Analytics for the Real-Time Web

Maxim Grinev Maria Grineva Martin Hentschel Donald Kossmann
Systems Group, Dept. of Computer Science, ETH Zurich, Switzerland
{grinevm, grinevam, hemartin, donaldk}@inf.ethz.ch

ABSTRACT

With the emergence of mobile devices constantly connected to the Internet, the nature of user-generated data has changed on most Web 2.0 sites. Today, people produce and share data more often and the lifespan of the data is shorter. Analyzing this data leads to new requirements for analytical systems: real-time processing and database-intensive workloads. Driven by these requirements, we have developed a new system for real-time analytics. Our system extends a key-value store, Cassandra, with push-based processing, transactional task execution, and synchronization. To demonstrate our system, we have built a service to reorganize news sites using real-time feedback from social media.

1. INTRODUCTION

The Web 2.0 era is characterized by the emergence of large amounts of user-generated content. So far, analyzing and making use of this data has been accomplished using batch-style processing. Data produced over a certain period of time is accumulated and then processed. Analytical processing consists mostly of computing aggregate values or generating indexes for data inspection. MapReduce [4] has become the state of the art for analytical batch processing of user-generated data.

Today, with the growing use of mobile devices constantly connected to the Internet, the nature of user-generated data has changed: it has become more real-time. People share their thoughts and discuss breaking news on Twitter and Facebook; they share their current locations and activities on location-based social networks such as Foursquare. The difference is that, today, people share more often and the lifespan of the data has become shorter.

This change implies new requirements for analytical systems. Processing data in batches is too slow to analyze data in real time. Accumulated data can lose its importance in several hours or, even, minutes. Therefore, analytical systems must aggregate values incrementally, as new data arrives. It follows that workloads are database-intensive because aggregate values are not produced at once, as in batch processing, but stored in a database constantly being updated.

Driven by these requirements, real-time and database-intensive, we have developed a system for real-time analytics. It extends a distributed key-value store, Cassandra [9], with push-style processing, transactional task execution, and synchronization. Push-style processing allows to immediately propagate the data to the analytical computations. Transactional task execution guarantees exactly-once execution in case of node failures. Synchronization is needed to ensure consistency of aggregate results. As we will discuss below, our system preserves the main advantages of the original, batch-oriented MapReduce framework, namely its programming model (adapted to push-style processing), fault tolerance, and scalability.

To demonstrate our system, we have developed a service for news site optimization using social media. As more than 40% of news are now shared via social networks [2], news companies are interested to understand the impact produced by recently published news. With real-time feedback from users of social networks, news companies can reorganize their front pages placing most discussed stories first and, thus, attract more readers.

2. SYSTEM

In this section we describe our real-time analytics system including programming model, execution model, fault tolerance, and scalability.

Programming Model. MapReduce [4] is a well-established programming model to express analytical applications. To support *real-time* analytical applications, we modify this programming model to support push-style data processing. In particular, we modify the *reduce* function. Originally, *reduce* combines a list of input values into a single aggregate value. Our modified function, *reduce**, incrementally applies a new input value to an already existing aggregate value. This modification allows to apply a new input value to the aggregate value as soon as the new input value is produced. This means, we are able to *push* new values to the *reduce** function. Figure 1 depicts our modified programming model. *reduce** takes as parameters a key, a new value, and the existing aggregate value. It outputs a key-value pair with the same key and the new aggregate value. Note that the aggregate value can be of complex type (e.g., a sorted list). We did not modify the *map* function as it already allows push-style processing. The difference between

Proceedings of the VLDB Endowment, Vol. 4, No. 12

$$map: \quad (k_1, v_1) \rightarrow list(k_2, v_2)$$
$$reduce^*: \quad (k_2, v_2, agg_{old}) \rightarrow (k_2, agg_{new})$$

Figure 1: Programming model.

map and *reduce** is that multiple *map*s can be executed in parallel for the same key, while the execution of *reduce** has to be synchronized for the same key to guarantee correct results. In order to setup a *map*/*reduce** job the developer has to provide implementations for both functions and define the input table, from which the data is fed into *map*, and the output table, to which the output of *reduce** is written.

Note that *reduce** exhibits some limitations in comparison to the original *reduce*. Not every reduce function can be converted to its incremental counterpart. For example, to compute the median of a set of values, the previous median and new value is not enough to compute the new median. To solve this issue, the complete set of values needs to be stored to compute the new median.

Execution Model. The incremental nature of real-time analytics requires intensive updates to a data store. Therefore, we built our system on top of a key-value store, which also provides fault tolerance and scalability.

There are three main assumptions about the underlying key-value store required to implement our approach. (1) The key-value store is partitioned and replicated across nodes according to some strategy. Any node of the store may be contacted to insert data. (2) Each node has a reliable and persistent storage. Persistent storage is needed for write-ahead logging. When a node recovers from a failure, it will read the log and continue in the state it was in before the failure. This assumption significantly simplifies implementing exactly-once semantics as we do not need to redundantly distribute and coordinate map/reduce* task execution—there is always a single node responsible to execute each task. We still replicate input and aggregate values to provide non-blocking execution of input data in case of node failures. (3) The system supports a quorum-like consistency protocol to update replicas, as for example supported in Amazon Dynamo [5]. Data can be consistently read and written from the database as long as the majority of replicas are available. It allows processing data when some of the nodes are down or not available. Note that these requirements are quite general and are supported by several databases; for example, by Apache Cassandra used to implement our system [9].

Based on these assumptions, we implemented additional mechanisms on top of the key-value store. (1) We extended the nodes of the key-value store with queues and worker threads. Each node maintains a queue that buffers map and reduce* tasks. Queues are persistent and transactional such that tasks can be durably stored or removed as part of a distributed transaction. Multiple worker threads drain the queues and execute buffered tasks in parallel. Buffering map and reduce* tasks allows to handle bursts of input data. Furthermore, the size of the queue allows a rough estimation of the load of a node. (2) We implemented a two-phase commit protocol in order to execute distributed transactions. Distributed transactions are used to ensure *exactly-once* semantics of map and reduce* tasks. (3) We added mechanisms to synchronize the execution of reduce* tasks. Reduce* tasks need to be synchronized because several tasks can potentially update the same aggregate value in parallel leading to inconsistent data. Synchronization is realized in two steps: by routing all key-value pairs output by *map* with the same key to a single *reduce** node; and by synchronizing the execution of *reduce** within a node using locks. Routing is implemented reusing the database's partitioning strategy—the primary replica storing the corresponding aggregate value is chosen. In case this primary replica fails, the output of *map* is routed to another replica that is alive. Therefore, synchronization between nodes is required. This is achieved using fixed-length leases [7]. The system has a central master node that serves as lease manager. Database nodes acquire leases from this manager. Leases expire after a fixed period of time (30 seconds in our case) and need to be renewed. Only the node that obtained the lease for the corresponding key range is allowed to execute reduce* tasks. In the normal case, the primary replica node already acquired the lease for its key range and can process reduce tasks for all keys routed to the node. In the failure case, when key-value pairs are routed to another replica, this replica must acquire the lease for the key range before starting to process these pairs.

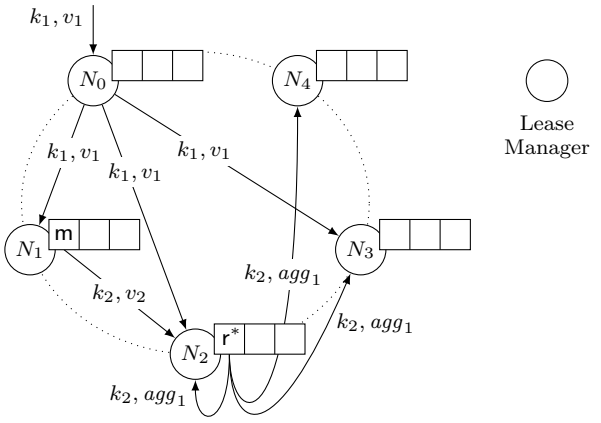

Figure 2: Execution of an initial insert and one *map*/*reduce** step (Replication shown).

The detailed steps to execute *map* and *reduce** are shown in Figure 2 and described below.

How to Insert Data. Whenever a new key-value pair is inserted into the system, the contacted node runs a distributed transaction. The transaction consists of three steps. First, the key-value pair is replicated to the majority of the nodes in the replication set. Second, one node among the replication set is elected to be the coordinator to execute the map task. As a result of these two steps, the key-value pair has been stored in a replicated fashion and exactly one node will execute the map task. Choosing different replicas to execute map tasks allows to balance load across replica nodes. Third, at the elected coordinator node, the map task is put into the queue. In Figure 2, the contacted node N_0 replicates the key-value pair k_1, v_1 and elects node N_1 as coordinator to execute the map task. The queue of N_1 contains the map task (denoted as m).

How to Execute map. Eventually, a worker thread executes the map task. The execution of the map task is also a distributed transaction. First, the map task is removed from the queue. Second, each key-value pair output by *map* is written to the corresponding *reduce** node as described above. And third, at those nodes a reduce* task is put into

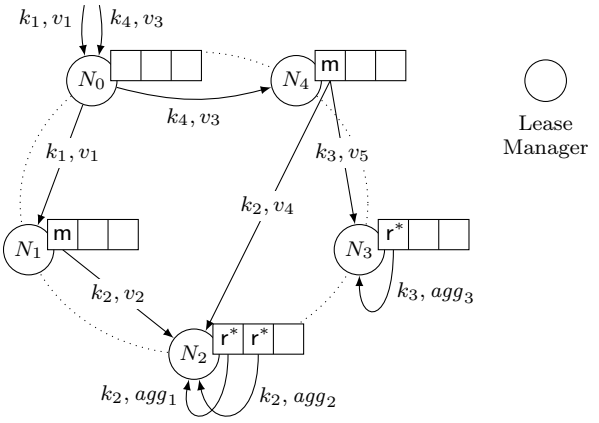

Figure 3: Synchronization of reduce* tasks (No replication shown).

the queue. In Figure 2, node N_1 executes the map task and writes k_2, v_2 to the responsible node N_2. The queue of N_2 contains the reduce* task (denoted as r*).

How to Execute reduce*. In Figure 3, we illustrate that output of *map* with the same key (k_2, v_2 and k_2, v_4) is routed to the same node (N_2) to ensure synchronization as described above. During execution of a reduce* task the worker thread reads the old aggregate value from the majority of nodes in the replication set. The output of the task is stored back to the majority of nodes in the replication set. The execution of a reduce* task is a distributed transaction to ensure that the output is replicated to the majority of the nodes in the replication set and the task is removed from the queue. In Figure 2, node N_2 executes the reduce* task and replicates its results. By writing the result, the node might fire a subsequent map/reduce* task for which the current node is the coordinator to execute the subsequent *map*.

Fault Tolerance. Our system tolerates node failures and provides the following two properties.

(1) Non-blocking execution. Similar to the original, batch-oriented MapReduce framework [4], we continue to make progress in case of node failures. In [4], if a node goes down the master node only assigns new jobs to alive nodes. Also, the master re-assigns the jobs of the failed node to nodes that are alive. In our implementation, if a node goes down and there is still a majority of nodes of the replication set alive, one of these alive nodes will execute new jobs. Only the majority of nodes of the replication set is needed to make progress. Our system blocks if the majority is lost and as long as majority is reached again. Jobs that were buffered in the queue when the node went down, will be executed whenever the node recovers from the failure. If the lease manager dies, the system stops working until the lease manager recovers. This single point of failure is similar to the master node in the original MapReduce framework. A solution is to make the lease manager fault tolerant by replicating its state.

(2) Exactly-once semantics. We provide exactly-once semantics by combining atomic task execution with durable storage of intermediate results. Each step of data processing is implemented as an atomic transaction which reads a task from a persistent queue, executes it, and durably stores the task to execute the next step into a subsequent queue. In case of any failure the transaction is rolled back and thus the initial task remains in the first queue. Atomic transactions are achieved via the two-phase commit protocol. Furthermore, there is always one single node chosen to execute a map or reduce task.

Scalability. In our system, the task execution is distributed across the nodes according to the data partitioning strategy. It allows to scale the system by adding nodes to the system and moving data (and thus, task execution) onto the newly added nodes. Distributed transactions do not prevent scalability because the number of nodes involved in a single distributed transaction is determined by the replication factor (= 3 in Figure 2) and does not depend on the size of the key-value store.

The proposed technique allows scaling when the overall load is distributed across many aggregate values with different keys. If a single aggregate value becomes a hotspot (e.g., a global counter reflecting the overall number of messages in a social network), our solution can be extended by aggregating values in a distributed fashion [11]. In [11], the aggregate value is replicated and load is shed across the replicas. The final result is obtained by combining the replicated aggregate values at query time.

3. RELATED WORK

Data warehouses and MapReduce [4] represent the state of the art in data analytics. They are designed for batch processing and do not meet the new requirements of analytics for the real-time web.

Google Percolator [12] is a recent example of a system to support real-time analytics. Percolator is similar to our work as it extends a distributed database, Google BigTable, with (1) push-based processing using database triggers (called observers in the paper), and (2) distributed transactions using the two-phase commit protocol and snapshot isolation. There are also important differences. First, the system provides at-most-once semantics of trigger execution. It is argued to be sufficient for the system's main application, Web indexing, because skipping some tasks still leads to statistically correct results of the PageRank computation. Our system guarantees exactly-once semantics and thus can be applied for a wider range of applications. Second, our system supports the (modified) MapReduce programming model, which has proven to be effective to express many kinds of analytical applications. In contrast to this, Percolator supports the low-level programming model of triggers. Moreover, there are a number of differences in the implementation of task distribution and synchronization, data replication, and transactions. A detailed analysis of these implementation differences is outside the scope of this paper.

The Hadoop Online Prototype (HOP) [3] is an approach to transform the MapReduce framework to support real-time analytics. HOP proposes a modified MapReduce architecture that allows a pipelined execution of operators. It still processes data in batches but users can see early approximations of the final result.

Recent research combines data stream management systems with databases to build real-time analytical applications, for example Truviso [6]. Compared to our system, Truviso does not distribute the computation and, therefore, cannot maintain large numbers of aggregate values (e.g., the popularity of links on social networks for *all* sites on the web). There are distributed stream management systems, such as IBM InfoSphere Streams [8] and Borealis [1], but

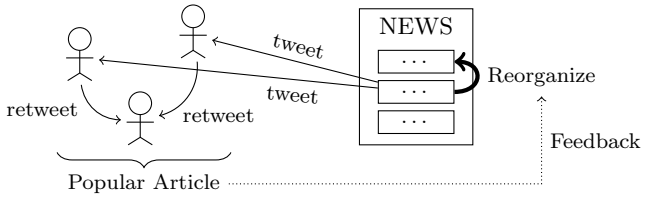

Figure 4: Reorganizing a newspaper using Twitter.

these do not provide database integration. In these systems, data is accumulated in windows. In analytical applications the size of a window is often not known in advance. According to our experience with these systems, expressing analytical computations with unknown window sizes results in complex application logic. Neither IBM InfoSphere nor Borealis provide persistent storage for windows that do not fit in main memory. Yahoo! recently developed the S4 streaming system [10]. S4 is built for real-time distributed analytical applications. It provides a programming model similar to MapReduce: data is routed between tasks by keys. But, S4 lacks database integration and does not provide fault-tolerance guarantees. In summary, our study on popular stream management systems (Truviso, IBM InfoSphere, Borealis, and Yahoo! S4) shows that none of these systems meet all requirements of real-time analytics provided by our system (database integration, distributed processing and scalability, fault tolerance, and expressiveness).

4. DEMONSTRATION

We will demonstrate our system using real data obtained from Twitter. Twitter allows to access a 1% sample of all current tweets in real time [14]. We will use this data as is to demonstrate real-time analytics. To demonstrate the scalability of our system, we artificially scale this workload to the 100% Twitter workload (1600 messages per second [13]) using data accumulated over seven days.

Application. We have built a service for news site optimization based on information posted by people on Twitter, illustrated in Figure 4. Using our service, news companies can monitor the popularity of published articles and reorganize their front pages featuring most discussed stories first. Hopefully, this will attract more readers.

Our service analyzes links in tweets. Many tweets contain links to news articles and blogs [2]. Typically, these links are abbreviated by URL shorteners such as *bit.ly*. We implement this application via two consecutive map/reduce* tasks. In our implementation, the first map/reduce* task extracts and resolves URLs from tweets and counts the number of times each URL is posted. The second map/reduce* task extracts domains from URLs, groups URLs by their domain, and keeps a sorted list of URLs based on their count (for each group). As result, the user can look up the most popular articles (i.e., their URLs) for each domain mentioned on Twitter. Thus, we demonstrate web-scale processing.

In all scenarios described below, we use a cluster of machines where each machine is equipped with an AMD Opteron 2.4GHz CPU and 6GB RAM running Ubuntu Server 10.

Scenario 1: Monitoring. A user will access our service through a standard web browser. The start page displays a text field in which the user can enter any domain, for example nytimes.com. For this domain, the service will then display a sorted list of URLs that are currently discussed on Twitter. The list is constantly updated in the browser as new tweets are received, which demonstrates the real-time character of our application. It allows the user to see the most popular articles of the New York Times on Twitter right now. The user can reset the displayed list for the domain by clicking a reset button. When the user clicks this button, all statistics are set to zero. In an instant though, the list will be populated again as new tweets are received.

Scenario 2: Scalability. In order to demonstrate the scalability of our system, we will artificially scale up the input stream to mimic a 100% Twitter load. That is, we replay data accumulated over seven days in a much shorter time frame. We will show that our system is able to handle this load on a cluster of 12 machines. The user will see that URL counters increase much quicker with the scaled-up input stream.

Scenario 3: Fault Tolerance. In order to demonstrate non-blocking execution of map/reduce* tasks in case of node failures, we will give the audience the chance to pick any node of the system that we will manually shut down by killing the corresponding Java process. To support this scenario, our system logs all fatal errors during task execution. Using the Linux command `tail -f error.log` (which outputs all errors to a console), we can show that no such errors occur when shutting down a node of the system. The audience may choose to kill more nodes, which at some point will cause fatal errors to appear in the logs. This happens when the majority of nodes of a replication set is down.

5. REFERENCES

[1] D. J. Abadi et al. The design of the Borealis stream processing engine. In *CIDR*, pages 277–289, 2005.
[2] CNN. Study into the power of news and recommendation. http://cnninternational.presslift.com/socialmediaresearch, 2010.
[3] T. Condie et al. Online aggregation and continuous query support in MapReduce. In *ACM SIGMOD Conf.*, pages 1115–1118, 2010.
[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
[5] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.
[6] M. J. Franklin et al. Continuous analytics: Rethinking query processing in a network-effect world. In *CIDR*, 2009.
[7] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP*, pages 202–210, 1989.
[8] InfoSphere streams. http://www-01.ibm.com/software/data/infosphere/streams, 2011.
[9] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.
[10] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDM Workshops*, pages 170–177, 2010.
[11] J. Oskarsson and K. Kakugawa. Increment counters. http://issues.apache.org/jira/browse/CASSANDRA-1072, 2010.
[12] D. Pen and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, pages 215–264, 2010.
[13] C. Penner. #numbers. http://blog.twitter.com/2011/03/numbers.html, 2011.
[14] Twitter Streaming API documentation. http://dev.twitter.com/pages/streaming_api, 2011.