Trigger-based Incremental Data Processing with Unified Sync and Async Model

Dong Dai, Yong Chen, Dries Kimpe, and Robert B. Ross

Abstract—In recent years, more and more applications in the cloud have needs to process large-scale on-line datasets, which evolve over time as new entries are added and existing entries are modified. Several programming frameworks, such as Percolator and Oolong, are proposed for such incremental data processing and can achieve efficient processing with an event-driven abstraction. However, these frameworks are inherently asynchronous, leaving the heavy burden of managing synchronization to applications' developers, which further significantly restricts their usabilities. In this study, we propose a trigger-based incremental computing framework in the cloud, called Domino, with both synchronous and asynchronous mechanisms to coordinate parallel triggers. With this new framework, both synchronous and asynchronous applications can be seamlessly developed. Use cases and extensive evaluation results confirm that it can deliver sufficient performance, and also is easy to use for incremental applications in large-scale distributed computing.

Index Terms—Programming Framework; Cloud; Incremental Computing;

1 Introduction

The availability of Cloud Computing services like Amazon EC2 [1] and Windows Azure [2] provide on-demand access to affordable large-scale computing resources without substantial upfront investments. However, designing and implementing different kinds of scalable applications to fully utilize the cloud can be prohibitively challenging requiring domain experts to address race conditions, deadlocks, distributed states while simultaneously concentrating on the problem itself. So, general distributed programming frameworks are widely adopted to make writing large-scale distributed applications under the cloud environment much simpler. Some of the most popular frameworks include MapReduce [3], Dryad [4], and Spark [5]. Although many applications have been implemented based on those frameworks, there are still many applications hard to write and inefficient to run under these frameworks [6], [7]. Typically, these applications contain iterative processing on continuous data streams, and they are of increasing importance to real-world analysis today. For example, the performance measurements in network monitoring and traffic management, log records or click-streams in web tracking and personalization, data feeds from sensor applications, and detailed call records in telecommunications, etc., all these applications require the ability to process external continuous data streams.

To process these data streams, incremental approaches are often more efficient. But, existing batch-processing programming frameworks, such as MapReduce [3], Spark [8] and Dryad [4], provide only shallow support for incremental processing. Although their extensions, like HaLoop [9], Twister [10], Spark Streaming [5], and Incoop [11], are recently proposed to to support a certain level of incremental processing based on batch model, there are still limitations on them: choosing the right interval to process the incoming streams in a batch way is challenging for developers. Additionally, the asynchronous applications are not supported by these strong synchronous models.

Recently, event-driven models also are proposed in cloud computing for incrementally processing data streams [12], [13], [14], [15], [16]. In these event-driven models, applications are triggered by external data streams. Thus, developers do not need to know the update frequency of their interested data streams in priori, and the framework schedules their applications reacting to streams based on system resources automatically. Most event-driven abstractions are designed and implemented based on triggers, like S4 [15], Percolator [13] and Oolong [12]. In a distributed environment, these triggers can run on different servers independently, but their results need to be aggregated together to form the final returns. These aggregations could be synchronous or asynchronous among different triggers. Most of existing trigger-based programming frameworks only support the asynchronous aggregation among different triggers [12], [13], which significantly limits their usabilities on incremental applications that need strong synchronization. These applications usually can deliver much better performance or have a clear inter-status inference and correctness proof under synchronous executing model [17].

Considering the limitations of existing frameworks, in this paper, we introduce the design and implementation of a trigger-based incremental computation framework, namely Domino, with both synchronous and asynchronous models. To our best knowledge, Domino is the first step towards a unified trigger-based programming system supporting both sync and asyn processing in Cloud. The main contributions of this study include:

- Focusing on a novel design and implementation of wait-free eventual synchronization on trigger-based programming model for incremental processing. Based on real use cases, our approach reduces the complexity of managing synchronization manually, delivers the desired performance, and supports a wide variety of application design patterns.
- Proposing and prototyping a set of optimization strategies including *execution preemption*, *result reuse*, and *gathered*

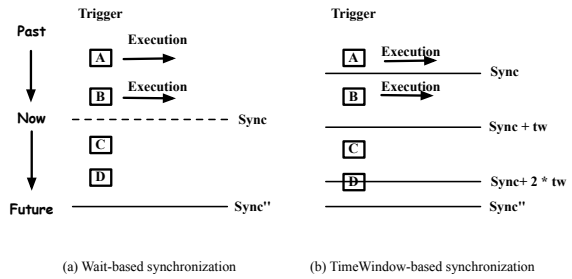
- *Dong Dai is with the Department of Computer Science, Texas Tech University, Lubbock, Texas, 79413. E-mail: dong.dai@ttu.edu.*
- *Yong Chen (corresponding author) is with the Department of Computer Science, Texas Tech University, Lubbock, Texas, 79413. E-mail: yong.chen@ttu.edu.*
- *Dries Kimpe is with the Mathematics and Computer Science Division, Argonne National Laboratory, Chicago, IL, 60439. E-mail: dkimpe@mcs.anl.gov.*
- *Robert R. Ross is with the Mathematics and Computer Science Division, Argonne National Laboratory, Chicago, IL, 60439. E-mail: rross@mcs.anl.gov.*

I/O for the trigger-based runtime system; and prototyping the fault tolerance and failure recovery based on HBase [18], a popular cloud storage system.
- Evaluating the Domino programming model and runtime system with extensive test cases to confirm that the flexible synchronization provided by the new Domino programming model is both easy to use and efficient for incremental applications.

The rest of this paper is organized as follows. In Section 2, we briefly discuss the challenge of synchronization in incremental computation. In Section 3, we introduce the programming model of Domino. In section 4, we show two applications and their implementation in Domino framework. In Section 5, we describe the design, implementation, and optimization strategies of Domino. In Section 6, we report the experimental results for both the Domino framework itself and its applications. In Section 7, we discuss related work and compare it with Domino. In Section 8, we summarize our conclusions and briefly outline future work.

2 Challenge of Incremental Sync

Trigger-based frameworks provide a low-latency way to process incremental data by reacting to incoming streams. An accumulation is often needed to gather the partial results from these parallel executions in different servers. Some applications, such as the SSSP (shortest single source path), belief propagation [19], and many machine learning algorithms [20], often conduct asynchronously accumulating. Many others, however, require the accumulation to be synchronized for performance or correctness. Many graph algorithms and most MapReduce-based applications belong to the later category.



(a) Wait-based synchronization (b) TimeWindow-based synchronization

Fig. 1: Two synchronization strategies in incremental scenario.

Trigger-based approaches do not support such synchronization because of the complexity of global synchronization in incremental scenarios. Fig. 1 shows an example of accumulating results from four triggers *A, B, C*, and *D*. The accumulation needs to gather partial results from those four triggers synchronously. As the triggers are activated by external data streams, they will generate results at different time.

Fig. 1 shows two commonly used mechanisms to synchronize these triggers. The first one is using distributed locks to enforce all triggers to wait until all of them are finished, as Fig. 1(a) shows. The challenge, however, is that in the incremental computing scenario, one cannot know how many updates there will be and how many trigger instances will run in the future. As shown in Fig. 1(a), the first *Sync* should already generate correct accumulation results because *C* and *D* will not execute in the future, but distributed locks require waiting until trigger *C* and *D* finish. This waiting time can be long or even infinite. The second possible solution is using a *time window*, which defines a fixed waiting time (t_w) and makes the accumulation progress each t_w even without knowing about the trigger instances status, as Fig. 1(b) shows. However, choosing t_w itself would be a challenge for developers. Moreover, it faces the problem of not knowing when the synchronization should stop and when the right answer is generated.

One of our goals in this research is to develop a wait-free synchronization mechanism for trigger-based frameworks: it should be able to avoid the unnecessarily long waiting for the uncertain future trigger executions, and also free developers from choosing appropriate synchronization intervals. We introduce our solution, namely Domino, in detail in the following sections.

3 Programming Model

The Domino programming model is a trigger-based programming model. It follows the classic *event-condition-action* (ECA) model [21], which defines an event-driven model which can be simply described as: *fired events that satisfy the conditions will trigger actions to execute*. For example, a web crawler which is used to fetch Internet web pages can be implemented in ECA model as follow: first, it starts from an initial empty URL set. Then a new web page or the updated version of an existed page is written into the URL set if it has not been retrieved before, a new *event* will be generated. The runtime system captures new events and run corresponding *actions* to analyze the page and store more URLs into the URL set to generate new events. This will repeat until the URL set is empty. Following the ECA model, we will introduce the data model, programming concepts and abstractions of Domino in the following subsections.

3.1 Data Model

Domino applications are triggered by the external data streams, which are usually collected from different on-line applications or services, so they have diverse formations. Domino provides a unified data model: *sparse tables* (as Tables 1–7 show), for all those datasets. Specifically, each *sparse table* represents a dataset. Each row indicates a data item in that set. Each row can contain millions of sparse fields (*columns*) to fit the unstructured nature of data. This table-based data model is similar to the Bigtable data model, which has been proven to be capable of describing diverse types of datasets in Cloud applications [22], [23], [24].

Specifically, the Domino runtime system is implemented on top of HBase [18]. Data stored in Domino is actually arranged into HBase tables that may contain billions of rows and millions of columns categorized in thousands of column-families. A column-family provides a higher level of abstraction of columns. It gathers all the similar columns and is guaranteed to be stored in the same server. Column within column-family can be referenced as *column-family:column-qualifier*. Each cell in the table will contain multiple versions of data, which plays an important role in Domino implementation. Inserting data into a table will generate events that need to be processed. Applications need to subscribe this event by specifying which table they are interested. Domino provides different granularities for subscribing: on columns, column-families, or the whole tables.

3.2 ECA Concept

Following ECA model, Domino programming model consists of three basic components (i.e., *Events*, *Conditions*, and *Actions*). The *event* is generated from the continuous external data streams that applications subscribed. Based on the *sparse tables* data model,

the continuous data streams mainly are *insert* or *update* operations on the sparse tables. The *condition* is used to filter events in order to control the execution of *actions*. It is a user-defined function that returns as *true* or *false* to denote whether the current event should be processed or not. One of the most important roles of conditions is to stop an iterative execution. We will show how it works in later examples. The *action* contains the actual logic of a user's functionality. It consumes the events and writes the results back persistently. Actions may be executed on multiple servers concurrently according to the data location of sparse tables. Domino prefers the actions to run on the server where the event is fired. This locality of action execution improves the performance and reduces the possibility of network congestion.

3.3 Domino Programming Abstraction

Combining these three basic concepts (event, condition, and action), Domino presents the basic programming abstraction: *trigger*. All applications in Domino are written as a series of dependent triggers. Users submit those dependent triggers together as an application. The dependency is implicitly expressed similar to a data flow system. During runtime, triggers can run on multiple physical servers according to the locations of events. Each running of trigger is called a trigger instance.

Domino supports three types of triggers for users to compose their applications: the *plain* trigger, the *asynchronous accumulator* trigger, and the *synchronous accumulator* trigger. Users need to implement one of these three basic triggers and pack them together as an application.

- The *plain* trigger works in the traditional single-event single-action way. It responds to an event generated from external data streams or partial results from other triggers, executes independently on different physical servers, and persists results to *sparse* tables. It is similar to the Map phase of the MapReduce model. It is not efficient to accumulate different data streams using *Plain* trigger.
- The *asynchronous accumulator* trigger keeps the traditional asynchronous semantic during accumulating: partial results arrive asynchronously and activate the trigger without any waiting or any coordination. Most event-driven models like S4, Percolator, and Oolong, etc. belong to this category. And also, many algorithms including the asynchronous PageRank [25], graph coloring, and single source shortest path [12], etc. can be implemented using this kind of triggers. However, as we have described, this asynchronous semantic, although simple, is limited towards certain scenarios, where the performance or the correctness of the applications relies on synchronization. Domino uniformly supports both sync and async through a newly designed trigger, discussed as follows.
- The *synchronous accumulator* trigger, which provides the synchronous semantics between different trigger instances. Its semantic is similar to the Reduce phase in MapReduce model. But, unlike MapReduce, Domino is designed for continuous, incremental applications, hence can not stall and wait for global synchronization as MapReduce does. In Domino, each partial result will activate the user-defined synchronous accumulator trigger immediately, but the accumulated results are not considered as final yet. These results will have versioning information and could be updated in the future if other partial results arrive and activate the trigger. So, while more partial results are generated, the final value will be eventually generated. This eventual synchronization avoids unnecessary global blocking and helps the applications make progress quickly. It is also simpler to use because developers do not need to explicitly set up global locks or barriers. In the next section we will describe how eventual synchronization is implemented and the key optimizations for performance improvement.

3.4 Comparison with MapReduce

Although the trigger-based abstractions can act like MapReduce model (consider the plain trigger as map and the synchronous accumulator trigger as reduce), there are several key differences making Domino a more general computing framework for both incremental applications and batch applications.

First, the map functions in MapReduce work only on static datasets. But, plain triggers in Domino work on data streams. Actually, the plain triggers can also process the static datasets if we treat them as fast streams filling a sparse table instantly. Second, the reduce function in MapReduce only works in a synchronous way: they need to wait for all the previous map functions finish. But, the accumulator triggers in Domino support both async and sync mechanisms. Even for the synchronous accumulation in Domino, it is still different from reduce. It provides the guarantee for synchronized results as the traditional reduce functions does, but does not need to globally wait for slow streams. Third, MapReduce provides a very simple linear dependency between map functions and reduce functions, which limits its usage in more complex applications. Domino allows developers to specify complex dependencies between triggers by simply subscribing the outputs of other triggers.

In fact, we can easily port huge amount of existing MapReduce applications to Domino, and enable them to process both static and streaming datasets simultaneously. To achieve this, developers need to re-write their map/reduce functions into different types of Domino triggers, specify those triggers to work on certain datasets, and use the synchronous accumulator triggers to subscribe the results to perform the reduce function. In fact, it is actually possible to automatically do this transformation in the runtime without re-writing the map/reduce functions. This is still our on-going work.

4 Examples of Domino Applications

4.1 Incremental WordCount

WordCount counts the frequency of each word from a huge text dataset. In Domino, the text dataset could be static history data or collected on-line. In either way, they should be stored in the *sparse table* as Table 1 shows. The counting results are also stored in a *sparse table* as Table 2 shows. To implement WordCount, we use one *plain* trigger (t_p) to subscribe the whole Table 1 to parse each text data. This trigger (t_p) monitors the Content column-family. Whenever new files are added, t_p will be fired to execute and split the document into words and then apply an atomic *'add 1'* write on the corresponding Acc:*Value* column in Table 2 for certain words. These *'add 1'* writes will be combined locally for each word to reduce I/O times. As we have described, this trigger-style application is similar to its MapReduce counterpart expect it can work on changing datasets. Moreover, by considering static data in table 1 as changing datasets filled instantly, we can deploy this Domino application to process existing dataset too.

TABLE 1. Content Table

Rowkey	**Cont** *En*
id-1	*'any string'*
...	...

TABLE 2. Word Table

Rowkey	**Acc** *Val*
'the'	*1200*
...	...

4.2 Incremental PageRank

PageRank algorithm [26] has been well studied in the past years. There were many variations including incremental versions [25] of it. But in this example, we only show how the initial version (i.e., with strong synchronization) to be implemented incrementally under Domino as a general demonstration.

PageRank calculates the rank of web pages based on the links between them. Using a transition matrix M to represent the whole web graph, PageRanks calculates the rank vector (v) for every page based on this equation:

$$v^{new} = \beta M v^{old} + (1-\beta)e/n \quad (1)$$

Where β is a constant, usually in the range 0.8 to 0.9, e is a vector of all 1s with the appreciate number of components, and n is the number of nodes in the web graph. v_{old} denotes current rank value of all pages. PageRank runs iteratively until the difference between v_{new} and v_{old} less than a small value ε. In each iteration, the processing contains two phases: in the first phase, every page spreads its page rank value to all its neighbors equally; in the second phase, every page aggregates all the received rank values and calculate using equation (1).

To implement PageRank in Domino, we abstract the whole web repository as Table 3 with URL as row key. In this *sparse table*, the *Meta:Rank* column stores PageRank value, *Meta:OutEdges* represents all the out edges of current page, and *Cot:En* stores the actual contents of a page.

TABLE 3. WebRepo Table

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

TABLE 4. Accumulation Table

RowKey	*Partial-results:url4*	*Partial-results:url3*	...
url_1	0.1	0.15	...
url_2	0.25	0.32	...
...	...	... , ...	...

Based on the two phases in PageRank algorithm, we deploy two triggers to implement it. The first one is a *plain* trigger (t_p) monitoring on the *Meta* column-family of Table 3. So, whenever *OutEdges* (e.g., new connection is created in th web) or *Rank* changes, the *plain* trigger will be fired to calculate the outgoing edges' weights. The new weights of all the *OutEdges* will contribute to the destination URLs and change their weights. Those contributions should be aggregated. To do so, we will write those weights to a *synchronous accumulator* trigger (t_a) for aggregation. In Domino, each t_a will implicitly host a table (i.e., accumulation table) using the destiny URL as key as Table 4 shows. It contains all the incoming edges' weights for each URL for calculating the new rank value. The trigger t_a is automatically monitoring this table, and each time a new weight is written into the table, t_a will execute to aggregate all the columns and write the result back to the *Meta:Rank* column of Table 3 and activate the next round. To check whether when to stop iterations, we add a *condition* in t_p to see whether the new rank value is close enough to the old one (converged); if yes, we eliminate this event to stop whole execution. Eventually, the whole application stops.

The *synchronous accumulator* in this example subscribes and monitors the so-called 'accumulation table'. In Domino, for each accumulator trigger, runtime will automatically create such an accumulation table, which only contains one column-family (*partial-results*) and allows multiple columns in it. This *sparse table* is invisible to other applications so that it avoids misbehaviors from shared users. We show the invisible table in Table 4 for trigger t_a. Here, different columns indicate weights from different source URLs. This example shows a typical iterative and synchronous application implemented in Domino.

This incremental version of PageRank needs strong synchronization during accumulating. If the accumulator trigger (t_a) just sums the latest incoming edges' weights without synchronization, the execution flow will be different from what the initial algorithm requires. In many algorithms, this may introduce uncertain results. But using the *synchronous accumulator* trigger abstraction, developers can easily guarantee the synchronous executions.

4.3 Incremental Clustering

TABLE 5. Items Repo

RowKey	*vec k1:k2:...*
$item_1$	12.7:2.3:...
$item_2$	...
...	...

TABLE 6. Clustering

RowKey	*central k1:k2:...*
$cluster_1$	2.0:0.5:...
$cluster_2$	0.2:0.3:...
...	.....

TABLE 7. Accumulation Table

RowKey	*Partial-results:c1*	*Partial-results:c2*	...
$cluster_1$	$item_1$	$item_2$	...
$cluster_2$	$item_3$	$item_4$	...
...	...	... , ...	...

The clustering algorithm is widely used to classify items based on their similarities or distances. It starts from some randomly chosen central points and iterates certain times to get the optimal clustering. To implement it in Domino, we will need two sparse tables: Table 5 shows the items repository and Table 6 shows the k existing clusterings and their central points. In each iteration, the clustering algorithm calculates the distances between items and the central points of each clustering to chooses the nearest clustering for each item. At the same time, the central point of each clustering also changes due to the added or removed items. After the central points changed, all items need to recalculate the distances and hence the belonging clusters again.

Three triggers are deployed for implementing such clustering algorithm in Domino. Plain trigger (t_{p1}) monitors Table 5 to calculate the clusters that the item belongs to, and write the item into an synchronous accumulator trigger (t_a). The *synchronous accumulator* trigger will accumulate updated rows to in Table 7, calculate new central points, and write them into Table 6. Another plain trigger (t_{p2}) monitors Table 6 and call *All-Activate* on plain trigger (t_{p1}) to run over all items, and calculate new clusterings again. The application will finally stop when all central points do not change anymore. For better performance, Domino allows developers to add filter-condition to eliminate the unnecessary computation in a fine-grained way during all-activation. For example, in clustering example, it is not necessary to recalculate the distances for items whose clusters are far away from the changing clustering. This

kind of optimization is possible in Domino because triggers can access any intermediate or global statues during execution.

This clustering algorithm represents an extreme situation where a small update may cause re-calculation on all the datasets. For such applications, manually generating huge amount of events is not efficient, hence Domino introduces *All-Activate* API to issue processing over all current data without generating large amounts of individual events.

5 Design and Implementation

In this section, we will introduce the core design and implementation details of Domino framework to support the abstractions and use cases we described in the previous section.

Domino was designed and implemented in Java and tightly integrated with HBase in the current implementation. However, it can work well with other distributed storage systems as long as they are able to support the sparse table data model with version capacity and persistence guarantee.

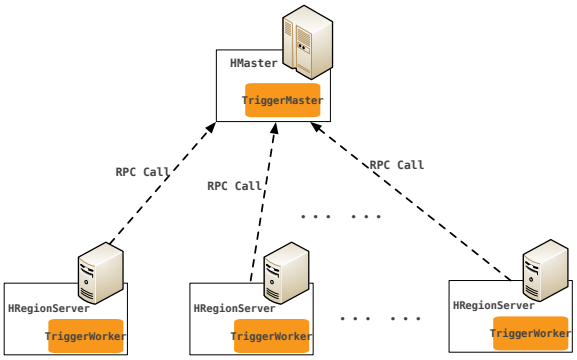


Fig. 2: Architecture of Domino with HBase.

Figure 2 provides a high-level overview of the Domino working with HBase cluster. Here, HMaster and HRegionServer are components of HBase, and the Domino instance runs along with the HBase instance on each server. There is one TriggerMaster node performing trigger management, collocating on the HMaster node. Other TriggerWorker nodes run along with the HRegionServer. TriggerWorkers communicate with TriggerMaster using a synchronous RPC protocol just as HBase does. TriggerMaster needs to continuously sync data location information from HBase through the sync RPC once it starts. The submitted triggers will be assigned to different servers according to this location information. Note that, both the HMaster and TriggerMaster node is chosen by the electing instead of user configuration, so whenever the master node fails, other nodes will be elected to keep the cluster running and avoid single point failure.

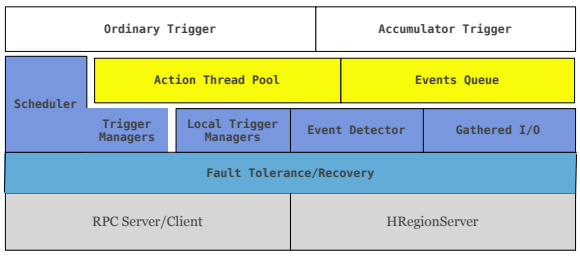


Fig. 3: Block diagram of different modules in Domino.

Figure 3 shows the logic diagram of each Domino instance. At the bottom, the RPC server and client components are used to communicate with other Domino servers and HBase. HRegionServer (from HBase) is slightly modified to provide both the public interfaces that HBase originally provides and also private APIs that only Domino can use. The fault tolerance/recovery layer detects and recovers trigger failures. Above this, an event detector detects updates on HBase; a gathered I/O component encapsulates all the data accesses in action functions to improve the performance. The scheduler and trigger managers (only run on the TriggerMaster node) manage and schedule submitted triggers. The local trigger managers (run on each TriggerWorker node) provide local management. All trigger actions are executed in a separate thread, which is allocated from the action thread pool. The event queues cache all the triggered events for processing.

In following subsections, we will concentrate on several core components affecting the behaviors of triggers including the events detector, the eventual synchronization mechanism, the gathered I/O system, and a series of optimizations with them.

5.1 Event Detecting and Processing

Event detector plays the central role of generating events. In Domino implementation, it is implemented through intercepting the core execution path of Write-Ahead-Log (WAL) appending procedure in HBase. All event detectors run on different servers concurrently to build event objects containing the information collected from the WAL logs and send them to their local event queues. To best work with Domino, we modify the HBase and add additional information on each key/value entry in WAL: old key/value pairs and last access timestamps.

All the fired events are managed by local event-queue and distinguished by their keys. The key of an event is a vector consisting of the monitored *table name*, *column-family*, *column*, and *version* information. Different triggers can subscribe on the same sparse table, so an event may activate multiple triggers. This is done by a consumer waiting on the event-queue: whenever there are events pending on the queue, it will be notified and send events to the corresponding triggers. For execution, each action function will be attached to a preallocated thread. All action threads are managed by the *Action Thread Pool* component in Fig.3.

In most cases, triggers are activated by the event detector when data in *sparse table* was changed. However, as Domino allows applications to run on static datasets in a MapReduce-style, the triggers should be able to directly run on all existing items in sparse tables. Domino provides *All-Activate* interface to support such behavior. Users can call *All-Activate* while submitting triggers or inside the action function of triggers. Once it is called, the runtime will make a snapshot of all the existing data items in the monitored *sparse table* and iterate them bypassing the event detector or event-queue and also ignoring any new items. In addition to this, all the intermediate results will not create new events until the iteration is fully finished.

5.2 Wait-free Eventual Synchronization

One of the major contributions of Domino is the wait-free eventual synchronization mechanism for synchronous accumulator triggers. The *Wait-free eventual synchronization* has two properties. First, the accumulation will be performed whenever any partial result arrives (*wait-free*); second, while more partial results arrive later, correct accumulated results will be eventually generated and overwrite previous results (*eventual*). In this section, we first describe the typical trigger-based execution flows and introduce

the reason of having eventual synchronization on the trigger-based execution models. Then, we introduce different parts of the eventual synchronization implementation, including: (1) the synchronous accumulator triggers itself, (2) the version manager that guarantees the correct results, and (3) the redundant execution preemption and incomplete result reuse, which help deliver better performance.

5.2.1 Eventual Synchronization in Trigger-based Model

We show a typical execution flow of a trigger-based (Domino) application in Fig.4. It contains an *plain* trigger and a synchronous accumulator trigger. The *plain* trigger is concurrently executed in three different servers (M_a,M_b,M_c), and generates partial results (D_a,D_b,D_c), which are written into the *accumulator trigger* for further aggregation. Assume the partial results arrive in the order of $M_a \rightarrow M_b \rightarrow M_c$.

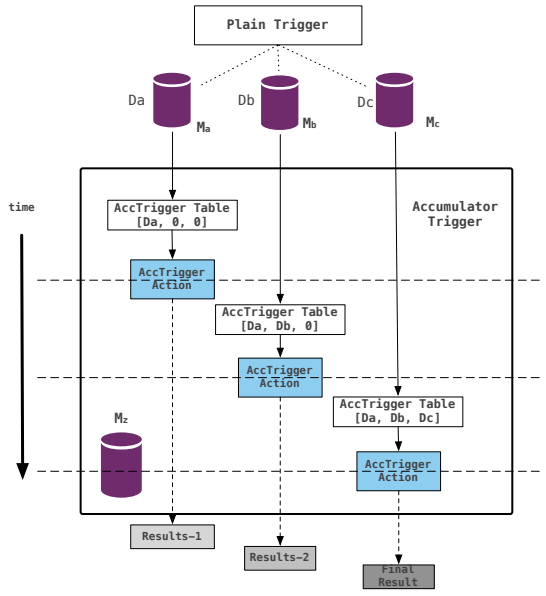


Fig. 4: A Domino application example, containing one plain trigger and one synchronous accumulator trigger

Based on the*wait-free* feature of Domino synchronous accumulator trigger, each time a new partial result is written, the accumulator trigger will execute immediately without waiting. Hence, there will be three *blue* boxes representing three executions as Fig.4 shows: the first action executes right after D_a was written (as the first dashed line shows), it calculates *Result-1* based on partial results of [D_a, 0, 0] (here, 0 indicates `NULL` instead of real value 0). The second action executes after D_b was written. And the third action calculates the result based on the complete inputs ([D_a,D_b,D_c]).

If we knew there will be three partial results, it is obvious that the first two of these three executions are useless (i.e., redundant). But, in incremental processing, due to the asynchronous nature of data streams, it is not possible for the applications to know how many partial results there might be. Taking incremental PageRank (see Section 4.2) as an example, the synchronous accumulator trigger needs to aggregate all the incoming edges' weights for a certain URL. In streaming case, it is highly possible that only parts of the URL's incoming edges changed and the aggregation needs to run once this happens. So that, during aggregating, both the programmers and the runtime systems do not know how many changes there will be and whether currently arrived one is the last one (e.g, D_c) or just the first one (e.g., D_a). To work in streaming cases, accumulator triggers should always execute whenever the intermediate data is arriving to keep progressing and generating new results as we required. In addition, we also need to distinguish the correct one or the redundant ones generated from multiple *wait-free* executions. To understand the intermediate results and guarantee the final results are correct, we proposed the eventual synchronization and its optimization techniques described later.

5.2.2 Accumulator Triggers Design

As we have described, for each accumulator trigger, Domino implicitly creates an invisible table as Fig.5 shows. It contains a predefined column-family named *partial-results*, which is automatically monitored by the newly created accumulator trigger. All writes to accumulator triggers should contain row key (r_i), column indicator (c_j), and value (v). They will be written into this table based on the key. The value (v) will be written with an added version information inherited from previous trigger.

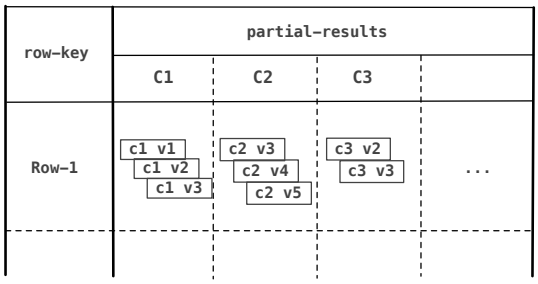


Fig. 5: Invisible table for accumulator trigger.

The key of eventual synchronization of accumulation triggers is to choose the data with right version to generate right results. To implement this, Domino uses the multi-version feature of HBase to store and trace both the trigger actions and external data streams.

5.2.3 Version Management

Domino allows tables to store multi-version data. Hence, when applications access data, they need to specify the requested version, or else, the newest version will be returned by default. This multi-version data was used to trace the partial results that executions generated and coordinate the executions.

First, in Domino, each application obtains a unique application id (id_{job}), which is shared by all its running trigger instances across different servers. To trace the executions inside one application, we introduced two extra ids: round id (id_r) and provenance id (id_p), both of which are 32-bit and can start over if run over. The round Id id_r represents iteration rounds of current execution (each trigger execution indicates a new iteration), and provenance Id id_p represents the external stream version (each external data updates on the same cell can be viewed as a new data stream version).

To show that these ids are sufficient to describe executions in different incremental applications, we categorize these applications into three types as Fig. 6 shows. The first is plain iterative applications (Fig. 6(a)), where each trigger simply writes into the *sparse table* subscribed by itself and activate itself to run iteratively. In this case, the round Id id_r of each execution will increase every time, so it can be used to distinguish different iterations. The second one is plain incremental applications shown in Fig. 6(b), where triggers are activated only by external inputs. We can see that the round Id id_r remains the same, whereas the provenance Id id_p increases. Fig. 6(c) shows the executions of

both iterative and incremental applications, which are the most complex and Domino targets at. In this case, both the id_r and id_p are changing based on extern inputs and internal iterations.

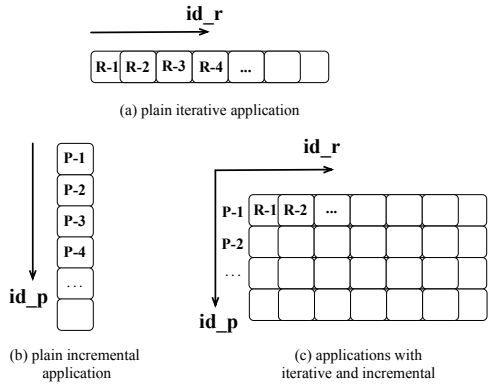


Fig. 6: Three types of applications in Domino.

We implement the version management based on these Ids. First, during application running, each time a trigger writes into *sparse table* and activates another trigger, we will merge the trigger's id_r and id_p into one 64-bit version number ($id_p : id_r$) and append it with each data item as version number. When these new data further activate other triggers, the events will inherit the same version number from data, and Domino will read the version number, parse it into id_p and id_r, and set the new trigger execution with versions: the id_r will increase '1', and id_p will keep the same. Second, whenever external inputs were written into a the sparse table, Domino will set the new $id_p^{'}$ as id_p+1, and initialize id_r to 0. In this way, we are able to trace each trigger execution by knowing the $id_p : id_r$ pair. These version information will be used for correct aggregation.

In Domino, to guarantee the synchronous results, the key is to restrict the accumulator triggers to only access data with the smaller or equal id_p and id_r comparing with its own id_p and id_r. Specifically, assuming there are k columns in the *partial-results* column-family receiving partial results. Each of these columns has multi-version data, and their largest version numbers will be id_p^j and $id_r^j (j \in 1 \rightarrow k)$. Whenever a partial result on $Column_i$ arrives, corresponding accumulator trigger should be activate and start to execute. This execution will inherit version information from partial result stored in $Column_i$. When acc_i executed, the partial results from other columns may have larger or smaller version numbers than acc_i has. The larger versions indicate newer executions; smaller versions show older executions. Based on the data access rule in Domino, acc_i will only access partial results in other columns ($j \in 1 \rightarrow k$) with version $id_r^j \leq id_r^i$ and $id_p^j \leq id_p^i$. In this way, the faster and slower executions are synchronized correctly.

In this way, when the last update inserted into the *partial-results* column-family, the accumulator trigger action will run and get the latest results from all columns to aggregate. This is a strict sync way, hence gurantees the return of the correct results.

5.2.4 Redundant Execution Preemption

Wait-free eventual synchronization avoids the global waiting for unpredictable upcoming events in steam processing. But it does introduce some possible redundant executions during accumulating. As described, Domino should execute accumulator trigger each time new partial results were generated without immediately. But, if there are new updates in the near future, previous execution would be redundant and quickly overwritten. Domino can reduce this kind of execution by rearranging events in the event queue.

Figure 7 shows how the event detector generates events and sends events to corresponding triggers. Here, each event has two Ids: [d_r, d_p]. Different events are distinguished based on the row-key of the accumulator table they are from. In each trigger, events are cached in queue waiting for processing. However, there might be redundant events initialized by different trigger as shown as the red framework in Fig. 7. As those events will activate multiple executions that read the same data in the near future, Domino will remove the first ones and only keep the last one, thus prune and postpone the events. This pruning will also cut all the children of the redundant executions, so it will reduce the execution number significantly.

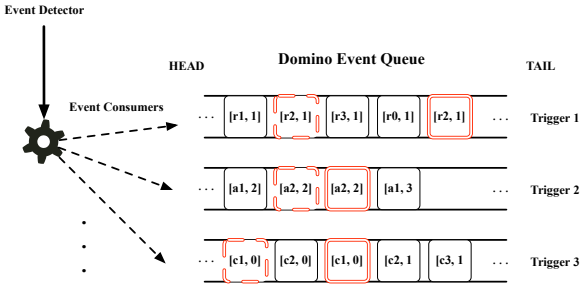


Fig. 7: Redundant executions exist inside each event queue. The dashed red box denotes that the event should be preempted.

Another advantage of queue rearranging is that it provides dynamic waiting time based on the event frequency and trigger execution speed. Frequent events will make the queue longer and increase the opportunity of preemption to save resources; a shorter queue will also accelerate the processing. The frequency of queue rearrange itself is based on how often the redundant events will show in the queue.

5.2.5 Incomplete Accumulation Results Reuse

Another optimization for eventual synchronization is incomplete results reuse. In *synchronous accumulator* trigger, each accumulating will generate a result, which may be the final result or just an incomplete result that will be overwritten by later accumulating. The incomplete results reuse mechanism reuses them to reduce the future accumulation time. Note that this is limited by the accumulation logic. Thus, it provides optimization only for certain applications.

Assuming x_i^j denotes the jth version of the ith column inside an accumulator table, we can abstract any accumulation logic as a function: $acc(x_0^{j_0},...,x_k^{j_k},...,x_n^{j_n})$. Whenever a new data stream causes an update on any column ($x_k^{j_{k+1}}$), we need to calculate this function again based on the new version as $acc(x_0^{j_0},...,x_k^{j_{k+1}},...,x_n^{j_n})$. However, since only a small part of these partial results changed, one can use those previous results only if there is a function *Incr()* that satisfies this equation for the current *acc* function.

$$acc(x_0^{j_0},...,x_k^{j_{k+1}},...) = Incr(\delta(x_k), acc(x_0^{j_0},...,x_k^{j_k},...))$$

In this equation, $\delta(x_k)$ indicates the difference between $x_k^{j_k}$ and $x_k^{j_{k+1}}$. In Domino, if developers implement the *Incr()* function, the runtime system will calculate the new aggregation results based on the old results automatically, following the code provided by developers. To implement results reuse, we add an assistant

column into the *partial-results* column-family of each accumulator table. It stores the incomplete results generated by the previous round. The user-defined $Incr()$ will retrieve the previous results from this column each time and calculate new accumulated results. A typical example of such optimization is accumulation logic like getting the maximal, minimal, or summation, which we can only compare the new one and existing incomplete result to get the updated value.

5.3 Gathered I/O

Domino allows applications to read arbitrary data and write the intermediate results into any *spare tables* during executing. This may introduce serious performance problems because of the heavy I/O workloads and data inconsistency problem if failure happens. To solve these problems, Domino introduces *gathered I/O*, which encapsulates the input and output data operations, in order to address both consistency and performance issues.

Gathered input is for accumulator triggers only. Domino framework automatically collects all the partial results while processing accumulating. The gathered inputs contain all the relevant values with the proper version (described in Section 5.2.1) in *partial-results* column-family. For *synchronous accumulator* triggers, the proper versions refer to less than or equal to the fired data versions; for others, they would be the newest version. Since all columns inside one column-family are stored on the same physical server, constructing these values is highly efficient.

Gathered output is general for all triggers. It is a per-server component called *WritePrepared*. In each trigger, all the data is first written into *WritePrepared* by calling the *append* method instead of directly writing into the sparse table. *WritePrepared* will cache all writes, combine them, and flush them to HBase later, based on the flush interval configuration and available buffer. *WritePrepared* will flush writes on their initial order of calling the append method. Hence, all writes in the action functions of applications are visible only after the flush method completed.

Domino also allows developers to write their own strategy to combine their local cached writes. For example, in the Word-Count example, developers can combine the '`+1`' writes to the same word, in order to reduce I/O times. Since the cached I/O operations may be lost because of server failures, Domino deploys a mechanism described in Section 5.4 to keep the fault-tolerance.

Each time we flush a *WritePrepared* instance, it will register itself to the global consistency service (*ZooKeeper*) and get a global sequential number S_f. If two writing requests from different triggers are not conflict, HBase will not feel any difference from the ordinary writes. However, if they are conflict with different sequential number $S_{f1} < S_{f2}$, then the second flush would be suspended until the first one finished. During the suspending, all the successful writes by S_{f2} will be over-written. This strategy does not mean all writes are sequential, but just means Domino chooses the write operations with the currently minimal sequential number, and once it started, it will not be preempted.

5.4 Fault Tolerance

Domino is able to sustain server failures by using HBase as basic fault-tolerant infrastructure. Moreover, as we store all the intermediate results into HBase table, failed triggers can be recovered on other nodes from the failing point instead of starting over. To detect the failures, each TriggerWorker issues periodical heartbeat signal to TriggerMaster node. If nodes crashed, the lost heartbeat signal will remind Domino into a recovery stage. It will reschedule triggers previous deployed on failed nodes to other nodes and replay the unfinished events.

There are mainly two in-memory data structures need to be recovered during replaying failed events: in-memory event queue and cached I/O operations. The in-memory event queue is built by processing new entries in HBase WAL (write-ahead-log) files. Whenever we lost a server, its event queues can be rebuilt from WAL logs again. The rebuilt events can be added into the backup nodes' event queues any time because the version information will make sure they will be processed correctly. To avoid losing cached I/O operations in *WritePrepared* instance, each time the trigger actions issue an *append* operation, the *WritePrepared* instance will add a new write entry into ZooKeeper. This write entry only contains the table name, column-family, and column information. The actual data will be persisted in WAL files. Once the *WritePrepared* instance starts to flush cached I/O operations, each successful write will change the corresponding entry in ZooKeeper to status *flushed*. When failure happens and some cached gathered I/O operations lost, we can simply run this failed action again and re-write the part of data that has not flushed.

6 EVALUATIONS

The evaluation includes two parts: the micro-benchmark evaluations and application evaluations. From the experimental results, we summarize several principal findings.

- The design and implementation of Domino are efficient: less than 12% I/O performance degradation comparing with original HBase even with heavily fired trigger execution.
- Applications running in Domino outperform MapReduce easily by 10x processing streaming datasets. But, the performance drops quickly when the streams arrive in batch. This means Domino is a good incremental framework instead of a batch processing framework.
- Redundant execution preemption and incomplete result reuse play an important role in performance optimization for the eventual synchronization mechanism in Domino.

We utilize two platforms to conduct the evaluations. The first platform is Amazon EC2. We use up to 64 m1.medium instances (128 cores) to deliver the scalability evaluation of Domino. This platform is used for scalability and micro-benchmark evaluation reported in Section 6.1. The second platform is a CloudLab Utah APT cluster [27]. We allocated 32 nodes for this evaluation. The Cloudlab server has 8-core Xeon E5-2450 processor, 16GB RAM, and 2 TB local hard disk drives. All nodes are still connected through 1G network card. This platform is used for evaluating different applications, reported in Section 6.2.

6.1 Evaluations with Micro-benchmark

As the Domino event detector intercepts WAL appending operations, build the event instances, adds all of them into queues, and activate corresponding triggers, this will introduce possible overheads. We use the PerformanceEvaluation [28] test suite, which is a pure I/O benchmark, to evaluate the overheads introduced from Domino components. The results are measured under two different scenarios: without trigger executions and with heavily fired trigger executions. In heavily fired triggers case, we only perform a simple '`+1`' computation in corresponding action function.

Fig. 8 shows a write performance comparison of Domino and HBase with different scalability in a cluster up to 64 nodes (128 cores). The performance difference between HBase and

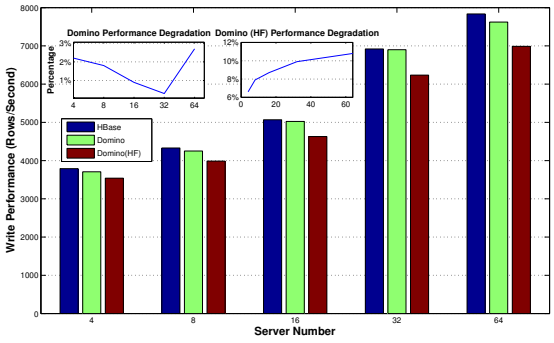


Fig. 8: Performance comparison between Domino and HBase. Domino(HF) means Domino with heavily fired triggers.

Domino is small and nearly constant while the server number increases: we can see that the Domino performance degradation is less than 3% when there is no trigger executing. Even under an events heavily fired situation, the performance degradation is still constantly small (essentially arround 12% based on the *Domino(HF)* column). We also note in this figure that the Domino system maintains good scalability, just as HBase does. From these results we conclude that Domino is designed and implemented to be a highly efficient framework.

To improve the performance of Domino, we introduced two optimizations: redundant execution preemption and incomplete result reuse. The best use case of them is in the iterative applications with accumulations. In this evaluation, we run the PageRank application on a synthetic web graph, where each node has n link-in edges. The number of link-in edges indicates the frequency of accumulating, so they are labeled as the *accumulator number* as the x-axis in Fig. 9 and Fig. 10.

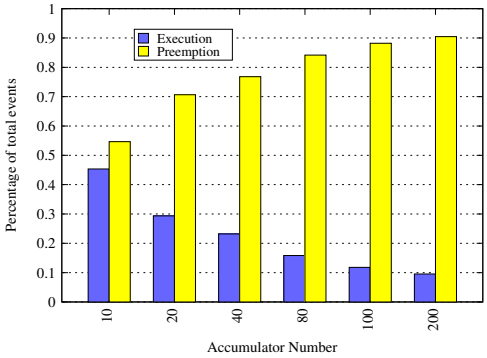


Fig. 9: Redundant execution preemption while accumulating.

The web graph generated for redundant execution preemption includes 100K pages, and each page is randomly assigned with 10 $\rightarrow$ 200 link-in edges. The y-axis in Fig. 9 means the total fired events. From Fig. 9 we can see that the possibility of preemption is increasing when the number of accumulator is increasing. This result shows that most of the preemptions happen in accumulating, and the execution preemption is able to reduce the useless executions in order to improve the applications performance in such cases.

To evaluate the performance improvement introduced by the *Incr*() function, we also implemented an *incr* function in PageRank's accumulation. We run the application on a more dense dataset, where each page is assigned with 100 $\rightarrow$ 800 random edges. Fig. 10 shows the time cost of each accumulation when the

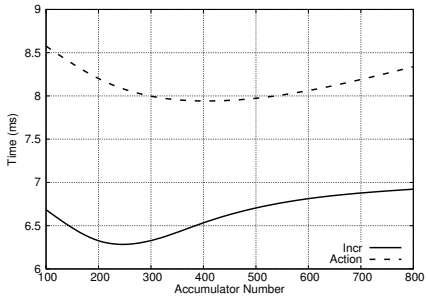


Fig. 10: Incomplete result reuse in PageRank implementation.

average link-in pages increasing from 100 to 800. We can see that along with bigger accumulator numbers, the *Incr*() version with result reuse gains more performance benefits. Although the *Incr*() function can improve accumulating significantly, it is not general for all applications. Therefore, in the following evaluations, we do not use *Incr*() except explicitly specified.

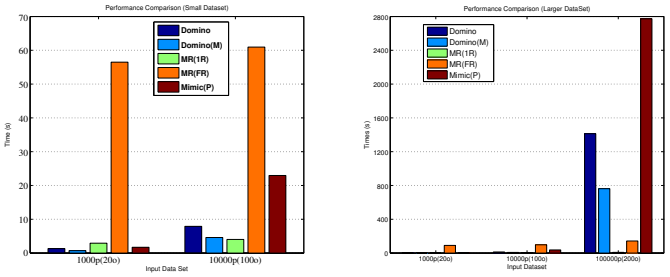


Fig. 11: Performance comparison of PageRank algorithm under different changing sets.

6.2 Evaluations with Applications

The Domino framework supports a large range of applications. In this section, we evaluate three different categories of applications: WordCount, PageRank, and Clustering. All results reported in this subsection were conducted on CloudLab cluster.

Since Percolator is not public, we cannot make direct comparisons with it. An identical implementation of it is also not an option because most of its foundations (Bigtable [22], GFS [29], Chubby [30], etc.) are not publicly available. Therefore, in this evaluation, we mimic the behavior of Percolator based on open source software stacks currently accessible. Specifically, we use HBase Coprocessor [31] to mimic the observers of Percolator and use the combination of Omid [32] and ZooKeeper [33] to simulate the transactions and distributed locks. Omid is a client-side transaction layer of HBase and was proven to have a comparable TPS (transactions per second) with Percolator [32]. ZooKeeper is a distributed service that provides a high-performance distributed lock service similar to Chubby used in Percolator. Current Domino is implemented with HBase 0.90 and The Hadoop MapReduce we used in following evaluations is 1.1.1 (with default scheduler).

6.2.1 Evaluation Strategy

The most critical performance measurement of incremental applications is how fast they can absorb the incremental inputs. If applications can process new inputs faster, they also can provide results with lower latency, which indicates *wait-free*. To detect how fast the applications can absorb the incoming data, we

give burst external inputs instead of slow streams. This extreme situation shows us the potential of an incremental framework.

For different evaluations, we created a large static dataset simulating the whole dataset and a small dataset simulating the stream. These changing datasets will be written into the Domino as fast as possible to trigger deployed Domino or Percolator applications. We also ran the MapReduce applications on both the whole input dataset and the smaller changing datasets as a comparison. Note that, only run MapReduce on the smaller changing datasets (streams) actually shows the best performance because they process only 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 this evaluation, we measured the performance of Domino by comparing it with classic MapReduce and another incremental framework (mimic Percolator). In the following evaluation results, 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 MapReduce was executed on the changing data sets; and Mimic(P) represents our mimic Percolator.

6.2.2 Evaluate Incremental WordCount

We evaluated the WordCount example on a synthetic dataset, which is stored as a sparse table as Table 3 shows. The overall input datasets are 200 GB; each time we added a new dataset (from around 140 M to 2.25 G) as the changing set. Larger total dataset size will not affect performance of Domino applications. It will mostly only increase the MapReduce application time. Fig. 12 shows the performance with different data sizes. We can see that Domino application is even slightly better than the case that MapReduce on changed dataset only. This performance advantage comes from the asynchronous nature of WordCount itself and also the local I/O combiner provide by Domino. However, the Mimic Percolator version does not scale well because it does not provide the *gathered I/O* optimization. During execution, it issues too many time-consuming '`+1`' to HBase.

Although in WordCount, it is possible to deploy MapReduce style applications only on the changed dataset and still obtain the right results, yet the frequency of running MapReduce is hard to decide towards possible changing stream speed. Domino, on the other hand, provides an easy way to run it in incremental situations, and it achieves similar performance.

6.2.3 Incremental Synchronous PageRank

Evaluation of incremental PageRank implementation is running on a synthetic web repository with one million pages. The streaming pages are ranging from 10^3 to 10^5. Since PageRank is iterative, in order to compare different implementations fairly, the MapReduce version needs to run enough times to make every page converged. This normally indicates more global iterations than Domino since most pages will converge fast, but MapReduce still run on all pages, leading to a much longer execution. So, in the result, we show the execution time of one MapReduce iteration (MR(R1)) as a hint to show the performance of a single iteration. The Mimic-Percolator does not support synchronous accumulation in its design. Therefore, we mimic such behavior by setting up a time window to wait before accumulating.

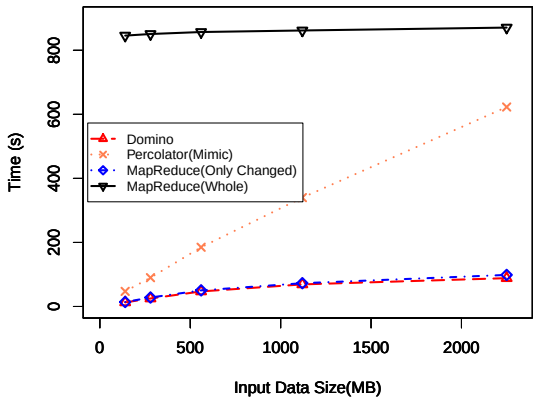


Fig. 12: WordCount performance on different changing datasets sizes.

Fig. 11 shows the execution time of different implementation towards changing datasets from 10^3 to 10^5 pages. The changing pages sets are having different distributions in terms of how many edges of each page in average. The distribution is labeled in the x-axis: p means the page number and o means the average out-edges of each page. To better show the performance differences, we divided the results into two parts, one stores results of 10^3 and 10^4 pages; another stores results of 10^4 and 10^5 pages. We can see that the Domino significantly outperforms the MapReduce in these two cases, simply because Domino only operates on changed pages instead of the whole repos. Because applications in our Mimic Percolator system have to wait for a fixed time window to accumulate, it is much slower than Domino, but still outperforms MapReduce. However, when the input size increasing, the MapReduce execution time is increasing much more slowly than Domino and Mimic Percolator do. As shown in Fig. 11(b), the Domino PageRank costs more time than MapReduce when 10^5 new pages are coming at once. We also show the performance of Domino running with in-memory mode (Domino(M)) to show the bottleneck is not from I/O operations. Instead, it comes from the inherit design of Domino: the event-based strategy will face challenges when the events are generated too fast. However, in real-world system, the burst, large volume streams in this evaluation are actually rare. Continuous small updates are the normal scenario, where Domino outperforms MapReduce considerably.

6.2.4 Incremental Clustering

Clustering represents an important kind of incremental applications where a single new input will cause re-computations on all existed data. To evaluate Domino performance on such kind of application is critical. However, this also means that our previous strategy of introducing burst new inputs and running applications only on new inputs will not work here. In this evaluation, we still try to change the inputs size, but both MapReduce and Domino applications need to run on the whole dataset each time to get correct results. In clustering algorithms, parameter (n, m, k) (n means the items number, m means their dimensions, k indicates the final clusterings number) defines the complexity of the algorithm. In this evaluation we simply set m=50 and k=120 as a demonstration, and change n from 10^3 to 10^5 by adding new items to evaluate the performance.

As Percolator does not provide *All-Activate* operation, it needs to generate numerous events in order to activate executions on all inputs. Its performance is far slower than that of the other

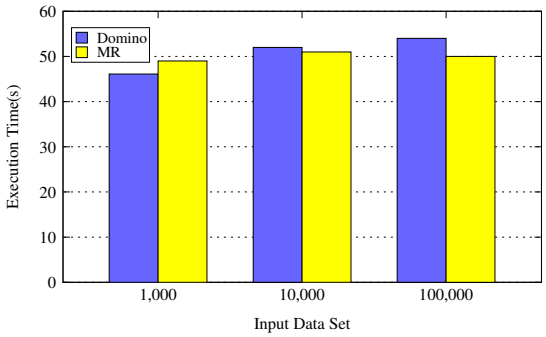


Fig. 13: Clustering performance.

two, so we eliminate its result. Fig. 13 shows the performance comparison of Domino and MapReduce. We can see that Domino is slightly slower than MapReduce with different new items. But, the performance does not change much for either Domino or MapReduce with different input sizes. The reason that Domino is able to achieve similar performance with MapReduce instead of being too slow like Percolator is that Domino supports an optimized *All-Activate* operation for this kind of applications.

In this section, we have showed the performance of different kinds of applications in Domino. Based on the results, we believe that Domino is a general incremental processing framework supporting both asynchronous and synchronous applications with performance that is better than or comparable to that of MapReduce and other incremental frameworks. We also conclude that for the synchronous iterative applications, it would be more practical to combine trigger-based and MapReduce-style execution: MapReduce-style could be used first to process the entire data set, and trigger-based semantics then could be deployed to process new inputs continuously.

7 Related Work

Trigger-based computation, especially in the active database field, was widely investigated in the 1990s and has been implemented in many commercial database productions [34], [35], [36]. They also follow the ECA model as Domino does. Most of these triggers in databases are used in the context of data validity and materialization of views. Integrity or validity of data is specified by a set of *semantic constraints* or *assertions*, and triggers may be used as extended assertions to enforce the validity. Triggers in Domino are different from triggers in databases since Domino is a general programming model.

Many popular distributed programming frameworks focus on synchronous computation and rely heavily on global barriers between iterations, for example, MapReduce [3], Dryad [4], and their extensions [9], [10], [37], [38], [39]. Compared with these frameworks, Domino supports the synchronous incremental applications effectively. Some computation frameworks support incremental applications in an asynchronous way, including Oolong [12] and Percolator [13]. Oolong [12], based on the Piccolo [40] storage system, provides asynchronous support for applications whose intermediate states can fit in the aggregated memory of the cluster. Percolator [13] provides a transaction to support serialization. Neither Oolong nor Percolator, however, provide an efficient way to synchronize different executions. In Domino, we introduce the wait-free eventual synchronization, which extends the usability of a trigger-based model. Domino can be used in a variety of applications including many synchronous machine-learning or data-mining applications.

Recently, lots of distributed frameworks aiming for specific purpose have been proposed, like Storm[14], S4[15], GraphLab[20] etc. We can basically divided them into different categories according to their aiming applications. S4 and Storm usually are used in stream processing for real-time situation, but both of them are limited in asynchronous stream processing without supporting on static datasets processing like Domino can provide. So, in most production deployment, Storm or S4 need to work with Hadoop, which complicates the management of cluster. On the other hand, Domino was able to support both incremental streams processing and also static dataset processing. These advantages make Domino more suitable in production environment along facing the real-time or stream processing challenges. GraphLab[20] offers asynchronous features for graph-based problem. It focuses on providing sequential consistency for such computation. It is not designed for general incremental applications, however, and it requires developers to abstract their algorithms graphically. Domino provides an easily understood abstraction and is more universal for different kinds of incremental applications.

8 Conclusions & Future Work

In this study, we have presented Domino, a trigger-based incremental programming framework with unified sync and async mechanism based on event-driven programming models. We have introduced the necessity of synchronization in an incremental processing framework, and we have presented a novel design and implementation of this wait-free eventual synchronization. We have also introduced several optimization strategies, including *execution preemption*, *result reuse*, and *gathered I/O*, to achieve the desired performance. The use cases and experimental results have confirmed the efficiency and performance. Our next step will be providing native support for MapReduce applications in Domino runtime.

Acknowledgment

This material is based upon work supported by the U.S. Department of Defense; by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357; and by the National Science Foundation under grant CNS-1338078, IIP-1362134, CCF-1409946, and CCF-1718336.

References

[1] "Amazon ec2," in *http://aws.amazon.com/*.
[2] Azure, "http://www.windowsazure.com/en-us/."
[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113.
[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, 2007.
[5] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX Association, 2012, pp. 10–10.
[6] D. Dai, X. Li, C. Wang, and X. Zhou, "Cloud based short read mapping service," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*. IEEE, 2012, pp. 601–604.
[7] D. Dai, X. Li, C. Wang, J. Zhang, and X. Zhou, "Detecting associations in large dataset on mapreduce," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*. IEEE, 2013, pp. 1788–1794.

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2011.

[9] Y. Bu, B. Howe, M. Balazinska, and M. Ernst, "Haloop: Efficient iterative data processing on large clusters," in *Proceedings of the VLDB Endowment*, vol. 3. VLDB Endowment, 2010.

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

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

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

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

[14] "Storm project," in *http://storm-project.net/*.

[15] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE, 2010.

[16] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 439–455.

[17] X. Chenning, C. Rong, G. Haibing, Z. Binyu, and C. Haibo, "Sync or async: Time to fuse for distributed graph-parallel computation," Shanghai Jiao Tong University, Tech. Rep., 2014.

[18] "Hbase project," in *http://hbase.apache.org*.

[19] J. Gonzalez, Y. Low, and C. Guestrin, "Residual splash for optimally parallelizing belief propagation." Aistats, 2009.

[20] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8. [Online]. Available: http://arxiv.org/pdf/1204.6078

[21] D. McCarthy and U. Dayal, "The architecture of an active database management system," *ACM Sigmod Record*, vol. 18, no. 2, pp. 215–224, 1989.

[22] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[23] D. Dai, X. Li, C. Wang, M. Sun, and X. Zhou, "Sedna: A memory based key-value storage system for realtime processing in cloud," in *Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on*. IEEE, 2012, pp. 48–56.

[24] W. Xie and Y. Chen, "Elastic consistent hashing for distributed storage systems," in *Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 876–885.

[25] G. Kollias, E. Gallopoulos, and D. B. Szyld, "Asynchronous iterative computations with web information retrieval structures: The pagerank case," *arXiv preprint cs/0606047*, 2006.

[26] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web," *Standford InfoLab*, 1999.

[27] R. Ricci and E. Eide, "Introducing cloudlab: Scientific infrastructure for advancing cloud architecturesand applications," *; login:*, vol. 39, no. 6, pp. 36–38, 2014.

[28] HBasePerformance, "https://issues.apache.org/jira/browse/hbase-399."

[29] S. Ghemawat, H. Gobioff, and S. Leung, "The google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM.

[30] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006.

[31] "Hbase coprocessor," in *http://blogs.apache.org/hbase/entry/coprocessor-introduction*.

[32] "Omid project," in *https://github.com/yahoo/omid*.

[33] "Zookeeper project," in *http://zookeeper.apache.org/*.

[34] M. Darnovsky and J. Bowman, "Transact-SQL user's guide," *Sybase Inc., Doc*, pp. 3231–2, 1987.

[35] T. Andrews and C. Harris, "Combining language and database advances in an object-oriented development environment," in *ACM Sigplan Notices*, vol. 22, no. 12. ACM, 1987, pp. 430–440.

[36] G. Schlageter, R. Unland, W. Wilkes, R. Zieschang, G. Maul, M. Nagl, and R. Meyer, "Oops–an object oriented programming system with integrated data management facility," in *Proceedings. Fourth International Conference on Data Engineering, 1988.* IEEE, 1988, pp. 118–125.

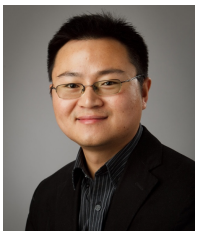
[37] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. USENIX Association, 2010, pp. 10–10.

[38] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: A distributed framework for prioritized iterative computations," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 13.

[39] H. Yang, A. Dasdan, R. Hsiao, and D. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of Data*. ACM, 2007, pp. 1029–1040.

[40] R. Power and J. Li, "Piccolo: building fast, distributed programs with partitioned tables," in *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*, 2010.

Dong Dai Dong Dai is a research assistant professor and associate director of the Data-Intensive Scalable Computing Laboratory in the Computer Science Department at Texas Tech University. His research interests include cloud computing, programming frameworks, HPC I/O scheduling, metadata management, and graph computing.



Yong Chen Yong Chen is an Assistant Professor and Director of the Data-Intensive Scalable Computing Laboratory in the Computer Science Department of Texas Tech University. He is also an Associate Director of the Cloud and Autonomic Computing center at Texas Tech University. His research interests include data-intensive computing, parallel and distributed computing, high-performance computing, and cloud computing. More information about him can be found at http://www.myweb.ttu.edu/yonchen/.

Dries Kimpe Dries Kimpe is a Senior Software Developer at KCG and a lecturer at the University of Chicago. His research interests include parallel I/O, distributed algorithms and high-performance communication.

Robert Ross Rob Ross is a senior scientist at Argonne National Laboratory and senior fellow in the Northwestern-Argonne Institute for Science and Engineering and in the University of Chicago/Argonne ComputationInstitute. His focus area for over two decades has been data and communication system software forhigh performance computing. He leads a team with a strong track record in the development of open-source software packages for scientific computing such as the PVFS parallel file system, ParallelnetCDF I/O library, ROMIO MPI-IO implementation, and Darshan I/O characterization projects.