reason is that traditional inverted indices are mainly designed for text retrieval applications where the main goal is to efficiently find the top K (with a typical value of 20 or 50 for K) most relevant text documents regarding a query comprising a set of keywords. To achieve this goal, information about the frequency and position of keywords in the documents is stored and used for computing relevance scores between documents and keywords during query evaluation. In contrast, social media data queries are designed for analysis purposes, meaning that they have to process all the related tweets, instead of the top K most relevant ones, to generate the results. This means data regarding frequency and position are extra overhead for the storage of the index structures, and relevance scoring is unnecessary in the query evaluation process. The query evaluation performance can be further improved by removing these items from traditional inverted indices.

Another issue with traditional text indexing techniques is that one separate inverted index structure is maintained for every indexed field. However, social media queries do not favor query execution plans using such separate one-dimensional indices. For example, Figure 1-5 illustrates a typical query execution plan for *get-tweets-with-meme*, using two separate indices on memes and tweet creation time. This plan uses the meme index to locate the IDs of all tweets containing the given memes and utilizes the time index to find the set of tweet IDs within the given time window, finally computing the intersection of these two sets to get the results. Assuming the size of the posting lists for the given memes to be m, and the number of tweet IDs coming from the time index to be n, the complexity of the whole query evaluation process will be $O(m + n) = O(\max(m, n))$, using a merge-based or hashing-based algorithm for the intersection operation. However, due to the characteristics of large social media and microblogging datasets, there is normally an orders-of-magnitude difference between m and n, as discussed in Section 1.2.1. As a result, although the size of the query result is bounded by $\min(m, n)$, a major part of query evaluation time is actually spent on scanning and checking irrelevant entries of the time index. In classic text search engines, techniques such as skipping or frequency-ordered inverted lists [170] may be utilized to quickly return the top K most relevant results without evaluating all the related documents. Such optimizations are not applicable to the analysis-oriented social media data queries. Furthermore, in case of a high cost estimation for accessing the time index, the search engine may choose to only use the meme index and generate the results by checking the content of relevant tweets. But valuable time is still wasted in checking irrelevant tweets falling out of the given time window. The query evaluation performance can be further improved if the unnecessary scanning cost can be avoided.

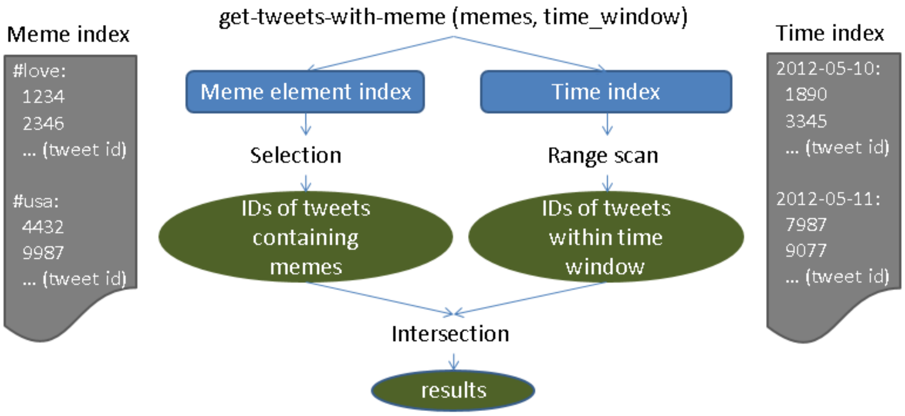
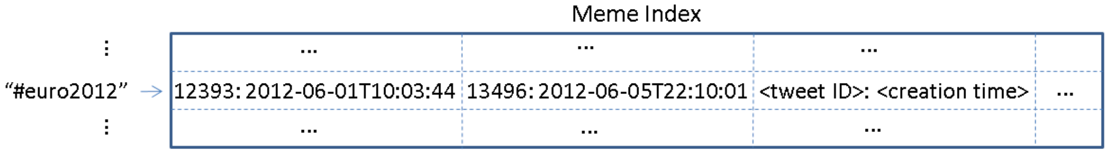


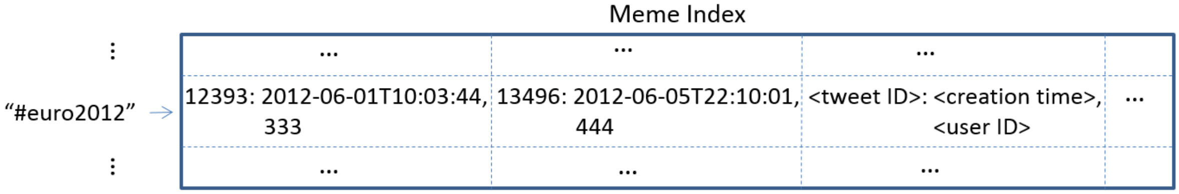
Figure 1-5. A typical query execution plan using separate indices on meme and creation time

To avoid the above-mentioned problems, a more suitable index structure would be the one given in Figure 1-6. It merges the meme index and time index, and replaces the frequency and position information in the posting lists of the meme index with creation time of corresponding tweets. Facilitated by this customized index structure, the query evaluation process for *get-tweets-with-meme* can be easily implemented by going through the index entries related to the given memes and selecting the tweet IDs associated with a creation time within the given time window. The complexity of the new query evaluation process is O(m), which is significantly lower than O(max(m, n)). Moreover, if we can further extend this index structure to also include the user ID of each tweet, as shown in Figure 1-7, it will be possible to evaluate the advanced query *user-post-count* by only accessing the index, without touching the original data at all.



⋮	...	...	...	
“#euro2012” →	12393: 2012-06-01T10:03:44	13496: 2012-06-05T22:10:01	<tweet ID>: <creation time>	...
⋮	...	...	...	

Figure 1-6. A customized meme index structure including time



⋮	...	...	...	
“#euro2012” →	12393: 2012-06-01T10:03:44, 333	13496: 2012-06-05T22:10:01, 444	<tweet ID>: <creation time>, <user ID>	...
⋮	...	...	...	

Figure 1-7. A customized meme index structure including time and user ID

The ideas behind these index structures are similar to the features of multi-dimensional indices and included columns that have been supported by relational databases for non-text data. However, they are not supported by current state-of-the-art text indexing systems, such as Lucene [22] and distributed Solr [57]. To enable them, **a fully customizable text indexing framework** is needed. Considering the special characteristics of social media data, this framework must provide both a scalable batch indexing mechanism for the static historical data and an efficient online indexing mechanism for the high-speed streaming data. So building such a framework in a scalable way is a major research challenge, as is knowing how to make use of the various customized index structures to support efficient queries and analysis tasks.

1.3.2 Efficient Execution of Analysis Workflows

As demonstrated in Figure 1-1, social media data analysis workflows normally consist of multiple stages, and each stage may apply a diversity of algorithms to process the target data subsets. These algorithms demonstrate a high level of complexity in their computation and communication patterns, including sequential, MapReduce, iterative MapReduce, and graph style. Different patterns are suitable for different processing frameworks such as Hadoop [18], Twister [60], Spark [166], and Giraph [17]. Moreover, to support online stream analysis applications, distributed stream processing engines like Storm [25] may also be used. To achieve efficient overall execution of the workflow, the analysis architecture must be able to dynamically adopt suitable processing frameworks to complete different steps from these stages. Achieving this in a distributed and shared environment is another major challenge. In case of integrated workflows involving both queries and analysis tasks, how to explore the value of indices in supporting sophisticated analysis algorithms (beyond the scope of queries) is also an interesting research question.

1.3.3 Parallel Processing of High-Speed Stream Data

Due to the high speed of social media data streams, parallel processing is necessary for many stream analysis applications such as clustering and classification. To support efficient parallel processing of streaming data, many distributed frameworks have been proposed including Apache Storm [25] and S4 [108]. Most of these frameworks organize the parallel stream processing workers in the form of a direct acyclic graph (DAG), which makes it difficult to complete dynamic status synchronization among the parallel workers, a crucial step for ensuring the correctness of the parallel analysis algorithms. This is because the synchronization step requires the parallel workers to send their local status updates either to each other or to a global updates collector, which will then broadcast the updated global state back to the parallel workers. Both ways will inevitably create cycles in the communication channel, which conflicts with the DAG model. Meanwhile, to achieve high-quality analysis results, many stream analysis applications represent the the social messages in the stream as multiple high-dimensional vectors that reflect both the textual content and the social context of the data. The high-dimensionality and sparsity of such vectors may bring extra complexity and cost to the synchronization mechanism, and designing proper synchronization strategies to enable efficient parallel stream analysis algorithms is an important research issue.

1.4 Contributions and Outline

To address the research challenges discussed in Section 1.3, this dissertation proposes a scalable and integrated analysis architecture as illustrated in Figure 1-8 to support modern scientific data analysis pipelines in the cloud. The three stages in the pipeline demonstrate how Information, Knowledge, and Wisdom [143] are eventually generated from Data. Correspondingly, we name our architecture Cloud DIKW. The whole architecture comprises three modules; each module extends and combines a set of big data storage and processing tools to tackle the corresponding challenges.

At the bottom layer, we use NoSQL databases as the storage substrate, which can provide scalable storage of large social media datasets and efficient random access to fine-grained social messages. To address the requirements for novel indexing techniques, we propose a fully customizable indexing framework that can be generally integrated with most NoSQL databases. With this framework, users can define customized index structures that contain the exact necessary information about the original social media data, so as to achieve efficient evaluation of queries about interesting social events and activities. By choosing proper mappings between the abstract index structures and the storage units provided by the underlying NoSQL database, efficient indexing of historical and streaming data can be achieved. We realize and verify our framework with IndexedHBase [83], a specific implementation on HBase [19].

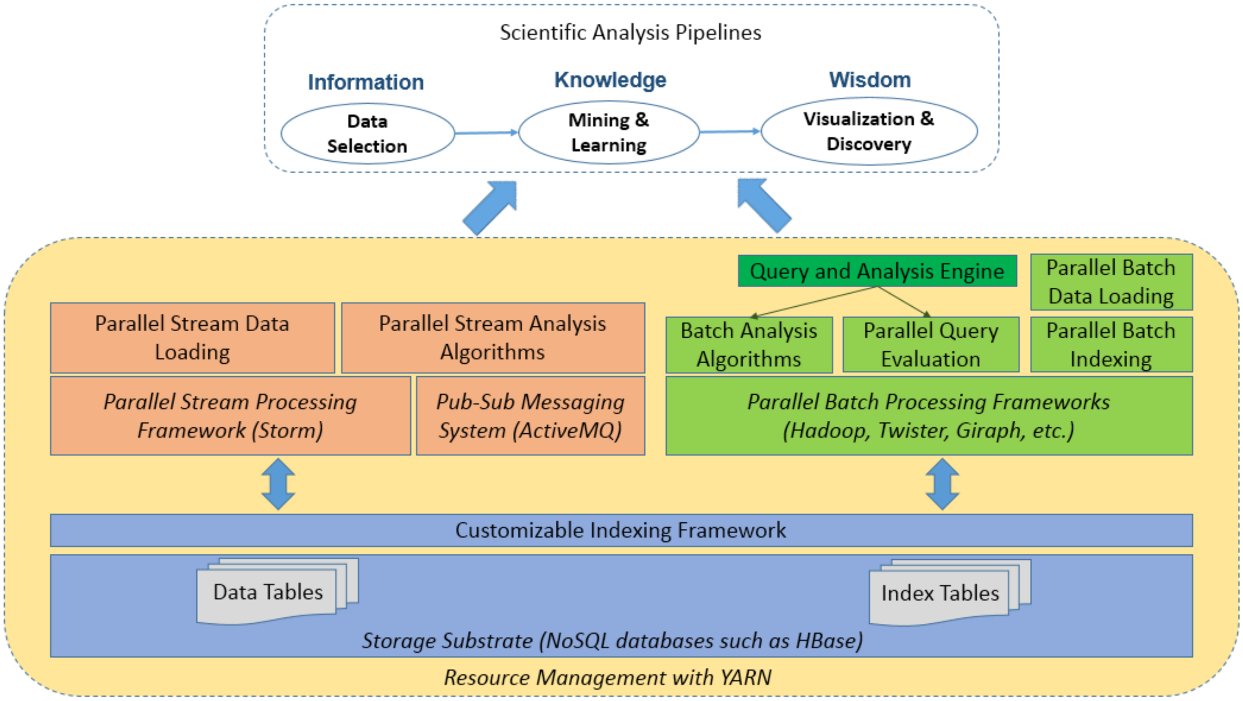

Scientific Analysis Pipelines
Information
Knowledge
Wisdom
Data Selection
Mining & Learning
Visualization & Discovery
Query and Analysis Engine
Parallel Batch Data Loading
Parallel Stream Data Loading
Parallel Stream Analysis Algorithms
Batch Analysis Algorithms
Parallel Query Evaluation
Parallel Batch Indexing
Parallel Stream Processing Framework (Storm)
Pub-Sub Messaging System (ActiveMQ)
Parallel Batch Processing Frameworks (Hadoop, Twister, Giraph, etc.)
Customizable Indexing Framework
Data Tables
Index Tables
Storage Substrate (NoSQL databases such as HBase)
Resource Management with YARN

Figure 1-8. Integrated architecture for social media data analysis

To achieve efficient execution of the whole analysis workflow, we extend IndexedHBase to build an analysis architecture based on YARN (Yet Another Resource Negotiator) [154], which is specially designed for dynamic scheduling of analysis tasks using different parallel processing frameworks. In the batch analysis module, we develop a parallel query evaluation strategy and a set of analysis algorithms using various parallel processing frameworks, including Hadoop MapReduce [99], Twister [60], etc. These can be used as basic building blocks for composing different workflows. In addition, parallel batch indexing and data loading mechanisms for handling historical data are developed based on the functionality of the customizable indexing framework. Moreover, we extend the usage of customized indices beyond basic queries to sophisticated mining and analysis algorithms, and demonstrate the significant performance improvement that can be achieved by exploring the value of indexing. We use real data, queries, and previously published analysis workflows from Truthy to evaluate the performance of these modules. Our results demonstrate that compared with implementations based on existing text indexing techniques on a widely adopted NoSQL database (Riak [125]), our data loading mechanism is faster by multiple times, and our query evaluation strategy can be faster by up to two orders of magnitude. Finally, parallel analysis algorithms are tens to hundreds of times faster than the old sequential implementation on Truthy, thus leading to much more efficient analysis workflows.

To achieve efficient parallel processing of stream data, we use the Storm stream processing engine as the basis of the stream analysis module, and develop a parallel stream data loading mechanism based on the online indexing functionality of the customizable indexing framework. Preliminary performance tests show that we are able to process a stream whose speed is five times faster than the current Twitter gardenhose stream [67] with only 8 parallel stream loaders. To support more sophisticated stream clustering algorithms, we create a separate synchronization channel by using the pub-sub messaging system ActiveMQ [16], and combine its functionality together with Storm to coordinate the synchronization process. Furthermore, to deal with the problem caused by the high-dimensionality of the data, we propose a novel synchronization strategy that broadcasts the dynamic changes (“deltas”) of the clusters rather than the whole centroid vectors. Performance evaluation shows that this synchronization mechanism can help the parallel algorithm achieve nice sub-linear scalability, and the algorithm can process the Twitter 10% data stream (“gardenhose”) in real-time with 96-way parallelism.

The rest of this dissertation is structured as follows. Chaper 2 describes the customizable indexing framework, the parallel data loading mechanism, and the parallel query evaluation strategy, and compares them to solutions based on existing distributed text indexing techniques. Chapter 3 explains the internal mechanism of the batch analysis module, and presents the implementation of multiple parallel analysis algorithms based on different processing frameworks. In addition we repeat a previously published analysis workflow [44] based on our parallel queries and analysis algorithms, and demonstrate the significant speedup that can be achieved for the overall execution of the whole workflow. Chapter 4 elaborates on the stream analysis module, analyzes the research challenges related to parallel clustering of social media data streams, and discusses our novel synchronization strategy for supporting the parallel algorithm on Storm. We demonstrate the scalability of our algorithm by comparing it against an implementation using the traditional synchronization strategy that directly broadcasts the whole centroids of the clusters. Chapter 5 concludes and proposes interesting future work for every module of the integrated architecture.

Chapter 2

Storage Layer - Customizable and Scalable Indexing Framework over NoSQL Databases

2.1 Overview

As discussed in Section 1.3, the major research challenge to the storage layer of Cloud DIKW roots in the scalability of the storage substrate and the efficiency of the query mechanisms for finding interesting data subsets. This chapter makes the following contributions to resolve this challenge:

- We compare the features of two options for constructing a scalable storage solution – parallel relational databases and NoSQL databases, and choose the latter as the the basis of our storage layer based on analysis of the characteristics of social media data.
- We provide a detailed review of multiple representative NoSQL database systems, and reveal that compared with the query patterns against social media data, the level of indexing support on current NoSQL databases is uneven and inadequate.
- To enhance the indexing support of NoSQL databases, we propose a general customizable indexing framework that can be implemented on most NoSQL databases. This framework allows users to define customizable text index structures that are not supported by current distributed text indexing systems, so as to achieve the best query performance for social media data.
- We provide one implementation of this framework, IndexedHBase, over HBase and develop parallel data loading and query evaluation strategies on top. Performance evaluation with real data and queries from Truthy shows that compared with solutions based on existing text indexing techniques provided by current NoSQL databases, our data loading strategy based on customized index structures is faster by multiple times, and our parallel query evaluation strategy is faster by one to two orders of magnitude.

The first step towards Cloud DIKW is to provide a scalable storage substrate. Specifically, it has to properly handle the following characteristics of social media data:

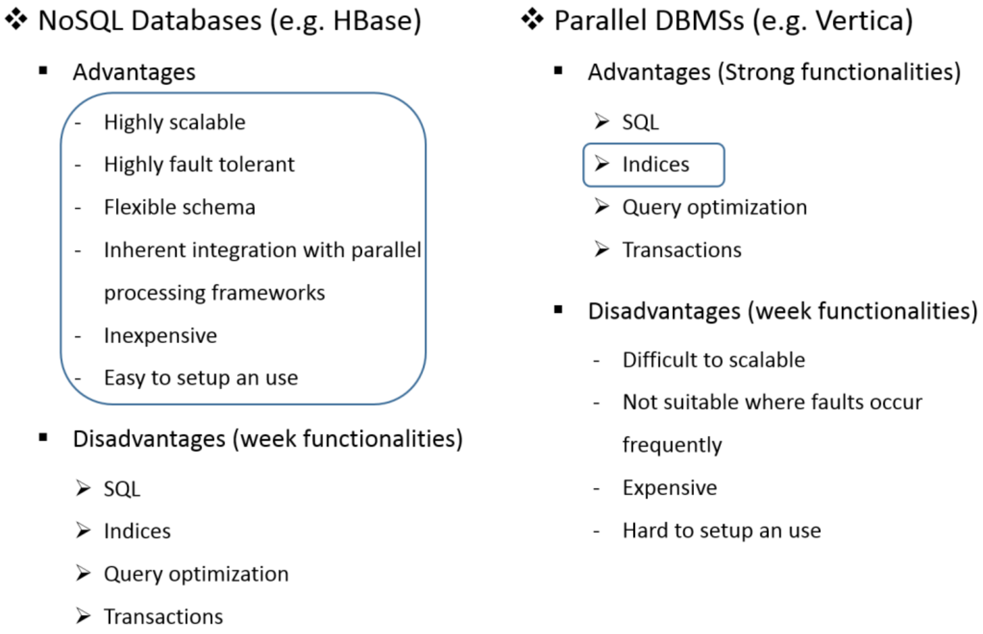
First of all, since the whole dataset is composed of both a large scale historical dataset and a high-speed stream, the storage substrate must support both scalable batch loading for static data and real-time online insertion of streaming data. Data access pattern is mostly write-once-read-many, because historical data is rarely updated. Therefore, in cases where data is replicated, consistency among dynamically changed replicas is not a strong requirement. And since data processing is mostly analysis-oriented, sophisticated data manipulation through online transcations with ACID (Atomicity, Consistency, Isolation, Durability) properties is not required either.

Next, since data comes in the form of separate social messages, the storage substrate has to support efficient random access to fine-grained data records. As illustrated in Figure 1-2, a social message may be structured as a hierarchy of multiple levels that may evolve over time i.e. fields could be dynamically deleted or added based on requirements of the application. For instance, for tweets coming from the Twitter Streaming API [148], fields like "id_str", "entities.symbols", "entities.media" are all added as the application evolves. The storage substrate should ideally be able to handle such dynamic data schema changes in a seamless way.

Since analysis workflows normally start with queries, the storage substrate must be equipped with necessary indexing techniques to enable the most suitable index structures for efficient evaluation of the queries.

Finally, due to the requirement for integrated queries and analysis tasks, the storage substrate should have a natural integration with parallel processing frameworks such as Hadoop MapReduce to support various analysis algorithms.

Due to the strong requirements for scalability and random data access, we mainly consider two types of systems: parallel relational databases and NoSQL databases. The advantages and disadvantages of both sides were compared by Kyu-Young Whang in 2011 [161], as illustrated in Figure 2-1 where we extend the advantages of NoSQL databases with two more features: flexible schema and inherent integration with parallel processing frameworks.



❖ NoSQL Databases (e.g. HBase)

- Advantages
  - Highly scalable
  - Highly fault tolerant
  - Flexible schema
  - Inherent integration with parallel processing frameworks
  - Inexpensive
  - Easy to setup an use
- Disadvantages (week functionalities)
  - ➢ SQL
  - ➢ Indices
  - ➢ Query optimization
  - ➢ Transactions

❖ Parallel DBMSs (e.g. Vertica)

- Advantages (Strong functionalities)
  - ➢ SQL
  - ➢ Indices
  - ➢ Query optimization
  - ➢ Transactions
- Disadvantages (week functionalities)
  - Difficult to scalable
  - Not suitable where faults occur frequently
  - Expensive
  - Hard to setup an use

Figure 2-1. Comparison between NoSQL databases and Parallel DBMSs

Parallel databases support the relational data model, and provide SQL for schema definition and data manipulation. Most systems use a “shared nothing” architecture [97], where each computer node in the system maintains a partition of the data. Sophisticated indexing and query optimization techniques are supported, and indices built for each data partition are maintained by the same node hosting the data partition. A query execution plan is first decomposed into a number of “sub-plans”, which are sent to every relevant node for local execution. Then the local results from each node are combined to generate the global final result. Since supporting efficient online transactions is a major goal of parallel database systems, most of them do not scale to a large number of nodes because concurrency control grows more and more complicated as further nodes are involved. To achieve low execution latencies for transactions, they are usually deployed on a small number of powerful machines, which are likely to be expensive.

Another disadvantage of parallel relational databases is they are hard to configure and maintain [58][161].

Many new systems have been proposed since 2011 with the goal of supporting ACID transactions at a much larger scale, represented by VoltDB [137][156] and Spanner [45]. Specifically, VoltDB relies on a new in-memory architecture, a single-threaded execution model, and heavy use of stored procedures to eliminate the necessity of a big portion of locking. Spanner builds a globally distributed architecture that may span across thousands of nodes, and leverages a global time API backed by GPS and atomic clocks to achieve external consistency of general transactions. Nonetheless, as discussed above, support for transactions is not a strong requirement for our case of social media data analysis.

NoSQL databases, on the other hand, mainly sacrifice the ability to handle ACID transactions to achieve high scalability over a large number (hundreds to thousands) of computer nodes. Data is replicated across different nodes for fault tolerance, and many systems allow eventual consistency among replicas of the same piece of data. Many NoSQL database systems are open-source or free to download, and can be easily set up across a variety of heterogeneous commodity machines. Flexible data schemas are usually allowed, meaning that every data record can have a different set of fields and values. This provides a perfect fit for the evolving data schemas of social media applications. Moreover, most NoSQL databases are inherently integrated with parallel processing frameworks such as Hadoop MapReduce [99], which makes it easier to support integrated queries and post-query analysis tasks.

Features required for handling the unique characteristics of social media data are marked with blue squares in Figure 2-1. Basically we need all the advantages from the NoSQL database side, plus a proper indexing mechanism for dealing with the special queries of social media data. Since the access pattern for social media data is mostly write-once-read-many, eventual consistency on NoSQL databases does not cause a big issue. As analyzed in Section 1.3.1, the most suitable index structures required for handling the text queries with social context contraints are not currently supported by the text indexing techniques from either side. Therefore, we choose to use NoSQL databases as the storage substrate, and enhance them with a customizable indexing framework to enable novel index structures for efficient query evaluation.

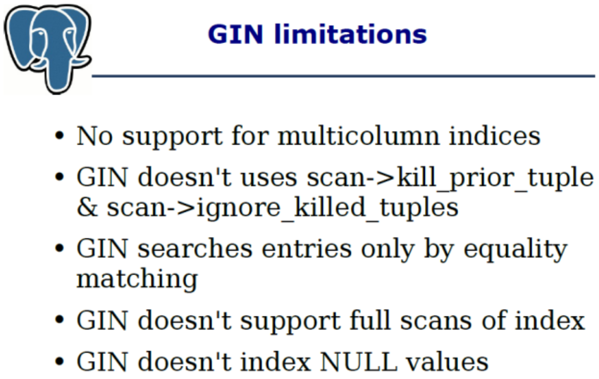
2.2 Related Work

The customized index structures we propose in this chapter aim to address the temporal challenge in social media analytics scenarios. Derczynski et al. [54] provide a more complete list of related work about temporal and spatial queries involving social data. In particular, Alonso et al. [14] give a detailed discussion about the challenges in extracting temporal information from online data and applying such information in text queries. Weikum et al. [158] further elaborate on the open research problems in the context of longitudinal analytics on web archive data. These papers focus on information retrieval applications, where ranking is still important. In addition, they mainly deal with versioned documents in datasets like web archives, so similarities and inter-connections among documents need to be considered in index structure designs. In our case, social updates with different timestamps are independent, and the number of tweets within a given time window is much larger than the number of versions per document. Information retrieval queries need not analyze contents of the documents. In contrast, our queries need to process the content of the related social updates to extract necessary information such as retweet network, so parallel processing of data records is needed after accessing the index. Our experience in this chapter may shed light on possible solutions for the problems discussed in these papers in multiple aspects, including customizable index structures, scalable storage platforms, and efficient index building strategies.

The customizable index structures we use share similar inspiration to composite indices used in relational databases, but index a combination of full-text and primitive-type fields. Compared with traditional inverted indices [170], our framework provides more flexibility about what fields to use as keys and entries to achieve more efficient query evaluation with less storage and computation overhead. Lin et al. [96] proposes text cubes to support multidimensional text database analysis. Due to the requirement for efficient online indexing of high-speed stream data, text cube is not suitable for our case.

Our online indexing mechanism for handling streaming data is comparable to early research on online indexing and incremental index maintenance [32][33][93][94][100][130][144], but is different in that we leverage the functionality of the underlying NoSQL databases to support these features. By using a write ahead log (WAL), HBase helps our framework achieve persistency of even unflushed index data in the memory, which is a missing feature in most existing online indexing systems.

The problem of supporting extendable input data models has been a well researched topic in the communities of object-oriented databases, nested relational databases, and object-relational databases, represented by O2 [27], ANDA (A Nested Database Architecture) [47], and PostgreSQL [127][135][136]. Based on a complete and clearly defined theoretical object-oriented data model, O2 achieves many nice features. It provides well-defined semantics of object identity, inheritance, and methods in a database context. The physical data storage and index organization mechanisms take object sharing, class inheritance, and composition graphs into consideration, and the seamless integration of programming languages and query languages are inspiring for our case of integrated queries and analysis tasks. Similarly, the VALTREE and RECLIST structures in ANDA are optimized to support efficient nesting and unnesting operations. Compared with these systems, our requirements for an extendable input data model are simpler. We don't need to handle the complexity of class hierarchies, composition graphs, or associated methods. Nor do we require nesting operations among subtuples. On the other hand, we need to tackle more complicated issues in other respects, including customizable text index structures, distributed index storage, scalable indexing performance, and dynamic load balancing. The data model of PostgreSQL aims at supporting new abstract data types using general storage management and index maintenance mechanisms. For example, the Generalized Search Tree (GiST) [78] is designed to cover a wide range of tree-structured indices for data with dynamically defined abstract data types. In comparison, our work emphasizes customizability of the elements of index structure itself – namely what to use as index key, entry, and entry fields (for included or computed columns). PostgreSQL extends GiST to build Generalized Inverted Index (GIN) [28] for text data. However, as illustrated in Figure 2-2, it lacks several important features that are needed in our queries for social media data, including multicolumn indices, range queries, and full scans of indices.



GIN limitations

- No support for multicolumn indices
- GIN doesn't uses scan->kill_prior_tuple & scan->ignore_killed_tuples
- GIN searches entries only by equality matching
- GIN doesn't support full scans of index
- GIN doesn't index NULL values

Figure 2-2. Limitations of Generalized Inverted Index in PostgreSQL [28]

Hadoop++ [58], HAIL [59], and Eagle-Eyed Elephant [61] are recent systems that try to extend Hadoop with various indexing mechanisms to facilitate MapReduce queries. Since data is directly stored as files in HDFS [132], these systems do not support efficient random access to fine-grained data records, and thus do not address the requirements of social media data access. Additionally, these systems all schedule MapReduce tasks based on data blocks or splits stored on HDFS (or at least 'relevant' splits), and tasks may have to scan irrelevant data records during query evaluation. In contrast, by using NoSQL databases as the storage substrate, we aim to support record-level indexing in our customizable indexing framework, and limit the post-query analysis computation to only relevant data records.

Google's Dremel [103] achieves efficient evaluation of aggregation queries on large-scale nested datasets by using distributed columnar storage and multi-level serving trees. Power Drill [75] explores special caching and data skipping mechanisms to provide even faster interactive query performance for certain selected datasets. Percolator [115] replaces batch indexing system with incremental processing for Google search. The columnar storage of table data used by both Power Drill and Dremel are inspiring to IndexedHBase in terms of more efficient query evaluation. Conversely, our customizable indexing strategy could also potentially help Dremel for handling aggregation queries with highly selective operations.

2.3 Review of NoSQL Databases

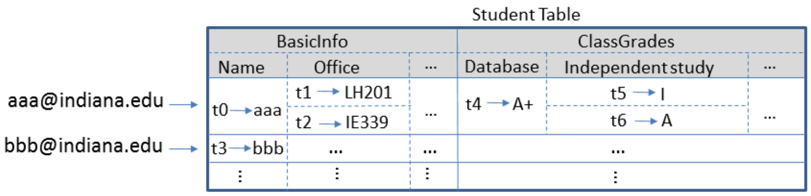
This section investigates and compares four representative distributed NoSQL database systems, namely HBase, Cassandra, MongoDB, and Riak, in terms of five dimensions: data model, data distribution mechanism, data replication and consistency management, data indexing support, and distributed data processing support. Discussions here about NoSQL databases form the basis for the customizable indexing framework presented in the next section.

2.3.1 Data Model

Data model defines the logical organization of data that is presented to the user or client application by a NoSQL database system.

HBase

HBase supports the BigTable data model [37] that was originally proposed by Google. Figure 2-3 illustrates this data model. Data is stored in **tables**; each table contains multiple **rows**, and a fixed number of **column families**. For each row, there can be a varied number of **qualifiers (columns)** within each column family, and at the intersections of rows and qualifiers are table **cells**. Cell contents are uninterpreted byte arrays. Cell values are versioned using **timestamps**, and a table can be configured to maintain a certain number of versions. **Rows are sorted** by row keys, which are also implemented as byte arrays. Within each column family, columns are sorted by column names. Cell values under a column are further sorted by timestamps.



Column families: BasicInfo, ClassGrades
Qualifiers: Name, Office, Database, Independent Study
Row keys: aaa@indiana.edu, bbb@indian.edu
Version timestamps: t0, t1, t2, t3, t4, t5, t6

Figure 2-3. An example of the BigTable data model

Compared with the data model defined by “relations” in traditional relational databases, HBase tables and columns are analogous to tables and columns in relational databases. However, there are **four significant differences**:

(1) Relational databases do not have the concept of “column families”. In HBase, data from different columns under the same column family is stored together (as one file on HDFS). In comparison, data storage in relational databases is either row-oriented, where data in the same row is consecutively stored on physical disks, or column-oriented, where data in the same column is consecutively stored.

(2) In relational databases, each table must have a fixed number of columns (or “fields”); thus every row in a given table has the same set of columns. In HBase, each row in a table can have a different number of columns within the same column family.

(3) In HBase, cell values can be versioned with timestamps. The relational data model does not have the concept of versions.

(4) Generally, NoSQL databases such as HBase do not enforce relationships between tables through mechanisms such as foreign keys in the way relational databases do. User applications have to deal with dependencies among tables through their application logics or mechanisms such as “Coprocessors” supported by HBase [88].

Cassandra

The data model of Cassandra [89][150] is similar overall to HBase, but with **several major differences:**

(1) In Cassandra, the concept of a **table** is equal to a "**column family**"; each table contains only one column family. Different column families are totally separate logical structures containing different sets of row keys. Therefore, compared with the relational data model, Cassandra column families are analogous to tables, and **columns** under column families are analogous to columns in relational tables. Consider the example in Figure 2-3. In Cassandra, the "Student Table" will be implemented either as one "Student" column family containing all the columns in Figure 2-3, or as two separate column families, "Student-BasicInfo" and "Student-ClassGrades".

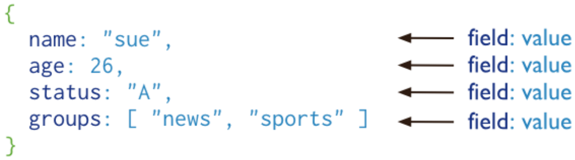
(2) Beyond column families, Cassandra supports an extended concept of "**super column family**", which can contain "**super columns**". A super column is comprised of a (super) column name and an ordered map of **sub-columns**. The limitation of super columns is that all sub-columns of a super column must be deserialized in order to access a single sub-column value.

(3) The order of row keys in a column family depends on the data partition strategy used for a Cassandra cluster. By default the *Random Partitioner* is used, which means row keys are not sorted within a column family and there is no way to do range scans based on row keys without using external facilitating mechanisms such as an extra user-defined indexing column family. Row keys are sorted when the *Order Preserving Partitioner* is used, but this configuration is not recommended [113][162].

(4) Cassandra does not support explicit maintenance of multiple 'versions' of the **column (cell) values**. Column values do have associated timestamps but they are internally used for resolving conflicts caused by **eventual consistency**. Column values with obsolete timestamps are eventually deleted as a result of conflict resolution.

MongoDB

MongoDB is a distributed document database that provides high performance, high availability, and automatic scaling. It uses the concept of "**collections**" and "**documents**" to model data [104]. A collection is a grouping of MongoDB documents which normally have similar schemas. A collection is similar to a table in relational databases and a document takes the place of a table record. Documents are modeled as a data structure following the JSON format, which is composed of field and value pairs. Each document is uniquely identified by a "_id" field as the primary key. The values of fields may include embedded documents, arrays, and arrays of documents [84]. Figure 2-4 shows an example MongoDB document. MongoDB can support access to a sorted list of documents by performing a query with sorting on a document field [122].



```
{
   name: "sue",
   age: 26,
   status: "A",
   groups: [ "news", "sports" ]
}
```

Figure 2-4. An example of the MongoDB document data model [84]

Relationships between documents can be modeled in two ways: references and embedded documents [49].

Riak

Riak is a distributed database designed for key-value storage. Its data model follows a simple "**key/value**" scheme, where the key is a unique identifier of a data object, and the value is a piece of data that can be of various types, such as text and binary [124]. Each data object can also be tagged with additional metadata, which can be used to build secondary indices to support query of data objects [153]. A concept of "**bucket**" is used as a namespace for grouping key/value pairs. Figure 2-5 illustrates an example of the Riak data model.

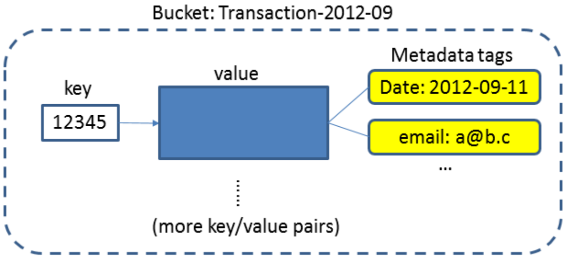


Figure 2-5. An example of the key/value data model in Riak

2.3.2 Data Distribution Mechanism

The data distribution mechanism determines how data operations are distributed among different nodes in a NoSQL database cluster. Most systems use two major mechanisms: **key-range-based** distribution and **hash-based** distribution. Key-range based distribution can easily support range scans of sorted data, but may face the problem of unbalanced access load to different value ranges. Hash-based distribution has the advantage of balanced access load across nodes, but does not support range scans very well.

HBase

HBase uses a **key-range-based** data distribution mechanism. Each table is horizontally split into regions, and regions are assigned to different region servers by the HBase master. Since rows are sorted by row keys in the HBase data model, each region covers a consecutive range of row keys. Figure 2-6 illustrates the architecture of HBase. HBase dynamically splits a region in two when its size goes over a limit, or according to a user-specified ***RegionSplitPolicy***. Users can also force region splits to handle "hot" regions [134]. Since table data is stored in HDFS, region splits do not involve much data movement and can be finished very quickly. Region splits happen in the background and do not affect client applications.

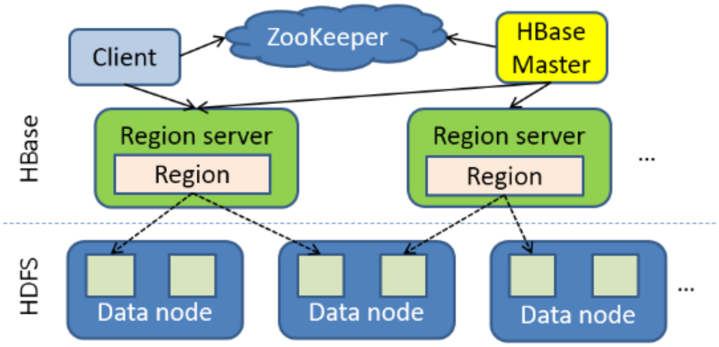


Figure 2-6. HBase architecture

Cassandra

Depending on the configuration about *data partitioner*, a Cassandra cluster may apply either **key-range-based** distribution or **hash-based** distribution.

When the *Random Partitioner* is used (which is the default configuration), nodes in the cluster form a Distributed Hash Table (DHT). Cassandra partitions data across the cluster using consistent hashing. The output range of a hash function is treated as a fixed circular space or “ring" (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space which represents its position on the ring. After position assignment, each node becomes responsible for the region in the ring between it and its predecessor node [89].

To handle a data operation request, the row key of the data operation is first hashed using the MD5 hashing algorithm, and then the operation is sent to the node that is responsible for the corresponding hash value to process. The MD5 hashing step ensures a balanced distribution of data and workload even in cases where the application data has an uneven distribution across the row keys, because the hash values of the possibly preponderant sections of row keys will still demonstrate an even distribution [162].

When the *Order Preserving Partitioner* is used, each node becomes responsible for the storage and operations of a consecutive range of row keys. In this case, when the application data has an uneven distribution across the row key space, the nodes will have an unbalanced workload distribution [162].

Load skew may be further caused by two other factors. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. Second, the basic data distribution algorithm is oblivious to the heterogeneity in the performance of nodes. To address these issues, Cassandra analyzes load information on the ring and **moves lightly loaded nodes on the ring** to alleviate heavily loaded nodes [89]. Also, every time a new node is added, Cassandra will assign a range of keys to that node such that it takes responsibility for half the keys stored on the node that currently stores the most keys. In a stable cluster, data load can also be rebalanced by careful administrative operations, such as manual assignment of key ranges or node take-down and bring-up [162].

MongoDB

MongoDB also supports both **key-range-based** distribution and **hash-based** distribution through configurations. The working logic is similar to Cassandra. MongoDB organizes nodes in units of **shards** and partitions the key space of data collections into **chunks**. Chunks are then distributed across the shards. Dynamic load balancing among shards is achieved through ***chunk splitting*** and ***chunk migration*** [128].

Riak

Riak also uses a DHT to support **hash-based** distribution. When the client performs key/value operations, the bucket and key combination is hashed. The resulting hash maps onto a 160-bit integer space. Riak divides the integer space into equally-sized partitions, each managed by a process called a virtual node (or “vnode”). Physical machines evenly divide responsibility for vnodes. Figure 2-7 illustrates an example partition distribution of the hash value space among 4 nodes.

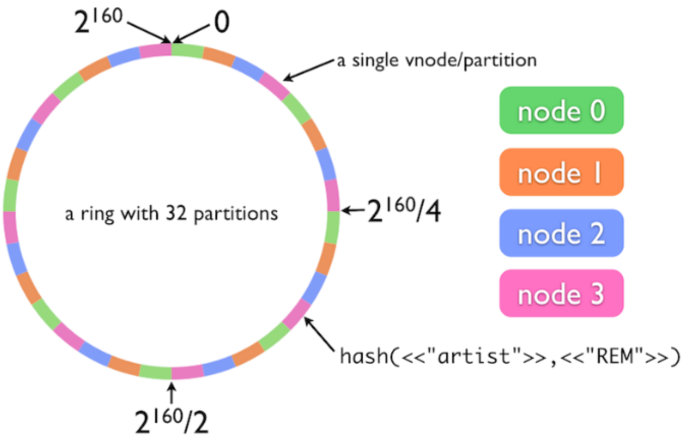


Figure 2-7. Hash-based data distribution in Riak [124]

2.3.3 Data Replication and Consistency Management

Almost all NoSQL database systems rely on replication to ensure high data availability in distributed deployments. However, these systems use different strategies to manage the consistency of multiple replicas of the same piece of data. This section only covers **data-object-level** consistency, i.e. consistency among replicas of single data objects. Most NoSQL database systems do not address **transaction-level** consistency, which may involve a series of updates to multiple related data objects. Supporting transaction-level consistency will require additional synchronization extensions [115].

HBase

Since HBase uses HDFS for data storage, it inherits the **replication and consistency management from HDFS**. Specifically, the replication factor and replica location method is decided by HDFS. Since HDFS enforces complete consistency – a write operation does not return until all replicas have been updated – HBase also ensures **complete consistency** for its data update operations. Upon receiving a data update operation, the HBase region server first records this operation in a write-ahead log (WAL), and then puts it in *memstore* (an in-memory data structure). When the *memstore* reaches its size limit, it is written to an HFile [30]. Both the WAL file and the store file are HDFS files. Therefore, complete consistency is guaranteed for all data updates. HDFS and HBase do not originally support deployment with **data center awareness**.

Cassandra

Each data item in Cassandra is replicated at N hosts, where N is the replication factor. The node responsible for the key of the data item is called a coordinator node. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 nodes in the ring. Cassandra provides various **replication policies** such as "**Rack Unaware**", "**Rack Aware**" (within a datacenter) and "**Datacenter Aware**". Replicas are chosen based on the replication policy of the application. If the "Rack Unaware" replication strategy is chosen, then the non-coordinator replicas are chosen by picking N-1 successors of the coordinator on the ring.

Cassandra allows **eventual consistency** among data replicas to achieve high availability, partition tolerance and short response time for data operations. Cassandra extends the concept of eventual consistency by offering **tunable consistency**. For any given read or write operation, the client application decides how consistent the requested data should be. The consistency level can be specified using values such as “ANY”, “ONE”, “QUORUM”, “ALL”, etc. Some values are specially designed for multiple data center clusters, such as “LOCAL_QUORUM” and “EACH_QUORUM” [3]. To understand the meaning of consistency levels, take “QUORUM” for write as an example. This level requires that a write operation will be sent to all replica nodes, and will only return after it is written to the commit log and memory table on a quorum of replica nodes.

Cassandra provides a number of built-in repair features to ensure that data remains consistent across replicas, including ***Read Repair***, ***Anti-Entropy Node Repair***, and ***Hinted Handoff*** [3].

MongoDB

MongoDB manages data replication in the units of **shards**. Each shard is a replica set, which can contain one **primary** member, multiple **secondary** members, and one **arbiter**. The primary is the only member in the replica set that receives write operations. MongoDB applies write operations on the primary and then records the operations on the primary’s oplog. Secondary members replicate this log and apply the operations to their data sets. All members of the replica set can accept read operations. However, by default, an application directs its read operations to the primary member. If the current primary becomes unavailable, an election determines the new primary. Replica sets with an even number of members may have an arbiter to add a vote in elections for primary [123]. Replica sets can be made **data center-aware** through proper configurations [48].

Data synchronization between primary and secondaries are completed through **eventual consistency** [111]. If *Read Preference* is set to **non-primary**, read operations directed to secondaries may get stale data [121]. MongoDB also supports **tunable consistency** for each write operation through the “Write Concern” parameter [163].

Riak

Riak allows the user to set a **replication number** for each **bucket**, which defaults to 3. When a data object's key is mapped onto a given partition of the circular hash value space, Riak automatically replicates the data onto the next two partitions (Figure 2-8). Riak supports multi-data center replication through the concept of “primary cluster” and “secondary clusters” [107].

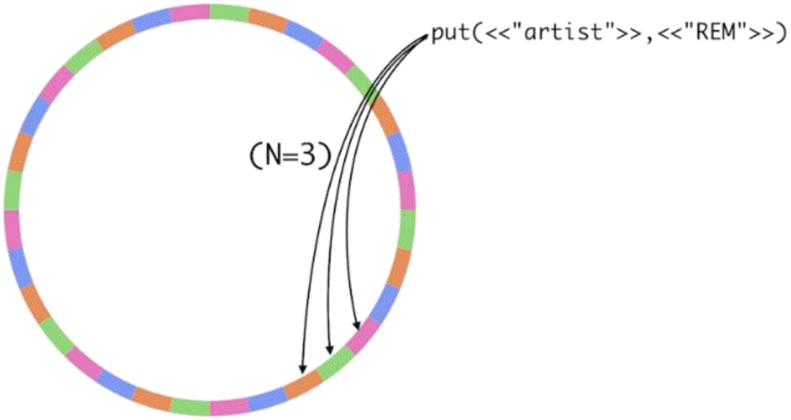


Figure 2-8. Data replication in Riak [124]

Similar to Cassandra, Riak also supports **tunable consistency** for each data operation [62]. It relies on mechanisms such as ***Vector Clock***, ***Hinted Handoff***, and ***Read Repair*** to resolve conflicts and ensure consistency [124].

2.3.4 Data Indexing Support

There are two major categories of indexing involved in distributed NoSQL database systems: **primary indexing** and **secondary indexing**. In terms of distributed index storage, there are two ways of **index partitioning**: **partition by original data** or **partition by index key**. “Partition by original data" means that each node in the cluster only maintains the secondary index for the portion of the original data that is locally hosted by that node. In this case, when a query involving an indexed field is evaluated, the query must be sent to every node in the cluster. Each node will use the local portion of secondary index to do a "partial evaluation" of the query, and return a subset of results. The final result is generated by combining results from all the nodes. Figure 2-9 illustrates partition by original data. "Partition by index key" means that a global index is built for the whole data set on all the nodes, and then distributed among the nodes by making partitions with the key of the index. To evaluate a query about an indexed field value, only the node maintaining the index for that queried field value is contacted, and it processes all related index entries to get the query result. Figure 2-10 illustrates partition by index key.

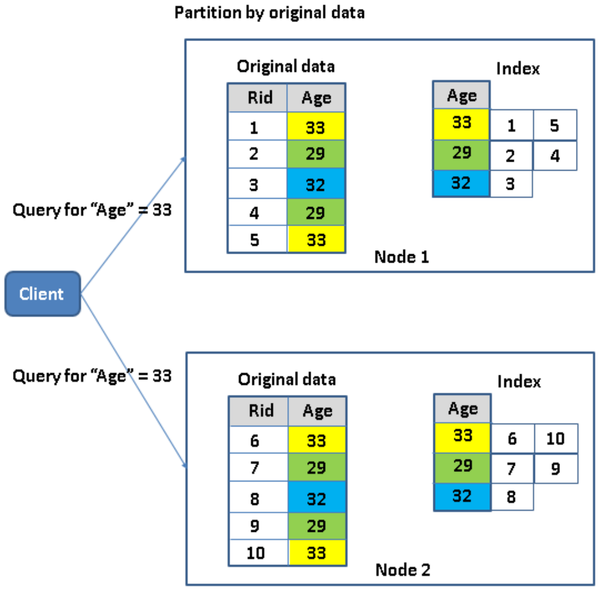


Figure 2-9. Partition by original data

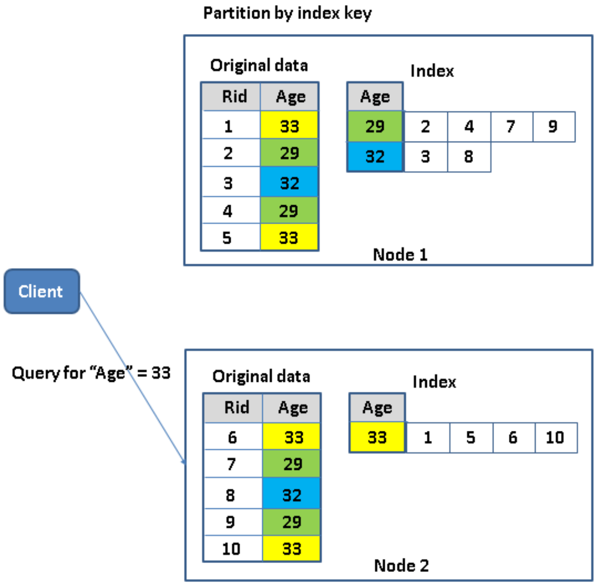


Figure 2-10. Partition by index key

Partition by original data is good for handling complicated queries involving multiple fields and constraints, because each node can partially evaluate the query by only accessing local data. Although the query has to be broadcast to all nodes, the total amount of communication is much smaller than the size of the relevant part of the indices for each field. Partition by index key works better when queries are simple: the major part of evaluation is the processing and transmission of the related index entries, and only the exact related node(s) need to be contacted.

HBase

Primary indexing. HBase builds a **primary index** on the row keys, which is conceptually similar to a distributed multi-level **B+-tree** index. HBase maintains two global catalog tables: ***ROOT*** and ***META***. *ROOT* always has only one region, and its location is stored in ***ZooKeeper***. *ROOT* keeps track of the regions of the *META* table, and *META* keeps a list of all regions in the system, as well as which region servers are hosting them [36]. On the region server, data is read from and written to HFiles on HDFS, and the HFile format contains information about a multi-level B+-tree-like data structure [30]. The primary index is a clustered index because the data records are stored directly in the index entries.

Secondary Indexing. HBase does not originally support secondary indices for cell values.

Cassandra

Primary Indexing. The DHT architecture of Cassandra basically builds a distributed **primary key hash index** for the row keys of column families. This primary index is a **clustered index** since data records are contained in the index entries.

Secondary Indexing. Beyond primary key index, Cassandra supports creation of secondary indices on any column values [4]. The internal secondary index implementation depends on whether the data type of the column values is **non-text** data and **text** data.

For **non-text** column values, Cassandra can create **hash indices** which are internally maintained as **hidden index column families** [81]. This index column family stores a mapping from index values to a sorted list of matching row keys. Since the index is a hash index, query results are not sorted by the order of the indexed values. Furthermonre, range queries on indexed columns cannot be completed by using the index, although an “equal” match in the index returns an ordered list relevant row keys.

For **text** column values, the commercial version of Cassandra, DataStax, supports secondary indices on text data through integration with Solr [51]. Moreover, the indices are stored as Lucene index files [21], which means various query types, including equal queries, wildcard queries, range queries, etc. can be supported.

Consistency between data and index. Data update + index update is an atomic operation, so **immediate consistency** is ensured between the original data and index data.

Secondary index partition scheme. Each node maintains the secondary indices for its own local part of original data. Therefore, secondary indices are **partitioned by original data**.

Limitations. Cassandra secondary indices currently have several **limitations**. First, they can only index values from single columns; multidimensional indices as used in [69] are not supported. Second, as mentioned above, indices for **non-text** columns cannot be used to evaluate range queries. Finally, even if a query specifies constraints on multiple indexed columns, only one index will be used to quickly locate the related row keys. Range constraints can be specified on additional columns in the query, but are checked against the original data instead of using indices [4].

MongoDB

Primary Indexing. MongoDB automatically forces the creation of a **primary key index** on the *_id* field of the documents. Index entries are sorted by *_id*, but note that this primary key index is **not a clustered index** in Database terms, meaning the index entries only contains pointers to actual documents in the MongoDB data files. Documents are not physically stored in the order of *_id* on disks.

Secondary Indexing. Beyond the primary index, MongoDB supports various secondary indices for field values of documents, including single field index, multidimensional index, multikey index, geospatial index, text index, and hashed index [82]. Single field, multidimensional, and multikey indices are organized using **B-tree** structures. The geospatial index supports indexing using **quad trees** [1] on 2-dimensional geospatial data. The official documentation does not provide details about how the text indices are implemented, but it is known that basic features such as stopping, stemming, and scoring are supported [139]. Text index in MongoDB is still in beta version. The hashed index can be used to support both hash-based data distribution and equality queries of field values in documents, but obviously cannot be used for range queries.

Consistency between data and index. Data is indexed on the fly in the same atomic operation. Therefore, **immediate consistency** is ensured between the original data and index data.

Secondary index partition scheme. Each shard maintains the secondary index for its local partition of the original data. Therefore, secondary indices are **partitioned by original data**.

Riak

Primary Indexing. As explained in section 2.4, Riak builds a **primary key hash index** for its key/value pairs through DHT. This index is a **clustered index** because data objects are directly stored together with the index keys.

Secondary Indexing. Riak supports secondary indices on the tagged attributes of the key/value pairs and inverted indices for text data contained in the value. For secondary indices on tagged attributes, exact match and range queries are supported. However, current Riak implementation forces the limitation that one query can only use secondary index search on one indexed attribute (field). Queries involving multiple indexed attributes have to be broken down as multiple queries; then the results are then merged to get the final result [153]. No details are given about the internal structures used for secondary indices in the official Riak documentation. According to the brief mention in [6], it seems that a flat list of key/entries is used.

For inverted indices on values of text type, text data contained in the values of key/value pairs are parsed and indexed according to a predefined index schema. Similar to DataStax, Riak also tries to integrate with the interface of Solr, and stores indices using the Lucene file format so as to support various types of queries on text data, such as wildcard queries and range queries [152].

Consistency between data and index. Data update + index update is an atomic operation, so **immediate consistency** is ensured between the original data and index data.

Secondary index partition scheme. For secondary indices on tagged attributes, each node maintains the indices for its local part of original data. Therefore, the indices are **partitioned by original data**, while the text index is **partitioned by terms** (keys in inverted index). In Riak, text index schemas are configured at the level of buckets. All the key/value pairs in a configured bucket will be parsed and indexed according to the same given schema. A global inverted index is created and maintained for all key/value pairs added to that bucket, then partitioned by terms in the inverted index and distributed among all the nodes in the ring.

2.3.5 Distributed Data Processing Support

HBase and Cassandra

HBase and Cassandra both support parallel data processing by integration with Hadoop MapReduce [74][77][99], which is designed for fault tolerant parallel processing of large batches of data. It implements the full semantics of the MapReduce computing model and applies a comprehensive initialization process for setting up the runtime environment on the worker nodes. Hadoop MapReduce uses disks on worker nodes to save intermediate data and does grouping and sorting before passing them to reducers. A job can be configured to use zero or multiple reducers.

MongoDB

MongoDB provides two frameworks to apply parallel processing to large document collections: **aggregation pipeline** [13] and **MapReduce** [98].

The aggregation pipeline completes aggregate computation on a collection of documents by applying a pipeline of data operators, such as *match*, *project*, *group*, etc. By using proper operators such as *match* and *skip* at the beginning of the pipeline, the framework is able to take advantage of existing **indices** to limit the scope of processing to only a related subset of documents in the collection and thus achieve better performance. Currently MongoDB implementation enforces several important limits on the usage of aggregation pipelines, including input data types, final result size, and memory usage by operators [12]. This implies that the pipeline operators work completely in memory and do not use external disk storage for computations such as sorting and grouping.

The MapReduce framework is designed to support aggregate computations that go beyond the limits of the aggregation pipeline, as well as extended data processing that cannot be finished by the aggregation pipeline. MapReduce functions are written in JavaScript, and executed in

MongoDB daemon processes. Compared with Hadoop MapReduce, MongoDB MapReduce is different in several aspects. In the MongoDB version, *reduce* is only applied to the *map* outputs where a key has multiple associated values. Keys associated with single values are not processed by *reduce*. Furthermore, besides *map* and *reduce*, an extra *finalize* phase can be applied to further process the outputs from *reduce*, and a special "incremental MapReduce" mechanism is provided to support dynamically growing collections of documents. This mechanism allows *reduce* to be used for merging the results from the latest MapReduce job and previous MapReduce jobs. Also the framework supports an option for choosing the way intermediate data is stored and transmitted. The default mode stores intermediate data on local disks of the nodes, but the client can specify to only use memory for intermediate data storage, in which case a limit is enforced on the total size of key/value pairs from the *map* output. Finally, functions written in JavaScript may limit the capabilities of *map* and *reduce*. For example, it is hard or even impossible to access an outside data resource such as a database or distributed file system [95][38] to facilitate the computation carried out in *map* and *reduce*.

Riak

Riak provides a lightweight MapReduce framework for users to query the data by defining MapReduce functions in JavaScript or Erlang [151]. Furthermore, Riak supports MapReduce over the search results by using secondary indices or text indices. Riak MapReduce is different from Hadoop MapReduce in several ways. There is never more than one reducer running for each MapReduce job. Intermediate data is transmitted directly from mappers to the reducer without being sorted or grouped. The reducer relies on its memory stack to store the whole list of intermediate data, and has a default timeout of only five seconds. Therefore, Riak MapReduce is not suitable for processing large datasets.

2.3.6 Summary

In summary, there is no standard data model currently shared by the NoSQL database systems. Each system may adopt models suitable for a specific type of applications, but flexible schemas are usually allowed in most data models. Systems may adopt various data distribution and replication mechanisms to achieve scalability and high availability. In case a **hash based** data distribution is used, sorted secondary indices will be needed to do range scans of data. Most systems provide native support for parallel data processing models such as MapReduce, but in order to handle large datasets or query results, a sophisticated and fault-tolerant framework like Hadoop MapReduce [99] is required. Finally, secondary indexing is an area where current NoSQL databases are not performing well. Figure 2-11 summarizes the four representative systems discussed above in terms of two categories of index structures that have been well studied in the database community: multi-dimensional indices and single-dimensional indices. It is obvious that a varied level of secondary indexing support is demonstrated by different systems. Moreover, as is evident in Section 1.3.1, the single-field inverted indices currently supported by some systems do not work well for the special query patterns of social media data. As such it is necessary to extend them with a fully customizable indexing framework. It will be a great contribution to the NoSQL world if such a framework can be generally integrated with most NoSQL databases, as it will help equalize the ragged level of indexing support across different systems.

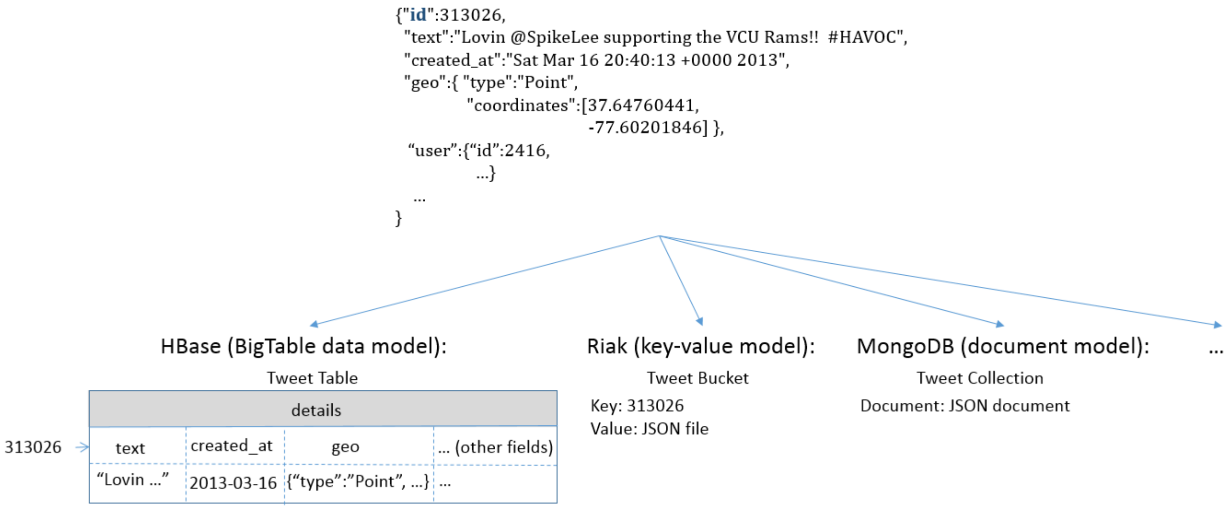
	Single-dimensional indexes					Multidimensional indexes		
	Sorted (B+ tree)		Unsorted (Hash)		Inverted index (Lucene)	R tree (PostGIS)	K-d tree (SciDB)	Quad tree
	Single -field	Composite	Single -field	Composite	Single -field			
HBase								
Cassandra			Yes		Yes			
Riak	Yes				Yes			
MongoDB	Yes	Yes	Yes		Yes			Yes

Figure 2-11. Varied level of indexing support among existing NoSQL databases

2.4 Customizable Indexing Framework

In this section we propose a fully customizable indexing framework that can be generally implemented over most NoSQL databases. Although our motivation derives from the lack of customizability in existing text indexing techniques, the framework can actually be used to define customized index structures for both text and non-text data.

2.4.1 Input Data Model



```
{"id":313026,
 "text":"Lovin @SpikeLee supporting the VCU Rams!!  #HAVOC",
 "created_at":"Sat Mar 16 20:40:13 +0000 2013",
 "geo":{ "type":"Point",
          "coordinates":[37.64760441,
                         -77.60201846] },
 "user":{"id":2416,
          ...}
  ...
}
```

Figure 2-12. An example of the input data model to the customizable indexing framework

The customizable indexing framework uses the concept of **data record** and **record set** to model the input data to be indexed. Equation (1) gives the conceptual definition of a **record set**. A **record set** is composed of zero to multiple **data records.** Each can be modeled by a JSON type of nested key-value pair list data structure uniquely identified by an "**id**" field, as shown in Figure 2-12. On the one hand, this data model is consistent with many existing social media data sources such as the Twitter Streaming API [148]. But the model of a record set and a data record can be easily mapped to the data storage units of various NoSQL databases. For example, a **record set** can be implemented as a table in HBase [19], a bucket in Riak [125], or a collection in MongoDB [105]. Correspondingly, a **data record** can be implemented as a row in HBase, an object in Riak, or a document in MongoDB.

$$\text{Record set } S = \{<ID, field_1, field_2, \dots field_N> \mid N > 0\}. \qquad (1)$$

2.4.2 Abstract Index Structure

Equation (2) gives our conceptual definition of an index. We define *index()* as a function that takes one data record *r* as input and generates a set of **index entries** as output. An **index entry** is defined as a tuple $<key, EID, EF_1, EF_2, \dots EF_N>$ $(N \geq 0)$, where *key* denotes an **index key**, *EID* denotes an **entry ID**, and $EF_1, EF_2, \dots EF_N$ denote a varied number of **entry fields**. An index *I* over a record set *S* is defined as the union of all the sets of index entries generated by applying *index()* on each data record in *S*.

$$\text{Index } I = \bigcup_{\forall r \in S} index(r);\ index : r \rightarrow \{<key, EID, EF_1, EF_2, \dots EF_N> \mid (N \geq 0)\}. \qquad (2)$$

Figure 2-13 illustrates the abstract index structure used by our framework. The overall structure is organized by a sorted list of index keys. Index entries with the same key are grouped together, and further sorted by their entry IDs. Each entry usually corresponds to one original data record that contains the index key on the indexed field. The entry fields can be used to embed additional information about the indexed data, which could be either fields directly from the original data record (included columns), or computation results based on them (computed columns). This structure is similar to the posting lists used in inverted indices [170], but the major difference is that our framework allows users to customize what to use as index keys, entry IDs, and entry fields through an index configuration file in XML format, as illustrated in Figure 2-14. The configuration file contains multiple "index-config" elements that hold the mapping information between source record sets and customized index structures. Each element can flexibly define how to generate index entries off a given data record from the source record set. For more complicated index structures, users can implement their own *index()* function (UDF) and use it by setting the "indexer-class" element.

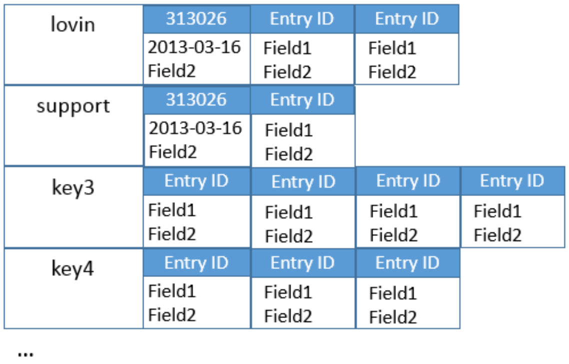
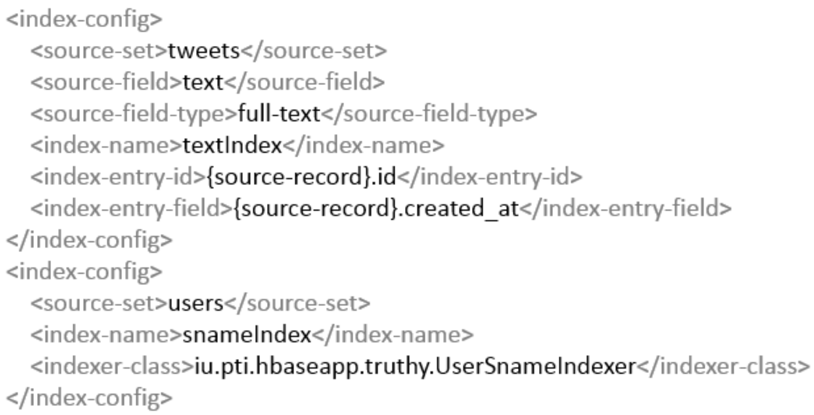


Figure 2-13. Abstract index structure



```
<index-config>
  <source-set>tweets</source-set>
  <source-field>text</source-field>
  <source-field-type>full-text</source-field-type>
  <index-name>textIndex</index-name>
  <index-entry-id>{source-record}.id</index-entry-id>
  <index-entry-field>{source-record}.created_at</index-entry-field>
</index-config>
<index-config>
  <source-set>users</source-set>
  <index-name>snameIndex</index-name>
  <indexer-class>iu.pti.hbaseapp.truthy.UserSnameIndexer</indexer-class>
</index-config>
```

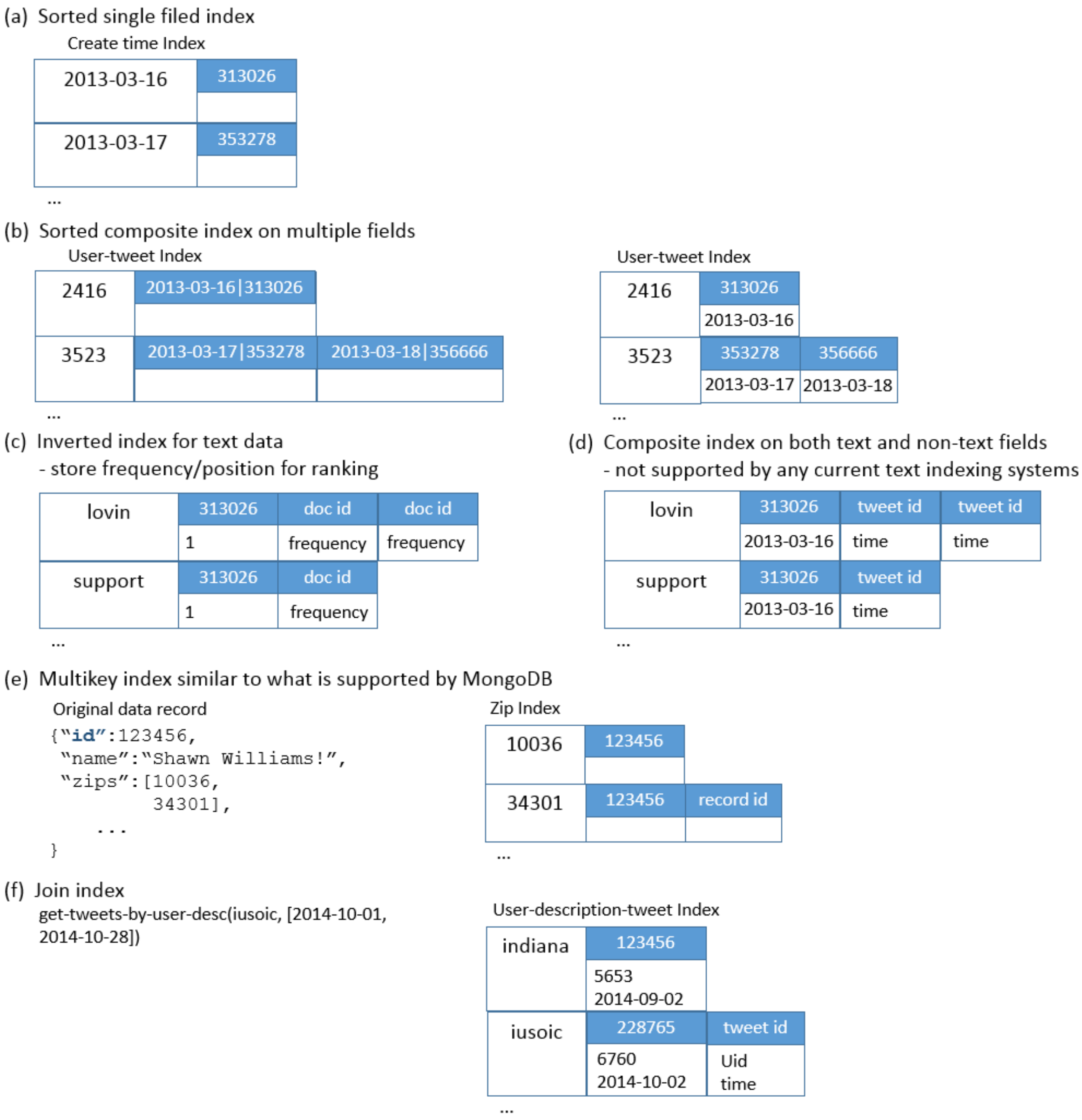
Figure 2-14. An example index configuration file

By defining proper index configurations or UDFs, it is possible to create various index structures, seen in Figure 2-15. In the simplest form, we can create sorted single-field indices by directly using the indexed field values as the index keys, and the ID of the corresponding original data records as the index entry ID (Figure 2-15 (a)). Beyond this, sorted composite indices can be created by adding additional field values to either the entry ID or the entry fields (Figure 2-15 (b)). Traditional inverted indices can be created by using the tokenized text terms as the index keys, the document IDs as the entry IDs, and the term frequency and position information as the entry fields (Figure 2-15 (c)). Moreover, by replacing the frequency information in the entry fields with values from other fields of the original data records, composite indices on both text and non-text fields can be defined (Figure 2-15 (d)). Such customization is not supported by current text indexing systems, but is exactly what is needed for evaluating the text queries with constraints on the social context. For array values in the original data records, it is possible to create index structures similar to the “multikey index” supported by MongoDB [82] by using a UDF that delves into the array field, creating one index key for every unique value in the array (Figure 2-15 (e)). Finally, inspired by research from the area of data warehouses [114], it is also possible to create join index structures for evaluating queries involving multiple record sets

(Figure 2-15 (f)). For instance, suppose we have a new query ***get-tweets-by-user-desc****(keyword, time-window)*, which is supposed to return all the tweets created within the given *time window* that are posted by users who have the given *keyword* in the personal description text of their profiles. Assuming the tweet information and the user information are stored in two different record sets, evaluating such a query will require a **Join** operation of the two record sets without a proper index. However, by building a customized index structure that uses keywords from the user description text as keys, tweet IDs posted by the corresponding users as entry IDs, and user ID and tweet creation time as the entry fields, it will be possible to evaluate the query by only accessing the index.

2.4.3 Interface to Client Applications

Figure 2-16 presents the major operations provided by our customizable indexing framework to client applications. The client application can use a **general customizable indexer** to index a data record. Upon initialization, the **general customizable indexer** reads the index configuration file provided by the user. If a user-defined indexer class is specified, a corresponding indexer instance will be created. Both general and user-defined indexers must implement the *index()* method. This method takes a data record from a source record set as input, and returns a mapping from related index names to their corresponding index entries. When *index()* is invoked on the general indexer during runtime, all related “index-config” elements are used to generate index entries, either by following the rules defined in “index-config” or by invoking a user-defined indexer.



```
{"id":123456,
 "name":"Shawn Williams!",
 "zips":[10036,
         34301],
    ...
}
```

Figure 2-15. Example index structures that can be created with the customizable indexing framework

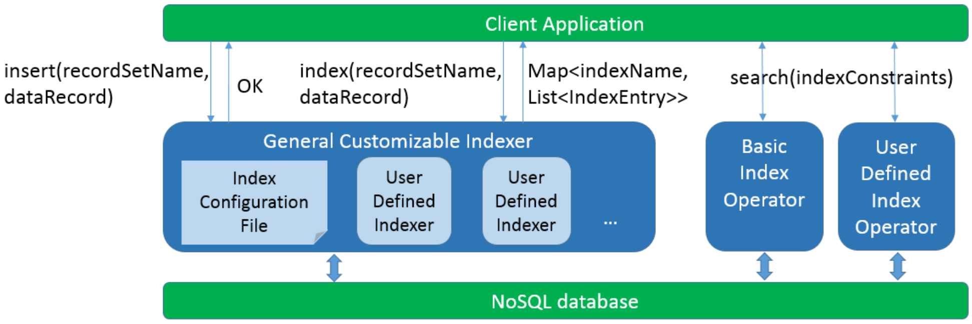

Client Application
insert(recordSetName, dataRecord)
OK
index(recordSetName, dataRecord)
Map<indexName, List<IndexEntry>>
search(indexConstraints)
General Customizable Indexer
Index Configuration File
User Defined Indexer
User Defined Indexer
...
Basic Index Operator
User Defined Index Operator
NoSQL database

Figure 2-16. Interface to client applications

Based on the general customizable indexer, two indexing mechanisms can be supported: **online indexing** and **batch indexing**. Online indexing is implemented through the *insert()* method. The client application invokes this method to insert one data record into a source record set. The indexer will first do the insertion, then generate index entries for this data record by invoking *index()* and insert them into the corresponding index structures. From the client application's perspective, data records are indexed "online" when they are inserted into the source record set. Efficient online indexing is crucial for the loading of streaming data. Batch indexing assumes original data records are already stored in the NoSQL databases as record sets, and does indexing for the whole sets in batches. The batch indexing application scans the source record set, and invokes the *index()* method for every data record. The returned index entries are inserted into the corresponding index structures.

To complete a search using an index structure, the client application can invoke a **basic index operator** provided by the framework, or a **user-defined index operator**. Multiple constraints can be specified as parameters to filter the index entries by their keys, entry IDs, or entry fields. Constraint types currently supported are **value set constraint**, **range constraint**, and **regular expression constraint**. A value set constraint is specified in the form of *{val1,val2,...}*, and can be used to select index keys, entry IDs, or entry fields that match any of the values in the set. Similarly, a range constraint is expressed in the form of *[lower, upper]*, and a regular expression is in the form of <*regular expression*>. For a special class of regular expression constraints, prefix constraints, we also support a simplified expression in the form of ~*prefix**~ (multiple characters following *prefix*) or ~*prefix?*~ (single character following *prefix*).

2.4.4 Implementation on HBase – IndexedHBase

As discussed in section 1.3.1, an actual implementation of the indexing framework needs to not only provide customizable index structures, but also support scalable index data storage and efficient indexing speed for high-volume streaming data. Taking these factors into consideration, our key observation about existing distributed NoSQL databases is that they already support scalable data storage and efficient random access to individual data records following their respective data models. Therefore, by defining a proper mapping between the abstract index structures and the actual storage units and data models of the underlying NoSQL databases, it is possible to leverage their existing data distribution and load balancing mechanisms to achieve scalable indexing for our framework. Figure 2-17 illustrates this idea.

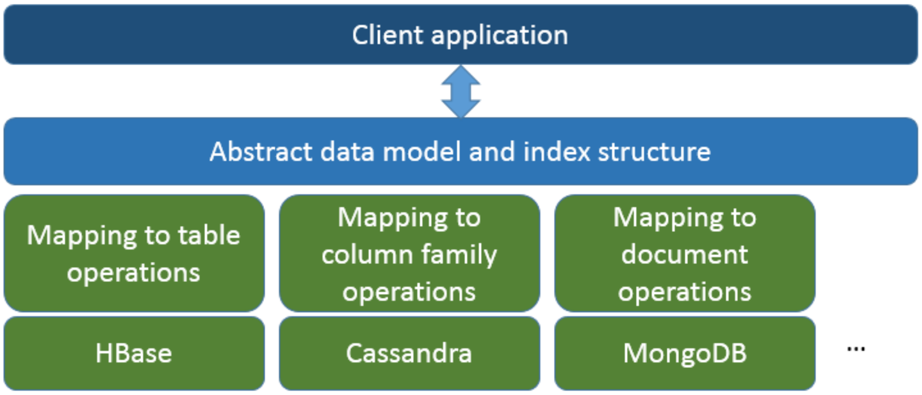

Client application
Abstract data model and index structure
Mapping to table operations
Mapping to column family operations
Mapping to document operations
HBase
Cassandra
MongoDB
...

Figure 2-17. Implementation through mapping to the data model of NoSQL databases

We have developed an implementation over HBase in our scalable analysis architecture which we call IndexedHBase. Figure 2-18 illustrates the mapping we designed for IndexedHBase. Specifically, we use an HBase table to implement an index structure, a row key for an index key, a column name for an entry ID, and a column value for all the entry fields. Since HBase stores table data under the hierarchical order of <row key, column name, timestamp>, it is easy to support range scans over the index keys or entry IDs. Based on the region split and load balancing mechanisms provided by HBase, we are able to achieve efficient and scalable real-time indexing of streaming data. HBase supports fast atomic row-level mutations; hence the insertion of a dynamic data record only involves random write operations to a limited number of rows in the index tables, and does not affect operations on any other rows. Moreover, write operations do not block read even on the same row, so the impact to concurrent query evaluations is minimum. Updates of original data records and index entries are completed as consecutive write operations to different tables, and eventual consistency between index and original data can be guaranteed at the level of milliseconds. Finally, since HBase is inherently integrated with the Hadoop software stack, we can leverage the Hadoop MapReduce framework to effectively support integrated queries and analysis workflows of social media data.

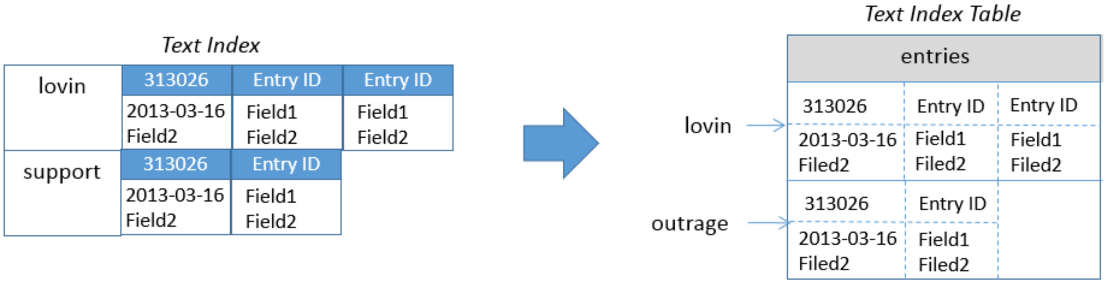


Figure 2-18. Mapping between an abstract index structure and an HBase table

The online indexing mechanism on IndexedHBase is implemented by translating the actions in *insert()* into the corresponding table operations in HBase. The batch indexing mechanism is implemented as a "map-only" Hadoop MapReduce job using the table for the source record set as input. The job accepts a source table and index table name as parameters and starts multiple mappers to index data in the source table in parallel, each processing one region of the table. Each mapper works as a client application to the general customizable indexer and creates one indexer instance at its initialization time. The indexer is initialized using the given index table name so that when *index()* is invoked, it will only generate index records for that single table. The *map()* function takes a <key, value> pair as input, where "key" is a row key in the source table and "value" is the corresponding row data. For each row of the source table, the mapper uses the general customizable indexer to generate index table records and write these records as output. All output records are handled by the table output format, which will automatically insert them into the index table.

2.4.5 Implementation on Other NoSQL Databases

It is possible to implement our customizable indexing framework on other NoSQL databases by designing proper mapping between the abstract index structure and the data model of the corresponding system. Such mapping should take the practical requirements for the indexing framework and the granularity of data access of the specific NoSQL database system into consideration. In order to achieve range scans of index keys and entries on systems using hash-based data distribution mechanisms, it is often necessary to leverage their native secondary indexing support. Table 2-1 provides a list of suggested mapping for the other three representative NoSQL databases discussed in section 2.2.

Table 2-1. Suggested mappings for other NoSQL databases

Feature needed	Cassandra	Riak	MongoDB
Fast real time random insertion and updates of index entries.	Yes. Index key as row key and entry ID as column name, or index key + entry ID as row key.	Yes. Index key + entry ID as object key.	Yes. Index key + entry ID as "_id" of document.
Fast real time random read of index entries.	Yes. Index key as row key and entry ID as column name, or index key + entry ID as row key.	Yes. Index key + entry ID as object key.	Yes. Index key + entry ID as "_id" of document.
Scalable storage and access speed of index entries.	Yes.	Yes.	Yes.
Efficient range scan on index keys.	Yes with order preserving hash function.	Doable with a secondary index on an attribute whose value is the object key.	Doable with Index key + entry ID as "_id" of document.

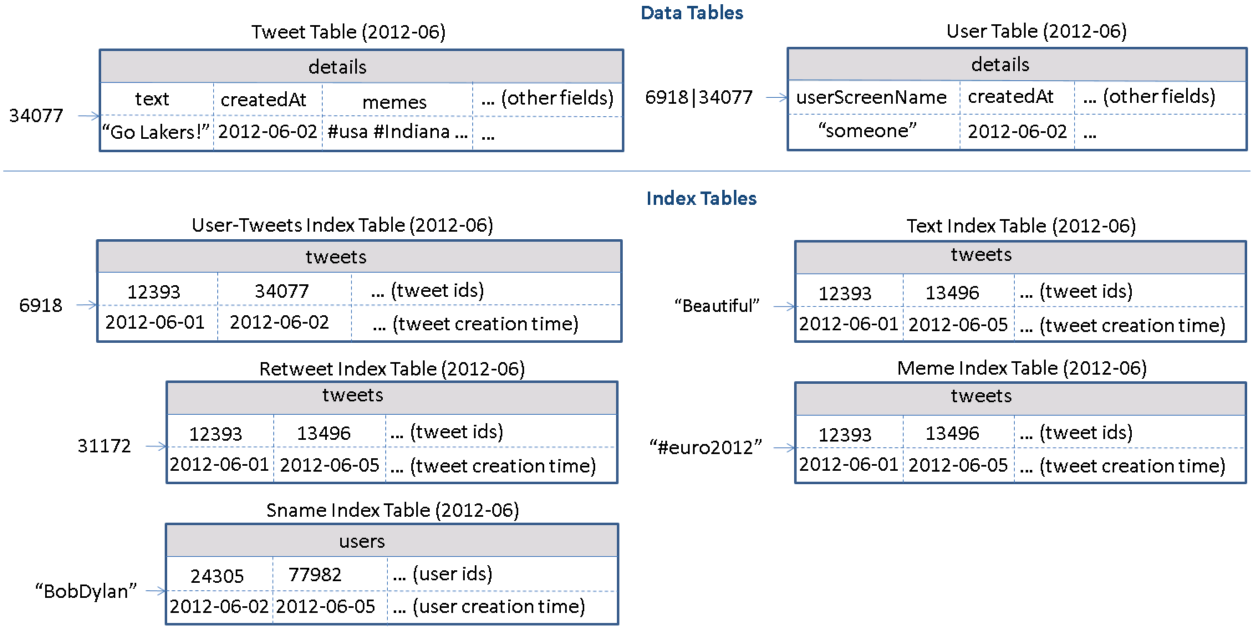
Efficient range scan on entry IDs.	Yes with order preserving hash function and index entry ID as column name.	Doable with a secondary index on an attribute whose value is the object key.	Doable with Index key + entry ID as "_id" of document.

2.5 Performance Evaluation

This section evaluates the effectiveness and efficiency of our customizable indexing framework by measuring its impact on the performance of data loading/indexing and query evaluation. Specifically, by defining customized index structures that eliminate unnecessary information from traditional text indices and embed useful information about the social media data, we expect to receive scalability, faster indexing and data loading speed, as well as better query evaluation performance. In order to verify this, we use real data and queries from Truthy, as described in section 1.2.1. Based on IndexedHBase, we develop parallel data loading and query evaluation strategies and compare their performance against another set of implementations on Riak using its natively supported text indexing techniques, which is based on distributed Solr at the backend [57].

2.5.1 Table Schemas on IndexedHBase

Working off the HBase data model, we design the table schemas in Figure 2-19 for storing the original data from Truthy and necessary indices for query evaluation. Specifically, we maintain the tweet and user information contained in a JSON message (Figure 1-2) in separate tables. To achieve efficient evaluation of the queries listed in Section 1.2.1, we create multiple customized indices with structures similar to Figure 2-15 (b) and Figure 2-15 (d). We split the whole dataset by months, maintaining a separate set of data and index tables for every month. This method of table management actually creates a **hybrid index partition mechanism** that inherits the advantage of both partition by index key and partition by original data. For instance, since the regions of every index table are maintained independently by HBase, index distribution is decoupled from original data distribution. At the same time, for queries with time windows covering multiple months, index access for different months can work in parallel, and the amount of index and original data accessed during query evaluation is limited by the scope of the time window. Another benefit is that the loading of streaming data only changes the tables relative to the current month, and does not interfere with access to all the other tables.



Data Tables

Tweet Table (2012-06)

	details			
	text	createdAt	memes	... (other fields)
34077 →	“Go Lakers!”	2012-06-02	#usa #Indiana ...	...

User Table (2012-06)

	details		
	userScreenName	createdAt	... (other fields)
6918|34077 →	“someone”	2012-06-02	...

Index Tables

User-Tweets Index Table (2012-06)

	tweets		
6918 →	12393	34077	... (tweet ids)
	2012-06-01	2012-06-02	... (tweet creation time)

Text Index Table (2012-06)

	tweets		
“Beautiful” →	12393	13496	... (tweet ids)
	2012-06-01	2012-06-05	... (tweet creation time)

Retweet Index Table (2012-06)

	tweets		
31172 →	12393	13496	... (tweet ids)
	2012-06-01	2012-06-05	... (tweet creation time)

Meme Index Table (2012-06)

	tweets		
“#euro2012” →	12393	13496	... (tweet ids)
	2012-06-01	2012-06-05	... (tweet creation time)

Sname Index Table (2012-06)

	users		
“BobDylan” →	24305	77982	... (user ids)
	2012-06-02	2012-06-05	... (user creation time)

Figure 2-19. Table schemas used in IndexedHBase for data from Truthy

Some details need to be clarified before proceeding further. Each table contains only one column family, e.g. “details” or “tweets”. The user table employs a concatenation of user ID and tweet ID as the row key, because analysis benefits from tracking changes in a tweet’s user metadata. For example, a user can change profile information, which can give insights into their behavior. A separate meme index table is created for indexing the *hashtags*, *user-mentions*, and *URLs* contained in tweets. This is because some special cases, such as expandable URLs, cannot be handled properly by the text index. The memes are used as row keys, each followed by a different number of columns, named after the IDs of tweets containing the corresponding meme. The timestamp of the cell value marks the tweet creation time.

2.5.2 Data Loading Strategies

We develop parallel loading strategies for both streaming data and historical data. Figure 2-20 shows the architecture of the streaming data loading strategy, where one or more distributed loaders are running concurrently. The stream distributer connects to the external Twitter streaming API [148] and distributes the sequence of social updates among all concurrent loaders. It can be implemented as a simple Storm topology [25] that does data distribution in a random or round-robin fashion. Each loader is assigned a unique ID and works as a client application to the general customizable indexer. Upon receiving a tweet JSON string, the loader first generates records for the tweet table and user table, then loads them into the tables by invoking the *insert()* method of the general customizable indexer, which will complete online indexing and update all the data tables as well as the relevant index tables.

The historical data loading strategy is implemented as a MapReduce program. One separate job is launched to load the historical files for each month, and multiple jobs can be running simultaneously. A job starts multiple mappers in parallel, each responsible for loading one file. At running time, every line in the .json.gz file is given to the mapper as one input, which contains the string of one tweet. The mapper first creates records for the tweet table and user table and then invokes the general customizable indexer to get all the related index table records. All table records are handled by the multi-table output format, which automatically inserts them into the related tables. Finally, if the JSON string contains a “retweeted_status”, the corresponding substring will be extracted and processed in the same way.

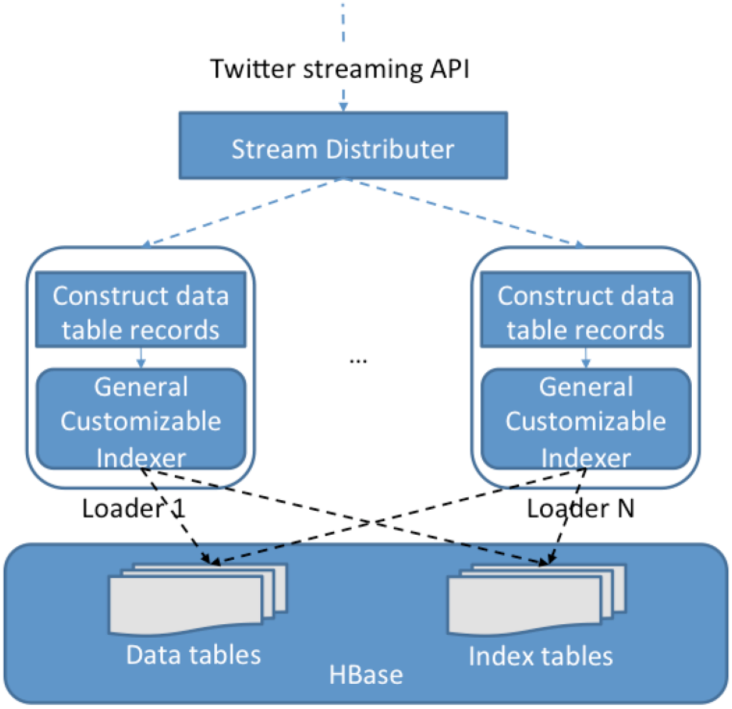

Twitter streaming API
Stream Distributer
Construct data table records
General Customizable Indexer
...
Construct data table records
General Customizable Indexer
Loader 1
Loader N
Data tables
HBase
Index tables

Figure 2-20. Streaming data loading strategy

2.5.3 Parallel Query Evaluation Strategy

We develop a two-phase parallel query evaluation strategy viewable in Figure 2-21. For any given query, the first phase uses multiple threads to find the IDs of all related tweets from the index tables and saves them in a series of files containing a fixed number (e.g., 30,000) of tweet IDs. The second phase launches a MapReduce job to process the tweets in parallel and extract the necessary information to complete the query. This means to evaluate *user-post-count*, each mapper in the job will access the tweet table to figure out the user ID corresponding to a particular tweet ID, count the number of tweets by each user, and output all counts when it finishes. The output of all the mappers will be processed to finally generate the total tweet count of each user ID.

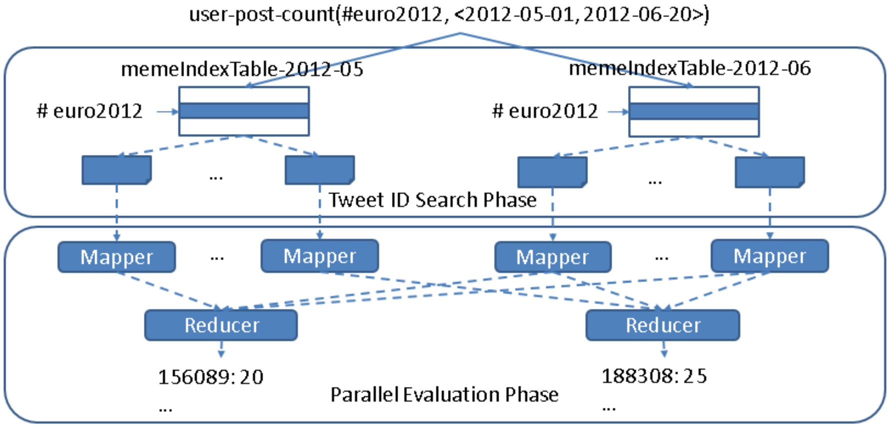


Figure 2-21. Two-phase parallel evaluation process for an example user-post-count query

Two aspects of the query evaluation strategy deserve further discussion. First, as described in Section 1.2.1, prefix queries can be constructed by using parameters such as "#occupy*". We provide two index operators for getting the related tweet IDs in the first phase. One is simply to complete a sequential range scan of rows in the corresponding index tables. The other uses a MapReduce program to complete parallel scans over the range of rows. The latter option is only faster for parameters covering a large range spanning multiple regions of the index table.

Second, the number of tweet IDs in each file implies a tradeoff between parallelism and scheduling overhead. When this number is set lower, more mappers will be launched in the parallel evaluation phase, which means the amount of work done by a mapper decreases while the total task scheduling overhead increases. The optimal number depends on the total number of related tweets and the amount of resources available in the infrastructure. We set the default value of this number to 30,000 and leave it configurable by the user.

2.5.4 Testing Environment Configuration

We use eight nodes on the Bravo cluster of FutureGrid to complete tests for both IndexedHBase and Riak. The hardware configuration for all eight nodes is listed in Table 2-2. Each node runs

CentOS 6.4 and Java 1.7.0_21. For IndexedHBase, Hadoop 1.0.4 and HBase 0.94.2 are used. One node hosts the HDFS headnode, Hadoop jobtracker, Zookeeper, and HBase master. The other seven nodes are used to host HDFS datanodes, Hadoop tasktrackers, and HBase region servers. The data replication level is set to two on HDFS. The configuration details of Riak will be given in Section 2.4.5. In addition to Bravo, we also use the Alamo HPC cluster of FutureGrid to test the scalability of the historical data loading strategy of IndexedHBase, since Alamo can provide a larger number of nodes through dynamic HPC jobs. Software configuration of Alamo is mostly the same as Bravo.

Table 2-2. Per-node configuration on Bravo and Alamo Clusters

Cluster	CPU	RAM	Hard Disk	Network
Bravo	8 * 2.40GHz (Intel Xeon E5620)	192G	2T	40Gb InfiniBand
Alamo	8 * 2.66GHz (Intel Xeon X5550)	12G	500G	40Gb InfiniBand

2.5.5 Configuration and Implementation on Riak

As mentioned in Section 2.2.4, Riak provides a “Riak Search” module that can build distributed inverted indices on data objects for full-text search purposes. Users can assign buckets to organize their data objects and configure indexed fields on the bucket level. Beyond the basic inverted index structure, Riak supports a special feature called “inline fields.” If a field is specified as an “inline” field, its value will be attached to the document IDs in the posting lists, as illustrated in Figure 2-22.

Similar to our customized index tables in IndexedHBase, inline fields can be used to carry out an extra filtering operation to speed up queries involving multiple fields. However, they are different in two basic aspects. First, inline fields are an extension of traditional inverted indices, which means overhead such as frequency information and document scoring still exist in Riak

Search. Second, customizable index structures are totally flexible in the sense that the structure of each index can be independently defined to contain any subset of fields from the original data. In contrast, if one field is defined as an inline field on Riak, its value will be attached to the posting lists of the indices of all indexed fields, regardless of whether it is useful. As an example, the “Sname index table” in Figure 2-19 uses the creation time of user accounts as timestamps, while the “meme index table” uses creation time of tweets. Such flexibility is not achievable on Riak.

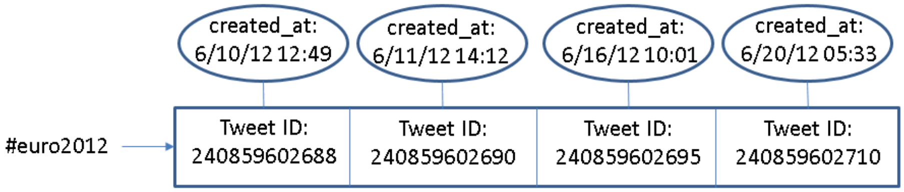
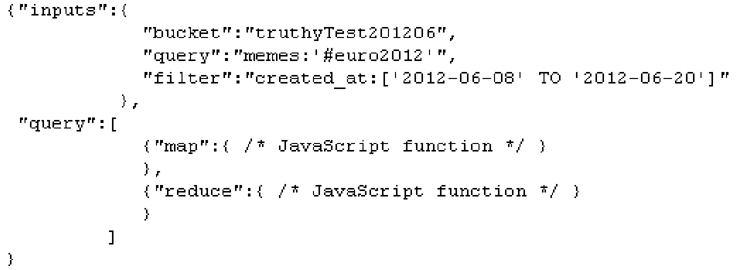

created_at: 6/10/12 12:49
created_at: 6/11/12 14:12
created_at: 6/16/12 10:01
created_at: 6/20/12 05:33
#euro2012
Tweet ID: 240859602688
Tweet ID: 240859602690
Tweet ID: 240859602695
Tweet ID: 240859602710

Figure 2-22. An example of inline field (*created_at*) in Riak

In our tests, all eight nodes of Bravo are used to construct a Riak ring. The nodes run Riak 1.2.1, using LevelDB as the storage backend. We create two different buckets to index data with different search schemas. The data replication level is set to two on both buckets. The tweet ID and JSON string of each tweet are directly stored into <key, value> pairs. The original JSON string is extended with an extra “memes” field, which contains all the hashtags, user-mentions, and URLs in the tweet, separated tab characters. Riak Search is enabled on both buckets, and the *user_id*, *memes*, *text*, *retweeted_status_id*, *user_screen_name*, and *created_at* fields are indexed. Specifically, *created_at* is defined as a separate indexed field on one bucket, and an “inline only” field on the other bucket, meaning that it does not have a separate index but is stored together with the indices of other fields.

Riak provides a lightweight MapReduce framework for users to query the data by defining MapReduce functions in JavaScript. Additionally Riak supports MapReduce over the results of

Riak Search. We use this feature to implement queries, and Figure 2-23 shows an example implementation. When this query is submitted, Riak will first use the index on “memes” to find related tweet objects (as specified in the “input” field), then apply the map and reduce functions to these tweets (as defined in the “query” field) to get the final result.



```
{"inputs":{
            "bucket":"truthyTest201206",
            "query":"memes:'#euro2012'",
            "filter":"created_at:['2012-06-08' TO '2012-06-20']"
          },
 "query":[
            {"map":{ /* JavaScript function */ }
            },
            {"reduce":{ /* JavaScript function */ }
            }
         ]
}
```

Figure 2-23. An example query implementation on Riak

2.5.6 Data Loading Performance

Historical Data Loading Performance

We use all the .json.gz files from June 2012 to test the historical data loading performance of IndexedHBase and Riak. The total data size is 352GB. With IndexedHBase, a MapReduce job is launched for historical data loading, with each mapper processing one file. With Riak, all 30 files are distributed among eight nodes of the cluster, so every node ends up with three or four files. Then an equal number of threads per node were created to load all the files concurrently to the bucket where “created_at” is configured as an inline field. Threads continue reading the next tweet, apply preprocessing with the “created_at” and “memes” field, and finally send the tweet to the Riak server for indexing and insertion.

Table 2-3 summarizes the data loading time and loaded data size on both platforms. We can see that IndexedHBase is over six times faster than Riak in loading historical data and uses significantly less disk space for storage. Considering the original file size of 352GB and a replication level of two, the storage space overhead for index data on IndexedHBase is moderate.

Table 2-3. Historical data loading performance comparison

	Loading time (hours)	Loaded total data size (GB)	Loaded original data size (GB)	Loaded index data size (GB)
Riak	294.11	3258	2591	667
IndexedHBase	45.47	1167	955	212
Riak / IndexedHBase	6.47	2.79	2.71	3.15

We analyze these performance measurements below. By storing data with tables, IndexedHBase applies a certain degree of data model normalization, and thus avoids storing some redundant data. For example, many tweets in the original .json.gz files contain retweeted status, and many of them are retweeted multiple times. With IndexedHBase, even if a tweet is retweeted repeatedly, only one record is kept for it in the tweet table. As for Riak, such a "popular" tweet will be stored within the JSON string of every corresponding retweet. The difference in loaded index data size clearly demonstrates the advantage of a fully customizable indexing framework. By avoiding frequency and position information and only incorporating useful fields in the index tables, IndexedHBase saves 455GB of disk space in storing index data, which is more than 1/3 of the total loaded data size of 1167GB. Also note that IndexedHBase compresses table data using Gzip, which generally provides a better compression ratio than Snappy on Riak.

The difference in loaded data size only explains part of the improvement in total loading time. Two other reasons are:

(1) The loaders of IndexedHBase are responsible for generating both data tables and index tables. Therefore, the JSON string of each tweet is parsed only once when it is read from

the .json.gz files and converted to table records. By contrast, Riak uses servers for its indexing so each JSON string is actually parsed twice – first by the loaders for preprocessing, and again by the server for indexing;

(2) When building inverted indices, Riak not only uses more space to store the frequency and position information, but also spends more time collecting it.

Scalable Historical Data Loading on IndexedHBase

We test the scalability of historical data loading on IndexedHBase with the Alamo cluster of FutureGrid. In this test we take a dataset for two months, May and June 2012, and measure the total loading time with different cluster sizes. The results are illustrated in Figure 2-24. When the cluster size is doubled from 16 to 32 data nodes, the total loading time drops from 142.72 hours to 93.22 hours, which implies a sub-linear scalability due to concurrent access from mappers of the loading jobs to HBase region servers. Nonetheless, these results clearly demonstrate that we get more system throughput and faster data loading speed by adding more nodes to the cluster.

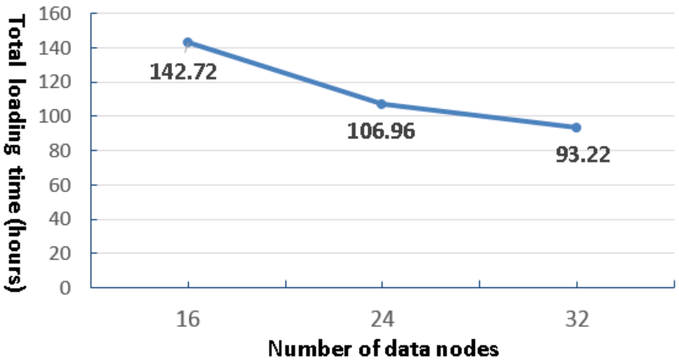


Figure 2-24. Historical data loading scalability to cluster size

Streaming Data Loading Performance on IndexedHBase

The purpose of streaming data loading tests is to verify that IndexedHBase can provide enough throughput to accommodate the growing data speed of the Twitter streaming API. To test the performance of IndexedHBase for handling potential data rates even faster than the current streams, we designed a simulation test using a recent .json.gz file from July 3, 2013. We varied the number of distributed streaming loaders and tested the corresponding system data loading speed. For each case, the whole file was evenly split into the same number of fragments as the loaders and then distributed across all the nodes. One loader was started to process each fragment. The loader reads data from the stream of the local file fragment rather than from the Twitter streaming API. So this test measures how the system performs when each loader gets an extremely high data rate that is equal to local disk I/O speed.

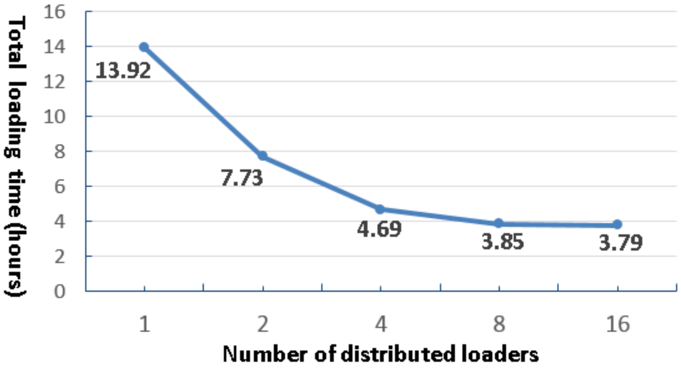


Figure 2-25. Results for streaming data loading test

Figure 2-25 shows the total loading time when the number of distributed loaders increases by powers of two from one to 16. Once again, concurrent access to HBase region servers results in a decrease in speed-up as the number of loaders is doubled each time. The system throughput is almost saturated when we have eight distributed loaders. For the case of eight loaders, it takes 3.85 hours to load all 45,753,194 tweets (less than 2.4ms on average to index a tweet), indicating the number of tweets that can be processed per day on eight nodes is about six times the current daily data rate. Therefore, IndexedHBase can easily handle a high-volume stream of social media data. In the case of vastly accelerated data rates, as would be the case for the Twitter firehose (a stream of all public tweets) [147], one could increase the system throughput by adding more nodes.

2.5.7 Query Evaluation Performance

Separate Index Structures vs. Customized Index Structures

One major purpose of using customized index structures is to achieve lower query evaluation complexity compared to traditional inverted indices on separate data fields. To verify this, we use a simple *get-tweets-with-meme* query to compare the performance of IndexedHBase with a solution using separate indices on the fields of memes and tweet creation time, which is implemented through the Riak bucket where "created_at" is defined as a separately indexed field.

In this test we load four days' worth of data to both IndexedHBase and the Riak bucket and measure the query evaluation time with different memes and time windows. For memes, we choose "#usa", "#ff", and "@youtube", each contained in a different subset of tweets. The "#ff" hashtag is a popular meme for "Follow Friday." For each meme, we use three different time windows with a length between one and three hours. Queries in this test only return tweet IDs – they don't launch an extra MapReduce phase to get the content. Figure 2-26 presents the query execution time for each indexing strategy. As shown in the plots, IndexedHBase not only achieves a query evaluation speed that is tens to hundreds of times faster, but also demonstrates a different pattern in query evaluation time. When separate meme index and time index are used, the query evaluation time mainly depends on the length of time window; the meme parameter has little impact. In contrast, using a customized meme index, the query evaluation time mainly depends on the meme parameter. For the same meme, the evaluation time only increases marginally as the time window gets longer. These observations confirm our theoretical analysis in Section 1.3.1.

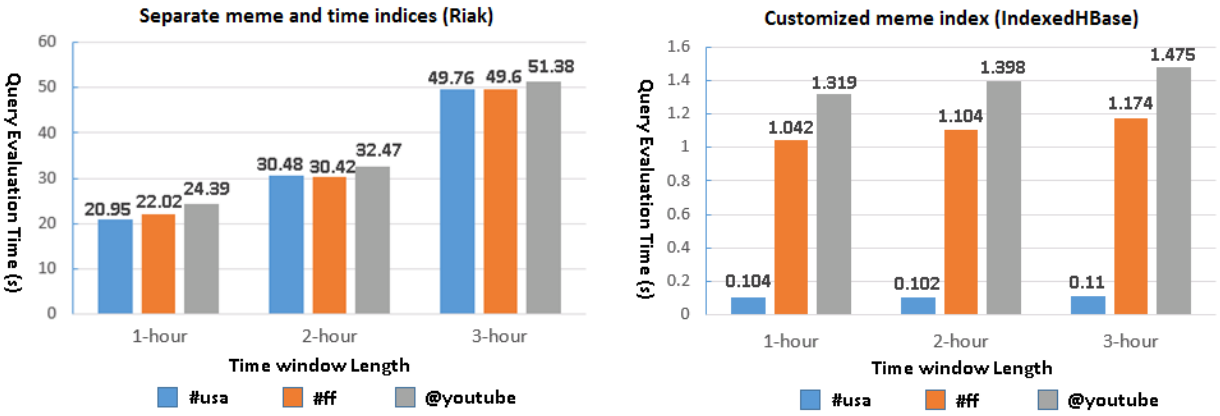


Figure 2-26. Query evaluation time: separate meme and time indices vs. customized index

Query Evaluation Performance Comparison

This set of tests is designed to compare the performance of Riak and IndexedHBase for evaluating queries involving different numbers of tweets and different result sizes. Since using separate indices has proven inefficient on Riak, we choose to test the query implementation using “created_at” as an inline field. Queries are executed on both platforms against the data loaded in the historical data loading tests. For query parameters, we choose the popular meme “#euro2012,” along with a time window with a length varied from three hours to 16 days. The start point of the time window is fixed at 2012-06-08T00:00:00, and the end point correspondingly varies exponentially from 2012-06-08T02:59:59 to 2012-06-23T23:59:59. This covers a major part of the 2012 UEFA European Football Championship. The queries can be grouped into three categories based on the manner in which they are evaluated on Riak and IndexedHBase.

(1) No MapReduce on either Riak or IndexedHBase

The *meme-post-count* query falls into this category. On IndexedHBase, query evaluation is done by simply going through the rows in meme index tables for each given meme and counting the number of qualified tweet IDs. In the case of Riak, since there is no way to directly access the index data, this is accomplished by issuing an HTTP query for each meme to fetch the “id” field of matched tweets. Figure 2-27 shows the query evaluation time on Riak and IndexedHBase. As the time window gets longer, the time increases for both. However, the absolute evaluation time is much shorter for IndexedHBase because Riak has to spend extra time to retrieve the “id” field.

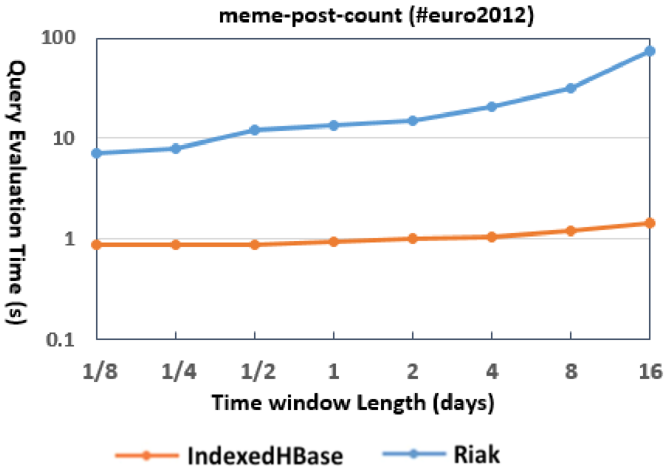


Figure 2-27. Query evaluation time for *meme-post-count*

(2) No MapReduce on IndexedHBase; MapReduce on Riak

The *timestamp-count* query belongs to this category. Inferring from the schema of the meme index table, this query can also be evaluated by only accessing the index data on IndexedHBase. On Riak it is implemented with MapReduce over Riak search results, where the MapReduce phase completes the timestamp counting based on the content of the related tweets. Figure 2-28 shows the query evaluation time on both platforms. Since IndexedHBase does not need to analyze the content of the tweets at all, its query evaluation speed is orders of magnitude faster than Riak.

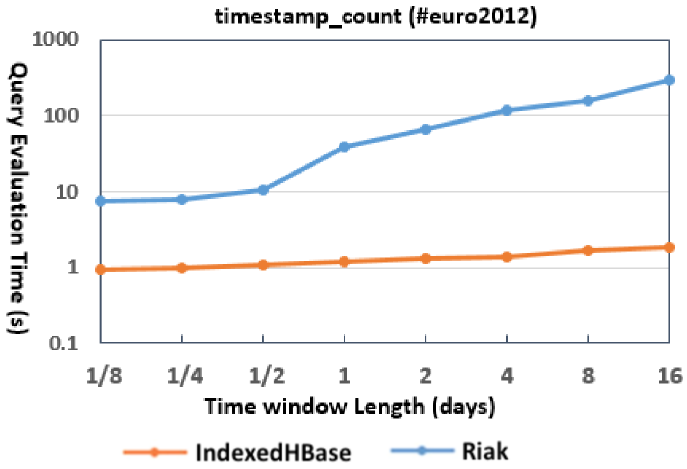


Figure 2-28. Query evaluation time for *timestamp-count*

(3) MapReduce on both Riak and IndexedHBase

Most queries require a MapReduce phase on both Riak and IndexedHBase. Figure 2-29 shows the evaluation time for several of them. An obvious trend is that Riak is faster on queries involving a smaller number of related tweets, but IndexedHBase is significantly faster on queries involving a larger number of related tweets and results. Figure 2-30 lists the results sizes for two of the queries. The other queries have a similar pattern.

The main reason for the observed performance difference is the characteristics of the MapReduce framework on these two platforms. IndexedHBase relies on Hadoop MapReduce, which is designed for fault tolerant parallel processing of large batches of data. It implements the full semantics of the MapReduce computing model and applies a comprehensive initialization process for setting up the runtime environment on the worker nodes. Hadoop MapReduce uses disks on worker nodes to save intermediate data and does grouping and sorting before passing them to reducers. A job can be configured to use zero or multiple reducers. Since most social media queries use time windows at the level of weeks or months, IndexedHBase can handle these long time period queries well.

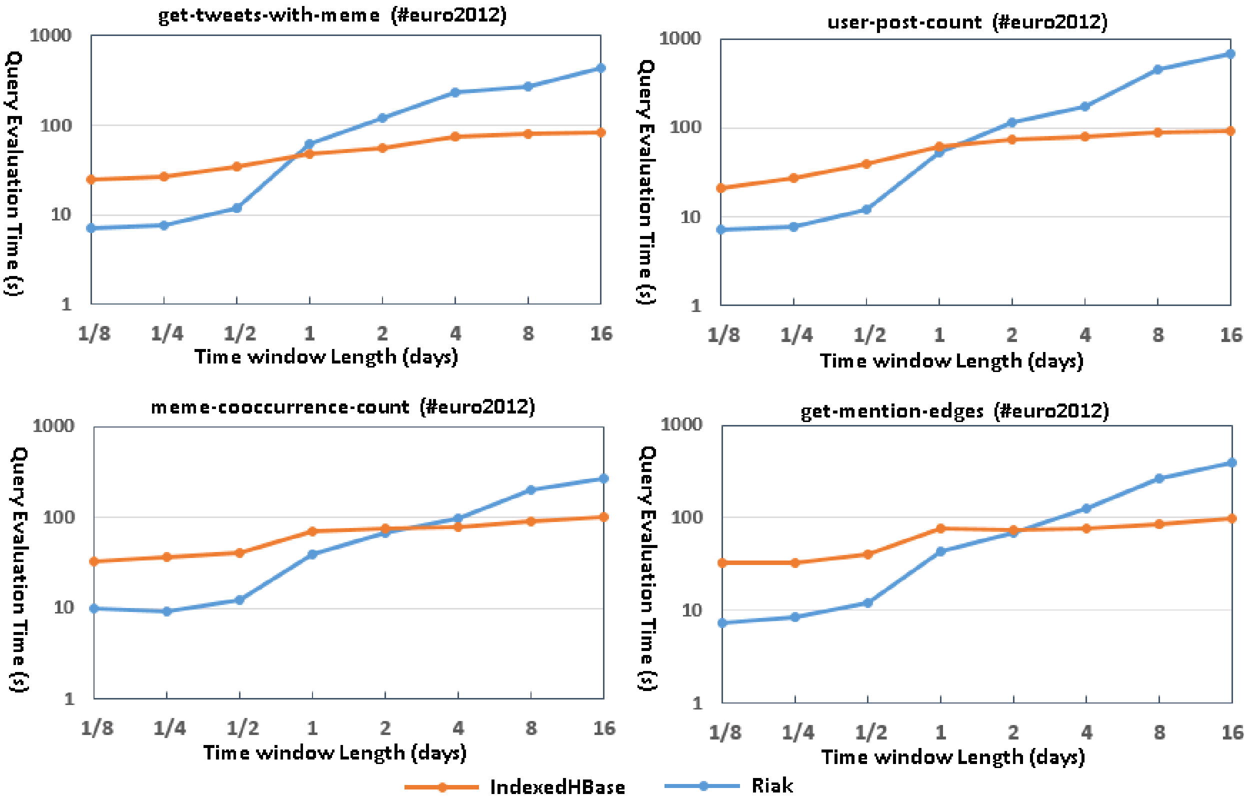


Figure 2-29. Query evaluation time for queries requiring MapReduce on both platforms

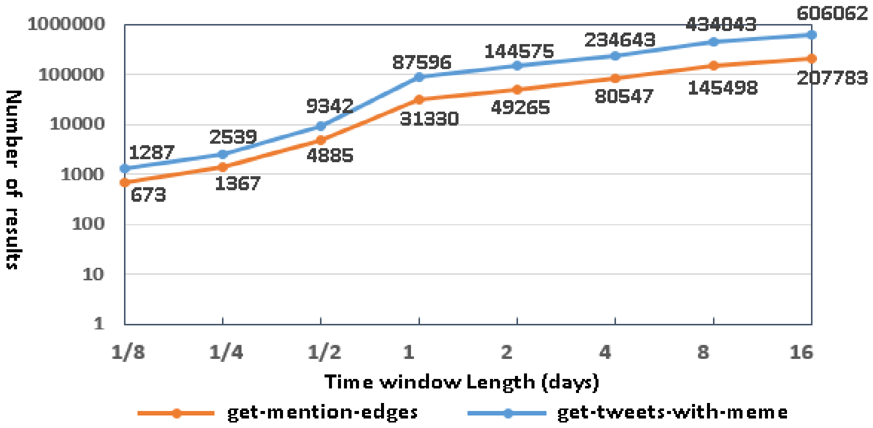


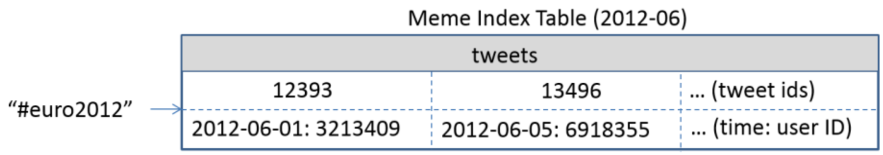
Figure 2-30. Result sizes for *get-tweets-with-meme* and *get-mention-edges*

The MapReduce framework on Riak, on the other hand, is designed for lightweight use cases where users can write simple query logic with JavaScript and get it running on the data nodes quickly without a complicated initialization process. There is always only one reducer running for each MapReduce job. Intermediate data is transmitted directly from mappers to the reducer without being sorted or grouped. The reducer relies on its memory stack to store the whole list of intermediate data, and has a default timeout of only five seconds. Therefore, Riak MapReduce is not suitable for processing the large datasets produced by queries corresponding to long time periods.

Improving Query Evaluation Performance with Modified Index Structures

IndexedHBase accepts dynamic changes to the index structures for efficient query evaluation. To verify this, we extend the meme index table to also include user IDs of tweets in the cell values, as illustrated in Figure 2-31. Using this new index structure, IndexedHBase is able to evaluate the *user-post-count* query by only accessing index data.

We use the batch indexing mechanism of IndexedHBase to rebuild the meme index table, which takes 3.89 hours. The table size increases from 14.23GB to 18.13GB, which is 27.4% larger. Figure 2-32 illustrates the query evaluation time comparison. The query with the new index structure is faster by more than an order of magnitude. In cases where *user-post-count* is frequently used, the query speed improvement is clearly worth the additional storage required. As will be demonstrated in Section 3.2, the extended meme index structure is also useful for analysis tasks.



Meme Index Table (2012-06)

	tweets		
"#euro2012"	12393	13496	... (tweet ids)
	2012-06-01: 3213409	2012-06-05: 6918355	... (time: user ID)

Figure 2-31. Extended meme index including user ID information

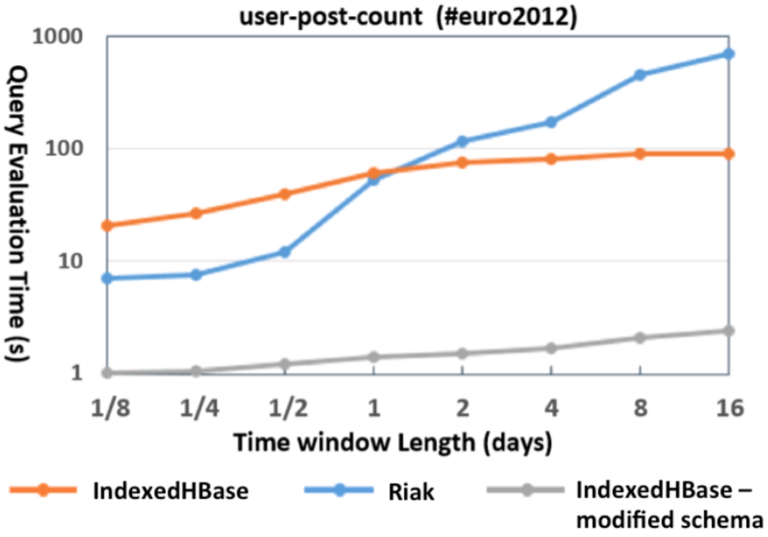


Figure 2-32. Query evaluation time modified meme index structure

2.6 Conclusions

This chapter presents and evaluates the storage layer of our scalable architecture for social media data analysis. In particular, we leverage the HBase system as the storage substrate, and extend it with a customizable indexing framework to support novel text index structures for handling the special queries of social media data. To the best of our knowledge, IndexedHBase is a first in developing a fully customizable indexing framework on a distributed NoSQL database. Performance evaluation with real data and queries from Truthy demonstrates that data loading and query evaluation strategies based on our customized index structures are significantly more efficient than implementations using current state-of-the-art distributed text indexing techniques. Our experimentation with IndexedHBase leads to serveral interesting conclusions of general significance.

First of all, parallelization and indexing are key factors in addressing the challenges brought by the sheer data size and special queries of social media data analysis. In particular, parallelization should be explored through every stage of data processing, including loading, indexing, and query evaluation. Also index structures should be flexible and customizable, rather than static, to effectively take advantage of the special characteristics of the data and achieve the best query evaluation performance at the cost of less storage and computation overhead. In order to achieve this, a general customizable indexing framework is necessary. Finally, to deal with the large size of intermediate data and results involved in the query evaluation process, complete and reliable parallel processing frameworks such as Hadoop MapReduce are needed. Lightweight frameworks like Riak MapReduce are not capable of handling queries involving analysis of large datasets.

Chapter 3

Batch Analysis Module – an Integrated Analysis Stack based on YARN

3.1 Overview

As discussed in Section 1.3.2, a social media data analysis workflow usually consists of multiple stages, and each stage may apply a diversity of algorithms that demonstrate different computation and communication patterns. To achieve efficient execution of the whole integrated workflow, two more issues beyond the queries must be addressed.

First of all, each individual algorithm needs be implemented in an efficient way using a proper processing framework that is good at handling its computation and communication pattern. To illustrate, for algorithms that process small intermediate datasets with a low level of computational complexity, a sequential implementation may be enough. Algorithms that complete a single-pass processing over a large dataset need parallelization through a framework like Hadoop MapReduce [18]. More sophisticated algorithms that need to carry out iterative computation and collective communication can use an iterative MapReduce framework such as Twister [60] or Spark [166]. Finally, for algorithms designed to process high-throughput streaming data, stream processing frameworks such as Storm [25] are the most suitable for the parallelization.

Moreover, the analysis architecture must be able to dynamically switch to different processing frameworks to execute different analysis algorithms and finish the end-to-end analysis workflow.

Targeting these issues, we extend IndexedHBase to an integrated analysis architecture (Figure 1-8) based on YARN [154], which is designed for accommodating tasks from various processing frameworks in a distributed environment with shared computing and storage resources. This chapter describes the batch analysis module of this architecture, and Figure 3-1 illustrates the internal interactions between the components in the batch analysis module and the storage layer.

The user application can define an analysis workflow in the form of a workflow driver script. This script invokes the *query-and-analyze* interface of the query and analysis engine to execute queries and analysis tasks. The engine converts these requests into jobs on different parallel processing frameworks, and dynamically employs different frameworks to complete the queries and analysis algorithms. During runtime, analysis algorithms may use either the data table records selected by the queries or the index table records as input.

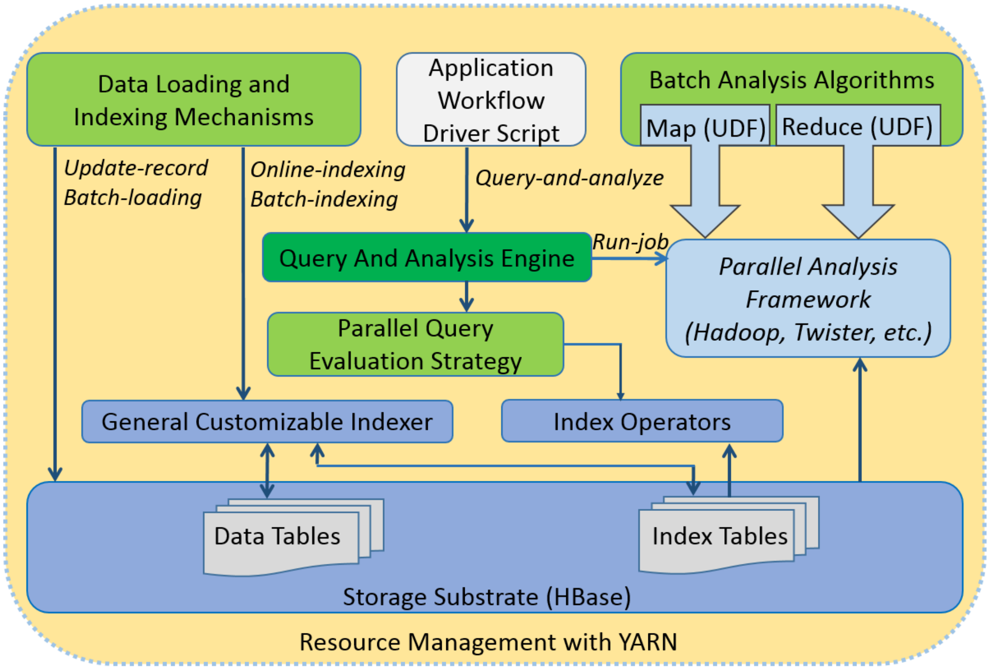

Data Loading and Indexing Mechanisms
Application Workflow Driver Script
Batch Analysis Algorithms
Map (UDF)
Reduce (UDF)
Update-record
Batch-loading
Online-indexing
Batch-indexing
Query-and-analyze
Query And Analysis Engine
Run-job
Parallel Analysis Framework (Hadoop, Twister, etc.)
Parallel Query Evaluation Strategy
General Customizable Indexer
Index Operators
Data Tables
Index Tables
Storage Substrate (HBase)
Resource Management with YARN

Figure 3-1. Internal interaction between batch analysis module and storage layer

Based on this architecture, we develop the following set of analysis algorithms that are generally useful in the analysis workflows of many research scenarios:

A **related hashtag mining** algorithm using Hadoop MapReduce. For a given *seed hashtag* (e.g. #ncaa), it finds all *related hashtags* that co-occur frequently with the seed hashtag during a specific time window. This algorithm is useful in all scenarios where the social event of concern can be identified by a set of related hashtags. It mainly relies on index tables to do mining, and only accesses a limited number of data table records according to the seed hashtag.

A **meme daily frequency generation** algorithm using Hadoop MapReduce. Given a time window, this algorithm generates the daily frequencies of all hashtags during that time. It is useful for many research purposes, such as generation of meme evolution timelines [41] and analysis of meme lifetime distribution [159]. It completely relies on parallel scans of index tables.

A **domain name entropy computation** algorithm using Hadoop MapReduce. Given a time window, this algorithm collects the URLs posted by each user during that time, generates the distribution of domain names in these URLs for each user, and computes the entropy of the distribution. This algorithm is useful for projects related to analysis of user interest allocation or comparison between social networks and search engines.

A **graph layout** algorithm (known as "Fruchterman-Reingold algorithm") using the Twister iterative MapReduce framework. Given a graph in the form of a set of nodes and edges, this algorithm generates a nice layout of all the nodes on a canvas, so that nodes connected with edges are positioned close to each other, and non-connected nodes tend to be apart. This algorithm is useful in many workflows involving visualization of social network structures, such as the one presented in [44]. Since it is computation intensive, a well-parallized implementation can achieve near-linear scalability.

A summary of these algorithms is given in Table 3-1. In this chapter, we describe the implementation of these algorithms and analyze their performance by comparing them to their sequential or raw data scanning counterparts. In addition we use several of them to reproduce a workflow from a previous publication about political polarization [44] and demonstrate efficient execution of the whole workflow.

Table 3-1. Summary of analysis algorithms

Algorithm	Key feature	Time complexity
Related hashtag mining	Mostly relies on index; only accesses a small portion of original data.	O(H*M + N). M is the number of tweets containing the seed hashtag in the given time window. H is the toal number of co-occuring hashtags. N is the total number of index entries associated with the co-occuring hashtags.
Meme daily frequency generation	Totally based on parallel scan of customized index.	O(N). N is the total number of index entries associated with all the hashtags in the given time window.
Domain name entropy computation	Totally based on parallel scan of customized index.	O(N). N is the total number of index entries associated with all the URLs in the given time window.
Graph layout	First parallel implementation on iterative MapReduce; near-linear scalability.	O(M*N^2). M is the number of iterations. N is the number of vertices in the graph.

3.2 Analysis Algorithms

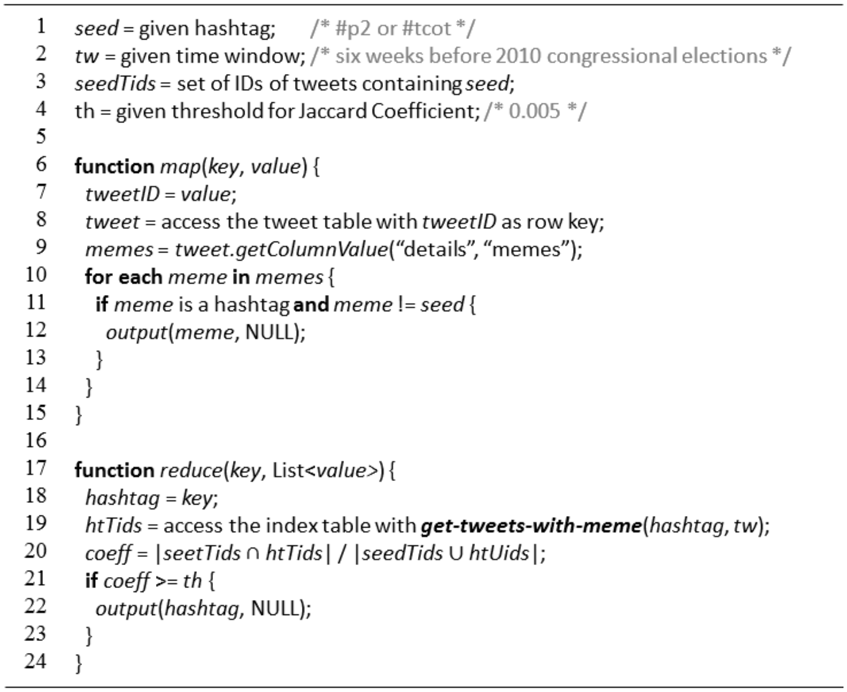
3.2.1 Related Hashtag Mining

Given a seed hashtag and a time window, the **related hashtag mining** algorithm finds all the other hashtags related to the seed by using the Jaccard coefficient. For a seed hashtag *s* and a target hashtag *t*, the Jaccard coefficient between *s* and *t* is defined as:

$$\sigma(S,T) = \frac{|S \cap T|}{|S \cup T|} \qquad (3)$$

Here S is the set of social updates containing s, and T is the set of social updates containing t. When this coefficient is large enough, the two hashtags are recognized as related.

We implement this algorithm as a *query-and-analyze* process. An index operator is first applied against the meme index table (Figure 2-19) to find the IDs of all the tweets containing the seed hashtag *s*. The query and analysis engine will automatically split these tweet IDs into multiple partitions. A Hadoop MapReduce job is then scheduled to process all the partitions in parallel (Figure 3-2). Each mapper processes one partition, and for every tweet ID therein, the mapper will access the corresponding row in the tweet table and output all the hashtags that co-occur with *s* as intermediate results. After the shuffling phase, each reducer will receive a list of unique target hashtags. For every target hashtag *t* in the list, the reducer again uses an index operator against the meme index table to find the corresponding set of tweet IDs, *T*. Then the Jaccard coefficient between *s* and *t* is calculated according to equation (3); if the value reaches a given threshold, *t* will be output as a final result.



```
seed = given hashtag;      /* #p2 or #tcot */
tw = given time window; /* six weeks before 2010 congressional elections */
seedTids = set of IDs of tweets containing seed;
th = given threshold for Jaccard Coefficient; /* 0.005 */

function map(key, value) {
  tweetID = value;
  tweet = access the tweet table with tweetID as row key;
  memes = tweet.getColumnValue("details", "memes");
  for each meme in memes {
    if meme is a hashtag and meme != seed {
      output(meme, NULL);
    }
  }
}

function reduce(key, List<value>) {
  hashtag = key;
  htTids = access the index table with get-tweets-with-meme(hashtag, tw);
  coeff = |seetTids ∩ htTids| / |seedTids ∪ htUids|;
  if coeff >= th {
    output(hashtag, NULL);
  }
}
```

Figure 3-2. MapReduce algorithm for mining related hashtags

Assuming the number of tweets containing the seed hashtag is M, the initial step will use the index operator to retrieve M entries from the meme index table. The map phase of the MapReduce job retrieves and analyzes M tweets from the tweet table. Assuming the total number of hashtags that co-occur with the seed hashtag in any tweet (and thus will have a Jaccard Coefficient larger than 0) is H, then the map phase will output H **candidate hashtags**. We denote these candidate hashtags as h_1, h_2, … h_H, and the set of index entries associated with each of them as T_1, T_2, … T_H. Suppose $|T_1| + |T_2| + \ldots + |T_H| = N$, so the reduce phase will retrieve H index entries from the index table. For each candidate h_i, a merge-based algorithm can be used to calculate the insersection and union of S and T_i. So the time spent on computing the Jaccard Coefficient is $(M + |T_1|) + (M + |T_2|) + \ldots + (M + T_H) = H*M + N$. Therefore, the time complexity of the whole algorithm is $O(M) + O(H*M + N) = O(H*M + N)$.

3.2.2 Meme Daily Frequency Generation

Given a time window, the **meme daily frequency generation algorithm** generates the daily frequencies of all hashtags during that time. This algorithm can be used in many research projects such as generation of meme evolution timelines [41] and analysis of meme lifetime and popularity distribution [159]. Figure 3-3 shows an example of meme timelines available on the website of Truthy [145]. Considering the schema of meme index table in Figure 2-19, it is obvious that this can be done by solely scanning the index without touching any original data. The algorithm is implemented as a Hadoop MapReduce program illustrated in Figure 3-4. Each mapper takes one region of the meme index table as input, and generates the daily frequencies for each hashtag by going through the corresponding row and doing simple counting.

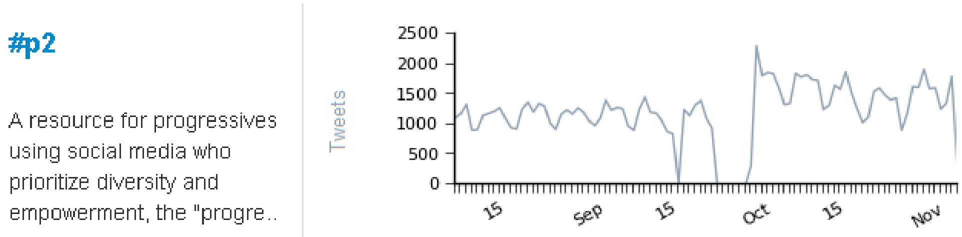


Figure 3-3. An example meme evolution timeline on the Truthy website [145]

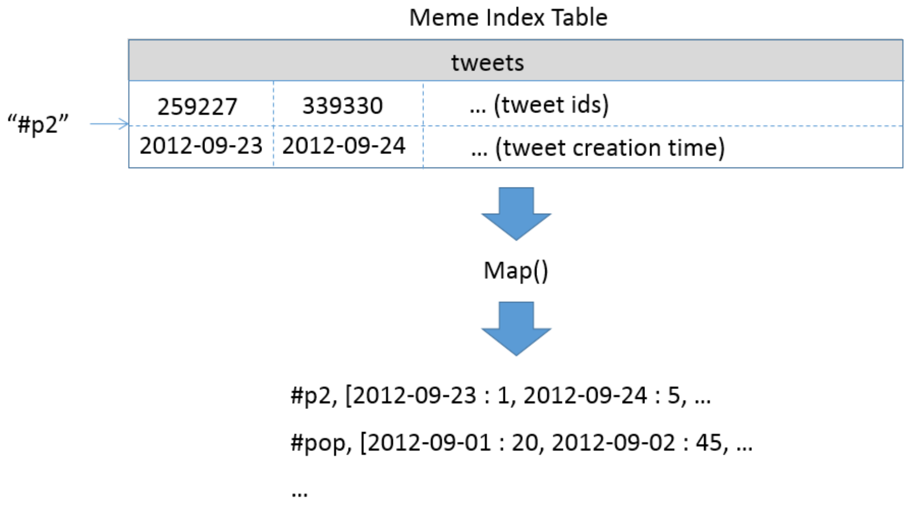

Meme Index Table
tweets
"#p2"
259227
339330
... (tweet ids)
2012-09-23
2012-09-24
... (tweet creation time)
Map()
#p2, [2012-09-23 : 1, 2012-09-24 : 5, ...
#pop, [2012-09-01 : 20, 2012-09-02 : 45, ...
...

Figure 3-4. Map-only job for meme daily frequency generation

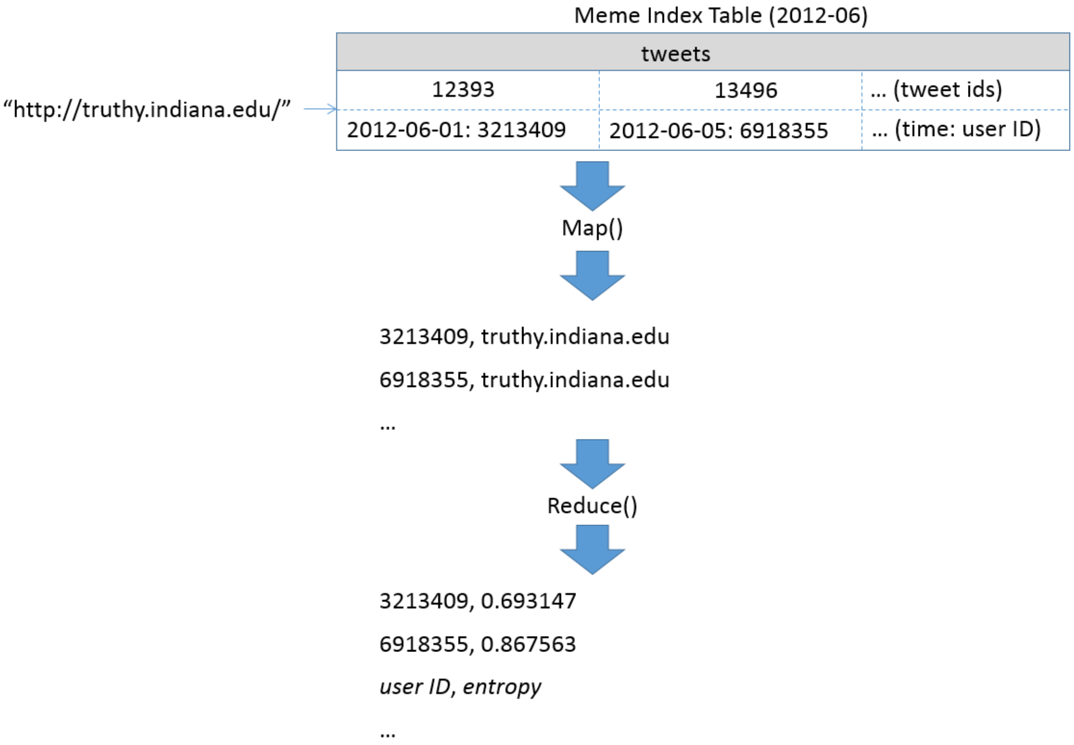
The total amount of data scanned by the mappers is the total number of index entries associated with hashtags in the index tables. It is obvious that the amount of computation spent on generating the results is linear to the number on index entries. So the overall complexity of the algorithm is O(N), assuming N is the total number of index entries scanned.

3.2.3 Domain Name Entropy Computation

For a given time window, the **domain name entropy computation** algorithm collects the URLs posted by all users in their tweets. Then for each user, it extracts the tweeted domain names, generates the probability distribution of these domain names, and computes the entropy value based on the distribution. By analyzing the entropy for a large number of users, it is possible to study users' interest allocation on the social network, and compare the results against search engines to further investigate whether social networks play a special role in shaping users' interest online.

This algorithm can be implemented as a single MapReduce job over extended meme index tables that incorporate user IDs as an entry field, as Figure 3-5 displays. Recall from Chapter 2 that the meme index tables will index hashtags, user-mentions, and URLs contained in the tweets, and we could leverage this extended index for improving the efficiency of queries like *user-post-count*. Here we show its value for supporting analysis tasks. The input to the map phase of the MapReduce job is the range of the index table that covers all the index keys for URLs. The number of mappers launched depends on the number of regions within this range. Index entries are passed to a mapper as a sequence of <*key*, *value*> pairs, where *key* is a row key of the index table (i.e. a URL), and *value* contains a number of index entries associated with the URL. For each entry, the mapper extracts the domain name from the URL and the user ID from the entry field, then emits a pair <*userID*, *domainName*> to the output. The output of all mappers are distributed to multiple reducers, each handling a subset of user IDs. For each user ID, the reducer counts the frequency of each domain name that he/she has tweeted about, generates the distribution, and computes the entropy.

The amount of data scanned by the mappers is the total number of index entries for URLs. Time spent by the mappers on the conversion is linear to the number of input index entries. This means the time spent by reducers for generating the domain name distribution and computing the entropy is linear to the number of <*userID*, *domainName*> pairs, which is the same as the number of index entries. Therefore, the overall complexity of the algorithm is O(N), assuming N is the total number of index entries for URLs.


Meme Index Table (2012-06)

	tweets		
	12393	13496	… (tweet ids)
“http://truthy.indiana.edu/”	2012-06-01: 3213409	2012-06-05: 6918355	… (time: user ID)

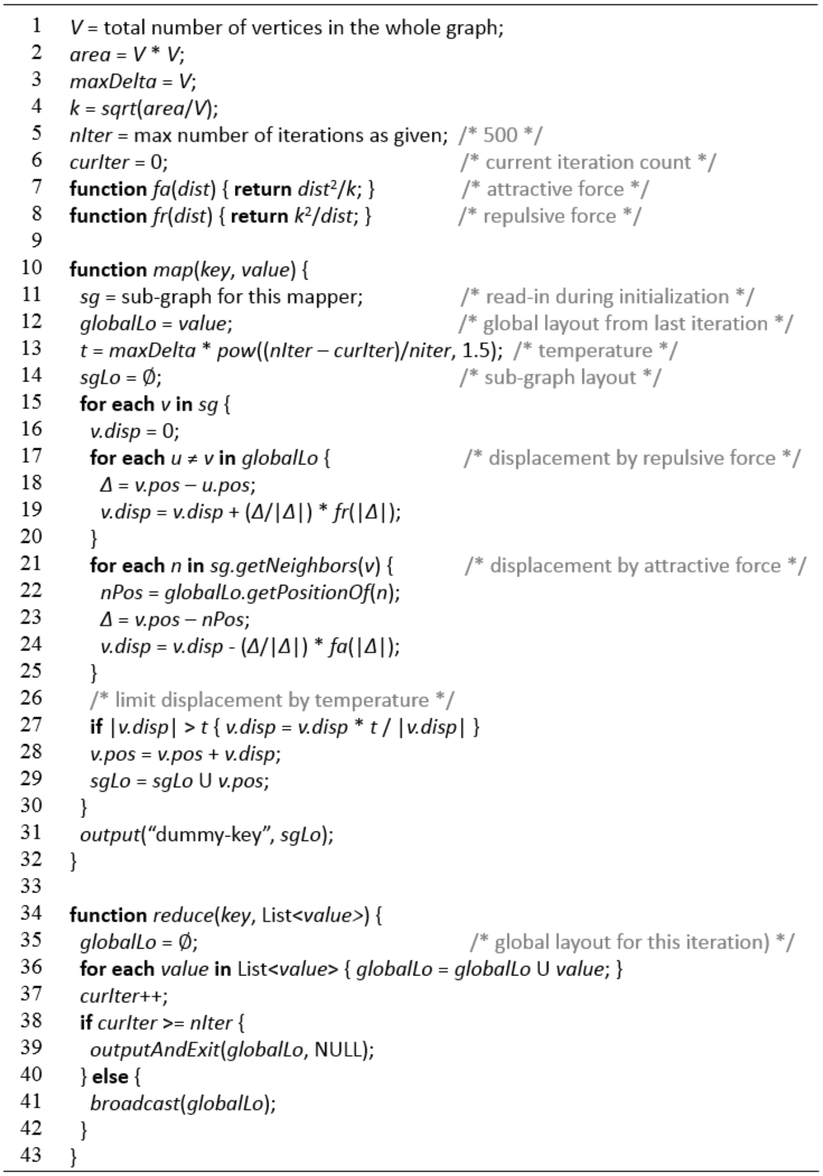
Figure 3-5. MapReduce algorithm for domain name entropy generation

3.2.4 Graph Layout

The **graph layout** algorithm we developed is a parallelization of the “Fruchterman-Reingold” force-directed layout algorithm. The idea of the algorithm is to compute the layout of a graph by simulating the behavior of a physical system where vertices of the graph are taken as atomic particles and edges as springs. A repulsive force exists between each pair of atomic particles, which tends to push them away from each other. An attractive force exists on each spring, pulling the vertices at the two ends closer to each other. Both forces are defined as functions of distances between vertices. Therefore, starting from an initial state of random layout, in each iteration, disconnected vertices are pushed further apart, and vertices connected with edges are pulled closer together. Over multiple iterations, the whole system eventually evolves to a ‘low-energy’ state. Besides the forces, a “temperature” parameter is used to limit the maximum displacement of vertices in each iteration. The temperature eventually ‘cools’ down as iterations proceed.

We implement this algorithm as an iterative MapReduce job on Twister [60], which is specially designed to support large-scale iterative computation. For simplicity, we call this algorithm MRFR (itervative-MapReduce version of Fruchterman-Reingold). The mapper and reducer functions used are given in Figure 3-6. Before the job starts, the graph is partitioned into multiple sub-graphs, each containing a subset of vertices associated with their neighbors. During job initialization time, an initial random layout of the whole graph is broadcasted to all the mappers. Each mapper reads a sub-graph during task initialization time, then saves it in memory for usage across all iterations. Within every iteration, each mapper receives the global layout of the whole graph from the last iteration through its input <*key*, *value*> pair. Then for every vertex in the sub-graph, the mapper first calculates its displacement based on the repulsive forces it receives from every other vertex as well as the attractive forces it receives from its neighbors, and finally decides its total displacement by taking the temperature into consideration. Then a new layout of the sub-graph is generated based on the displacements and output as an intermediate result from the mapper. The reducer collects the output from all mappers to generate the global layout. If the maximum number of iterations is reached, the reducer will output the global layout as the final result. Otherwise, the global layout is broadcasted to all mappers for the next iteration.

Within each iteration, the processing time is dominated by the computation of repulsive forces between each pair of vertices, which is O(N^2), where N denotes the total number of vertices in the graph. Thus the overall complexity of the algorithm is O($M*N^2$), where M is the number of iterations. It takes less than 100 iterations in most cases to generate an elegant layout of the input graph. Since the algorithm is computation intensive, we can achieve near-linear scalability for large graphs, as will be demonstrated in Section 3.2.5.



```
V = total number of vertices in the whole graph;
area = V * V;
maxDelta = V;
k = sqrt(area/V);
nIter = max number of iterations as given;  /* 500 */
curIter = 0;                                /* current iteration count */
function fa(dist) { return $dist^2/k$; }    /* attractive force */
function fr(dist) { return $k^2/dist$; }    /* repulsive force */

function map(key, value) {
  sg = sub-graph for this mapper;           /* read-in during initialization */
  globalLo = value;                         /* global layout from last iteration */
  t = maxDelta * pow((nIter – curIter)/niter, 1.5);  /* temperature */
  sgLo = ∅;                                 /* sub-graph layout */
  for each v in sg {
    v.disp = 0;
    for each u ≠ v in globalLo {            /* displacement by repulsive force */
      Δ = v.pos – u.pos;
      v.disp = v.disp + (Δ/|Δ|) * fr(|Δ|);
    }
    for each n in sg.getNeighbors(v) {      /* displacement by attractive force */
      nPos = globalLo.getPositionOf(n);
      Δ = v.pos – nPos;
      v.disp = v.disp - (Δ/|Δ|) * fa(|Δ|);
    }
    /* limit displacement by temperature */
    if |v.disp| > t { v.disp = v.disp * t / |v.disp| }
    v.pos = v.pos + v.disp;
    sgLo = sgLo U v.pos;
  }
  output("dummy-key", sgLo);
}

function reduce(key, List<value>) {
  globalLo = ∅;                             /* global layout for this iteration) */
  for each value in List<value> { globalLo = globalLo U value; }
  curIter++;
  if curIter >= nIter {
    outputAndExit(globalLo, NULL);
  } else {
    broadcast(globalLo);
  }
}
```

Figure 3-6. Parallel Fruchterman-Reingold algorithm using iterative MapReduce

3.2.5 Performance Analysis

A major advantage of the **related hashtag mining**, **meme daily frequency generation**, and **domain name entropy computation** algorithms is that they mainly rely on indices to finish their computation. Since the size of an index is much smaller than the original data (Table 2-3), these algorithms are significantly more efficient than solutions that process the original data.

To demonstrate this, we compare the efficiency of these algorithms to their counterparts based on parallel scans of the original data, which are implemented as Hadoop MapReduce jobs that directly process the .json.gz files. We call these jobs "Hadoop-FS" versions of implementation.

Specifically, the "Hadoop-FS" version of related hashtag mining is implemented as two consecutive map-only jobs. The first job launches multiple mappers, each processing the .json.gz file for one day. The mapper reads each tweet from the file, and outputs a <*hashtag, tweetID*> for each hashtag contained in the tweet. If the tweet contains the seed hashtag, then all hashtags in this tweet will be written to a file containing the co-occuring hashtags. The second job reads this file and launches multiple mappers to only process the <*hashtag*, *tweetID*> pairs for the co-occuring hashtags, then computes the Jaccard Coefficient for them.

The "Hadoop-FS" job for daily meme frequency generation also launches multiple mappers to process multiple .json.gz files in parallel. Each mapper reads tweets from the file and outputs <*hashtag*, *tweetTimestamp*> pairs for every hashtag contained. The reducers will group the pairs for the same hashtag together and generate the daily frequencies.

For the "Hadoop-FS" job for domain name entropy computation, each mapper also processes one .json.gz file. It reads tweets from the file, then outputs <*userID*, *domainName*> pairs for every URL contained. The reducers will group the pairs for the same user ID together, then generate the domain distribution and compute the entropy.

Figure 3-7 illustrates the performance comparison between the "Hadoop-FS" versions and the versions based on IndexedHBase. All tests are done on a private eight-node cluster called

"Madrid". The hardware configuration of the nodes is listed in Table 3-2. Each node runs RHEL 6.5 and Java 1.7.0_45. For the deployment of YARN and IndexedHBase, Hadoop 2.2.0 and HBase 0.96.0 are used. One node is used as the HDFS name node, YARN resource manager, HBase master, and Zookeeper. The other seven nodes are used as HDFS data nodes and HBase region servers.

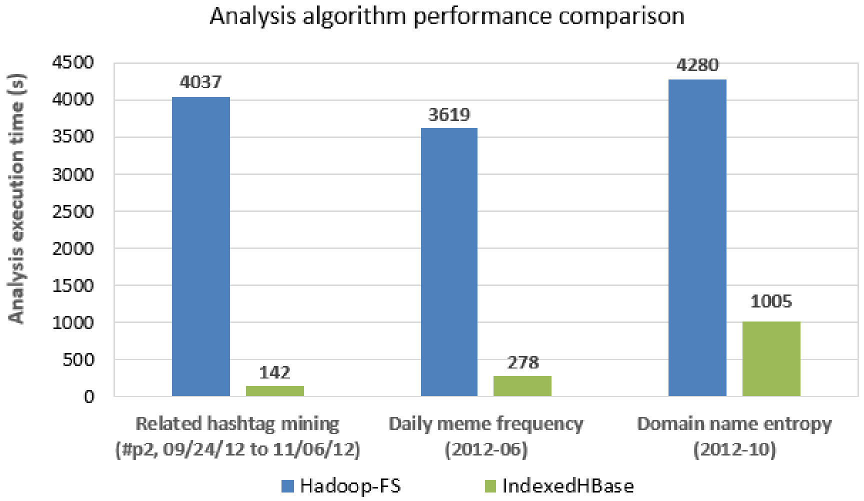


Figure 3-7. Analysis algorithm performance comparison

Table 3-2. Hardware configuration of each node of the Madrid cluster

CPU	RAM	Hard Disk	Network
4 * 4 Quad-Core AMD Opteron 8356 2.3G Hz	48GB	4TB HDD + 1TB SSD	1Gb Ethernet

As shown in Figure 3-7, the algorithms based on IndexedHBase are tens of times faster than the "Hadoop-FS" versions for the case of related hashtag mining and daily meme frequency generation. The processing time for the domain name entropy computation algorithm is longer because 2012-10 has more data and the size of index entries for URLs is larger. Yet it is still four times faster than its "Hadoop-FS" counterpart. Beyond the execution time, these algorithms are also more efficient in terms of resource usage. Each MapReduce job over the index tables launches seven to eight mappers, which equals the number of related regions of the tables. In comparison, the “Hadoop-FS” jobs launch 30 to 31 mappers, because there is one .json.gz file for each day.

We measure the scalability of the graph layout algorithm using 33 nodes of the Alamo HPC cluster on FutureGrid [157]. The per-node hardware configuration is given in Table 1. All nodes are installed with CentOS 5.9 and Java 1.7.0_40. One of the nodes is used to host a Broker of ActiveMQ 5.4.2, and the other 32 nodes run daemons of Twister-Ivy. We take a retweet graph containing 477,111 vertices and 665,599 edges as input, then measure the per-iteration execution time of MRFR using different numbers of mappers. Figure 3-8 illustrates the results. According to Figure 3-6, each iteration is composed of a computation stage (map) and a communication stage (reduce and broadcast). For a graph containing more than 470,000 vertices, the execution time of each iteration is dominated by the map phase that computes the forces between pairs of vertices. Therefore, by parallelizing this intensive computation with multiple mappers, we are able to achieve a near-linear scalability. Details about how the retweet graph was generated will be described in the next section.

Table 3-3. Hardware configuration of each node of the Alamo cluster

CPU	RAM	Hard Disk	Network
8 * 2.66GHz (Intel Xeon X5550)	12GB	500GB	40Gb InfiniBand

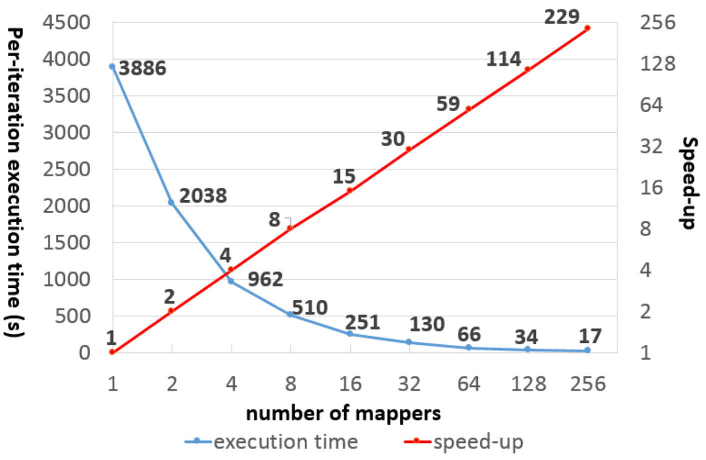


Figure 3-8. Per-iteration execution time and speed-up of MRFR

3.3 Composition and Execution of Analysis Workflows

Using the queries and analysis algorithms based on IndexedHBase, users can compose analysis workflows for various research projects. In this section, we demonstrate the composition and execution of workflows by reproducing the end-to-end analysis presented in a published research project [44] using the dataset of Truthy. The project investigated how social media shapes the networked public sphere and facilitates communication between communities with different political orientations. More than 250,000 politically relevant tweets were extracted from the Truthy dataset during the six weeks leading up to the 2010 U.S. congressional midterm elections. Then the characteristics of the *retweet network* and *mention network* generated from these tweets were examined. The results showed that the retweet network exhibited a highly modular structure, segregating users into two homogenous communities corresponding to the political left and right. In contrast, the mention network did not exhibit such political segregation.

We will first try to reproduce the analysis and results in [44] on Cloud DIKW using the same dataset from 2010, after which we extend the same analysis process to another dataset collected during the six weeks before the 2012 U.S. presidential election to verify if a similar pattern in the social communication networks can be observed. Our explanation in this section focuses on analysis of the retweet network, and implementations for the mention network are similar.

3.3.1 Analysis Workflow for Political Polarization Investigation

Figure 3-9 illustrates the major steps of the analysis workflow in [44]. The first two steps in the workflow try to find a set of political hashtags that can be used to identify politically related tweets from all those collected during the selected six-week time window. In **Step (1)**, two of the most popular political hashtags, #p2 ("Progressives 2.0") and #tcot ("Top Conservatives on Twitter") are manually selected as seed hashtags. **Step (2)** tries to extend this initial set with other related hashtags with the related hashtag mining algorithm, using a threshold of 0.005.

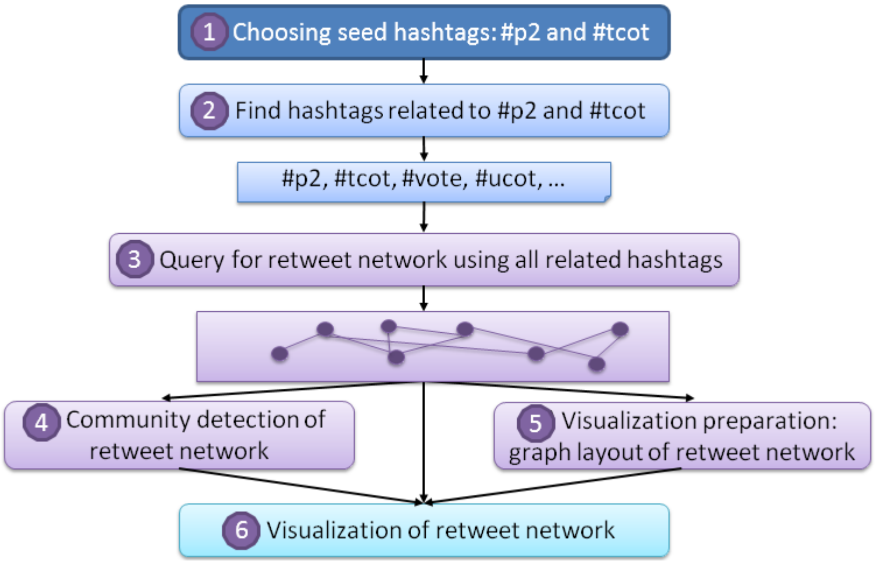

1 Choosing seed hashtags: #p2 and #tcot
2 Find hashtags related to #p2 and #tcot
#p2, #tcot, #vote, #ucot, ...
3 Query for retweet network using all related hashtags
4 Community detection of retweet network
5 Visualization preparation: graph layout of retweet network
6 Visualization of retweet network

Figure 3-9. End-to-end analysis workflow in [44]

Step (3) executes the ***get-retweet-edges*** query, using all hashtags found in Step (2) as the *memes* parameter and the six-week time window as the *time-window* parameter. It does this to get the retweet network among users from both political orientations. The retweet edges compose a graph structure, with vertices representing users and edges representing ‘retweet’ relationships that happened during the time window.

Step (4) uses a combination of two algorithms, leading eigenvector modularity maximization [109] and label propagation [118] to detect communities on the retweet network. Here a “community” is defined as a set of vertices on a graph that are densely inter-connected and sparsely connected to the other parts of the graph. After this step, vertices from different communities are labeled with different colors for visualization in Step (6).

In order to achieve a high-quality visualization of segregated communities in the retweet network, **Step (5)** uses the “Fruchterman-Reingold” algorithm [66] to generate a desirable layout of the retweet network. **Step (6)** makes a final plot of the retweet network from Step (3) using the color labels computed in Step (4) and layout information generated in Step (5).

3.3.2 Analysis of Twitter Data from 2010

We compose the workflow using our queries and analysis algorithms on the scalable architecture, and compare them to the original implementations in [44]. The experiments are carried out on 35 nodes of the same Alamo cluster as Section 3.2.5 (Table 3-3). We use Hadoop 1.0.4, HBase 0.94.2, Twister-Ivy (together with ActiveMQ 5.4.2), and R 2.10.1 in our experiments. Among the 35 nodes, one is used to host the Hadoop jobtracker and HDFS namenode, another hosts the HBase master, and a third hosts Zookeeper and Active MQ broker. The other 32 nodes host HDFS datanodes, Hadoop tasktrackers, HBase region servers, and Twister daemons.

As explained in Section 3.1, **Step (1)** is fixed to a manual choice of #p2 and #tcot. **Step (2)** is completed by running the related hashtag mining algorithm twice, once for #p2, and again for #tcot. Overall, it takes 109.3 seconds to find related hashtags for #p2, which involves analysis of the content of 109,312 tweets with 4 map tasks. The same process for #tcot spends 128.1 seconds in analyzing 189,840 tweets with 8 map tasks. Merging the results for both seed hashtags, we found the same 66 related hashtags as [44].

Step (3) is completed with the ***get-retweet-edges*** query. This step takes 93.3 seconds, and returns the same retweet network as in [44], which contains 23,766 non-isolated nodes.

Steps (4), (5), and (6) are completed by using the igraph [141] library of R in [142], which provides a baseline benchmark with sequential implementation. Table 3-4 lists the execution time of these three steps with R on a single node. It can be observed that Step (5) is significantly more time consuming than the other two steps, and may potentially become a bottleneck of the analysis workflow as we apply it to larger-scale datasets. Therefore, we use our parallel MRFR algorithm to complete this step. To facilitate it further, we modified ***get-retweet-edges*** to get ***get-retweet-adjmtx***, a new query that generates the **adjacency matrix** of the retweet network instead of only the edges. This query outputs a list of lines, and each line is in the form of ‘<vertex ID> <neighbor vertex ID> <neighbor vertex ID> …’, i.e. a vertex ID followed by a list of IDs of other vertices that are connected with this vertex by edges. This matrix representation is then given to MRFR as input.

Table 3-4. Sequential execution time (seconds) on R for Step (4) - (6) for 2010

(4) Community Detection	(5) Graph Layout (500 iterations)	(6) Visualization
3.4	4508.3	1.6

Figure 3-10 illustrates the per-iteration execution time and speed-ups of MRFR under different levels of parallelism. It is obvious that MRFR can effectively speed up the graph layout step. Specifically, with 64 mappers on 8 nodes, MRFR runs 18 times faster than the sequential implementation in R, completing 500 iterations within 300 seconds. However, MRFR does not achieve very good scalability for the 2010 retweet network, mainly because the amount of computation required in mappers is not large enough compared to the scheduling and communication overhead. For example, in the case of 64 mappers, the slowest mapper finishes in 250 ms, while the total overhead stays consistent at about 350 ms across different numbers of mappers. Figure 3-11 shows the final visualization of the retweet network using the layout generated by MRFR. The layout is almost the same as the plot in [44], with only a slight difference caused by a different initial random layout. As identified in [44], the red cluster is made of 93% right leaning (conservative) users, and the blue cluster is made of 80% left leaning (progressive) users. Since we generate the same result as [44] in each step of the analysis workflow, our solution on IndexedHBase is validated.

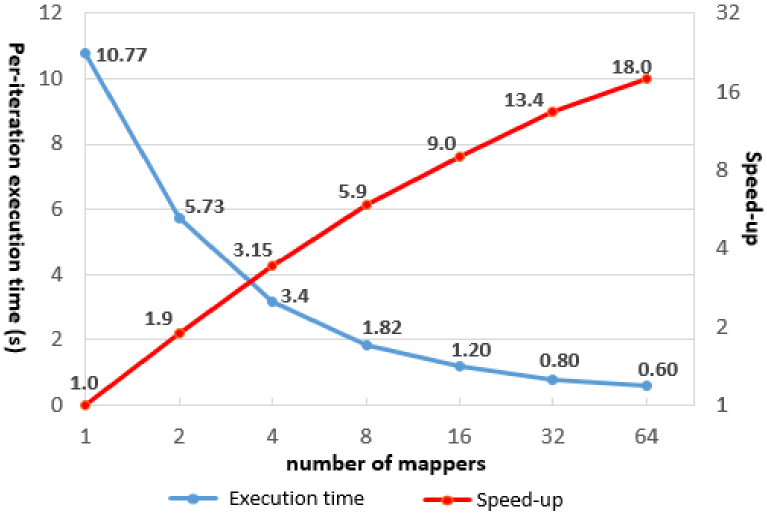


Figure 3-10. Per-iteration execution time and speed-up for MRFR for 2010

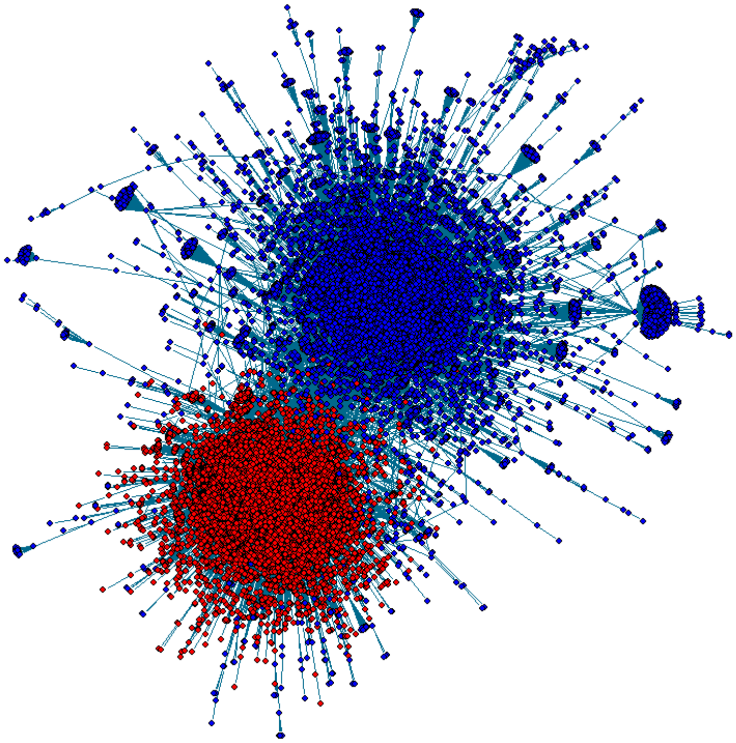


Figure 3-11. Final plot of the retweet network in 2010

3.3.3 Analysis of Twitter Data from 2012

Here we extend the analysis workflow in Figure 3-9 to a later dataset collected during the six weeks (09/24/2012 to 11/06/2012) before the 2012 U.S. presidential election, and verify if the corresponding retweet network demonstrates a similar polarized pattern. The average data size for each day in 2012 is about 6 times larger than 2010.

Step (1) still starts from #p2 and #tcot. **Step (2)** spends 142 seconds in mining related hashtags for #p2, and 191 seconds for #tcot. The number of tweets analyzed is 160,934 and 364,825 respectively. In total, 66 related hashtags are found (see Table 3-5). In **Step (3)**, 80 mappers need 150 seconds to analyze 2,360,361 politically related tweets, and the result is a retweet network that is 20 times larger, with 477,111 vertices and 665,599 edges.

Table 3-5. Related hashtags for 2012

Related to #p2: #2futures #47percent #4jobs #connecttheleft #cspj #mittromney #ofa #vote #votedem #wiright #ctl #dems #sensata #waronwomen #1u #benghazi #dem #p1 #fem2 #p2b #romnesia #tcot #dnc #forward #lgbt #msnbc #tpot #wiunion
Related to both: #obama #resist44 #romney #teaparty #tiot #cnn #lnyhbt #mitt2012 #news #ocra #ohio #ows #p21 #topprog #twisters #election2012 #gop #mapoli #masen #ncpol #sgp #sot #war #ccot #debate #obama2012 #romneyryan2012 #tlot
Related to #tcot: #debates #p2 #benghazigate #dems #gop2012 #benghazi #nobama #tpp #cantafford4more #nra #oh #prolife

Step (4) requires 2,402 seconds on R to complete community detection for this large network. In **Step (5)**, it takes as long as 6,044 seconds to finish only one iteration of the Fruchterman-Reingold algorithm on R. This demonstrates that due to the fast growth of data volume, sequential algorithms quickly become infeasible for social data analysis scenarios. In order to address this challenge, we use more mappers in MRFR to complete Step (5), and achieve nice speed-ups as shown in Figure 3-8. The near-linear scalability clearly demonstrates that MRFR is especially good at handling large networks. In particular, using 256 mappers on 32 nodes, MRFR can finish one iteration 355 times faster than the sequential implementation in R. **Step (6)** runs for 32 seconds on R, and Figure 3-12 shows the final plot of the two largest communities of the retweet network. On the one hand, we can still observe a clearly segregated political structure in the 2012 network; on the other hand, the two sides also seem to demonstrate a ‘merging’ trend by having more edges reaching out to each other.

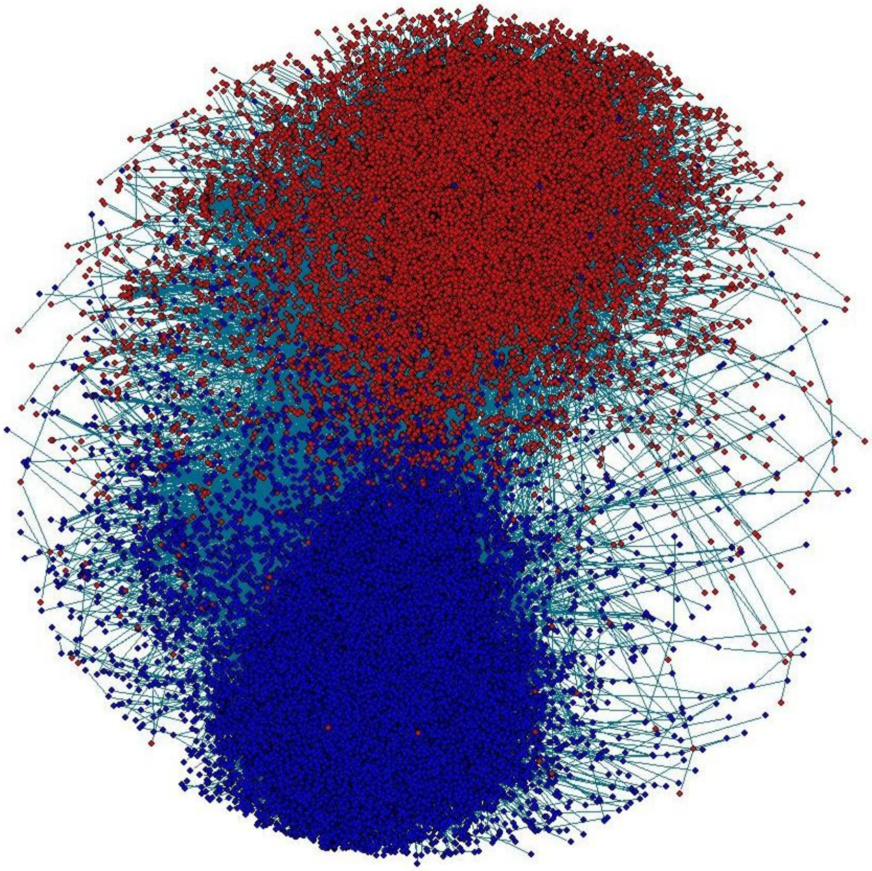


Figure 3-12. Final plot of the retweet network (2012)

3.4 Related Work

Compared with existing relational databases [78] and NoSQL databases [52][105][125], we not only support novel customizable text index structures, but also make innovative use of them. Instead of hiding them behind the queries, we expose direct operator interfaces so that they can be used in post-query analysis algorithms. Also by leveraging the inherent integration of IndexedHBase and Hadoop MapReduce, we are able to support efficient parallel scans of the indices. The significant performance difference between our analysis algorithms and their "Hadoop-FS" counterparts clearly demonstrates the value of indices in supporting analysis tasks beyond the basic queries.

By integrating components from Hadoop, Hive [21], and relational databases [136], HadoopDB [5] provides a hybrid solution that can utilize the indexing techniques offered by relational databases to achieve efficient query evaluation. Despite this, HadoopDB applies deep changes to the Hadoop framework; thus is difficult to configure and maintain. The SQL queries supported by HadoopDB also do not cover sophisticated iterative analysis algorithms.

By using Spark [166] as the execution engine and applying various optimizations to its in-memory processing model, Shark [16] is able to support both efficient SQL queries and sophisticated iterative analytics at a large scale. Compared with Shark, our architecture supports efficient fine-grained data operations, putting an emphasis on building customizable index structures to support both queries and analysis tasks. IndexedHBase can be integrated with Shark to further improve the performance of analysis jobs by only loading relevant data records as RDDs in Spark. The columnar storage of table data used by Shark is inspiring to us in terms of more efficient iterative analysis tasks.

To the best of our knowledge, MRFR is the first iterative MapReduce implementation for the Fruchterman-Reingold layout algorithm. There have been previous efforts on parallelizing this algorithm with MPI [106] and GPUs [129], but for commodity cluster environments where GPUs are not available, MRFR is the best fit and delivers near-linear scalability. We may consider extending our solution with the usage of GPUs on each node to handle larger-scale problems.

3.5 Conclusions

In summary, we make the following contributions in this chapter:

First of all, we extended IndexedHBase to a scalable architecture, which not only encapsulates efficient indexing and query mechanisms, but can also be integrated with various parallel processing frameworks such as Hadoop and Twister to support sophisticated analysis of the query results.

Based on this architecture, we develop a set of analysis algorithms, including related hashtag mining, meme daily frequency generation, domain name entropy computation, and graph layout, which are generally useful for composing analysis workflows in many research scenarios. Our experience with the first three algorithms demonstrates that indices are not only useful for query evaluation, but also valuable for analysis and mining purposes. Our index-based algorithms have proven to be significantly more efficient than the corresponding implementations based on parallel scans of original data, in terms of both execution time and resource usage. These are made possible by exposing proper index operator interfaces and leveraging the inherent integration between IndexedHBase and Hadoop MapReduce. Our graph layout algorithm is the first iterative MapReduce implementation of the Fruchterman-Reingold algorithm. It can achieve near-linear scalability for processing large graphs in distributed environments.

Finally, based on the queries and analysis algorithms, we demonstrate the composition and execution of analysis workflows by reproducing the end-to-end analysis process from a published research project about political polarization [44] and further extending it to another data subset about the 2012 U.S. presidential election. Experiments demonstrate that our solutions on Cloud DIKW can consistently provide efficient and scalable solutions for the analysis workflows, despite the significant data size growth over time.

Chapter 4

Stream Analysis Module - Parallel Clustering of High-Dimensional Social Media Data Streams

4.1 Background

As introduced in Chapter 1, Cloud DIKW is designed to support scientific analysis pipelines that require the integration of both sophisticated batch data processing algorithms and non-trivial streaming algorithms. By “non-trivial” algorithms, we refer to the cases where parallel workers not only process stream partitions independently, but also dynamically synchronize with the global state from time to time. The synchronization strategy could either leverage a pub-sub messaging system, reuse the communication mechanisms in batch algorithms, or a combination of both.

This chapter presents our contribution in applying Cloud DIKW to support one representative application: clustering of social media data streams. Specifically, we analyze the unique challenges brought by high-dimensional social media data streams and propose our extensions to current state-of-the-art stream processing frameworks, as well as an innovative synchronization method, for addressing the challenges.

As an important data mining technique, clustering is used in many applications involving social media stream analysis, such as meme [63][85], event [10], and social bots detection [63]. As an example, Figure 4-1 illustrates the analysis pipeline of the DESPIC (Detecting Early Signatures of Persuasion in Information Cascades) platform [63] that is being developed by the Center for Complex Networks and Systems Research at Indiana University. This platform first clusters posts collected from social streams (e.g., tweets from Twitter) into groups of homogenous memes, according to various measures of similarity, and then uses classification methods to detect memes generated by real users and separate them from those produced by social bots [64].

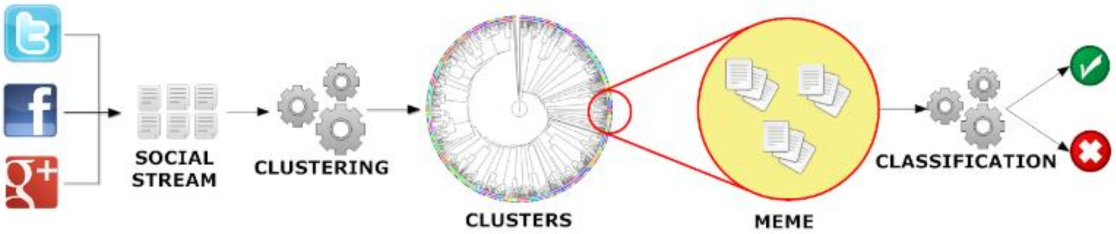


Figure 4-1. DESPIC architecture for meme clustering and classification [63]

Social media data streams come in the form of continuous sequences of atomic posts, e.g. Twitter tweets or Facebook status updates. The target of the clustering process is to group messages that carry similar meaning together, while capturing the dynamic evolution of the streams that is closely related to social activities in the real world. For example, two tweets, “Step up time Ram Nation. #rowdyrams” and “Lovin @SpikeLee supporting the VCU Rams!! #havoc”, should be grouped into the same cluster because they both talk about the VCU (Virginia Commonwealth University) basketball team. Furthermore, the appearance of “@SpikeLee” in the cluster is an indicator of the event that the famous director Spike Lee was wearing a VCU T-shirt while watching the VCU and UMass game courtside on Mar 16th, 2013.

In order to design a high-quality clustering algorithm, some unique characteristics of social posts must be considered. For instance, the length of the textual content of a social message is normally short, which makes clustering methods solely based on lexical analysis ineffective [10][29][63]. Social messages also carry rich information about the underlying social network (e.g. through the functionality of ‘retweet’ and ‘mention’ on Twitter), which can be valuable for measuring the similarity among data points and clusters. In addition they may contain other metadata such as temporal and geographical information, hashtags, URLs, etc., which can also be leveraged to effectively guide the clustering process.

Domain researchers in the area of social media data analysis have recently invested a great deal of effort toward developing proper data representations and similarity metrics to generate high-quality clusters [10][63][29][85]. An important conclusion is that the data representation should not only describe the textual features of the social messages, but also capture the temporal, geographical, and social network information attached therein. For example, Aggarwal and Subbian [10] proposed an event-detection system that uses two high-dimensional vectors to describe each social post: one content vector that represents the textual word frequencies, and another binary vector housing the IDs of the social message's recipients (e.g., the followers of a tweet's author on Twitter). To compute the similarity between two social messages, an independent score is first computed using each vector, and then a linear combination of the two scores is taken as the overall similarity between the two messages. It has been demonstrated that the quality of the resulting clusters can be significantly improved by using the combined similarity rather than just the textual content similarity. JafariAsbagh et al. [63] proposed to first group the social messages into 'protomemes' according to shared metadata such as hashtags and URLs, and then use the protomemes as input data points to the clustering algorithm. They use four high-dimensional vectors to describe each protomeme and define a new 'diffusion network' vector to replace the full followers vector used in [10], which is hardly available in a practical streaming scenario. The authors show that a combination of these new techniques can help generate better clustering results than previous methods when measured against a common ground truth data set.

To achieve efficient processing of social media data streams, these special data representations and similarity metrics are normally applied in a single-pass clustering algorithm such as online K-Means and its variants [10][85][87]. The algorithm can be further equipped with mechanisms like sliding time window [15][85], weighted data points [87][8][9][11], and outlier detection [8][10][35][85] to deal with the dynamic evolution of the streams. However, due to the high cost of similarity computation coming from the high-dimensional vectors, sequential implementations of such single-pass streaming algorithms are not fast enough to match the speed of real-world streams. For example, the fastest implementation presented in [10] can only process less than 20,000 tweets per hour, while the Twitter gardenhose stream [67] generates over 1,000,000 tweets in one hour. According to a test we carried out, it takes 43.4 hours for a sequential implementation of the algorithm in [85] to process one hour's worth of data collected through the gardenhose Twitter streaming API. It is therefore clear that parallelization is a necessity in order to handle real-time data streams.

In this chapter we describe our work in parallelizing a state-of-the-art social media data stream clustering algorithm presented in [85], which is a variant of online K-Means incorporating sliding time window and outlier detection mechanisms. We use Apache Storm [25] stream processing engine in Cloud DIKW for data transmission and workload distribution, and tackle two system-level challenges emerging from parallelization of such type of algorithms.

The first challenge concerns the fact that most stream processing engines organize the distributed processing workers in the form of a directed acyclic graph (DAG); this makes it difficult to dynamically synchronize the state of the parallel clustering workers without breaking the "live" processing of the stream. The reason is that the synchronization step requires parallel workers to send their local updates either to each other or to a global updates collector, which will then broadcast the updated global state back to the parallel workers. Both methods inevitably create cycles in the communication channel, which is not supported in the DAG-oriented stream processing frameworks. To address this challenge, we create a separate synchronization channel by incorporating the pub-sub messaging system ActiveMQ [16] into Cloud DIKW, and combine its functionality with Storm to coordinate the synchronization process.

The second issue is that the sparsity of high-dimensional vectors may cause the cluster centroids to greatly increase in size with the addition of new data points to the clusters. Figure 4-2 illustrates a cluster containing two tweets about VCU basketball as mentioned earlier. Due to the sparsity of the content vector (assuming the hashtags and user mentions are extracted as another separate vector) of each data point, they only overlap along one dimension: “ram”. As a result, the length of the content vector of the centroid, which is computed as an average of the two data points, is close to the sum total length for two separate vectors. Due to the high dimensionality of these vectors, this trend can continue as more data points are added, and the length of the centroid vectors increases dramatically. A sliding time window mechanism may help to limit the total size by removing old data points, but the full centroids data remains large and difficult to transfer over the network. Consequently, the classic synchronization strategy of directly broadcasting the cluster centroids becomes infeasible and hampers scalability of the parallel algorithm. To solve this problem, we propose a new strategy that broadcasts the dynamic changes (i.e. the “deltas”) of the clusters rather than the complete centroids data. Since the size of the delta is small, we are able to keep the synchronization cost at a low level and achieve good scalability. For sake of simplicity, we name the traditional synchronization strategy *full-centroids strategy*, and our new synchronization strategy *cluster-delta strategy*.

We use a real dataset collected through the Twitter streaming API 10% sample (“gardenhose”) [67] to verify the effectiveness of our solutions and evaluate the scalability of our parallel algorithm. The results demonstrate that we can keep up with the speed of the Twitter gardenhose stream with 96-way parallelism. By natural improvements to Cloud DIKW, including advanced collective communication techniques developed in our Harp [169] project, we will be able to process the full Twitter data stream in real-time with 1000-way parallelism. Our use of powerful general software subsystems will enable many other applications that need integration of streaming and batch data analytics.

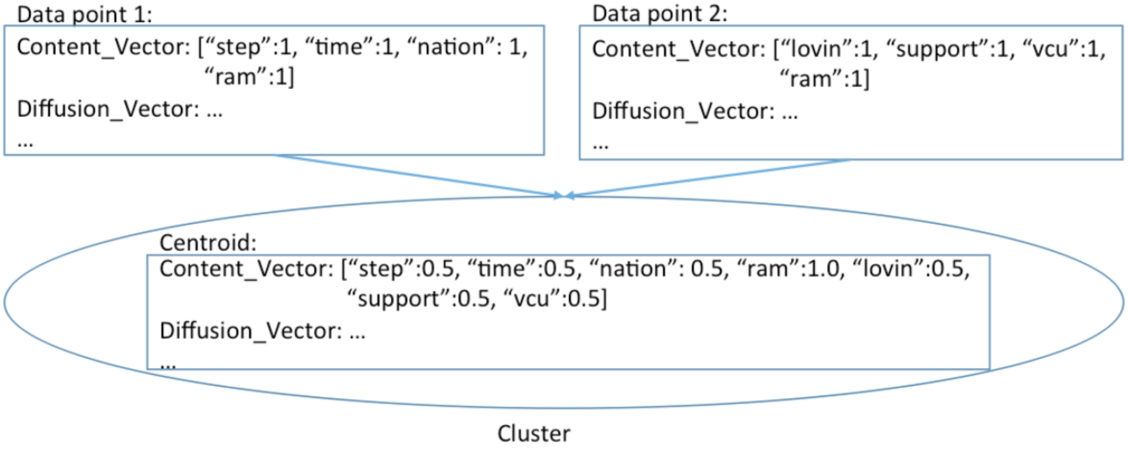

Data point 1:
Content_Vector: [“step”:1, “time”:1, “nation”: 1, “ram”:1]
Diffusion_Vector: …
…
Data point 2:
Content_Vector: [“lovin”:1, “support”:1, “vcu”:1, “ram”:1]
Diffusion_Vector: …
…
Centroid:
Content_Vector: [“step”:0.5, “time”:0.5, “nation”: 0.5, “ram”:1.0, “lovin”:0.5, “support”:0.5, “vcu”:0.5]
Diffusion_Vector: …
…
Cluster

Figure 4-2. An example of growing vector size of centroids

4.2 Related Work

Data stream clustering algorithms have been an active research area for many years as witnessed by Ding et al. review work [56]. For the problem of high-dimensional data stream clustering, techniques such as projected/subspace clustering [8][9][138] and density-based approaches [15][35][138] have been proposed and investigated. Due to the unique data representations (multiple high-dimensional vectors from totally independent spaces) and similarity metrics used for social media data streams, it seems hard to apply these existing techniques to the case of social media streams. We listed and discussed practical limitations in a previous work [63]. Here we inherit the high-dimensional data representation and similarity metrics that have been proven effective, and focus on improving the efficiency of the clustering algorithm through parallelization.

The algorithm presented in [10] uses sketch tables [7] to deal with the growing size of tweet followers network information maintained for the clusters. However, sketch tables only approximate vector values and thus may impact the accuracy of the clustering results. In the case of our algorithm, since the size of the centroid vectors is constrained by the size of the sliding time window, we are not forced to use sketch tables in the cost of accuracy so far. For faster data streams or longer time windows, a sketch table-based implementation could eventually become more efficient in terms of both space and time for computing the similarities between data points and cluster centroids. Nonetheless, our cluster-delta synchronization strategy may still achieve better efficiency than broadcasting the whole sketch tables in such cases since the sketch tables have to be large enough to ensure accuracy.

A similar work to ours is the parallel implementation of the Sequential Leader Clustering [76] algorithm presented in [164], which also leverages Storm [25] for parallel processing and data stream distribution. The parallel clustering algorithm by Wu et al. is simplified, because it only considers the textual content of social messages and uses Locality-Sensitive Hashing [31] to guide the stream distribution, which avoids synchronization among the parallel clustering workers. Yet this type of algorithms is unable to make use of the valuable social network information contained in the data streams. Callau-Zori proposed a distributed data stream clustering protocol based on sequential (a, b)-approximation algorithms for the K-Means problem [34]. Although the author provides a theoretical analysis of its accuracy and efficiency, it does not address the special case of high-dimensional data, and only considers the situation within a single time window.

Compared with streaming databases such as Aurora [39] and Borealis [2], the functionality of our clustering workers in Storm is more complicated than their streaming operators for evaluating SQL queries. Cloud DIKW can utilize other stream processing engines such as Apache S4 [108] and Spark Streaming [167]. We choose Storm because its pull-based data transmission mode makes it easy to carry out controlled experiments at different levels of parallelism. Storm gives us more flexibility to implement and test different synchronization strategies. Interested readers may refer to [86] for a survey of major distributed stream processing frameworks.

4.3 Sequential Clustering Algorithm

The sequential algorithm we parallelize was originally proposed in [85] for clustering memes in the Twitter streams of tweets. In order to generate high-quality clusters, the algorithm first groups tweets into 'protomemes', and then uses these protomemes as input data points for the clustering process. We start by introducing the definition of a protomeme and its data representation.

4.3.1 Protomemes and Clusters

A *protomeme* is defined as a set of tweets grouped together according to a shared entity of one of the following types:

- **Hashtags**. Tweets containing the same hashtag.
- **Mentions**. Tweets mentioning the same user. A mention is identified by a user's screen name preceded by the '@' symbol in the text body of a tweet.
- **URLs**. Tweets containing the same URL.
- **Phrases**. Tweets sharing the same phrase. A phrase is defined as the textual content of a tweet that remains after removing the hashtags, mentions, URLs, and after stopping and stemming [170].

We call these four types of entities markers of protomemes. Note that according to this definition, a tweet may belong to multiple protomemes. Each protomeme is represented by its marker and four high-dimensional vectors:

(1) A binary *tid vector* containing the IDs of all the tweets in this protomeme: $V_T = [tid_1, tid_2, \ldots, tid_T]$;

(2) A binary *uid vector* containing the IDs of all the users who authored the tweets in this protomeme: $V_U = [uid_1, uid_2, \ldots, uid_U]$;

(3) A *content vector* containing the combined textual word frequencies for all the tweets in this protomeme: $V_C = [w_1{:}f_1, w_2{:}f_2, \ldots, w_C{:}f_C]$;

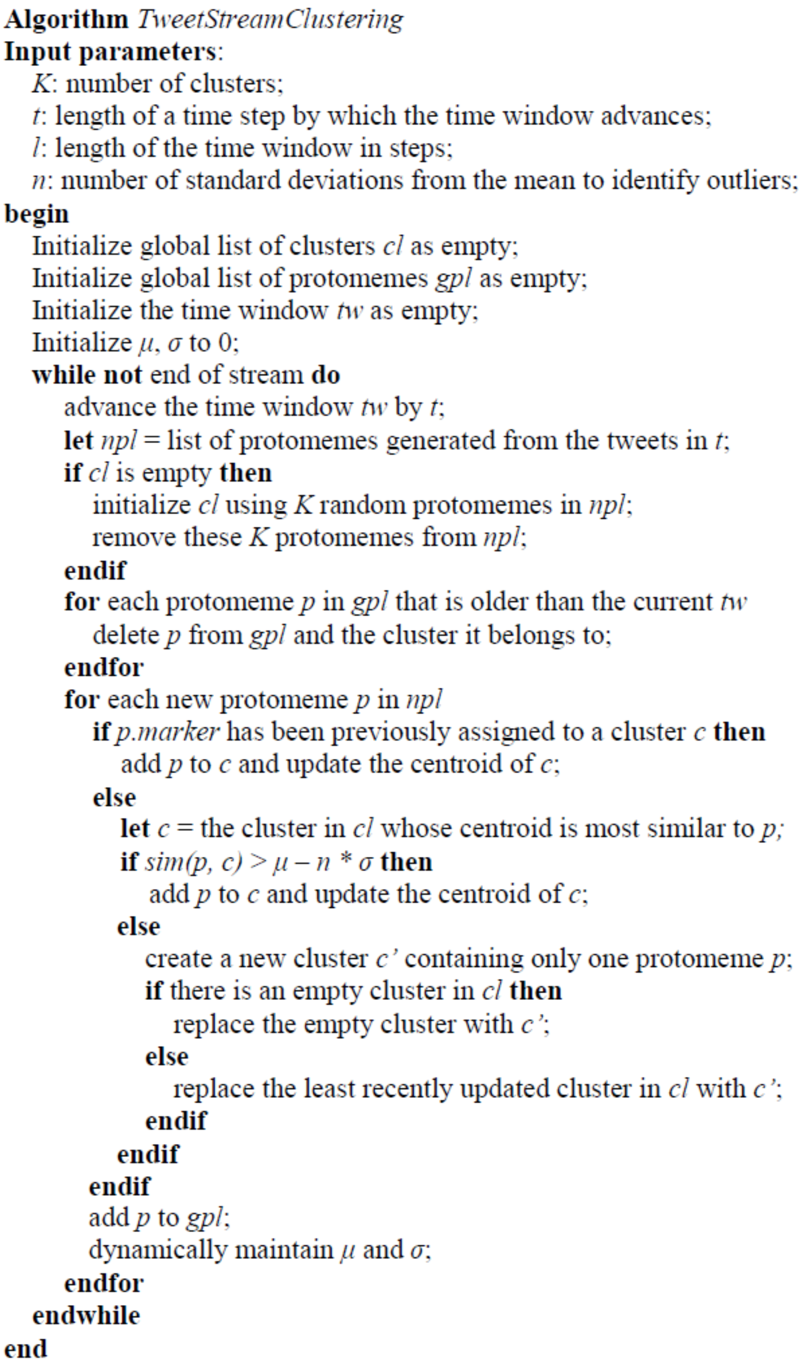
(4) A binary vector containing the IDs of all the users in the *diffusion network* of this protomeme. The diffusion network of a protomeme is defined as the **union** of the set of tweet authors, the set of users mentioned by the tweets, and the set of users who have retweeted the tweets. We denote this *diffusion vector* as $V_D = [uid_1, uid_2, \ldots, uid_D]$.

A *cluster* is defined as a set of protomemes grouped together according to a certain similarity metric. Since a tweet may belong to multiple protomemes, clusters can have overlap with respect to tweets. The centroid of each cluster is also represented by four high-dimensional vectors, which are the averages of the corresponding vectors of all the protomemes in the cluster. We denote the vectors of the cluster centroid as $\mathcal{V}_T$, $\mathcal{V}_U$, $\mathcal{V}_C$, and $\mathcal{V}_D$.

To compute the *similarity* between a protomeme and a cluster, the **cosine similarity** between each vector of the protomeme and the corresponding vector of the cluster centroid is first computed. Then the **maximum value** of all these cosine similarities is taken as the overall similarity between the two. It has been demonstrated in [63] that for the purpose of generating high-quality clusters, taking the maximum is as effective as using an optimal linear combination of all the cosine similarities. There are multiple ways to define *distance* based on the similarity; we use the simplest form *1 – similarity*.

4.3.2 Sequential Clustering Algorithm

Figure 4-3 illustrates the sketch of the sequential clustering algorithm from [85]. The algorithm controls its progress through a sliding time window that moves step by step. The length of a time step in seconds and the length of the time window in steps are given as input parameters. These are defined with respect to the timestamps of the social posts (i.e., the tweets), not the wall-clock time for running the algorithm. Every time the sliding window advances, old protomemes falling out of the current window are deleted from the clusters and new ones are generated using the tweets from the latest time step. For every new protomeme, the algorithm first checks whether others with the same marker have been previously assigned to a cluster. If so, the new protomeme will be added to the same cluster. Otherwise, the algorithm will compute the new protomeme's similarity with all the existing clusters, and decide whether or not this is an outlier. If not, the protomeme is assigned to the most similar cluster. Otherwise, a new cluster is created and initialized with this new protomeme, then inserted into the list of all clusters by replacing either an empty cluster or the least recently updated one. In order to determine whether the protomeme is an outlier, the algorithm maintains the mean μ and standard deviation σ of the similarities between all processed protomemes and the centroid of the clusters they belong to. If the similarity between a new protomeme and its closest cluster is smaller than the mean by more than *n* standard deviations, then the protomeme is identified as an outlier. μ and σ are maintained incrementally as in [10].



```
Algorithm TweetStreamClustering
Input parameters:
  K: number of clusters;
  t: length of a time step by which the time window advances;
  l: length of the time window in steps;
  n: number of standard deviations from the mean to identify outliers;
begin
  Initialize global list of clusters cl as empty;
  Initialize global list of protomemes gpl as empty;
  Initialize the time window tw as empty;
  Initialize $\mu$, $\sigma$ to 0;
  while not end of stream do
    advance the time window tw by t;
    let npl = list of protomemes generated from the tweets in t;
    if cl is empty then
      initialize cl using K random protomemes in npl;
      remove these K protomemes from npl;
    endif
    for each protomeme p in gpl that is older than the current tw
      delete p from gpl and the cluster it belongs to;
    endfor
    for each new protomeme p in npl
      if p.marker has been previously assigned to a cluster c then
        add p to c and update the centroid of c;
      else
        let c = the cluster in cl whose centroid is most similar to p;
        if $sim(p, c) > \mu - n * \sigma$ then
          add p to c and update the centroid of c;
        else
          create a new cluster c' containing only one protomeme p;
          if there is an empty cluster in cl then
            replace the empty cluster with c';
          else
            replace the least recently updated cluster in cl with c';
          endif
        endif
      endif
      add p to gpl;
      dynamically maintain $\mu$ and $\sigma$;
    endfor
  endwhile
end
```

Figure 4-3. The social media stream clustering algorithm from [85]

The quality of clusters generated by this algorithm was evaluated in [85] using a ground truth dataset collected from the Twitter gardenhose stream [67] during a week in 2013, which includes all the tweets containing the Twitter trending hashtags [65][150] identified for that period. A variant of the *Normalized Mutual Information* (NMI) [46] measurement, LFK-NMI [90], which is especially well suited for the case of overlapping clusters, was computed between the result clusters of the algorithm and the ground truth clusters. The results in [85] show that this algorithm can achieve better performance than previous state-of-the-art methods, including the one presented in [10]. We use the same ground truth dataset and LFK-NMI measurement to verify the effectiveness of our parallel implementation of the algorithm in Section 4.5.

4.3.3 Opportunities and Difficulties for Parallelization

We run the sequential algorithm on a raw dataset (without any filtering) containing six minutes of tweets (2014-08-29 05:00:00 to 05:05:59) collected from the Twitter gardenhose stream. By fixing the parameters *K*, *l*, and *n* to 120, 6, and 2, and varying the length of a time step, we collect some important runtime statistics that are informative to the development of the parallel version of the algorithm.

Table 4-1. Runtime Statistics for the Sequential Algorithm

Time Step Length (s)	Total Length of Content Vector	Similarity Compute time (s)	Centroids Update Time (s)
10	47749	33.305	0.068
20	76146	78.778	0.113
30	128521	209.013	0.213

Table 4-1 presents the results for the last time step of the whole clustering process when the time step length is increased from 10 to 30 seconds (which means the time window length is increased from 60 to 180 seconds). The numbers for the other time steps follow a similar pattern. The second column measures the total length of the content vectors of all the cluster centroids at the end of the last time step; the third column measures the time spent on computing the similarities between protomemes and cluster centroids in that time step; and the fourth column measures the time spent on updating the vectors of the cluster centroids.

Some interesting observations lead to our research of parallelizing the streaming algorithm: first, the whole clustering process is dominated by the computation of similarities. The ratio of **similarity compute time / centroids update time** in Table 4-1 increases from 490 to 981 as the length of the time window increases. This implies the feasibility of parallelizing the similarity computation, and processing the global updates of centroids with a central collector. Furthermore, the longer the time window, the more we can benefit from parallelization.

We also observed that the content vector size of the centroids expands as the length of the time window increases. In fact, the other vectors (V_T, V_U, $V_{D)}$ demonstrate the same trend. This confirms our analysis in Section I about the infeasibility of traditional synchronization strategies. To address this issue, we design the new cluster-delta strategy, which will be presented in Section 4.4.

4.4 Parallel Implementation on Storm

4.4.1 Storm

Apache Storm is a stream processing engine designed to support large-scale distributed processing of data streams. It defines a stream as an unbounded sequence of *tuples*, and provides an easy-to-use event-driven programming model to upper level applications. Stream processing applications are defined in the form of *topologies* in Storm, as exemplified in Figure 4-4. There are two types of *processing elements* in a topology, *spouts* and *bolts*, which are organized into a DAG through the streams connecting them. A spout is a source of streams that generates new tuples and injects them into the topology. A bolt can consume any number of input streams, do some processing to each tuple of the streams, and potentially generate and emit new tuples to the output streams. To define a topology, the application only needs to provide implementation logics of spouts and bolts, specify the runtime parallelism level of each type, and configure the data distribution patterns among them. The Storm framework will automatically take care of system management issues including data transmission, parallel spouts/bolts execution, work load distribution, and fault tolerance.

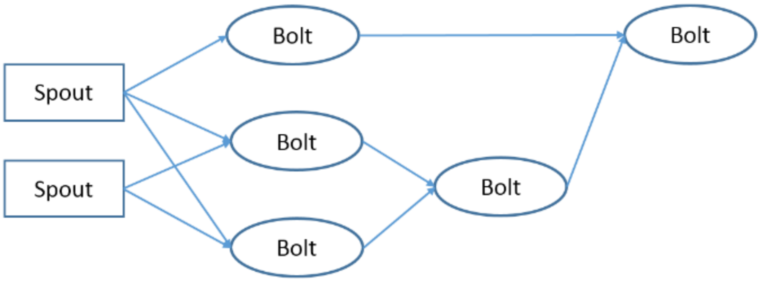


Figure 4-4. An example topology in Storm

Figure 4-5 illustrates the standard architecture of a Storm cluster. The whole cluster consists of two types of nodes: one master node and multiple worker nodes. The master node runs a daemon process called *Nimbus* responsible for assigning spout and bolt tasks to the worker nodes and monitoring their status for failures. Every worker node runs a *Supervisor* daemon process, which manages the resources on the local node and accepts task assignments from the *Nimbus*. Spout and bolt tasks are executed by parallel *executor threads* in *worker processes*. By default, one executor thread is spawned for each task. The number of worker processes on each node is configurable as a system parameter. The number of tasks to run for each type of spout and bolt in a topology can be configured through the parallelism parameters. Coordination between the *Nimbus* and the *Supervisors* is accomplished by using Zookeepers [26].

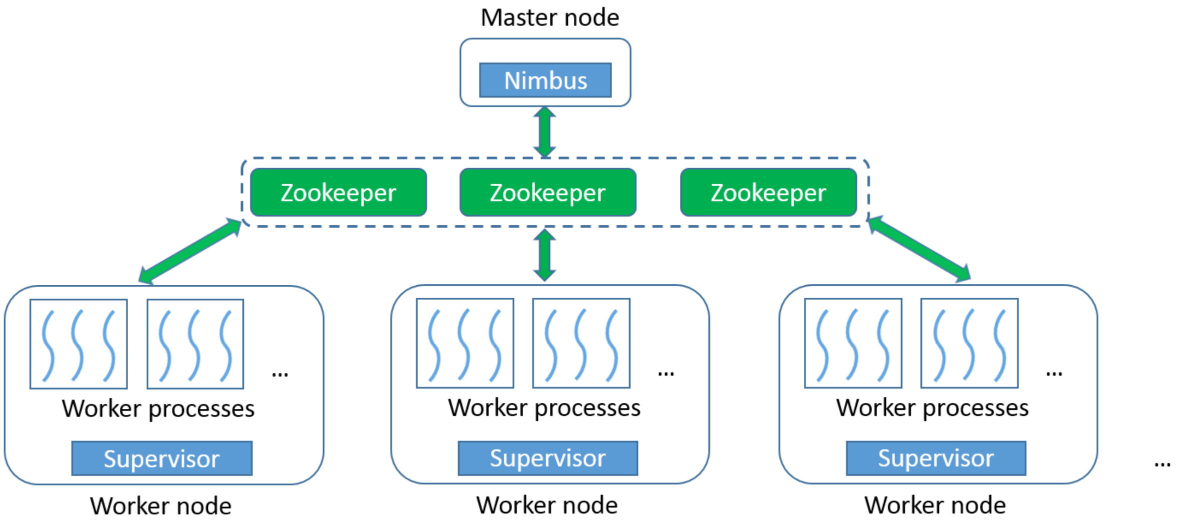


Figure 4-5. Storm architecture

Storm adopts the ‘pull-based’ message passing model between the processing elements. Bolts pull messages from the upstream bolts or spouts. This ensures that bolts will never get excessive workload that they cannot handle. Therefore, overflow can only happen at the spouts. This model allows us to test our algorithm easily at different levels of parallelism. For example, we can implement spouts that generate streams by reading data from a file, and control their paces based on the number of acknowledgements received for tuples that have been processed. This will prevent the topology from getting overwhelmed by too much data no matter how slowly the bolts are working.

4.4.1 Implementation with Cluster-Delta Synchronization Strategy

We implement the parallel version of the algorithm in a Storm topology, as illustrated in Figure 4-6. There is one type of spout, *Protomeme Generator Spout*, and two types of bolts, *Clustering Bolt* and *Synchronization Coordinator Bolt*. For simplicity, we call them *protomeme generator*, *cbolt*, and *sync coordinator*. At runtime, there is one instance of the protomeme generator, multiple instances of cbolts working in parallel, and one instance of sync coordinator. A separate synchronization channel is created between the cbolts and the sync coordinator using the ActiveMQ pub-sub messaging system [16]. ActiveMQ allows client applications to connect to *message brokers*, and register themselves as *publishers* or *subscribers* to various *topics*. Publishers can produce messages and publish them to a certain topic, and the message broker will automatically deliver the messages to all the subscribers of that topic. In our topology, the sync coordinator is registered as a publisher to a topic named "clusters.info.sync", and all the cbolts are registered as subscribers to this topic. The lifetime of the whole topology can be divided into two phases, an *initialization phase* and a *running phase*. We introduce the working mechanism of each type of spout and bolt in both phases.

Protomeme Generation

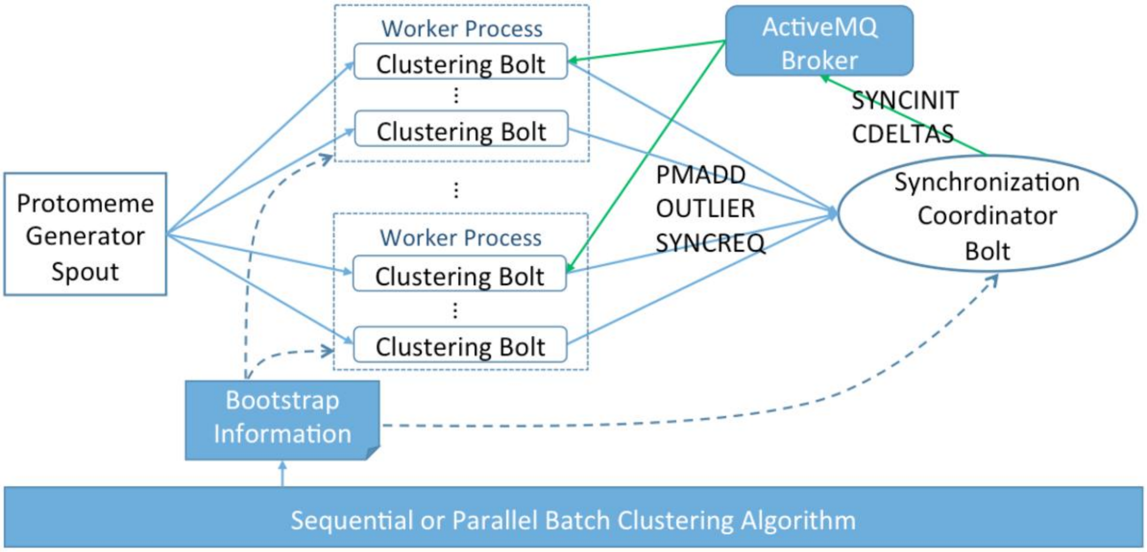


Figure 4-6. Storm topology for the parallel stream clustering algorithm

During the **initialization phase**, every processing element reads some information from a bootstrap file. The protomeme generator reads the start time of the current time step, the length of a time step in seconds, and the length of a time window in steps. After reading this information, the generator can either connect to an external stream of tweets or open a file containing tweets for generating protomemes.

Upon entering the **running phase**, the protomeme generator keeps reading and buffering tweets for the “current” time step, until it identifies a tweet falling into the next time step. Then it generates protomemes using the buffered tweets. Every protomeme is associated with a *creation timestamp* and an *ending timestamp*, which are set based on the timestamp of the earliest and latest tweet in the protomeme. To facilitate the construction of diffusion vectors of protomemes, an **in-memory index structure** is maintained to record the mapping between each tweet ID and the set of user IDs who have retweeted it. To construct the diffusion vector of a protomeme, the user IDs of the tweet authors and the user IDs mentioned in its tweets are first added to the vector. Then the index is queried for each tweet ID of the protomeme, and the corresponding user IDs found in the index are added to the vector. The protomeme generator emits one tuple to its output stream for every newly generated protomeme. The tuples are evenly distributed among all the parallel cbolts based on the hash values of their markers. Therefore, protomemes generated in different time steps but sharing the same marker will always be processed by the same cbolt.

Protomeme Clustering

During the **initialization phase**, the cbolts and sync coordinator first read the same time window parameters as the protomeme generator; then they read the input parameter *n* (number of standard deviations for outlier detection), and a list of initial clusters. The initial clusters are generated by running either a parallel batch clustering algorithm, or the sequential stream clustering algorithm over a small batch of data from recent history. The initial values of μ and σ are then generated based on the protomemes contained in the initial clusters.

During the **running phase**, protomemes are processed in small batches. A *batch* is defined as the number of protomemes to process, which is normally configured to be much smaller than the total number of protomemes in a single time step. Upon receiving a protomeme, the cbolt first checks its creation timestamp to see if it starts a new time step. If so, the cbolt will first advance the current time window by one step, and delete all the old protomemes falling out of the time window from the clusters. Then it performs the outlier detection procedure and protomeme-cluster assignment in the same way as in the sequential algorithm, based on the current clusters and μ, σ values. If the protomeme is an outlier, an *OUTLIER* tuple containing the protomeme will be emitted to the sync coordinator. If it can be assigned to a cluster, a *PMADD* tuple will be emitted. Note that the cbolt does not immediately create a new cluster with the outlier protomeme, because outlier protomemes detected by different cbolts may be similar to each other and thus should be grouped into the same cluster. Such global grouping can only be done by the sync coordinator, which collects *OUTLIER* tuples generated by all the cbolts. For the case of *PMADD*, the centroid of the corresponding cluster is not immediately updated either. Instead, clusters are only updated during the synchronization between two consecutive batches. This ensures that within the same batch, different cbolts are always comparing their received protomemes against the same set of global clusters.

Within each batch, the sync coordinator maintains a list of “cluster delta” data structures and another list of outlier clusters. Upon receiving a *PMADD*, it will simply add the protomeme contained in the tuple to the delta structure of the corresponding cluster, and change the latest update time of the delta structure to the ending timestamp of the protomeme in case the ending timestamp is larger. Since the sync coordinator collects *PMADD* from all parallel cbolts, the delta structures will contain the global updates to each cluster. For an *OUTLIER* tuple, it will first check whether the protomeme contained in the tuple can be assigned to any existing outlier cluster. If so, it is simply added to that outlier cluster; otherwise a new outlier cluster is created and appended to the list of outlier clusters. After processing each tuple, the values of μ and σ are dynamically updated.

Synchronization

As a final step of the **initialization phase**, the cbolts and sync coordinator connect to an ActiveMQ message broker and register as subscribers and the publisher. Since the cbolt tasks run as threads in worker processes, they first go through an **election** step to select one **representative thread** within each process. Only the representative thread will be registered as a subscriber, and the synchronization message received will be shared among the threads in the same process. This election step can significantly reduce the amount of data transmission caused by synchronization.

At the **running phase**, a synchronization procedure is launched when the number of processed protomemes reaches the batch size. The whole procedure consists of three steps as detailed in Figure 4-7: ***SYNCINIT***, ***SYNCREQ***, and ***CDELTAS***. The *SYNCINIT* step initiates the procedure and notifies the cbolts to start synchronization. In the *SYNCREQ* step, each cbolt will temporarily stop processing incoming protomemes, and emit a *SYNCREQ* tuple. After receiving *SYNCREQ* from all the cbolts, the sync coordinator will sort the deltas of all the clusters (including the outlier clusters) by the latest update time, and pick the top K with the highest values to construct a *CDELTAS* message, which also contains latest global values of μ and σ. The message is then published through ActiveMQ. Upon receiving *CDELTAS*, every cbolt will update their local copy of clusters and μ, σ values to a new global state, then resume processing the protomemes for the next batch. Note that the *SYNCINIT* step and the temporary stopping of the cbolts are necessary to ensure that protomemes processed by different cbolts and received by the sync coordinator are always handled with regards to the same global view of the clusters. Since the size of *CDELTAS* is normally small and stable, the synchronization step can usually finish in a short time, as will be demonstrated in Section 4.5.

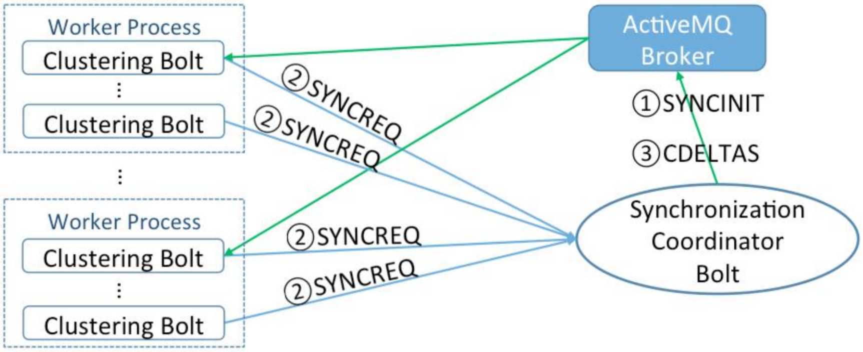


Figure 4-7. Synchronization process of the cluster-delta strategy

In order to achieve the best performance for the whole synchronization procedure, an optimal solution for *SYNCINIT* is also necessary. We tested three methods in this regard. With **spout initiation**, the protomeme generator counts the number of protomemes emitted and broadcasts a *SYNCINIT* tuple through Storm when the batch size is reached. With **cbolt initiation**, each cbolt counts the number of protomemes processed by itself and directly emits a *SYNCREQ* tuple when it reaches the expected average. This method is similar to the synchronization mechanism used in typical iterative batch algorithms. However, due to the buffering effect of Storm and varied processing speed among cbolts, both methods suffer from a large variance in the *SYNCREQ* time observed by different cbolts. The variance can reach the level of seconds and totally eliminate the benefits of the cluster-delta strategy. This suggests that, due to the dynamic nature of streaming analysis, synchronization should be handled differently than in batch algorithms. To address this issue, we propose **sync coordinator initiation** as illustrated in Figure 4-7. In this method, the sync coordinator counts the total number of *PMADD* and *OUTLIER* received, and publishes a *SYNCINIT* message using ActiveMQ if the batch size is reached. Because of the pushing-mode of message delivery and the small size of the message, it can be received by the cbolts within milliseconds. Therefore the large variance problem is avoided.

4.4.2 Implementation with Full-Centroids Synchronization Strategy

To verify the effectiveness of our cluster-delta synchronization strategy, we implement another version of the parallel algorithm using the full-centroids strategy for comparison. The protomeme generation and processing logics of the full-centroids version are mostly the same as the cluster-delta version. There are, however, major differences in the implementation caused by the full-centroids strategy: during the processing time of each batch, the sync coordinator will maintain a full list of existing clusters, instead of their delta structures. During the synchronization time, instead of the *CDELTAS* message, it will generate a *CENTROIDS* message, which contains the whole centroid vectors of the clusters with the top *K* latest update times. Upon receiving the *CENTROIDS* message, every cbolt will use the centroid vectors contained in the message to replace the centroids of the old clusters.

Since the cbolt receives the centroid vectors rather than the incremental protomemes of each cluster, it can no longer maintain a full record of all the protomemes in the clusters. Therefore, the task of new time step detection and old protomeme deletion is moved to the sync coordinator. The centroids update time is negligible if compared to the similarity compute time, so this has little impact on the overall performance of the algorithm.

4.5 Evaluation of the Parallel Algorithm

We verify the correctness of our parallel algorithm by comparing its results with the sequential implementation, and evaluate its efficiency and scalability through comparison with the full-centroids synchronization strategy. Our evaluation tests are done on the same Madrid cluster as described in Section 3.2.5. Each node runs RHEL 6.5, Java 1.7.0_45, and Apache Storm 0.9.2. Apache ActiveMQ 5.4.2 is deployed on the same node where the Storm Nimbus runs. Each node is configured to run at most four Storm worker processes, and the parallel instances of spouts and bolts are launched as threads spawned by these worker processes. The maximum heap size of each worker process is set to 11GB. Message compression with zip is enabled for ActiveMQ, and only one message broker is used in all tests of the parallel implementations.

4.5.1 *Correctness Verification*

To test the correctness of our algorithm, we use the same ground truth dataset and LFK-NMI measurement as [85]. The LFK-NMI value is a number between 0 and 1 that indicates the degree of matching between two sets of result clusters. A value of 1 corresponds to a perfect matching, while a value of 0 means that the two sets of clusters are completely disjointed. The ground truth dataset was collected from the Twitter gardenhose stream [67] within the week of 2013-03-23 to 2013-03-29. It includes all the tweets containing the Twitter trending hashtags [65][149] identified during that time.

We first define the ground truth clusters as the sets of tweets corresponding to each trending hashtag: all tweets sharing a common trending hashtag are grouped into one separate cluster. Note that, since a tweet may contain multiple trending hashtags, the ground truth clusters may have overlaps. We then remove the trending hashtags from the content of all tweets, and run both the sequential implementation from [85] and our parallel implementation over the remaining dataset. As a result, protomemes corresponding to the trending hashtags will not be created and used as input data points to the clustering process. This is done to avoid giving an unfair advantage to protomeme-based algorithms that use hashtag information. Finally, we compute three LFK-NMI values: results of the sequential algorithm versus the ground truth clusters, results of the parallel algorithm versus the ground truth clusters, and results of the sequential versus the parallel algorithm. We use the same input parameters as the experiments completed in [85]: $K = 11$, $t = 60$ minutes, $l = 6$, and $n = 2$. For the parallel algorithm, we use two parallel cbolts and a batch size of 40.

Table 4-2 presents the LFK-NMI scores using the final clusters generated by the two algorithms. The high value of 0.728 in the first column indicates that the clusters generated by our parallel implementation match very well with the results of the original sequential implementation in [85]. Moreover, values in the second and third column suggest that, when measured against the same ground truth clusters, our parallel implementation can achieve a degree of matching comparable or better (we observe an improvement of around 10%) than the sequential implementation. These scores show that our parallel implementation is correct and can generate results that are consistent with the sequential algorithm. The value 0.169 is consistent with the original test results in [85]. In addition, the slightly higher value of 0.185 indicates that processing the protomemes in small batches may be helpful for improving the quality of the clusters.

Table 4-2. LFK-NMI Values for Correctness Verification

Parallel vs. Sequential	Sequential vs. ground truth	Parallel vs. ground truth
0.728	0.169	0.185

4.5.1 *Performance Evaluation*

To evaluate the performance and scalability of our parallel algorithm in Cloud DIKW, we use a raw dataset collected from the Twitter gardenhose stream without applying any type of filtering. It contains a total number of 1,284,935 tweets generated within one hour (from 05:00:00 AM to 05:59:59 AM) on 2014-08-29. We first run the sequential algorithm over the whole dataset using input parameters K = 240, t = 30 seconds, l = 20, and n = 2, and measure the total processing time. Note that the time window has a length of 10 minutes and thus may contain a large number of protomemes. Then we run the two parallel implementations at different levels of parallelism, and measure their processing time, speedup, and other important statistics. We use the clusters generated for the first 10 minutes of data as the bootstrap clusters, and process the following 50 minutes of data using the parallel algorithms. The average number of protomemes generated in each time step is 19,908, and the batch size is set to 6,144.

The total processing time of the sequential algorithm is 156,340 seconds (43.43 hours), and the time spent on processing the last 50 minutes of data is 139,950 seconds (38.87 hours). Figure 4-8 compares the total processing time of the two parallel implementations, and some important statistics are given in Table 4-3 and 4-4. Numbers in brackets in the first column tell how many Storm worker processes were used for hosting the cbolt threads. These correspond to the total numbers of ActiveMQ receivers in each run. Here we list the numbers that delivered the best overall performance. The length of the synchronization message in the last column is measured before ActiveMQ runs any compression. Figure 4-9 compares the scalability of the two parallel implementations (the blue line and the red line).

Table 4-3. Statistics for Full-centroids Version Parallel Algorithm

Number of cbolts (worker processes)	Total processing time (sec)	Compute time / sync time	Sync time per batch (sec)	Avg. length of sync message
3 (1)	67603	31.56	6.45	22,113,520
6 (1)	35207	15.53	6.51	21,595,499
12 (2)	19228	7.79	6.60	22,066,473
24 (4)	10970	3.95	6.76	22,319,413
48 (7)	6818	1.92	7.09	21,489,950
96 (28)	5804	0.97	8.77	21,536,799

Table 4-4. Statistics for Cluster-delta Version Parallel Algorithm

Number of cbolts (worker processes)	Total processing time (sec)	Compute time / sync time	Sync time per batch (sec)	Avg. length of sync message
3 (1)	50377	289.18	0.54	2,525,896
6 (1)	22888	124.62	0.56	2,529,779
12 (2)	11474	58.45	0.58	2,532,349
24 (4)	6140	27.44	0.64	2,544,095
48 (7)	3333	11.96	0.76	2,559,221
96 (28)	1999	5.95	0.89	2,590,857

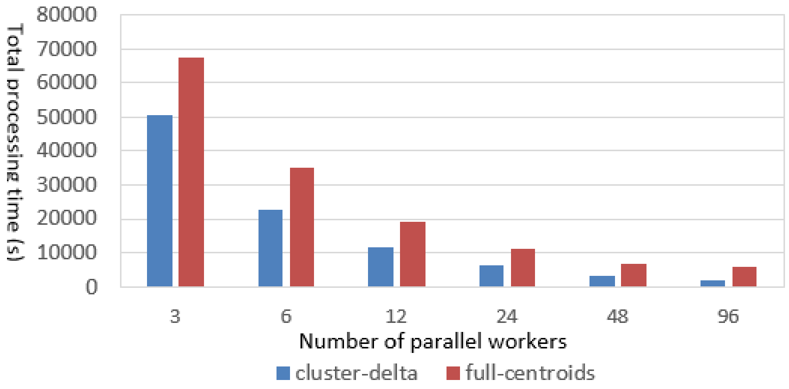


Figure 4-8. Total processing time of Cluster-delta vs. Full-centroids

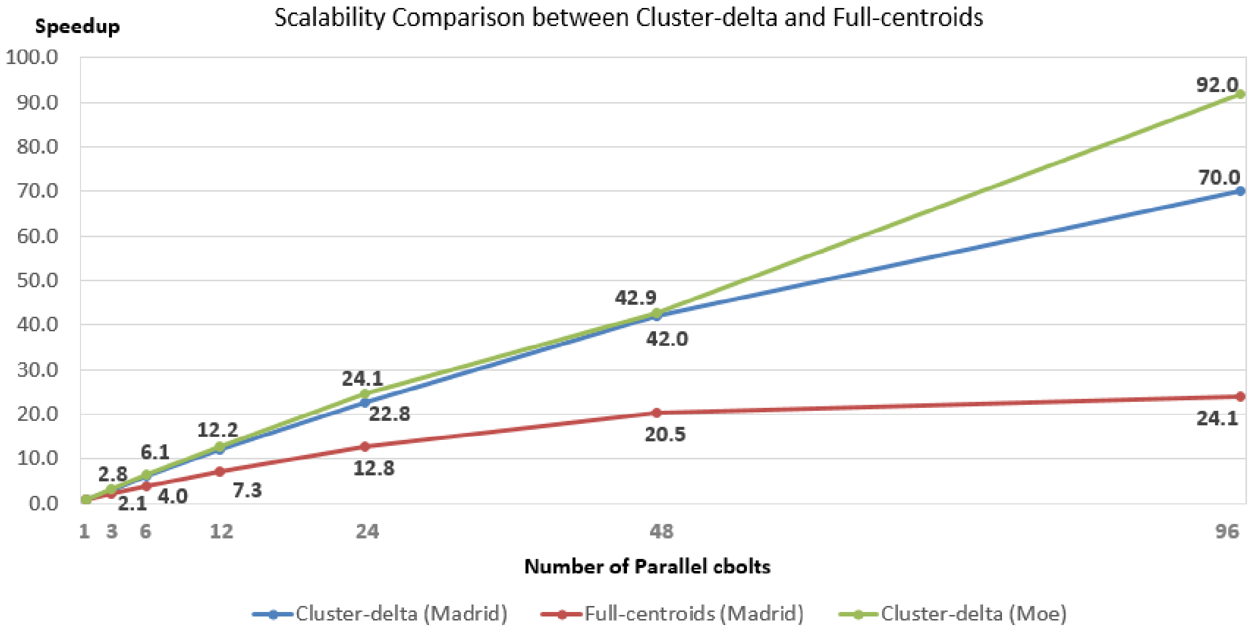


Figure 4-9. Scalability comparison between two versions of parallel implementations

Table 4-3 demonstrates that due to the large size of the cluster centroids, the full-centroids strategy generates a large synchronization message over 20MB, and incurs a long synchronization time in every batch. In addition, the synchronization time increases as the number of parallel cbolts increases, because the single ActiveMQ broker needs to send a large message to more subscribers. The total processing time for the case of 96 parallel cbolts is dominated by synchronization. As a result, the full-centroid algorithm demonstrates poor scalability, and stops getting faster after 48 parallel cbolts.

In comparison, the cluster-delta strategy generates a much smaller synchronization message and thus keeps the per-batch synchronization time at a low level, as shown in Table 4-4. The zip compression of ActiveMQ provides a compression ratio of about 1:6, so the actual message size sent over the network is less than 500KB. As the number of parallel cbolts increases, the computation time covers the major part of the total processing time for all cases. The parallel implementation using the cluster-delta strategy can achieve a near-linear scalability for up to 48 parallel cbolts. Overall, it demonstrates sub-linear scalability. Using 96 parallel cbolts, it finishes processing the 50 minutes' worth of data in 1,999 seconds (33.3 minutes), thus keeping up with and surpassing the speed of the Twitter gardenhose stream. Note that even for the case of 96 parallel cbolts, the per-batch synchronization time is still relatively low. A major reason for the relatively low speedup of 70.0 is lack of computation, because each cbolt only processes about 64 protomemes per batch. In case of longer time steps or faster data rate, it is possible to extend the near-linear-scalability zone to larger numbers of parallel cbolts by increasing the batch size. To verify this, we use a dataset containing 2,258,821 tweets for 1 hour (1:00:00 PM to 2:00:00 PM) on 2014-08-29, and run the same tests on a different computer cluster called "Moe" with better CPU and network configuration (Table 4-5). 1-2pm is the peak hour of day when gardenhose generates the most tweets. The average number of protomemes in each time step is 35,358, and we set the batch size to 12,288. The speed-ups are illustrated by the green line in Figure 4-9. Due to larger *CDELTAS* messages, the sync time per batch for 96 parallel cbolts increases to 0.979 seconds, despite the faster network. However, since the batch size is large, we are able to retain the near-linear scalability, and finish 50 minutes' worth of data in 2,345 seconds (39 minutes).

Table 4-5. Per-node hardware configuration of Moe

CPU	RAM	Hard Disk	Network
5 * Intel 8-core E5-2660v2 2.20GHz	128GB	48TB HDD + 120GB SSD	10Gb Ethernet

4.6 Conclusions

This chapter describes our contribution in the streaming analysis module of Cloud DIKW for supporting non-trivial parallel stream processing algorithms. Our research leads to some important conclusions.

Firstly, the distributed stream processing engines provide an easy way to develop and deploy large-scale stream processing applications. However, in order to properly coordinate the dynamic synchronization between parallel processing workers, their DAG-oriented processing models will need to be combined with facilitating tools such as pub-sub messaging systems. Whether such synchronization facilitating mechanisms should be directly built into the stream processing engines, as well as how this can be done, could become an interesting research issue for the distributed systems community.

Moreover, the parallelization and synchronization strategies may differ depending on the data representations and similarity metrics of the application. For example, we observed that the high-dimensionality and sparsity of the data representation in our application led to nontrivial issues for both computation and communication. By replacing the traditional full-centroids synchronization strategy with the new cluster-delta strategy, our parallel algorithm achieves good scalability, and keeps up with the speed of the real-time Twitter gardenhose stream with less than 100 parallel workers.

Chapter 5

Conclusions and Future Directions

5.1 Conclusions

As Big Data processing problems evolve, many research scenarios demonstrate special characteristics related to their data and analysis process. Social media data analysis is one such example. In this area, the data source contains not only a large historical dataset, but also a high-speed data stream generated by online users all over the world. Despite the large size of the whole dataset, most analyses only focus on smaller data subsets related to specific social events or special aspects of social activities. These characteristics raise the need for a scalable architecture that can support queries, batch analysis, and streaming analysis of social media data in an integrated way. In pursuit of that goal, this dissertation proposes Cloud DIKW, an integrated architecture that combines and extends multiple state-of-the-art Big Data storage and processing tools (Figure 1-8), and attempts to address the related research challenges in each module. Important conclusions can be drawn from our research experience in developing this architecture.

At the storage layer, we demonstrate that existing text indexing techniques do not work well for the special queries of social media data, which involve constraints on both text content and social context such as temporal or network information. To address this challenge, we leverage the HBase system as the storage substrate, and extend it with a customizable indexing framework – IndexedHBase. This framework allows users to define fully customizable text index structures that embed the exact necessary social context information for efficient evaluation of the queries. Based on this framework, we develop efficient online and batch indexing mechanisms, and a parallel query evaluation strategy. Performance evaluation shows that compared with solutions based on existing text indexing techniques provided by current NoSQL databases (e.g. Riak), our data loading strategy based on customized index structures is faster by multiple times, and our parallel query evaluation strategy is faster by one to two orders of magnitude.

In the batch analysis module, we extend IndexedHBase to an integrated analysis architecture based on YARN [154]. Two important insights were gained from our experience in developing analysis algorithms and composing analysis workflows on this architecture. First of all, indices are not only useful for query evaluation, but also valuable for analysis and mining algorithms. To explore such value, mechanisms for both random access and parallel scans of index entries are necessary. Moreover, social media data analysis workflows normally consist of multiple algorithms having different computation and communication patterns. As such, dynamically adopting diverse processing frameworks to handle different tasks is crucial to achieve efficient execution of the whole workflow.

In the streaming analysis module, we demonstrate that the high-dimensional data representation of social media data and the DAG-model organization of parallel workers in stream processing engines can pose special challenges to the problem of parallel clustering of social media data streams. To address such challenges, it is necessary to extend the stream processing frameworks with novel synchronization mechanisms. To this end, we leverage the ActiveMQ pub-sub messaging system to create a separate sychronization channel, and design a new synchronization strategy that broadcasts the dynamic changes of clusters rather than the whole centroids. Performance evaluation shows that our methods lead to much better scalability for the parallel stream clustering algorithm, and our algorithm can eventually catch up to the speed of real-world data streams with less than 100 parallel workers. By incorporating, including advanced collective communication techniques developed in our Harp project, we will be able to process the full Twitter data stream in real-time with 1000-way parallelism.

5.2 Future Work

As far as future work is concerned, there are interesting directions to explore in each module of our architecture.

For the storage layer, an important feature of our customizable indexing framework is that it could be generally implemented on most NoSQL databases. It will be interesting to extend it to other NoSQL databases and compare the performance with IndexedHBase. Inspired by the columnar storage used by both Power Drill [75], Dremel [103] and Shark [165], we can consider grouping frequently co-accessed columns in the HBase tables into separate column families to achieve more efficient queries and analysis algorithms. The query performance may also be further improved by taking data locality into consideration when launching the MapReduce jobs.

For the batch analysis module, it will be valuable to incorporate more parallel processing frameworks such as Giraph [17] and Harp [169] into the architecture, and develop more analysis algorithms that can be used in various workflows. There is on-going work attempting to extend Pig [23] to provide a high-level language for composing analysis workflows and model the analysis algorithms as basic operators in the language. But having more underlying analysis algorithms as building blocks is a pre-condition for such efforts. Additionally, domain researchers have written many legacy sequential analysis algorithms using various languages such as Python. A general mechanism that can easily parallelize such legacy codes will be very useful.

For the streaming analysis module, we will integrate advanced collective communication techniques as implemented by the Iterative MapReduce Hadoop plugin Harp [169] into Cloud DIKW, and use them to improve the synchronization performance of both batch and streaming algorithms. Instead of using a “gather and broadcast” communication model, Harp can organize the parallel workers in a communication chain, so that the local updates generated by each worker can be transmitted through all the other workers in a pipeline. According to our earlier attempts [69] to apply this technique in the Twister iterative MapReduce framework [60], it can significantly reduce the synchronization time and ensure that the algorithm achieves near linear scalability. With improved synchronization speed, we can process the data at the rate of the whole Twitter firehose stream [147], which is about 10 times larger than gardenhose. To support higher data speed and larger time window sizes, we may apply the sketch table technique as described in [7] in the clustering bolts and evaluate its impact on the accuracy and efficiency of the whole parallel program. Variations in arrival rate and jitter in event distribution exist in many real-time data streams. Therefore, we will also make the parallel algorithm elastic to accommodate this irregularity in event arrival.

Bibliography

[1] 2d Index internals. MongoDB documentation available at http://docs.mongodb.org/manual/core/geospatial-indexes/

[2] Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J., et al. The design of the Borealis stream processing engine. In Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR 2005).

[3] About data consistency in Cassandra. Apache Cassandra 1.1 documentation. Available at http://www.datastax.com/docs/1.1/dml/data_consistency

[4] About indexes in Cassandra. Apache Cassandra 1.1 documentation. Available at http://www.datastax.com/docs/1.1/ddl/indexes

[5] Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Rasin, A. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In Proceedings of the 35th International Conference on Very Large Data Bases (VLDB 2009).

[6] Advanced secondary indexes. Riak documentation. Available at http://docs.basho.com/riak/latest/dev/advanced/2i/

[7] Aggarwal, C. C. A framework for clustering massive-domain data streams. In Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE 2009).

[8] Aggarwal, C. C., Han, J., Wang, J., Yu, P. S. A framework for projected clustering of high dimensional data streams. In Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004).

[9] Aggarwal, C. C., Han, J., Wang, J., Yu, P. S. On high dimension projected clustering of uncertain data streams. Data Mining and Knowledge Discovery. 10(3): 251–273, 2009.

[10] Aggarwal, C. C., Subbian, K. Event detection in social streams. In Proceedings of SIAM International Conference on Data Mining, 2012.

[11] Aggarwal, C. C., Yu, P. S. A framework for clustering massive text and categorical data streams. Knowledge and Information Systems. 24(2): 171-196. August 2010.

[12] Aggregation pipeline limits. MongoDB documentation. Available at http://docs.mongodb.org/manual/core/aggregation-pipeline-limits/

[13] Aggregation pipeline. MongoDB documentation. Available at http://docs.mongodb.org/manual/core/aggregation-pipeline/

[14] Alonso, O., Strötgen, J., Baeza-Yates, R. A., Gertz. M. Temporal information retrieval: challenges and opportunities. TWAW, volume 813 of CEUR Workshop Proceedings, page 1-8. CEUR-WS.org 2011.

[15] Amini, A., Wah, T. Y. DENGRIS-Stream: a density-grid based clustering algorithm for evolving data streams over sliding window. In Proceedings of the 2012 International Conference on Data Mining and Computer Engineering (ICDMCE 2012).

[16] Apache ActiveMQ. http://activemq.apache.org/

[17] Apache Giraph. https://giraph.apache.org/

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

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

[20] Apache Hive. http://hive.apache.org/

[21] Apache Lucene - index file formats. Lucene documentation. Available at http://lucene.apache.org/core/3_5_0/fileformats.html

[22] Apache Lucene. https://lucene.apache.org/

[23] Apache Pig. http://pig.apache.org/.

[24] Apache Solr. http://lucene.apache.org/solr/

[25] Apache Storm. https://storm.incubator.apache.org/

[26] Apache ZooKeeper, http://zookeeper.apache.org/

[27] Bancilhon, F., Delobel, C., Kanellakis, P. Building an Object-Oriented Database System, The Story of O2. Morgan Kaufmann. June 15, 1992.

[28] Bartunov, O. Sigaev, T. Generalized Inverted Index.Presentation at PostgreSQL Summit 2006. Available at http://www.sigaev.ru/gin/Gin.pdf

[29] Becker, H., Naaman, M., Gravano, L. Learning similarity metrics for event identification in social media. In Proceedings of the 3rd ACM International Conference on Web Search and Data Mining (WSDM 2010).

[30] Bertozzi, M. Apache HBase I/O – HFile. Blog post available at http://blog.cloudera.com/blog/2012/06/hbase-io-hfile-input-output/. 2012.

[31] Broder, A. Z. On the resemblance and containment of documents. In Proceedings of the Compression and Complexity of Sequences 1997 (SEQUENCES 1997).

[32] Büttcher, S., Clarke, C. L. A. Indexing time vs. query time: trade-offs in dynamic information retrieval systems. In Proceedings of the 14th ACM International Conference on Information and Knowledge Management (CIKM 2005).

[33] Büttcher, S., Clarke, C. L. A., Lushman, B. Hybrid index maintenance for growing text collections. In Proceedings of the 29th ACM International Conference on Research and Development in Information Retrieval (SIGIR 2006).

[34] Callau-Zori, M. INDICIa: a new distributed clustering protocol. In Proceedings of the 28th ACM Symposium On Applied Computing (SAC13).

[35] Cao, F., Ester, M., Qian, W., Zhou, A. Density-based clustering over an evolving data stream with noise. In Proceedings of 2006 SIAM Conference on Data Mining (SDM 2006).

[36] Catalog tables. HBase documentation. Available at http://hbase.apache.org/book/arch.catalog.html

[37] Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A. and Gruber, R. Bigtable: a distributed storage system for structured data. In Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI 2006).

[38] Chapman, S. What Javascript can not do. Online article available at http://javascript.about.com/od/reference/a/cannot.htm

[39] Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Xing, Y., Zdonik, S. Scalable distributed stream processing. In Proceedings of 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003).

[40] Conover, M. D., Davis, C., Ferrara, E., McKelvey, K., Menczer, F., Flammini, A. The geospatial characteristics of a social movement communication network. PLoS ONE, 8(3): e55957. 2013.

[41] Conover, M., Ferrara, E., Menczer, F., Flammini, A. The digital evolution of Occupy Wall Street. PLoS ONE, 8(5), e64679. 2013.

[42] Conover, M., Gonçalves, B., Flammini, A., Menczer, F. Partisan asymmetries in online political activity. EPJ Data Science, 1:6, 2012.

[43] Conover, M., Gonçalves, B., Ratkiewicz, J., Flammini, A., Menczer, Filippo. Predicting the political alignment of twitter users. Proceedings of 3rd IEEE International Conference on Social Computing (SocialCom 2011).

[44] Conover, M., Ratkiewicz, J., Francisco, M., Goncalves, B., Flammini, A., Menczer, F. Political polarization on Twitter. Proceedings of the 5th International AAAI Conference on Weblogs and Social Media, (ICWSM 2011).

[45] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C. Furman, J., et al. Spanner: Google's Globally-Distributed Database. In Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI 2012).

[46] Danon, L., Díaz-Guilera, A., Duch, J., Arenas, A. Comparing community structure identification. Journal of Statistical Mechanics: Theory and Experiment, 2005(09): P09008, 2005.

[47] Dashpande, A., Van Gucht, D. An Implementation for Nested Relational Databases. In Proceedings of the 14th International Conference on Very Large Data Bases (VLDB 1988).

[48] Data Center Awareness. MongoDB documentation. Available at http://docs.mongodb.org/manual/data-center-awareness/

[49] Data Modeling Introduction. MongoDB documentation. Available at http://docs.mongodb.org/manual/core/data-modeling-introduction/

[50] Datasift. http://datasift.com

[51] DataStax Enterprise: Cassandra with Solr integration details. DataStax Enterprise 2.0 documentation. Available at http://www.datastax.com/dev/blog/datastax-enterprise-cassandra-with-solr-integration-details

[52] DataStax. http://www.datastax.com/

[53] Dean, J., Ghemawat, S. MapReduce: simplified data processing on large clusters. Communications of the ACM - 50th anniversary issue: 1958 – 2008. Volume 51 Issue 1, January 2008.

[54] Derczynski, L., Yang, B., Jensen, C. Towards context-aware search and analysis on social media data. In Proceedings of the 16th International Conference on Extending Database Technology (EDBT 2013).

[55] DiGrazia, J., McKelvey, K., Bollen, J., Rojas, F. More Tweets, More Votes: Social media as a quantitative indicator of political behavior. Available at SSRN: http://dx.doi.org/10.2139/ssrn.2235423. 2013.

[56] Ding, S., Wu, F., Qian, J., Jia, H., Jin, F. Research on data stream clustering algorithms. Artificial Intelligence Review. January 2013.

[57] Distributed Search. Solr Documentation. Available at https://wiki.apache.org/solr/DistributedSearch

[58] Dittrich, J., Quiané-Ruiz, J., Jindal, A., Kargin, Y., Setty, V., et al. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). In Proceedings of the 36th International Conference on Very Large Data Bases (VLDB 2010).

[59] Dittrich, J., Quiané-Ruiz, J., Richter, S., Schuh, S., Jindal, A., et al. Only aggressive elephants are fast elephants. In Proceedings of the 38th International Conference on Very Large Data Bases (VLDB 2012).

[60] Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S., Qiu, J., Fox, G. Twister: a runtime for iterative MapReduce. In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC 2010).

[61] Eltabakh, M., Özcan, F., Sismanis, Y., Haas, P., Pirahesh, H., et al. Eagle-eyed elephant: split-oriented indexing in Hadoop. In Proceedings of the 16th International Conference on Extending Database Technology (EDBT 2013).

[62] Eventual consistency. Riak documentation. Available at http://docs.basho.com/riak/latest/theory/concepts/Eventual-Consistency/

[63] Ferrara, E., JafariAsbagh, M., Varol, O., Qazvinian, V., Menczer, F., Flammini, A. Clustering memes in social media. In Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2013).

[64] Ferrara, E., Varol, O., Davis, C., Menczer, F., Flammini, A. The rise of social bots. http://arxiv.org/abs/1407.5225

[65] Ferrara, E., Varol, O., Menczer, F., Flammini, A. Traveling trends: social butterflies or frequent fliers? In Proceedings of the 1st ACM conference on online social networks (COSN 2013).

[66] Fruchterman, T., Reingold, E. M. Graph drawing by force-directed placement. Software: Practice and Experience. 21, 11 (Nov. 1991), pp. 1129-1164.

[67] Gannes, L. Twitter adopts graded approach for developer access to tweets. News article available at http://allthingsd.com/20101110/twitter-firehose-too-intense-take-a-sip-from-the-garden-hose-or-sample-the-spritzer/. 2010.

[68] Gao, X., Nachankar, V., Qiu. J. Experimenting Lucene index on HBase in an HPC Environment. 2011. In Proceedings of the 1st workshop on High-Performance Computing meets Databases at Supercomputing 2011 (HPCDB 2011).

[69] Gao, X., Qiu, J. Social media data analysis with IndexedHBase and iterative MapReduce. In Proceedings of the 6th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS 2013).

[70] George, L. HBase: the definitive guide. O'Reilly Media, Inc. September 2011.

[71] Google Cloud DataFlow. http://googlecloudplatform.blogspot.com/2014/06/sneak-peek-google-cloud-dataflow-a-cloud-native-data-processing-service.html. 2014.

[72] Graefe, G. Query evaluation techniques for large databases. ACM Computing Surveys (CSUR), 25(2): 73-169. 1993.

[73] Guo, R., Cheng, X., Xu, H., Wang, B. Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree. In Proceedings of the 16th ACM International Conference on Information and Knowledge Management (CIKM 2007).

[74] Hadoop support. Cassandra wiki page available at http://wiki.apache.org/cassandra/HadoopSupport

[75] Hall, A., Bachmann, O., Büssow, R., Gănceanu, S., Nunkesser, M. Processing a trillion cells per mouse click. In Proceedings of the 38th International Conference on Very Large Data Bases (VLDB 2012).

[76] Hartigan, J. Clustering algorithms. John Wiley and Sons, New York, 1975.

[77] HBase and MapReduce. HBase documentation. Available at http://hbase.apache.org/book/mapreduce.html

[78] Hellerstein, J. Naughton, J., Pfeffer, A. Generalized Search Trees for Database Systems. In Proceedings of the 21th International Conference on Very Large Data Bases (VLDB 1995).

[79] Hey, T., Tansley, S., Tolle, K. The Fourth Paradigm: Data-Intensive Scientific Discovery. Microsoft Research, Redmond, Washington, 2009.

[80] HIndex. https://github.com/Huawei-Hadoop/hindex

[81] How do secondary indices work? From the Cassandra users mailing group, available at http://cassandra-user-incubator-apache-org.3065146.n2.nabble.com/Re-How-do-secondary-indices-work-td6005345.html

[82] Index introduction. MongoDB documentation. Available at http://docs.mongodb.org/manual/core/indexes-introduction/

[83] IndexedHBase. http://salsaproj.indiana.edu/IndexedHBase

[84] Introduction to MongoDB. MongoDB documentation. Available at http://docs.mongodb.org/manual/core/introduction/

[85] JafariAsbagh, M., Ferrara, E., Varol, O., Menczer, F., Flammini, A. Clustering memes in social media streams. Social Network Analysis and Mining, 2014.

[86] Kamburugamuve, S. Survey of distributed stream processing for large stream sources. Technical report. Available at http://grids.ucs.indiana.edu/ptliupages/publications/survey_stream_processing.pdf. 2013.

[87] King, A. Online K-Means clustering of nonstationary data. Course project report. Available at http://coursepin.com/pin/3970. 2012.

[88] Lai, M., Koontz, E., Purtell, A. Coprocessor Introduction. Apache HBase blog post available at http://blogs.apache.org/hbase/entry/coprocessor_introduction. 2012.

[89] Lakshman, A., Malik, P. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review. 44(2): 35-40. 2010.

[90] Lancichinetti, A., Fortunato, S., and Kertész, J. Detecting the overlapping and hierarchical community structure in complex networks. New Journal of Physics, 11(3):033015, 2009.

[91] Lempel, R., Mass, Y., Ofek-Koifman, S., Sheinwald, D., Petruschka, Y., Sivan, R. Just in time indexing for up to the second search. In Proceedings of the 16th ACM International Conference on Information and Knowledge Management (CIKM 2007).

[92] Lester, N., Moffat, A., Zobel, J. Fast on-line index construction by geometric partitioning. In Proceedings of the 14th ACM International Conference on Information and Knowledge Management (CIKM 2005).

[93] Lester, N., Zobel, J., Williams, H. Efficient online index maintenance for contiguous inverted lists. Information Processing and Management: an International Journal, 42(4):916–933, 2006.

[94] Li, R., Chen, X., Li, C., Gu, X., Wen, K. Efficient online index maintenance for SSD-based information retrieval systems. In Proceedings of the 9th IEEE International Conference on High Performance Computing and Communication (HPCC 2012).

[95] Limitations with JavaScript. From online tutorial for JavaScript, available at http://cbtsam.com/jsl1/cbtsam-jsl1-012.php

[96] Lin, C. X., Ding, B., Han, J., Zhu, F., Zhao, B. Text Cube: Computing IR Measures for Multidimensional Text Database Analysis. In Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008).

[97] M. Stonebraker. The Case for Shared Nothing, IEEE Data Eng. Bulletin, Vol. 9, No.1, pp. 4-9, 1986.

[98] Map-Reduce. MongoDB documentation. Available at http://docs.mongodb.org/manual/core/map-reduce/#map-reduce-behavior

[99] MapReduce Tutorial. Hadoop documentation. Available at https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

[100] Margaritis, G., Anastasiadis, S. V. Low-cost management of inverted files for online full-text search. In Proceedings of the 18th ACM International Conference on Information and Knowledge Management (CIKM 2009).

[101] Mayo, M. Hybridizing Data Stream Mining and Technical Indicators in Automated Trading Systems. Lecture Notes in Computer Science Volume 6820, 2011, pp 79-90.

[102] McKelvey, K., Menczer, F. Design and prototyping of a social media observatory. In Proceedings of the 22nd International Conference on World Wide Web Companion (WWW 2013).

[103] Melnik, S., Gubarev, A., Long, J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T. Dremel: interactive analysis of Web-scale datasets. In Proceedings of the 36th International Conference on Very Large Data Bases, (VLDB 2010).

[104] MongoDB Glossary. MongoDB documentation. Available at http://docs.mongodb.org/manual/reference/glossary/#term-collection

[105] MongoDB. http://www.mongodb.org/

[106] Mueller, C., Gregor, D., Lumsdaine, A. Distributed force-directed graph layout and visualization. In Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization (EGPGV 2006).

[107] Multi data center replication: architecture. Riak documentation. Available at http://docs.basho.com/riakee/latest/cookbooks/Multi-Data-Center-Replication-Architecture/

[108] Neumeyer, L., Robbins, B., Nair, A., Kesari, A. S4: distributed stream computing platform. In Proceedings of 2010 IEEE International Conference on Data Mining Workshops (ICDMW 2010).

[109] Newman, M. Finding community structure in networks using the eigenvectors of matrices. Physical Review E 74, 036104 (2006).

[110] Nishimura, S., Das, S., Agrawal, D., Abbadi, A. E. MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. In Proceedings of the 16th IEEE International Conference on Mobile Data Management (MDM 2011).

[111] On distributed consistency - Part 2 - some eventual consistency forms. MongoDB blog available at http://blog.mongodb.org/post/498145601/on-distributed-consistency-part-2-some-eventual. 2010.

[112] Padmanabhan, A., Wang, S., Cao, G., Hwang, M., Zhao, Y., Zhang, Z., Gao, Y. FluMapper: an interactive CyberGIS environment for massive location-based social media data analysis. In Proceedings of 2013 Extreme Science and Engineering Discovery Environment: Gateway to Discovery (XSEDE 2013).

[113] Partitioners. Cassandra wiki. Available at http://wiki.apache.org/cassandra/Partitioners

[114] Patrick O’Neil, Dallan Quass. Improved Query Performance with Variant Indexes. In Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD 1997).

[115] Peng, D., Dabek, F. Large-scale incremental processing using distributed transactions and notifications. In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (USENIX 2010).

[116] PeopleBrowsr. http://peoplebrowsr.com.

[117] Pierce, M., Gao, X., Pallickara, S., Guo, Z., Fox, G. QuakeSim Portal and Services: New Approaches to Science Gateway Development Techniques. Concurrency & Computation: Practice & Experience, Special Issue on Computation and Informatics in Earthquake Science: The ACES Perspective. Vol. 22, Iss. 12, pp. 1732-1749, 2010.

[118] Raghavan, U., Albert, R., Kumara, S. Near linear time algorithm to detect community structures in largescale networks. Physical Review E 76, 036106 (2007).

[119] Ratkiewicz, J. Conover, M., Meiss, M., Goncalves, B., Patil, S., Flammini, A., Menczer, F. Truthy: mapping the spread of astroturf in microblog streams. In Proceedings of 20th International World Wide Web Companion Conference (WWW 2011).

[120] Ratkiewicz, J., Conover, M., Meiss, M., Gonçalves, B., Flammini, A., Menczer, F. Detecting and tracking political abuse in social media. In Proceedings of the 5th International AAAI Conference on Weblogs and Social Media (ICWSM 2011).

[121] Read preference. MongoDB documentation. Available at http://docs.mongodb.org/manual/core/read-preference/

[122] Reference for the "orderby" operator. MongoDB documentation. Available at http://docs.mongodb.org/manual/reference/operator/meta/orderby/

[123] Replica set members. MongoDB documentation. Available at http://docs.mongodb.org/manual/core/replica-set-members/

[124] Riak introduction. Riak documentation. Available at http://docs.basho.com/riak/latest/theory/why-riak/

[125] Riak. http://basho.com/riak/.

[126] Ripples. https://plus.google.com/ripple/details?url=google.com

[127] Rowe, L. A., Stonebraker, M. The POSTGRES Data Model. In Proceedings of the 13th International Conference on Very Large Data Bases (VLDB 1987).

[128] Sharding introduction. MongoDB documentation. Available at http://docs.mongodb.org/manual/core/sharding-introduction/

[129] Sharma, P., Khurana, U., Shneiderman, B., Scharrenbroich, M., Locke, J. 2011. Speeding up network layout and centrality measures for social computing goals. In Proceedings of the 4th International Conference on Social Computing, Behavioral-cultural Modeling and Prediction (SBP 2011).

[130] Shieh W.-Y., Chung, C.-P. A statistics-based approach to incrementally update inverted files. Information Processing and Management: an International Journal, 41(2):275-288, Mar. 2005.

[131] Shuai, X., Liu, X., Xia, T., Wu Y., Guo, C. Comparing the Pulses of Categorical Hot Events in Twitter and Weibo. In Proceedings of the 25th ACM conference on Hypertext and social media (HyperText 2014).

[132] Shvachko, K., Kuang, H., Radia, S., Chansler, R. The Hadoop distributed file system. In Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST 2010).

[133] SocialFlow. http://socialflow.com.

[134] Soztutar, E. Apache HBase region splitting and merging. Blog post available at http://hortonworks.com/blog/apache-hbase-region-splitting-and-merging/

[135] Stonebraker, M. Inclusion of New Types in Relational Data Base Systems. In Proceedings of the 2nd International Conference on Data Engineering. 1986.

[136] Stonebraker, M. The Design of the Postgres Storage System. In Proceedings of the 13th International Conference on Very Large Data Bases (VLDB 1987).

[137] Stonebraker, M., Madden, S., Abadi, D., Harizopoulos, S., Hachem, N., Helland, P. The end of an architectural era: (it's time for a complete rewrite). In Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB 2007).

[138] Sun, Y., Lu, Y. A grid-based subspace clustering algorithm for high-dimensional data streams. Lecture Notes in Computer Science Volume 4256, 2006, pp 37-48.

[139] Text indexes. MongoDB documentation available at http://docs.mongodb.org/manual/core/index-text/

[140] The Arabidopsis Genome Initiative. Analysis of the genome sequence of the flowering plant Arabidopsis thaliana. Nature 408, 796-815 (14 December 2000).

[141] The igraph library. http://igraph.sourceforge.net/.

[142] The R project. http://www.r-project.org/.

[143] Thomson, R., Lebiere, C., Bennati, S. Human, Model and Machine: A Complementary Approach to Big Data. In Proceedings of the 2014 Workshop on Human Centered Big Data Research.

[144] Tomasic, A., Garc ía-Molina, H., Shoens, K. Incremental updates of inverted lists for text document retrieval. In Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD 1994).

[145] Truthy. http://truthy.indiana.edu/.

[146] TwitInfo. http://twitinfo.csail.mit.edu.

[147] Twitter firehose stream. https://dev.twitter.com/streaming/firehose

[148] Twitter streaming API. https://dev.twitter.com/docs/streaming-apis

[149] Twitter trends. https://mobile.twitter.com/trends

[150] Understanding the Cassandra data model. Apache Cassandra 0.8 documentation. Available at http://www.datastax.com/docs/0.8/ddl/index

[151] Using MapReduce. Riak documentation. Available at http://docs.basho.com/riak/latest/dev/using/mapreduce/

[152] Using Search. Riak documentation. Available at http://docs.basho.com/riak/latest/dev/using/search/

[153] Using secondary indexes. Riak documentation. Available at http://docs.basho.com/riak/latest/dev/using/2i/

[154] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., et al. Apache Hadoop YARN: yet another resource negotiator. In Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC 2013).

[155] VisPolitics. http://vispolitics.com.

[156] VoltDB. http://voltdb.com/

[157] Von Laszewski, G., Fox, G., Wang, F., Younge, A., Kulshrestha, A., Pike, G. Design of the FutureGrid experiment management framework. In Proceedings of 2010 Gateway Computing Environments Workshop (GCE 2010).

[158] Weikum, G., Ntarmos, N., Spaniol, M., Triantafillou, P., Benczúr, A., Kirkpatrick, S., Rigaux, P., Williamson, M. Longitudinal analytics on web archive data: It’s about time! In Proceedings of the 5th Biennial Conference on Innovative Data Systems Research, (CIDR 2011).

[159] Weng, L., Flammini, A., Vespignani, A., Menczer, F. Competition among memes in a world with limited attention. Scientific Reports, (2) 335, Nature Publishing Group 2012.

[160] Weng, L., Ratkiewicz, J., Perra, N., Gonçalves, B., Castillo, C., Bonchi, F., Schifanella, S., Menczer, F., Flammini, F. The role of information diffusion in the evolution of social networks. In Proceedings of the 19th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD 2013).

[161] Whang, K. NoSQL vs. Parallel DBMS for Large-scale Data Management. Presentation at the Challenges in Managing and Mining Large, Heterogeneous Data Panel of 2011 Internaltional Conference on Database Systems for Advanced Applications (DASFAA 2011).

[162] Williams, D. Cassandra: RandomPartitioner vs OrderPreservingPartitioner. Blog post available at http://ria101.wordpress.com/2010/02/22/cassandra-randompartitioner-vs-orderpreservingpartitioner/. 2010.

[163] Write concern. MongoDB documentation. Available at http://docs.mongodb.org/manual/core/write-concern/

[164] Wu, G., Boydell, O., Cunningham, P. High-throughput, Web-scale data stream slustering. In Proceedings of the 4th Web Search Click Data workshop (WSCD 2014).

[165] Xin, R., Rosen, J., Zaharia, M., Franklin, M., Shenker, S., Stoica, I. Shark: SQL and rich analytics at scale. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD 2013).

[166] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., Stoica, I. 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 (NSDI 2012).

[167] Zaharia, M., Das, T., Li, H., Shenker, S., Stoica, I. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 2012).

[168] Zezeski, R. Boosting Riak search query performance with inline fields. Blog post available at http://basho.com/boosting-riak-search-query-performance-with-inline-fields/. 2011.

[169] Zhang, B., Ruan, Y., Qiu, J. Harp: Collective Communication on Hadoop. To appear at the Proceedings of 2015 IEEE International Conference on Cloud Engineering (IC2E2015).

[170] Zobel, J. Moffat, A. Inverted files for text search engines. ACM Computing Surveys, 38(2) - 6. ACM New York, 2006.