Domino: An Incremental Computing Framework in Cloud with Eventual Synchronization

Dong Dai[1], Xuehai Zhou[2], Dries Kimpe[3], Rob Ross[3], and Yong Chen[1]

[1]Computer Science Department, Texas Tech University, dong.dai@ttu.edu, yong.chen@ttu.edu

[2]Computer Science College, University of Science and Technology of China, xhzhou@ustc.edu

[3]Mathematics and Computer Science Division, Argonne National Laboratory, {dkimpe, rross}@mcs.anl.gov

ABSTRACT

In recent years, more and more applications in cloud have needed to process large-scale on-line data sets that evolve over time as entries are added or modified. Several programming frameworks, such as Percolator and Oolong, are proposed for such incremental data processing and can achieve efficient updates with an event-driven abstraction. However, these frameworks are inherently asynchronous, leaving the heavy burden of managing synchronization to applications developers. Such a limitation significantly restricts their usability. In this paper, we introduce a trigger-based incremental computing framework, called Domino, with a flexible synchronization mechanism and runtime optimizations to coordinate parallel triggers efficiently. With this new framework, both synchronous and asynchronous applications can be seamlessly developed. Use cases and current evaluation results confirm that the new Domino programming model delivers sufficient performance and is easy to use in large-scale distributed computing.

Categories and Subject Descriptors

D.4.7 [**OPERATING SYSTEMS**]: Organization and Design—*Distributed systems*

Keywords

Cloud Computing, Incremental Computing, MapReduce

1. INTRODUCTION

Continuous data streams are of increasing importance to real-world analysis today. In order to process these data streams, incremental approaches are often more efficient. Popular batch-processing programming models, like Dryad and MapReduce, provide only shallow support for incremental processing. Although there are extensions based on the batch-processing models to support incremental applications [5, 4], it is hard to eliminate the unnecessary processing on unchanged data and difficult to choose the right time to process new data streams, since such models are not aware of incremental streams.

In order to address these challenges, event-driven models have emerged in cloud computing [6, 7]. Event-driven applications are triggered by external data streams to process new updates. Most event-driven abstractions are designed and implemented based on triggers or observers to handle data updates, like Percolator [7] and Oolong [6]. In a distributed environment, these triggers/observers will run concurrently on different servers. Synchronization of these concurrent triggers/observers then becomes an important issue. However, most of them consider themselves as pure asynchronous frameworks without providing any synchronization mechanisms [6], or just provide write transactions to keep data consistency instead of synchronizing different executions [7]. This drawback has significantly limited their usability.

In this paper, we present Domino, a *trigger-based incremental programming framework with a flexible synchronization mechanism* to address these issues. To our best knowledge, Domino is the first system that provides flexible synchronization based on a trigger-based distributed programming model: it supports both *natural* asynchronization, which is brought by the trigger-based model inherently, and new *wait-free* eventual synchronization. Hence, it can support a broad range of applications including most MapReduce applications and asynchronous algorithms.

2. MOTIVATION

Many Cloud applications usually contain iterative procedures. Among different iterations, accumulation is used to gather partial results from last iteration, which usually leads to a global synchronization. How to handle these synchronizations could be very complex and challenging in any event-driven computing models. Figure 1 shows an example that an accumulation needs to synchronously gather partial results from four triggers. The most straightforward way is using distributed locks to enforce the triggers wait until all of them finished, as Fig. 1(a) shows. However, the challenge is that in the incremental computing scenario, we do not know how many updates will be there. As shown in Fig. 1(a), the first *Sync* could already generate correct accumulation results because C and D will not execute in the future, but distributed locks require waiting until trigger

HPDC'14, June 23–27, Vancouver, BC, Canada.

http://dx.doi.org/10.1145/2600212.2600705.

C and D finish. Another possible solution is using a *time window*, which defines a fixed waiting time (t_w) and makes the accumulation progress each t_w as Fig. 1(b) shows. However, choosing t_w itself would be a challenge for developers. Moreover, it still faces the problem of not knowing when the synchronization should return.

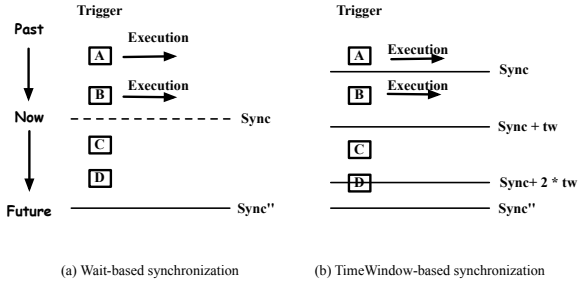

Figure 1: Synchronization in incremental scenario.

Our goal is to develop a programming model for incremental applications with a synchronization mechanism that avoids unnecessarily waiting for the uncertain future trigger executions, and frees developers from choosing synchronization intervals or managing the return.

3. DOMINO MODEL

In Domino, we abstract all data sets as *sparse tables*. The continuous data streams can be viewed as *insert* or *update* on tables. This table-based data model is similar to the Bigtable data model and has been proven to be capable of describing most of the data sets in cloud applications. Similarly, the sparse tables in Domino also provides concept of column-family, which gathers all the similar columns and is guaranteed to be stored on the same server. Besides, each cell in the sparse table contain multiple versions of data.

Based on *sparse tables* data model, the Domino programming model consists of three components: events, conditions, and actions. An *event* is generated from the continuous external data streams, which could be any operations on the sparse tables. To detect the events, applications need to declare which columns, column-families, or tables they are monitoring. A *condition* is used to filter events in order to control the execution of *actions*. It is a user-defined function that returns *true* or *false* to denote whether current event should be processed or not. One of the most important uses of conditions is to stop an iterative execution of triggers. An *action* is the real logic of a user's application. It consumes the events and writes the results back persistently. Actions always run on the server where event is fired. This locality of action execution significantly improves the performance and reduces the possibility of network congestion.

Combing these three parts, we form a *trigger*, which is the core concept of Domino: all the applications in Domino are written as a series of triggers. One trigger maybe run at different servers concurrently. Each run is called a *trigger instance*. Domino has three types of triggers: plain triggers, asynchronous accumulator triggers, and synchronous accumulator triggers.

Plain trigger is the simplest case. It responds to new data streams and executes independently in different servers in parallel. Other two triggers are both used to accumulate partial results from other triggers. For each accumulator trigger, Domino runtime will automatically create a table with only one column-family (*partial-results*). This new table is invisible to other applications so that it avoids misbehaviors from shared users. The *asynchronous accumulator trigger* represents the natural asynchronous semantic: partial results arrive and activate the execution of accumulator trigger without any coordination. The *synchronous accumulator trigger* provides a self-managed eventual synchronization mechanism between different triggers. The eventual synchronization avoids unnecessary global blocking and helps the applications make progress. It is also simpler to use because developers do not need to explicitly set up global locks or barriers.

Based on the concepts and definitions of Domino, we demonstrate the implementation of incremental PageRank algorithm, which iteratively calculates the priority of web pages based on links to them. Table *WebRepo* (Table. 1) shows the whole web repository with url as row key. Column *Meta:Rank* column stores current PageRank value, *Meta:OutEdges* represents all the out edges of current page, *Cot:En* stores the actual contents of a page.

Table 1: WebRepo Table

RowKey	*Meta:Rank*	*Meta:OutEdges*	*Cot:En*
url_1	1.0	$url_{11}, url_{12}, ...$	...
url_2	1.0	$url_{112}, url_{21}, ...$	...
...	...	... , ...	...

Incremental PageRank needs two triggers to work. The first trigger is a plain trigger monitoring on the *Meta* column-family in Table 1: whenever the *OutEdges* or *Rank* value changes, it will execute to calculate all the outgoing edges' weights of the that page. Those updates are written to the second trigger, which is a synchronous accumulator trigger. It accumulates all the incoming edges' weights to form a new PageRank value for that page, which will be written into the *Meta:Rank* column of Table 1 and activate the next round execution. To stop iterations, we can check whether the new PageRank value is close enough to the old one as a condition in the first trigger. This incremental version of PageRank needs strong synchronization during accumulating. Based on our Domino abstraction, developers can easily set accumulator trigger as a synchronous trigger, which will execute in an efficient wait-free way.

4. DESIGN AND IMPLEMENTATION

Domino was tightly integrated with HBase [1] in the current implementation. However, it can work well with other distributed storage systems as long as they support the sparse table data model with version capacity and persistence guarantee. The Domino instance runs along with the HBase instance on each server. There is one TriggerMaster node performing trigger management and collocated on the HMaster node. Other nodes, namely TriggerWorkers, run on these nodes along with the HRegionServer. The major components of Domino include event detector, trigger manager, scheduler, and gathered I/O.

Event detector detects updates on *sparse table*. It intercepts the core execution path of Write-Ahead-Log (WAL) appending in HBase. It will build an event object containing the information collected from the logs and send it to the local event queue. Events queue manages all the local fired

event objects. Inside this queue, events are ordered by their keys. A key is a vector consisting of the monitored table name, column-family, and column. Since different triggers are allowed to monitor on the same dataset, a fired event may cause multiplex triggers to execute. There is a consumer waiting on each event queue. Whenever an event is appended, it will be notified and send this event to the corresponding trigger actions. The event then will be processed by the user-defined action function. For execution, each action function is attached to a preallocated thread. All action threads are managed by the action thread pool component.

Gathered I/O component encapsulates all the data accesses in action functions into a delegate to accelerate I/O accesses. *Gathered input* is for accumulator triggers only: the Domino framework will automatically collect all the accumulated partial results while processing accumulating events. The gathered inputs contain all the columns' values with the proper id_p and id_r number (described later) in the *partial-results* column-family. For accumulator triggers with wait-free eventual synchronization, the proper versions refer to less than or equal to the fired data versions; for other accumulator triggers, they would be the latest value based on timestamps. *Gathered output* is general for all triggers. It is a per-server component: in each trigger, all the data is written into in-memory cache first instead of the HBase tables. Domino combines cached writes and flushes them later according to their initial order of calling the append method.

Eventual synchronization integrated with accumulator triggers concepts plays a critical role to guarantee the final result will be eventually synchronized and overwrite incomplete results with the asynchronous partial results. The eventual synchronization in Domino includes two parts: the accumulator triggers and the version management. For each accumulator trigger, Domino implicitly creates an invisible HBase table as Fig.2 shows. It contains a predefined column-family named *partial-results*, which is automatically monitored by the newly created accumulator trigger instance. Just like the other table, all writes to accumulator triggers should contain a row key (r_i), column indicator (c_j), and value with different versions (v). All the accumulator triggers execute in the same way except that the synchronous accumulation triggers need to choose data with the right version while accumulating. This strategy makes sure that the partial results from stragglers can also be accumulated with the right data in a synchronous way.

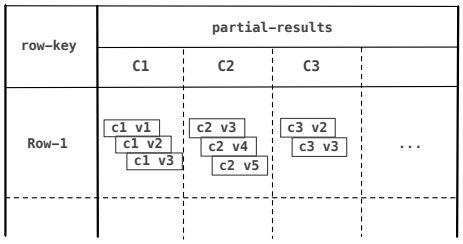

Figure 2: Invisible table for accumulator trigger.

To describe the executions of incremental applications, we categorize them into three types: iterative only, incremental only, and both iterative and incremental applications. In iterative only applications, each trigger simply writes into dataset monitored by itself and causes itself to run again. In this case, we can use a auto-increase index (id_r) to distinguish different execution rounds. In incremental only applications, triggers are activated only by external inputs. The id_r will remain the same, whereas external inputs are changing. We can use another index (id_p) to trace the inputs. For executions of both iterative and incremental, both the id_r and id_p change.

The sparse table in Domino stores multi-version data for each cell, which allows us to trace the executions status (id_r and id_p) to restrict the data access. First, we initialize id_r and id_p for each execution based on the version values stored with the data that triggers the execution. Each time a trigger execution writes into HBase, the Domino framework will merge its id_r and id_p into a version number ($id_p : id_r$) and write into HBase table. If this data activates another trigger execution, then the new execution will automatically read this version number, parse it into id_p and id_r, and set its new id_r^n equal to id_r+1. The id_p changes only if external inputs are written into a data cell, then Domino read the newest id_p, set a new id'_p=id_p+1, and initialize id_r equal to 0 for the new input. With these versions, Domino restricts the execution of accumulator triggers and makes sure they only read data with right version. In this way, Domino guarantees that when the last intermediate values are updated in the *partial-results* column family, the accumulator trigger action will read the same inputs as a strict synchronization action does and eventually return the correct results.

5. EVALUATIONS

The evaluation of the Domino system was based mainly on a 12-node cluster. We also evaluated Domino on the Amazon EC2 platform using up to 64 m1.medium nodes.

The Domino framework supports a large range of applications. In this section, we evaluate two typical applications: WordCount and PageRank. All evaluations were run on our local 12-node cluster. Since Percolator is not public, we can not make direct comparisons with it. In this evaluation, we mimic the behavior of Percolator based on open source software stacks currently accessible. Specifically, we use HBase Coprocessor to mimic the observers of Percolator and use the combination of Omid [2] and ZooKeeper [3] to simulate the transactions and distributed locks.

The most critical performance measurement of incremental applications is how fast they can absorb the new inputs. The faster the external data streams come, the higher the pressure that is placed on the underlying infrastructure. For different evaluations, we created a large number of static input datasets and different changing data sets. These changing datasets were written into the underlying HBase storage system as fast as possible. We also ran the MapReduce applications as a comparison. Running MapReduce on the smaller changing datasets shows the best performance MapReduce can achieve because they only process the updated data. Running MapReduce on the whole dataset shows the baseline performance and also represents the response time that current batch-based frameworks can achieve in incremental scenarios.

In the following evaluation figures, Domino indicates the Domino performance; MR(1R) means one round of MapReduce for iterative applications; MR(FR) means the whole execution time (full round) for iterative applications; MapReduce(Whole) means the execution time of applying MapReduce on the whole data set; MapReduce means that MapRe-

duce was executed on the changing data sets; and Mimic(P) represents our mimic Percolator.

Figure 3 shows the performance of WordCount with different data sizes. We can see that Domino achieves slightly better performance than MapReduce running only on the changed dataset. The Mimic Percolator version does not scale well because it does not provide the *gethered I/O* optimization, so during execution, it issues too many small writes to HBase. Compared with the MapReduce running on the whole dataset, Domino clearly has much better performance. Although in WordCount we can deploy MapReduce only on the changed dataset to obtain right results, the frequency of running MapReduce is hard to decide and difficult to manage with different stream speeds. Domino, on the other hand, provides an easy way to run it in incremental situations, and it achieves sufficient performance.

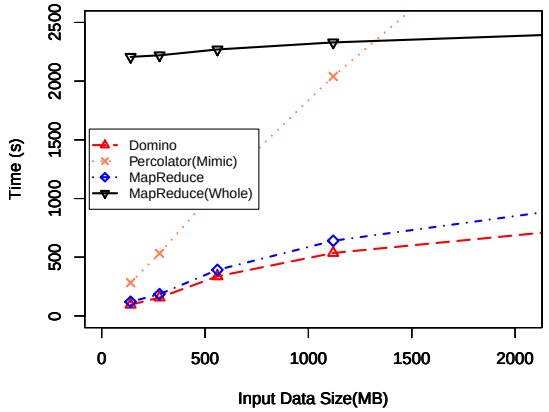

Figure 3: WordCount under different sizes.

The synchronous PageRank application runs on a synthetic web repository with increasing input size. PageRank is iterative, so in order to compare different implementations fairly, the MapReduce version needs to run enough times to make sure convergence of every page is met. Therefore, we also show the execution time of one MapReduce iteration (MR(R1)) in the result as a hint. The Mimic-Percolator does not support synchronous accumulation in its design. In this evaluation, therefore, we mimic such supports by setting up a time window to wait until the lock is released and continue to accumulate.

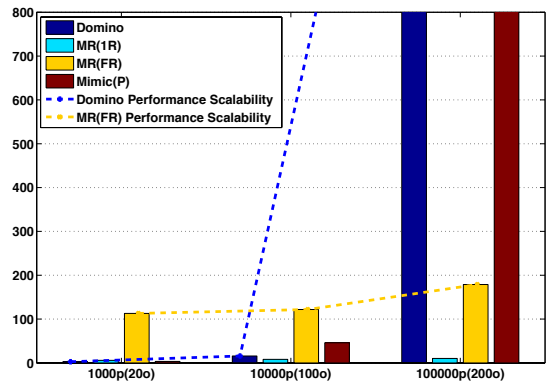

Figure 4: PageRank under different changing sets.

Figure 4 shows the execution time for different changing sets. We labeled each input data set in the x-axis of Fig. 4: p means the page number and o means the out-edges of each page. We can see that the Domino significantly outperforms the MapReduce in first two cases. Because of the fixed time window, which applications in our Mimic Percolator system have to wait for accumulating, the Mimic(P) is much slower than our Domino implementation, but it still outperforms MapReduce at these input data sets. However, we can also observe that while the input size increasing, the MapReduce execution time is increasing much more slowly than that of Domino and Mimic Percolator. But note that the MapReduce performance here is actually the upper bound of what MapReduce can achieve: it does not contain any processing on existing data. Furthermore, the burst new inputs in this evaluation are rare in real-world applications. Continuous small updates are the normal cases, where Domino outperforms MapReduce considerably.

6. CONCLUSION

In this study, based on existing event-driven programming models, we proposed Domino, a trigger-based incremental programming model with wait-free eventual synchronization for cloud applications. We introduced the necessity of synchronization in an incremental processing framework, and we presented a novel design and implementation of this wait-free synchronization (eventual synchronization). The current use cases and experimental results have confirmed the efficiency and benefits. The design and development experiences of Domino can help the community better support incremental processing of cloud applications.

Acknowledgment

This work was supported in part by the National Science Foundation under grant CNS-1162488 and by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357.

7. REFERENCES

[1] Hbase project. In *http://hbase.apache.org*.

[2] Omid project. In *https://github.com/yahoo/omid*.

[3] Zookeeper project. In *http://zookeeper.apache.org/*.

[4] Bhatotia, P., Wieder, A., Rodrigues, R., Acar, U., and Pasquin, R. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (2011), ACM.

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

[6] Mitchell, C., Power, R., and Li, J. Oolong: asynchronous distributed applications made easy. In *Proceedings of the Asia-Pacific Workshop on Systems* (2012), ACM, p. 11.

[7] Peng, D., and Dabek, F. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (2010), USENIX Association.