Tornado: A System For Real-Time Iterative Analysis Over Evolving Data

Xiaogang Shi Bin Cui Yingxia Shao Yunhai Tong
Key Lab of High Confidence Software Technologies (MOE), School of EECS, Peking University
{sxg, bin.cui, simon0227, yhtong}@pku.edu.cn

ABSTRACT

There is an increasing demand for real-time iterative analysis over evolving data. In this paper, we propose a novel execution model to obtain timely results at given instants. We notice that a loop starting from a good initial guess usually converges fast. Hence we organize the execution of iterative methods over evolving data into a main loop and several branch loops. The main loop is responsible for the gathering of inputs and maintains the approximation to the timely results. When the results are requested by a user, a branch loop is forked from the main loop and iterates until convergence to produce the results. Using the approximation of the main loop, the branch loops can start from a place near the fixed-point and converge quickly.

Since the inputs not reflected in the approximation is concerned with the approximation error, we develop a novel bounded asynchronous iteration model to enhance the timeliness. The bounded asynchronous iteration model can achieve fine-grained updates while ensuring correctness for general iterative methods.

Based on the proposed execution model, we design and implement a prototype system named Tornado on top of Storm. Tornado provides a graph-parallel programming model which eases the programming of most real-time iterative analysis tasks. The reliability is also enhanced by provisioning efficient fault tolerance mechanisms. Empirical evaluation conducted on Tornado validates that various real-time iterative analysis tasks can improve their performance and efficiently tolerate failures with our execution model.

1. INTRODUCTION

Real-time data analysis is very challenging as we must produce timely results at required instants while data continuously arrives and evolves over time. The ability of stream processing systems to achieve low latencies makes them attractive to real-time analysis. Recently, many stream processing systems have been proposed to facilitate real-time data analysis at scale [42, 35, 46]. But these systems are short of applicability as they provide little support for iterative methods which are often deployed in data analysis tasks.

Many machine learning and data mining problems are solved by starting with an initial guess and repeatedly refining the solution with iterations. These algorithms are traditionally performed on static datasets; but the requirement for timeliness now is leading to a shift to adaptive learning and stream mining [22, 49]. Given the wide adoption of iterative methods in data analysis and the increasing demand for timeliness, it's critical for us to develop a system that can perform real-time iterative analysis over evolving data.

A naive method to perform real-time computation over evolving data is collecting the inputs continuously and performing the computation when the results are required. It allows the deployment of existing systems designed for static data, e.g., Spark and GraphLab. Though these systems are efficient to process data with large volume, they usually experience high latencies because they have to load all collected data and perform computation from scratch.

Some attempts made on adaptive systems, e.g. Incoop [7] and Kineograph [13], provided methods to track the modification to the inputs so that they can reduce the cost of data collection. They also employed incremental methods to avoid redundant computation. But the incremental methods enforce a order on the execution of the requests. The enforced order disallows them to produce timely results at given instants because new inputs cannot be processed before the current computation completes. To reduce the dependencies between different epoches, Naiad [32, 33] decomposed difference traces into multiple dimensions and allowed them to be combined in different ways for different versions. The relaxation in dependencies allows an epoch to be processed without waiting for the completion of preceding epochs.

In all these systems, the latency is dominated by the amount of computation to perform at the required instant. Mini-batch methods are widely adopted to reduce the latency. In mini-batch methods, the inputs are organized into a series of epochs and the results are updated when an epoch ends. When the results at an instant is required, they can be incrementally computed from the results of the last epoch. Because the amount of updates to be processed at the required instant is limited by the epoch size, we can achieve a relatively low latency.

Though mini-batch methods are proved useful in non-iterative computation over data streams, they are not suited for real-time iterative computation. A mini-batch method works only when the cost to incrementally maintain the results scales sub-linearly or linearly with the number of updates. But the studies in adaptive computing suggest that the requirement is not satisfied in many iterative methods [16]. For example, when a graph is updated, the time needed to incrementally update PageRank results is proportional to the current graph size, but not the number of updated edges [5]. Therefore, shrinking epochs will not help reduce the latency.

In this paper, we propose a novel execution model that allows real-time iterative analysis over evolving data. We notice that *a loop starting with a good initial guess usually converges faster than*

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

ISBN 978-1-4503-3531-7/16/06. . . $15.00

DOI: http://dx.doi.org/10.1145/2882903.2882950

those starting with poor guesses. If we can obtain a good approximation of a loop's results, then we can efficiently reduce the computation time of the loop. In mini-batch methods, the results of the last completed epoch are used as the initial guesses. But because the most recent inputs are not reflected in the initial guesses, the approximation error is relatively high. The observation motivates us to organize the computation into two separated parts: a main loop and several branch loops.

The main loop continuously gathers the incoming inputs and approximates the results at each instant. There already exist many studies in the approximation for iterative methods in the context of streaming mining [40, 14, 16]. Users can deploy them to approximate the initial guesses for specific iterative methods. For a wider range of iterative methods, users can approximate the initial guesses by performing the iterative methods on partial inputs. These approximation methods can provide better initial guesses than mini-batch methods because their computation complexity are much lower than the incremental counterparts. They can catch up with the evolving data, adapting the approximation to the changes.

Different branch loops correspond to different queries over the data. When a user issues a request to retrieve the results at a certain instant, a branch loop will be forked from the main loop and start its execution to return the precise results. The computation of different branch loops are independent of each other. They can benefit from the approximation of the main loop. Because the main loop can provide a good initial guess, they can start from a place near the fixed-point and converge quickly to return results.

In our execution model, the key to achieve low latency is reducing the approximation error. Because synchronous iteration models require global barriers to perform iterations, they are unable to gather inputs before all local computation completes. Many inputs are not reflected in the approximation. These inputs significantly increase the approximation error. On the other hand, asynchronous iteration models allow local computation to proceed to the next iteration individually. The ability to achieve low latencies and fine-grained updates allow them to efficiently adapt the approximation. But unfortunately, asynchronous iteration models cannot ensure correctness for many iterative methods when the degree of asynchronism is not bounded [12, 19, 37, 26].

To allow fine-grained updates while ensuring correctness for most iterative methods, we deploy a bounded asynchronous model to perform iterations. Though there exist some bounded asynchronous iteration models implemented in Parameter Servers [15, 26, 29, 45], they are designed for specific iterative methods. It's difficult for them to perform common graph analysis like PageRank and Shortest Single-Source Path. They are also unable to deal with retractable data streams. Different from these existing models, our iteration model utilizes data dependencies to manage iterations. With a three-phase update protocol, it can ensure correctness and consistency for general iterative methods over evolving data.

Based on the execution model of loops and iterations, we design and implement a prototype system named Tornado on top of Storm [42]. Tornado adopts a graph-parallel programming model which is similar to those in popular iterative computation systems [31, 30, 23]. Users can easily develop their applications to perform iterative analysis over data streams. Efficient fault tolerance and recovery mechanisms are provided so that user applications can achieve fault-tolerated execution at scale.

We implement many data analysis algorithms in Tornado and conduct a comprehensive analysis to study the benefits and costs of our designs. The evaluation demonstrates that Tornado can achieve significant performance gains from the main loop's approximation and can efficiently tolerate failures.

To summarize, the main contributions of our work include:

- We carefully analyze the challenges of real-time iterative data analysis over evolving data as well as the benefits and limitations of existing methods.
- We propose a novel execution model which can improve the timeliness without enforcing any partial orders.
- We propose a bounded asynchronous iteration model which provides correct and efficient execution.
- We design and implement a prototype system which allows easy programming and fault-tolerated execution at scale.
- We conduct extensive empirical studies on the prototype system. The experiments with various common iterative workloads validate that the execution can efficiently improve their performance while tolerating unexpected faults.

The rest of the paper is organized as follows. We first briefly review iterative computation in Section 2. Then we introduce the execution model for loops in Section 3. The details of the bounded asynchronous iteration model is given in the Section 4. The implementation and evaluation of Tornado are presented in Section 5 and 6 respectively. Finally we review the related work in Section 7 and conclude the work in Section 8.

2. ITERATIVE COMPUTATION

The researchers in the community of numerical analysis have put a lot of effort in implementing iterative methods under parallel and distributed settings. In this section, we briefly review the iteration models and their convergence properties. By extending these studies, we lay a sound theoretical foundation for our work, achieving guaranteed correctness for many real-time iterative methods.

2.1 Iterative Methods

Many iterative methods have similar programming structures. Suppose x_i is the i-th variable in the program and D_i is the domain of x_i. The state of a program with n variables then can be represented by $\sigma = (x_1, x_2, ..., x_n), \sigma \in \Sigma$ and $\Sigma = \mathrm{D}_1 \times \mathrm{D}_2 \times ... \times \mathrm{D}_n$. x_i is also called a component of σ, $i = 1, 2, ..., n$.

Given a function $f : \Sigma \to \Sigma$, the problem is to find a fixed point of f which is an element $\sigma^\infty \in \Sigma$ and $f(\sigma^\infty) = \sigma^\infty$. It's usually solved by starting from an initial guess $\sigma^0 \in \Sigma$ and iteratively applying the function:

$$\sigma^{k+1} = f(\sigma^k) = f^k(\sigma^0), k \in \mathbb{N} \quad (1)$$

σ^k represents the value of σ in the k-th iteration and $f^k(\sigma^0)$ is the results if we start f from σ^0 and repeat the execution of f for k times. σ^∞ is approximated by $\sigma^{\hat{k}}$ where $|\sigma^{\hat{k}+1} - \sigma^{\hat{k}}| \leq \varepsilon$ and ε is the precision required in the computation.

2.2 Iteration Models

When performing iterative methods in parallel, the components of σ are distributed over a set of processing units. Different processing units communicate with each other through shared memory or message passing.

The execution of synchronous iteration models strictly obeys the definition of iterative methods. In each iteration, the processing units read the value of σ in the previous iteration and update the values of all components. The computation proceeds to the next iteration only when all components complete their updates.

The requirement for synchronization however is eliminated in asynchronous iteration models [12]. In asynchronous iteration models, a processing unit can update the components without the most

recent value of σ and can delay the computation of some components for several iterations.

Let $\tau^k_{i\to j}$ be the iteration number of x_i that is read in the computation of x^k_j, S_i be the scheduling strategy of x_i which contains the iterations in which x_i is updated. Both synchronous and asynchronous iterations can be mathematically modeled as follows [6]:

Definition 1 (General Iteration Model) *Given an initial guess $\sigma^0 \in \Sigma$ and the scheduling strategies for all components $\{S_0, S_1, ...\}$, the iterative method:*

$$x^k_i = \begin{cases} x^{k-1}_i, & i \notin S_i \\ f_i(x_1^{\tau^k_{1\to i}}, x_2^{\tau^k_{2\to i}}, ..., x_n^{\tau^k_{n\to i}}), & i \in S_i \end{cases} \quad (2)$$

is said valid if the following assumptions hold:

1. ***Causality.*** *Only the values produced in previous iterations can be used in the computation, i.e.*

$$0 \le \tau^k_{i\to j} \le k-1, k \in \mathbb{N}$$

2. ***Eventual Scheduling.*** *Though the updates of some components may be delayed, these components will eventually be updates as time goes on, i.e.*

$$|S_i| = \infty, i = 1, 2, ..., n$$

3. ***Eventual Propagation.*** *A component will eventually receive new values of other components, i.e.*

$$\lim_{k\to\infty} \tau^k_{j\to i} = \infty, i, j = 1, 2, ..., n$$

$\{k - \tau^k_{j\to i}\}$ are the *delay*s (or staleness in [26]) in the computation of x^k_i. When $S_i = \mathbb{N}$ and $k - \tau^k_{j\to i} = 1$, the iterations are synchronous; otherwise, the computation is asynchronous.

2.3 Bounded Asynchronous Iteration Models

As asynchronous iteration models eliminate the synchronization between different iterations, they can only ensure correctness for limited iterative methods. The convergence properties of asynchronous iterations are well understood in both linear and nonlinear cases [12, 19, 21]. These studies suggest that for most common iterative methods, including Stochastic Gradient Descent and Markov Chains, convergence is guaranteed only when the delays are bounded. Hence a bounded asynchronous iteration model is applicable for more iterative methods.

Definition 2 (Bounded Asynchronous Iterations) *A valid iterative method is said to be bounded asynchronous if there exists a delay bound $B \in \mathbb{N}$ s.t.*

1. *The delays of all components are less than B, i.e.*

$$k - B < \tau^k_{i\to j} < k, i, j = 1, 2, ..., n, k \ge 0$$

2. *All components are updated at least once in B iterations, i.e.*

$$\{k, k+1, ..., k+B-1\} \cap S^i \neq \emptyset, i = 1, 2, ..., n, k \ge 0$$

Obviously, a synchronous iteration model is also bounded asynchronous. Its delay bound B is 1.

3. REAL-TIME ITERATIVE COMPUTATION

In this section, we will introduce the problem of real-time iterative analysis over evolving data and describe the proposed execution model. We explicitly distinguish loops from iterations in this paper. We refer the procedure to obtain the value of f^∞ as a *loop* and each execution of f as an *iteration*. Obviously, the execution of a loop contains a number of iterations. A loop converges when its fixed-point is reached.

3.1 Problem Definition

In the real-time iterative analysis over evolving data, the inputs are continuously updated by external sources. The input data, $\mathcal{S}$ can be described as a unbounded sequence of stream tuples. Each stream tuple δ_t is associated with a timestamp t to identify the time when δ happens. In this paper, for simplicity and without loss of generality, we adopt the turnstile stream model [34] and assume that all stream tuples are updates to $\mathcal{S}$. The value of $\mathcal{S}$ at an instant $\bar{t}$ can be obtained by summing up all updates happening before $\bar{t}$:

$$\mathcal{S}_{[\bar{t}]} = \sum_{t\le\bar{t}} \delta_t$$

Definition 3 (Real-time Iterative Analysis over Evolving Data) *Given an instant t and a method f whose input data is $\mathcal{S}$, the problem of real-time iterative analysis over evolving data is to obtain the fixed-point of f at the instant t s.t.*

$$\sigma^\infty_{[t]} = \sigma = f(\mathcal{S}_{[t]}, \sigma)$$

A typical example of the problem is observed in search engines. In a simplified setting, the input of a search engine is a retractable edge stream which is produced by the crawlers. Iterative methods like PageRank are deployed to rank collected pages. When new pages are collected or old pages are updated, the search engine has to update its repository and adapt the ranking to the changes. To reflect the most recent updates, search engines usually updates its results at regular intervals, e.g. hourly or daily. In more general scenarios, a user may also commit ad-hoc queries to retrieve the results at specific instants. In all cases, the latencies of the queries should be as low as possible.

3.2 Design Consideration

Many existing systems organize the inputs into batches and deploy incremental methods to obtain timely results. They assume that the time complexity to incrementally update the results is proportional to the amount of new inputs. Under such assumption, the latency can be reduced by shrinking the batch as much as possible. But such assumption does not hold for most iterative methods. Consequently, the timeliness they achieve is limited.

The difficulty to develop efficient incremental methods to compute the results forces us to find alternative methods to satisfy the requirement of iterative analysis over evolving data. Our design is heavily encouraged by the following observations.

Observation One: Many iterative methods can converge to the same fixed-point with different initial guesses. The observation can be made in many iterative methods. For example, a Markov Chain starting from various initial distributions can converge to the same stationary distribution only if its transformation matrix is irreducible and aperiodic.

Definition 4 (Equivalence of Initial Guesses) *Given two different initial guesses σ^0_i and σ^0_j, we say they are equivalent in terms of input x if the loops starting from them can converge to the identical fixed-point, i.e.*

$$\sigma^0_i \sim \sigma^0_j \leftrightarrow f^\infty(x, \sigma^0_i) = f^\infty(x, \sigma^0_j)$$

Though loops from equivalent initial guesses will converge to the same fixed-point, the initial guesses have different effects on the convergence rate.

Observation Two: For many iterative methods, the execution time is closely relative to the approximation error. A loop starting from a good initial guess usually converges faster than those starting from bad initial guesses. If we can obtain a good initial guess, we can efficiently improve the performance of a loop.

Numerous methods have been proposed to approximate the results of iterative methods [40, 14]. These approximate methods all can provide a good approximation for the iterative computation over evolving data. Because their computation complexity is much less than the original iterative method, they can quickly adapt the approximation to the input changes.

Suppose at the time t, we have already obtained an approximation $\tilde{\sigma}_{[t]}$ with the approximation method g. We can produce the results of $\sigma^{\infty}_{[t]} = f^{\infty}(\mathcal{S}_{[t]}, \sigma^0)$ in two steps.

In the first step, we use an adaptive method $\mathcal{A}$ to obtain an valid initial guess from the approximation:

$$\sigma^0_{[t]} = \mathcal{A}(\tilde{\sigma}_{[t]}), \text{ where } \sigma^0_{[t]} \sim \sigma^0$$

In the second step, we iteratively perform the original method f to obtain the fixed point:

$$\sigma^{\infty}_{[t]} = f^{\infty}(\mathcal{S}_{[t]}, \sigma^0_{[t]})$$

As both the approximate method g and the original method f can be computed incrementally, we can avoid the difficult problem of incremental computation between different loops. Now the computation of $\sigma^{\infty}_{[t]}$ does not depend on previous results, $\sigma^{\infty}_{[t-\Delta t]}$. By not enforcing any order between the computation of different requests, we are able to produce timely results for ad-hoc queries. But to ensure correctness and benefit from the approximation, the following two conditions must be satisfied:

- **Correctness Condition.** $\tilde{\sigma}_{[t]}$ is equivalent to σ_0 in terms of the input $\mathcal{S}_{[t]}$.
- **Profitability Condition.** The execution time of f depends on the approximation error of the initial guesses and the approximate error made by g is kept small enough during the evolvement of $\mathcal{S}$.

Fortunately, the two conditions can be satisfied in most common iterative methods with minor changes. For example, Stochastic Gradient Descent (SGD) avoids the costly computation of gradients by approximating the gradient with an unbiased sampling of observation instances. The unbiased sampling is usually done by random sampling. But it's problematic because when the observation instances evolves over time, the old instances will be sampled more times than the new ones. The results produced by the approximate method hence are not valid initial guesses. In these cases, we should use reservoir sampling [44] instead to obtain the unbiased sampling of data streams. The correctness condition can be satisfied as reservoir sampling can ensure that all instances are sampled with identical possibility, regardless of the time when they come in.

3.3 Execution Model

Based on the ideas described above, we organize the iterative computation over evolving data into a main loop and several branch loops. The execution model of loops is illustrated in Figure 1.

The main difference made by the execution model of loops is that the collection of inputs and the computation of iterations are interleaved. In the main loop, we continuously gather incoming data and approximate the results at current instant. Let $\Delta\mathcal{S}_i$ be the inputs collected in the i-th iteration and $\tilde{\sigma}^i$ be the approximation in the i-th iteration, then we have $\tilde{\sigma}^i = g(\sum_{j=1}^{i} \Delta\mathcal{S}_j, \tilde{\sigma}^{i-1})$, where g is the approximation function.

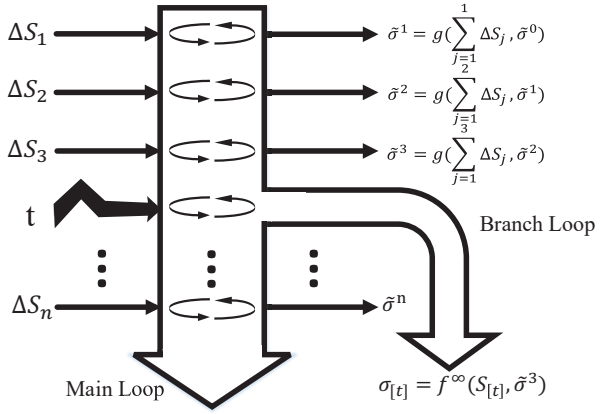


Figure 1: Illustration of the execution model of loops.

Users can issue their requests at regular intervals or send ad-hoc requests at specific instants. When the result at an instant t is required, we take a snapshot of $\mathcal{S}$ and fork a branch loop to perform iterations over the taken snapshot. Suppose by the time the user issues a request, the main loop has completed the execution of the i-th iteration. We can start the branch loop with the approximation obtained from the gathered input $\sum_{j=1}^{i} \Delta\mathcal{S}_j$. Though the inputs collected in the i-th iteration are not reflected in the approximation, the approximation error can be very small if $\|\mathcal{S}_{[t]} - \sum_{j=1}^{i} \Delta\mathcal{S}_j\|$ is small enough. The fixed-point at t, $\sigma^{\infty}_{[t]}$, is obtained when the branch loop converges.

Our execution model does not rely incremental methods to reduce the latency. Instead, the key to achieve low latency is reducing the approximation error. By starting from a place near the fixed-point, the performance of branch loops is improved. As the performance improvement is not gained from incremental methods, we avoid the partial orders between different loops. Besides, since the computation performed in the main loop can be shared by all branch loops, the overall amount of computation is also reduced.

4. ITERATION MODEL

To reduce the approximation error of initial guesses, the amount of the inputs that are not reflected in the approximation should be very small. To the satisfy the requirement, we propose a novel partially asynchronous iteration model which can guarantee correctness for most common iterative methods. We introduce the details of the iteration model in this section.

4.1 Determining Iterations

A key issue in the iteration management is to determine the iteration in which each update is performed. The iteration information then can be used to track progress and bound the iteration delays.

It's easy to determine iterations in synchronous models as all components must proceed to the next iteration with global synchronization barriers. Therefore, the iteration numbers of all components are the same. But as these global synchronization barriers don't allow fine-grained updates, we must develop a different method to determine the iteration of each update.

To support iterative computation over evolving data, we start from the general iteration model described in Section 2.2 and utilize the dependencies between different components to determine the iteration of each update.

Definition 5 (Dependency) *We say a component x_i depends on x_j if the value of x_j affects the value of x_i. If x_i depends on x_j, we call x_j is a* producer *of x_i. x_i in turn is a* consumer *of x_j.*

The dependencies can be easily obtained in many iterative methods where the dependencies are explicitly expressed as the edges of graphs. With the dependencies between different components, we can reason about the causality in the iterative computation. Suppose x and y are two components and y is a producer of x. We consider the cases when x receives the update of y to motivate our update rules. The iteration number of x and the update of y are τ_x and τ_y respectively:

- If $\tau_y \geq \tau_x$, it's indicated that y is updated in an iteration later than x. Hence according to Eq. 1, x should be updated in the next iteration. Therefore, we should set τ_x to $\tau_y + 1$. The iteration numbers of x may not be continuous as we allow delaying the update of a component. In this case, the update of x is not scheduled between the iterations from $\tau_x + 1$ to τ_y, i.e. $\{\tau_x + 1, ..., \tau_y\} \subseteq \mathbb{N} - S_x$.
- If $\tau_y < \tau_x$, y propagates to x an update which is produced in an iteration earlier than τ_x. Inconsistency arises in this case. Since x has proceeded to the iteration τ_x, it must have received an update from its producers in an iteration before τ_x. When x is updated, it doesn't observe the update of y in iteration τ_y. Now that y is updated, the fact is inconsistent with x's observation.

From the discussion above, we can see that to ensure consistency, the iteration in which a component is updated cannot be earlier than the iterations of its consumers. The following rules hence can be derived to ensure the validity of the iteration model:

1. When a component is updated in an iteration, the update should be observed by its consumers in the next iteration.
2. A component should be updated when it observes the update to its consumers, but the update can be delayed.
3. The iteration in which an update is performed cannot be later than the iterations of its consumers.

Rule 1 follows the definition of iterative methods, and Rule 3 is actually derived from the contraposition of Rule 1. By delaying the update of the component, we can achieve a valid and consistent iteration model with Rule 2. These update rules can ensure the validity and consistency of iteration models. An example which is executed using the update rules is illustrated in Figure 2.

Here we emphasize that the scheduling of updates is only *logical*. The concept of *scheduling strategy* helps us better understand the execution of our iteration model; but in practice, there does not exist a scheduler to update components according to the scheduling strategy. When a component receives an update, it can update itself and propagate its update immediately with the iteration information gained from its consumers.

4.2 The Three-Phase Update Protocol

The update rules imply that to determine the iteration number of a component, we should gather the iteration numbers of its consumers. Since all components are updated asynchronously and communicate with each other using message passing, it's possible that a component is updated after it replies to a producer with its iteration number. Inconsistency will arise in these cases.

To ensure that all updates are consistent with the observations, a component cannot update itself when it's still involved in the updates of its producers. Intuitively, the problem is a special case

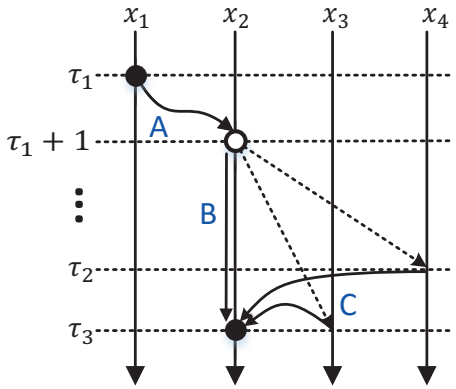


Figure 2: Illustration of the iteration model. x_2 is a consumer of x_1 and also a producer of x_3 and x_4. Suppose x_1 is updated in the iteration τ_1, then x_2 should update itself accordingly. The update of x_2 are performed in three steps: (A) When x_1 is updated, x_2 will observe the update in the next iteration and is prepared to update itself. (B) The update of x_2 however is *"delayed"* to make the scheduling strategy consistent with the observation of x_3 and x_4. (C) By the time x_2 is to update, x_3 and x_4 have proceeded to the iterations numbered with τ_2 and τ_3 respectively. In the progress towards the current iterations, they don't observe the update of x_2. Therefore, to make the scheduling strategy consistent with their observations, we update x_2 in an iteration whose number is no less than $\max(\tau_2, \tau_3)$.

of the well-known problem of *dining philosophers* [17]. When a component is to update itself, it must obtain the forks of its consumers and its consumer cannot update itself before the borrowed fork is returned. Generally, the dining philosophers problem can be solved by the Dijkstra [17] and Chandy-Misra [11] algorithms, but they both target at static dependency graphs and are not applicable in our cases where the dependency graph is continuously modified by the evolving inputs. Since the Dijkstra algorithm requires a component to acquire locks in sequence, it's inefficient when a component has a large number of neighbors or the list of neighbors is unknown at advance. The Chandy-Misra algorithm is also problematic as it avoids deadlocks by enforcing the resource priority graphs to be acyclic. When the algorithm starts, an initial acyclic priority graph is obtained by assigning forks to the nodes with lower IDs and no cycles are allowed to arise during the execution. But when new forks are created during the computation, it's difficult for us to assign these new forks. Cycles may arise if we continue assigning forks to the nodes with lower IDs.

To ensure the consistency of our iteration model, we adapt Chandy-Misra to our cases with a method based on Lamport Clocks [28]. Each component maintains a Lamport clock to record the time when it's updated. A component only replies to its producers when it's not updated or its update happens after that of its producers. Since an update is only blocked by the updates that *happen before* it, the implied partial orders make deadlocks and starvation impossible. As the partial orders in the execution are enforced by the logical clocks but not the acyclic priority graphs, we are flexible to deal with the evolvement of the dependency graph. The pseudo code of our iteration model is illustrated in Figure 3. The update of a component x is performed in three phases:

Update Phase. When x receives an input or an update, it first gathers the input/update and updates its iteration number. Suppose the iteration numbers of x and the update are $\tau(x)$ and $\tau(\delta)$ respectively. To preserve the causality, the new iteration number of

```
1  /* phase 1: gather updates from its producers */
2  OnReceiveUpdate(producer, iteration, data)
3    this.iteration = max(this.iteration, iteration+1);
4    /* gather the update with the user-defined function */
5    gather(id, iteration, data);
6    this.prepare_list.remove(producer);
7    /* prepare the update when it's not involved in any preparation */
8    if this.prepare_list is null and this.update_time is null
9      call OnPrepareUpdate();
10
11 /* phase 2: prepare to update itself */
12 OnPrepareUpdate()
13   this.update_time = LamportClock::get_time();
14   foreach consumer in this.consumer_list {
15     send (PREPARE, this.id, this.update_time) to the consumer
16     this.waiting_list.add(consumer);
17   }
18
19 OnReceivePrepare(producer, update_time)
20   this.prepare_list.add(producer);
21   /* only acknowledge those updates happening before */
22   if this.update_time is null or this.update_time > update_time
23     send (ACKNOWLEDGE, this.id, this.version) to the producer
24   else
25     this.pending_list.add(producer);
26
27 OnReceiveAcknowledge(consumer, iteration)
28   this.iteration = max(this.iteration, iteration);
29   this.waiting_list.remove(consumer);
30   /* commit the update when all its consumers have acknowledged */
31   if this.waiting_list is null
32     call OnCommitUpdate();
33
34 /* phase 3: commit the update to its consumers */
35 OnCommitUpdate()
36     this.update_time = null;
37     /* commit the update with the user-defined function */
38     scatter(UPDATE, this.id, this.iteration);
39     /* reply to all pending producers */
40     foreach producer in this.pending_list
41       send (ACKNOWLEDGE, this.id, this.iteration) to the producer
```

Figure 3: The pseudo code of the three-phase update protocol

x should be greater than $\tau(\delta)$. Therefore we update the iteration number of x as follows:

$$\tau'(x) = \max(\tau(x), \tau(\delta) + 1)$$

where $\tau'(x)$ is the new iteration number of x. If the update comes from one of the x's producers, x will also remove the producer from its `PrepareList`. When the `PrepareList` is empty, it can prepare its update. (`OnReceiveUpdate`, lines 1-9)

Prepare Phase. In the second phase, x first retrieves the Lamport clock to obtain the time it's updated. Then x should determine the iteration in which its update is scheduled. It sends `PREPARE` messages to its consumers to collect their iteration numbers. The timestamp obtained from the Lamport Clock is also attached in the message. (`OnPrepareUpdate`, lines 11-17)

Once a consumer y of x receives the `PREPARE` message from x, it replies to x with an `ACKNOWLEDGE` message which contains its iteration number. Then y will put x in its `PrepareList`. As a component can only start its update when its `PrepareList` is empty, to avoid the starvation of the update of itself, y will not acknowledge x if it's already updated and its update happens before that of x. Therefore y will not acknowledge a `PREPARE` message if it has updated itself and the update happens before the producer of the `PrepareMessage`. In these cases, y will put the `PREPARE` messages in a pending list. (`OnReceivePrepare`, lines 19-25)

When x receives the `ACKNOWLEDGE` message from its consumers, it will update its iteration number accordingly. During the preparation, x can continue gathering updates and acknowledging `PREPARE` messages. But it will not gather any input because new inputs may change the dependency graph. The results of the preparation will be incorrect if new consumers are added to x. (`OnReceiveAcknowledge`, lines 27-32)

Commit Phase. When x has received all the iteration numbers of its consumers and its `PrepareList` is empty, it can commit its update. The iteration in which x is updated, $\tau'(x)$, must be the maximum iteration number of all its consumers, i.e.

$$\tau'(x) = \max\Big(\tau(x), \tau(y_1), \tau(y_2), ..., \tau(y_n)\Big)$$

where y_i is a consumer of x. x will commit its new value and iteration number to all its consumers with `COMMIT` messages. It will also acknowledge all pended `PREPARE` messages (if any) with its new iteration number. (`OnCommitUpdate`, lines 35-41)

The timestamp obtained by x will be cleaned after the commitment. Now x can gather the inputs that come in during the preparation of the update. The update protocol will be executed again when x gathers any inputs or updates.

4.3 Terminating Iterations

In existing totally asynchronous iteration models, there is no method to detect the termination of an iteration. A loop converges when all components are updated and no updates are in transition. But in distributed settings, it's difficult to detect whether there are updates in transition. Some systems [30, 23] attempt to detect the convergence of loops using the Chandy-Lamport algorithm [10]. But their usage is problematic as the Chandy-Lamport algorithm only works for the detection of stable properties. The convergence of asynchronous loops however is not stable.

With the iteration numbers of each update, we can easily determine the termination of an iteration and step further to detect the convergence of a loop. According to the causality of iteration models, a component's update in an iteration will cause its consumers to update themselves a later iteration. Eventually all components will proceed to the succeeding iterations. The property implied by the causality allows us to terminate an iteration when all iterations preceding it have terminated and all vertices have proceeded to the iterations succeeding it.

The convergence of a loop is as same as the synchronous iteration models. When an iteration terminates, the loop can track the progress made in the iteration and evaluate the convergence condition with the progress. Generally a loop can converge when no updates are performed in an iteration.

A major difference made by our iteration model is that the iteration in which a vertex is updated is determined by the iterations of its consumers. In other systems, e.g., Naiad, the values of a vertex's versions form a continuous sequence. When a vertex completes its update in an iteration, its iteration number will be added by one. But in our iteration model, the iteration numbers of a vertex may not be continuous.

4.4 Bounding Delays

To ensure correctness for a wide variety of iterative methods, we must bound the iteration delays in the asynchronous execution. It can be done by blocking the propagation of all the updates that proceed too far.

Suppose the number of delays allowed in the execution is B and the first iteration that does not terminate is τ. We bound the delays by stopping the propagation of the updates in iteration $\tau + B - 1$. If a component is updated in iteration $\tau + B - 1$, then its update will not be gathered by its consumers. These updates are gathered only when iteration τ terminates. We can easily prove that the resulting iterations are valid bounded asynchronous iterations (see Appendix A for the proof).

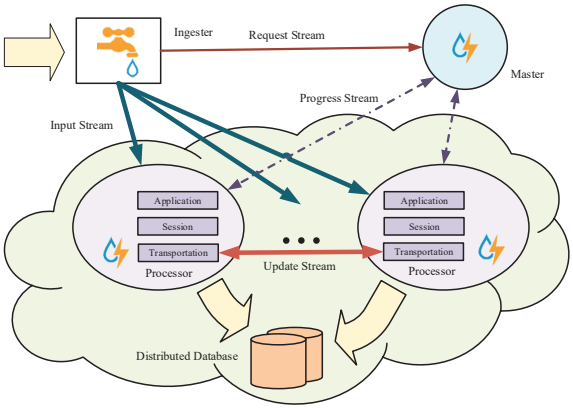


Figure 4: System architecture of Tornado.

The bounding of delays also allows the reduction in the number of PREPARE messages. If a component is updated in the iteration $\tau + B - 1$, we can commit its update without sending PREPARE messages. It's because the iteration numbers returned by its consumers cannot be larger than $\tau + B - 1$.

When B is set to 1, all updates will happen in the iteration $\tau + B - 1$ and they will not be gathered until all components have been updated in the iteration τ. In such cases, the execution of our partially asynchronous iteration model becomes exactly the same as the synchronous iteration models.

5. SYSTEM IMPLEMENTATION

Based on the execution model, we design and implement a prototype system named Tornado on top of Storm. All code is implemented in Java and the core library contains approximately 6000 lines of code. In this section, we will introduce the implementation details of Tornado, including the topology of Tornado, data storage and fault tolerance mechanisms.

Though this section focuses on the implementation on Storm, our techniques can be applied to any distributed systems that allow the components to communicate with each other by message passing.

5.1 Topology Setup

Storm is a popular system for distributed and fault-tolerated stream processing. An application in Storm can be developed by implementing two main components, *spout* and *bolt*, and connecting them to form a topology. A spout receives inputs from external sources and feeds them to the bolts. A bolt processes received messages and emits newly produced messages to other bolts. Both spouts and bolts are required to be stateless in Storm. Program states, if any, must be stored in external storage.

Tornado extends Storm with the ability to perform iterative computation over data streams. To ease the programming, Tornado adopts a graph-parallel programming model[1]. The execution of a user program is organized as a directed graph. Users can define the behavior of vertices and describe how the vertices are connected.

The topology of Tornado is illustrated in Figure 4. There are three different nodes in the topology, namely *ingester*, *processor* and *master*. Ingesters are implemented by spouts while the other ones are implemented by bolts. The ingesters collect inputs from external sources and send them to corresponding vertices. To utilize data parallelism, the vertices are partitioned and assigned to different nodes. The partitioning scheme of the vertices is stored in the shared data storage. Both the ingesters and processors retrieve the partitioning scheme to determine the location of a vertex.

Typically, a processor corresponds to a worker thread and is responsible for the management of vertices. The execution of the processors is organized into three layers. The transportation layer, the lowest layer, is concerned with the transmission and reception of messages. It packages the messages from higher layers and utilizes the interfaces provided by Storm to transfer them to other processors. It also ensures that messages are delivered without any error. When an input or an update is received, the transportation layer will unpack it and deliver it to the session layer. The session layer controls the updates of vertices, ensuring that the results are consistent and correct. The execution of the session layer strictly follows the iteration model described in Section 4. The three-phase update protocol is employed to update the vertex. The updates of vertices are coordinated so that the delays don't exceed the bound set by the users. The top most layer is the application layer which serves as the interface for users. By implementing the user-defined functions, users can describe how inputs and updates are processed and where outputs are sent.

The states of vertices are materialized in the external storage. When a vertex receives an input or an update, the processor first probes the vertex's state from the storage. After the vertex's update is committed in an iteration, the new version of the vertex will be deserialized and written to the storage.

When all vertices in an iteration have been updated, the processor will report the progress made in the iteration to the master. The master collects the progress from all processors and notifies the processors when an iteration terminates. The master also collects execution statistics, e.g. CPU usage and I/O utilization, to monitor the loads of processors. It will modify the partitioning when the load is heavily unbalanced. To ensure correctness, the master stops the computation before the modification to the partitioning scheme. After the partitioning scheme is modified, the computation will restart from the last terminated iteration.

5.2 Query Execution

Besides data collection, the ingesters are also responsible for the reception of users queries. When the results at an instant are required, the user can issue a request to the ingester. On receiving a request, the ingester will push it to the master. The master will start a branch loop to execute the query if there are sufficient idle processors. The branch loop will be forked at the place where the last iteration terminates. A message containing the iteration number will be sent to idle processors to fork the branch loop. When a processor receives the message, it will take a snapshot of the main loop and start the execution of the branch loop. If a branch loop is forked at iteration i, the most recent versions of vertices that are not greater than i will be selected in the snapshot.

The progress made in the branch loop will also be reported to the master. The master will detect the convergence of the branch loop with collected progress. Once the convergence condition holds, the processors will be notified to stop the execution of the branch loop. Users can retrieve the results from the storage with the number of the branch loop and the iteration at which the loop converges.

If there are no inputs are gathered during the computation of the branch loop, the results of the branch loop are also the precise results at the current instant. In these cases, the master will notify the processors to merge the results of the branch loop into the main loop to improve the approximation of the main loop. Though the computation of the main loop may be performed simultaneously during the merging, our execution model allows us to obtain a consistent snapshot of the main loop with little efforts.

[1] See Appendix B for the details of the programming model.

If the first iteration that does not terminate in the main loop is numbered τ, the results of the branch loop will be written with the iteration number $\tau + B$ into the main loop. During the merging of the results, the master will postpone the notification of the termination of iteration τ. The versions produced by the processors will not overwrite the results of the branch loop because the iteration numbers of the versions must be less than $\tau + B$. When the merging completes, the master can start the computation from iteration $\tau + B$. All the updates in transition will also be discarded because our iteration model requires that the iteration number of the a valid update must be no less than those of its consumers.

5.3 Fault Tolerance

Storm provides many mechanisms to tolerate faults and recover from failures. As Tornado is implemented on top of Storm, it can benefit from some of them. For example, as Storm can continuously monitor the status of the cluster, the failing workers can be detected and restarted efficiently.

Storm also provides some methods to achieve guaranteed message passing, but these methods do not work in our scenarios. For example, to ensure a tuple emitted by the spout is fully processed in the topology, Storm maintains a tree to track the propagation of the tuple and acknowledges the spout when all tuples produced by the tuple have been processed. But in Tornado, an update may lead to a large number of new updates. Besides, it's hard to track the propagation of the tuples because the topology is cyclic.

We deploy a mechanism based on checkpoints and message replaying to tolerate faults and recover from failures. We track all the passing of messages. When a sent message is not acknowledged in certain time, it will be resent to ensure at-least-once message passing. The causality of our iteration model ensures that our execution will not be affected by the duplicated messages. Any update whose iteration number is less than that of its receiver will be discarded.

When a processor has updated all its vertices in an iteration, it should report the progress to the master. But before doing that, it should flush all the versions produced in the iteration to disks. Hence when the master has received the progress of the iteration from all processors, we have already obtained a checkpoint of the computation. When some failures are detected, we can recover the computation with the results of the last terminated iteration.

Many efficient methods have been proposed to reduce the overheads of fault-tolerance and recovery. We now are investing them to help optimize our method. For example, the authors in [20] proposed an asynchronous parallel mechanism to minimize the processing interruption. Their techniques can also be applied in Tornado because currently Tornado deploys a synchronous method to perform checkpointing. Before terminating an iteration, the processor must flush all the updates made in the iteration onto disks. The computation hence is interrupted by the I/O operations. By deploying the asynchronous checkpointing method, we can proceed the computation without the waiting for the completion of the flushing. When the computation fails, we can recover from the last iteration whose results are all flushed onto disks.

6. EVALUATION

In this section, we present the experimental evaluation of the Tornado system. In the experiments, we will evaluate the effects of various factors on the execution of Tornado. We first introduce the experiment setup in Section 6.1. Then we will evaluate the effect of approximation and asynchronism in Section 6.2 and 6.3 respectively. Then we examine the scalability of Tornado in Section 6.4. Finally, we compare Tornado with other systems available for real-time data processing in Section 6.5.

6.1 Experiment Setup

The experiments are conducted on a cluster with 20 physical nodes. Each node has an AMD Opteron 4180 CPU clocked at 2.6GHz, 48GB memory and a 10TB hard disk. The version of Storm is 0.9.5. The maximum heap size of JVM is set to 16GB on each node. Unless otherwise stated, the number of threads is 200. Tornado allows the results to be stored in different external storage. In the following experiments, the intermediate results are materialized in Postgresql as default. Other different external storages will be described in the corresponding experiments.

Algorithm	Dataset	Description	Size
SSSP & PR	LiveJournal	4.84M nodes, 68.9M edges	1G
KMeans	20D-points	10M instances, 20 attributes	1.12G
SVM	HIGGS	11M instances, 28 attributes	7.48G
LR	PubMed	8.2M instances, 141K attributes	7.27G

Table 1: Data sets used in the iterative algorithms:

A range of analysis tasks and datasets, which are listed in Table 1, are used in our experiments. LiveJournal is downloaded from the SNAP project[2] while HIGGS and PubMed are from the UCI repository[3]. 20D-points dataset used in the experiments of KMeans is generated by choosing some initial points in the space and using a normal random generator to pick up points around them. PubMed is represented in the form of bag-of-words. In the experiments, we use LR to predict the relevance of each document to cancer.

6.2 Impact of Approximation

In this section, we evaluate the benefits of approximate methods and discuss the factors in the design of approximate methods.

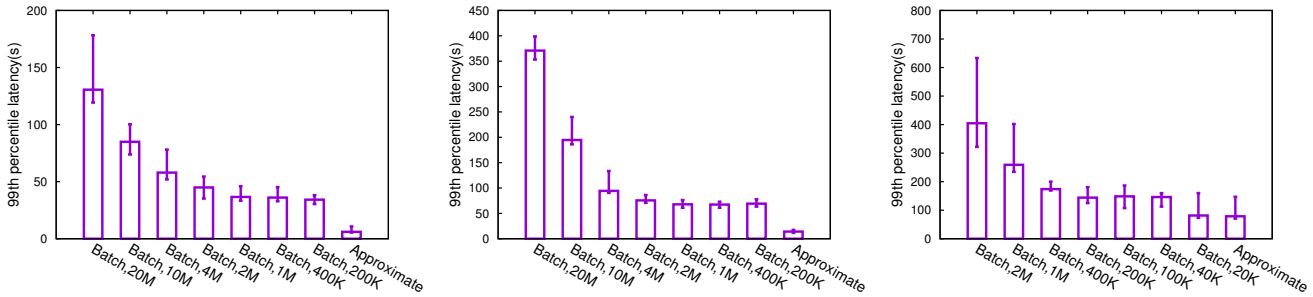
6.2.1 Comparison with Batch Methods

We first validate that the approximation in the main loop can efficiently improve the performance of real-time iterative analysis. We read inputs from the datasets and fork a branch loop to compute the results when the number of required inputs is reached. We first do not perform any approximation in the main loop and use the results of the last batch as the initial guesses. Then we perform approximation in the main loop and probe the results at the same instants as in the batch methods. Three different workloads, namely SSSP, PageRank and KMeans are adopted in the experiments. The 99th percentile latencies of these workloads with different methods are plotted in Figure 5.

Figure 5a illustrates the latencies in the computation of SSSP under different methods. In the experiments of SSSP, we first obtain a fixed-point at the time when 4 million edges are read. Then we use incremental methods to calculate the results of each batch with the last fixed-point obtained. The cost to incrementally compute the results of SSSP is proportional to the number of updated edges. Therefore, the latency degrades linearly with the batch size when the batch size decreases from 20M to 1M. But when the batch size continues to decrease, the latency is not reduced as expected. It's likely that the performance is dominated by the communication cost when the batch size becomes small. Because the updated vertices randomly distribute over the clusters, the number of messages don't decrease with the reduction of inputs. Finally, the lowest latency achieved by batch methods is approximately 34.24s. Similar results are also observed in the experiments of PageRank. Figure 5b reports that the latencies of batch methods degrade quickly at first, but become stable afterwards.

[2]http://snap.stanford.edu

[3]http://archive.ics.uci.edu/ml/index.html



(a) SSSP (b) PageRank (c) KMean

Figure 5: Comparison between batch and approximate methods.

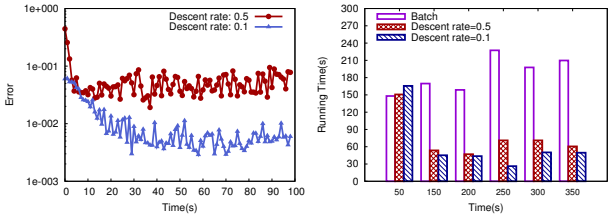
In both workloads, the approximate method achieves the lowest latency due to the smallest approximation error. As the incremental method of SSSP can catch up with the speed of data evolvement, we use it to approximate the results at each instant. In the main loop, each iteration takes approximately 4.13s to terminate and roughly 5K vertices are needed to be updated. When a branch loop is forked, the number of vertices to be updated in the approximate method is much less than the batch methods. Therefore, the branch loop can converge within only 6.10s on average.

Though the approximate method can also obtain good initial guesses for the KMeans workload, its latency roughly equals that of the batch method whose batch size is 20K (illustrated in Figure 5c. This is because the execution time of KMeans is not proportional to approximation error. KMeans has to rescan all the points even when only one centroid is updated. Hence, the initial guesses with less approximation error do not help reduce the latencies.

6.2.2 *Approximation Error vs. Adaption Rate*

We then exhibit the trade-off between the approximation error and adaption rate using SVM and LR workloads, both of which are solved by SGD. In the computation of SGD, the convergence rate is closely related to the descent rate. Typically the loop with a larger descent rate will converge faster. Therefore, if the main loop is configured with a large convergence rate, it will quickly adapt the approximation to the inputs. This assertion is ensured by Figure 6a. We plot the value of the objective function against the time in Figure 6a. We can see that the main loop with the descent rate whose value is 0.5 is reduced quickly. But the fast convergence rate does not help improve the performance of branch loops. As we see in Figure 6b, the branch loops forking from the main loop whose descent rate is 0.1 takes less time to converge. It's because the main loop can obtain better initial guesses when the descent rate is 0.1. We can see from Figure 6a that the approximation error in the main loop oscillates around 0.005 when the descent rate is 0.1. The achieved approximation error is much less than that in the main loop whose descent rate is 0.5.

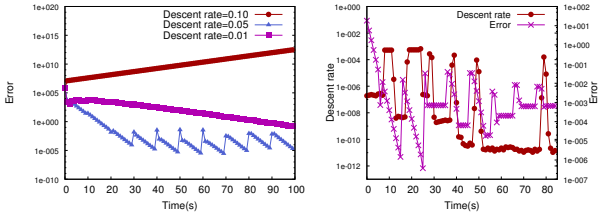
But it's not always good to choose a small descent rate. Figure 7a exhibits another aspect of the problem. When the descent rate is 0.1, we observe the growth in the objective function. Then we decrease the descent rate to 0.05. The main loop now can efficiently adapt the approximation to the input changes. But if we continue decreasing the descent rate, the main loop fails to catch up with the input changes. Among all the three settings, the main loop whose descent rate is 0.05 achieves the best performance. The quick adaption of the approximation helps the branch loops to start from initial guesses with small approximation error.



(a) SVM approximation error (b) SVM performance

Figure 6: Impact of approximation error on SVM.

The observation from Figure 6a and Figure 7a suggests that there exists trade-off between the approximation error and the adaption rate. A large descent rate allows the main loop to quickly adapt to the input changes, but it may achieve a relatively large approximation error. A small descent rate though can achieve small approximation error, but as they cannot adapt to the changes in time, the performance may still be degraded by the poor initial guesses.

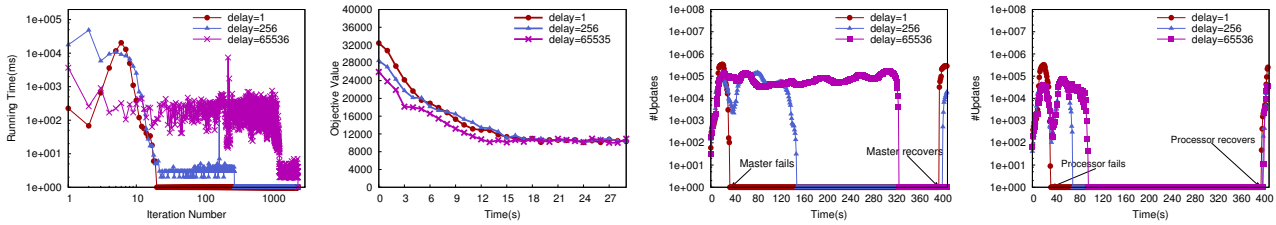


(a) LR with static descent rate (b) LR with dynamic descent rate

Figure 7: Approximation error vs. descent rate in LR.

Therefore we should adjust the descent rate adaptively. We notice that the choices of descent rate have been extensively studied in the context of stochastic optimization; but those methods, including Adagrad [18] and Adadelta [47], are not suitable for the approximation of the main loop. As the underlying model may evolve over time, the main loop will never converge, and should continuously adapt its approximation to the input changes. As those adaptive descent methods all produce a decreasing sequence of descent rates, they are unable to catch up with the input changes.

Currently, we deploy a heuristic method called *bold driver* [27, 38] to dynamically adjust the descent rate in the main loop. We decrease the descent rate by 10% when the value of the objective function increases and increase the descent rate by 10% when the value of the objective function decreases too slowly (less than 1%). The effect of the dynamic method on the LR workload is illustrat-



(a) SSSP branch loop (b) LR branch loop (c) Master failure (d) Single processor failure

Figure 8: Impact of asynchronism on performance and fault tolerance

ed in Figure 7b. When the main loop starts, the descent rate is set to an initial value and the approximation error drops sharply with the initial value. But when more inputs come in, the approximate error increases. Realizing the growth in the approximation error, the dynamic method increases the descent rate which helps the approximation to quickly adapt to the input changes. When the approximation error is small enough, the descent rate is decreased to achieve a better approximation. Compared with the static methods, the dynamic method can adjust the approximation to the inputs quickly and achieve relatively small approximation error.

6.3 Impact of Asynchronism

In this section, we verify that the degree of asynchronism has an impact on the performance and fault tolerance, and characterize the design trade-off. Three different delay bounds are chosen in the experiments, namely 1, 256 and 65536. In the loop whose delay bound is 1, the execution is equivalent to the one in synchronous iteration models, as described in Section 2.3. Hence we call this loop as the synchronous loop.

6.3.1 Impact on Performance

We first fork the branch loop of SSSP under three delay bounds. The branch loop starts from the default initial guess when the gathered inputs amount to half of the data sets. The computation time per iteration is plotted in Figure 8a and the summaries of the execution of these branch loops are listed in Table 2.

Bound	Time	#Iterations	#Updates	#Prepares
1	87.13s	22	104,668,914	0
256	227.811s	276	166,005,263	46,309,139
65536	335.521s	2370	119,634,766	119,634,766

Table 2: The summaries of the loops with different bounds.

We can see from Table 2 that the computation of the synchronous loop is performed without any prepare messages. That is because the computation progress completely replies on the notification of iteration termination, which performs as the synchronization barriers. All vertices are located at the same iteration, hence they do not need any `PREPARE` message to determine the iteration of each update. But as the delay bound increases, the loops become asynchronous and the number of prepare messages increases with the delay bound. When the delay bound is 65536, the numbers of prepare and update messages are the same. It indicates that the execution now does not depend on the information of iteration termination. The iteration in which each vertex is updated is completely determined by the vertex's consumers.

We also see that the synchronous loop can converge with the least number of iterations. Its computation completes in only 22 iterations while the other two loops need 276 and 2370 iterations respectively. Since the synchronous loop requires that the computation cannot proceed to the next iteration before all vertices have completed their updates, the computation can make sufficient progress in each iteration, which reduces the number of iterations. The reduction in the number of iterations has a measurable impact on the performance in our implementation. Figure 8a illustrates that the synchronous loop exhibits the best performance among the three loops. As explained in Section 5.3, to ensure data integrity, we require a processor to flush all the updates onto disks before it reports the progress to the master. Because asynchronous loops step more iterations to the fixed points, the performance of the asynchronous loops is degraded by the additional I/O cost. Besides, since more messages are needed in the computation, the performance is also hampered by the increasing communication cost.

But to make sufficient progress in each iteration, the synchronous loop takes a long time to terminate an iteration. An iteration in the synchronous loop takes 4.356s on average to terminate while in the loop whose delay bound is 65536, it takes only 0.141s on average. The growth in the computation time per iteration implies that we need a longer time to reflect the changes in the computation. In many workloads, it makes synchronous loops converge slower than their asynchronous counterparts.

In Figure 8b, we plot the objective value of the LR workloads versus time with different delay bounds. We increase the sample ratio to 10%. With the growth in the sample ratio, more computation is needed in each iteration and different workers will contend for computation resources. Consequently, the performance of the synchronous loop degrades significantly by the stragglers. On the other hand, the loop with the largest delay bound can quickly update the parameter model and is more tolerated to the unbalanced scheduling. Hence, it achieves a faster convergence rate than the synchronous one.

6.3.2 Impact on Fault Tolerance

Then we evaluate the impact of delay bound on the fault tolerance. We perform the branch loop of SSSP and kill the master node at a certain time. With the provisioned mechanism, the master node can recover from the failure and continue its execution. But during its recovery, the delay bound has different impact on the execution. In Figure 8c, we plot the number of updates performed per second against time under different delay bounds. To make the comparison fair, we ensure that all loops experience the same down time of the master node.

As the synchronous loop relies on the master node to detect the termination of iterations, its execution stops shortly after the failure of the master node. Different from the synchronous loop, the branch loop whose delay bound is 65536 is not affected by the failure of the master node. Because its execution can complete in 2370 iterations which is much smaller than the bound 65536, no updates will be blocked. The loop can continue its execution as nothing happens. The branch loop whose delay bound is 256 can continue

its execution after the master node fails. But as it needs 276 iterations to complete its execution, eventually all updates are blocked by the delay bound. By that time, the execution completely stops.

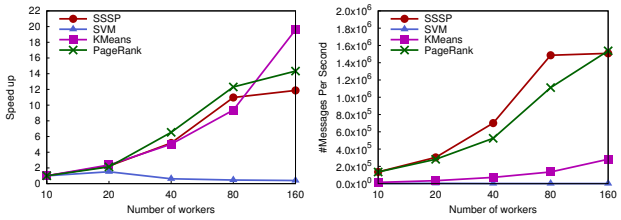
Now we evaluate the impact of asynchronism on fault tolerance when a single processor fails. We randomly choose a processor node and kill it. Similar to the master node, the processor node can correctly recover from the failures but the branch loops with different delay bounds exhibit very different behaviors which is illustrated in Figure 8d.

As in the case of master failure, the synchronous loop stops shortly after the processor node fails. It's because an iteration cannot terminate when the vertices in the failing processors have not completed their updates. The asynchronous loops however can continue their execution as they don't require all vertices to be updated in each iteration. But as each vertex must prepare its updates before the commitment, the vertices cannot complete their updates if one of its consumers resides in the failing processor. The effect of the processor's failure then is propagated to the vertex's producers. Eventually all the vertices in the connected component will be unable to complete their updates.

The experiments with machine failures suggest that (1) When the master node fails, the computation stops when all updates are blocked by the delay bound. The computation of synchronous loop stops immediately after the master node is down. But if the execution of an asynchronous loop does not depend on the iteration termination, the loop can continue its computation as nothing happens. (2) When a vertex fails, the effect of the failure will propagate to its producers. The computation stops when the effect of the failure is propagated to all vertices.

6.4 Scalability

We now study the scalability characteristics of Tornado on various workloads. Ideally the performance should improve proportionally when the number of workers increases.



(a) Speedup (b) Message throughput

Figure 9: Scalability of Tornado

In the computation of Tornado, each vertex has to materialize their updates when it's updated. With the increasing number of workers, the programs can fully take advantage of the additional I/O devices and efficiently reduce the computation time. Figure 9a shows that nearly linear speed-ups are achieved by SSSP, PageRank and KMeans when the number of workers increases from 10 to 80. But when the number of workers continues to increase, the communication cost increases as well. We measure the network utilization by the number of messages sent per second. We can see in Figure 9b that with the growth in the number of workers, the throughput stops to increase when the number of sent message per second reaches 1.5M. At that time, more workers will not help SSSP and PageRank to improve the performance.

SVM exhibits very different scalability characteristics than SSSP and KMeans. The performance degrades when the number of workers increases. In SVM, the amount of computation is determined by the batch size and only the parameters are updated in each iteration. Hence its performance does not benefit from the increasing number of workers. On the contrary, more workers incur more communication cost, which degrades the performance.

6.5 Comparison with Existing Systems

In this section, we compare Tornado with other related systems. Three different systems are adopted in the comparison, namely Spark, GraphLab and Naiad. Spark is one of the most popular real-time systems in the industry. It also provides Spark Streaming which can process data streams with mini-batches. But as Spark Streaming does not allow iterative semantics, we does not adopt Spark Streaming in the experiments. Graphlab is a graph processing prototype which stands out with its excellent performance. Different from Spark, it adopts a graph-parallel model to execute programs at scale. Naiad is the system closest to Tornado. By tracking the dependencies between the inputs and the outputs, the results can be incrementally updated when the input changes.

Program	Spark	GraphLab	Naiad	Tornado
SSSP, 1%	364.16±23.13	43.12±5.67	26.41±4.12	5.76±2.13
SSSP, 5%	343.73±31.19	43.27±4.98	30.53±3.87	6.17±3.44
SSSP, 10%	352.98±29.92	45.11±4.63	32.48±4.84	6.71±2.03
SSSP, 20%	355.67±22.94	47.56±3.36	37.42±7.38	5.83±3.79
PR, 1%	461.25±25.21	95.85±2.39	100.46±12.08	13.72±2.02
PR, 5%	452.70±33.42	96.91±7.63	114.20±7.29	13.64±4.28
PR, 10%	449.13±35.61	99.75±5.27	135.19±10.07	12.53±4.12
PR, 20%	466.38±38.12	105.35±3.45	163.56±9.21	13.57±4.76
SVM, 1%	64.07±13.20	69.04±8.35	31.25±16.86	6.12±2.78
SVM, 5%	67.12±9.31	68.83±12.30	37.19±14.78	6.09±3.07
SVM, 10%	78.76±21.87	65.51±14.31	42.76±20.43	8.13±3.58
SVM, 20%	73.98±45.09	68.97±18.39	67.20±24.14	7.85±3.26
KM, 1%	644.08±43.10	154.07±2.47	-	64.23±18.23
KM, 5%	723.27±61.32	156.36±3.16	-	79.85±15.25
KM, 10%	686.21±67.12	156.96±7.03	-	89.64±19.49
KM, 20%	694.37±63.96	159.39±6.61	-	96.51±17.26

Table 3: Latency (seconds) in different systems

We evaluate the performance of these systems as well as Tornado with four different workloads, namely SSSP, PageRank, KMeans and SVM. Since other systems store results in memory, for fair comparison, Tornado stores the intermediate results in a distributed in-memory database backed by LMDB[4]. We continuously feed the inputs to these systems, and measure the latencies of these systems when the percentage of the accumulated inputs amount to 1%, 5%, 10% and 20% respectively. The results are illustrated in Table 3. In the experiments of KMeans, Naiad is unable to complete because it consumes too much memory.

Because both Spark and GraphLab target at batch-processing, we have to collect all produced inputs and run the iterative methods only when the results are required. Though Spark has been optimized for iterative algorithms by keeping intermediate results in memory and reusing them to reduce redundant results, it exhibits the worst performance in most workloads due to the I/O overheads in the data spilling.

Different from Spark and GraphLab, Naiad and Tornado are designed for real-time analysis. Similar to other adaptive systems, Naiad also utilizes incremental methods to reduce redundant computation. By decomposing the different traces, Naiad relaxes the dependencies among different epochs. But the decomposition also increases the time complexity of the incremental maintenance. When the inputs change, Naiad has to combine all previous difference traces to restore the current version. The performance hence degrades linearly with the number of epochs and iterations.

[4]http://symas.com/mdb

In the experiments, we set the epoch size of Naiad to be 1% of the size of the data set. In the computation of SSSP and SVM, Naiad achieves a much better performance than Spark and Graphlab. But in the computation of PageRank, it experiences longer execution time due to the growth in the number of iterations. The time needed to incrementally compute the results is even longer than the time needed by GraphLab to compute from scratch.

Tornado achieves the best performance on all the workloads against these adopted systems. The speedups vary from a fold of 1 to 50 with the accumulated changes ranging from 1% to 20%. Particularly, higher speedups are achieved when we can approximate the results well and the penalty of approximation error is small. In the experiments other than KMeans, the latency is independent from the amount of the changes and the size of historic data. The results show that Tornado can effectively reduce the latency with the proposed execution model.

7. RELATED WORK

The problem of iterative analysis over evolving data is closely related to many research areas. We briefly review the related work in this section.

Stream processing. Stream processing has been studied for many years in the context of both systems and algorithms. Early attempts made in stream processing systems [1, 9], have their roots in relational databases and allow users to perform relational queries on data streams. Recently, many distributed stream processing frameworks [42, 35, 2, 46] are proposed to facilitate fault-tolerant stream processing at scale. These stream processing systems, however, cannot efficiently handle cyclic data flows. Chandramouli et al. extended the punctuation semantics [43] and proposed Flying Fixed-Point [8] to support iterative queries over streams. But as their method targets at totally asynchronous iterations, its applicability is limited. Apache Flink[5] supports iterative computation over streams, but it does not provide an efficient method for users to retrieve the results at certain instants. A user has to collect all the results in the iterations and construct the results by himself.

Adaptive systems. Adaptive systems target at the applications whose inputs are incrementally updated. They avoid the need to compute from scratch by tracking the dependencies in the computation and employing incremental methods. Typical examples include Nectar [24] and Incoop [7]. But both of them do not support iterative computation. The iterative computation over dynamic data makes it hard to track the dependencies in the computation. Naiad [32, 33] proposed to decompose the different traces into multiple dimensions and combine them to allow incremental computation. By grouping inputs into small epochs, Naiad can produce timely results over data streams. But as we saw in the experiments, the decomposition degrades the performance as well. With the growth in the number of epochs, the latency of Naiad will become very high.

Iterative computation. In recent years, many graph processing systems have been proposed to allow general graph analysis, e.g., Pregel [31], GraphLab [30, 23] and Trinity [39]. By reducing materialization overheads and allowing incremental computation, these graph processing systems can achieve good performance for common iterative methods. But all these graph systems cannot deal with continuously changed inputs. Our iteration model allows iterative methods over dynamic graphs, hence can be deployed in these graph processing systems to enhance their applicability. For example, GraphLab deploys Chandy-Misra in its asynchronous engine to ensure consistency. Our extension to Chandy-Misra with Lamport Clock can be implemented in GraphLab to allow evolving inputs.

[5]http://flink.apache.org/

Kineograph [13] extends the functionality of these systems by allowing the collection of dynamic data. But newly-arrived updates cannot be processed immediately and must wait for the completion of the current epoch. Signal/Collect [41] and HAMA [3] adopt similar methods to Kineograph, but they allow the application of input changes when an iteration terminates.

Naiad provides different consistency levels across iterations. By default, the computation proceeds synchronously with the notify messages which indicates the termination of iterations. But users are free to proceed their computation, ignoring the notify messages. In such cases, the computation is totally asynchronous and can only ensure correctness for a limited number of iterative methods.

Besides these general-purpose iterative systems, there are many other systems targeting at specific workloads. For example, Hogwild! [37] is an efficient method to compute SGD on multi-core servers without locking. But it cannot be extended to distributing settings as the assumption that write latencies are limited will not hold. Parameter Servers [15, 26, 29, 45] are based on distributed shared memory and utilize weak consistency to implement bounded asynchronous iterations. But as they specialize the communication model, they can not support general iterative methods whose dependency graph are not bipartite.

Approximation methods. Many stream mining algorithms are developed for specific analysis tasks, such as clustering [40], link analysis [14] and frequent pattern mining [25]. These algorithms are usually assumed to be executed in a system with limited computation resources and the inputs will be discarded after they are processed [4]. Hence they usually approximate the results with constructed sketches and trade off space for passes [16]. These approximation methods can be adopted in Tornado to accelerate the convergence rate of branch loops.

Our method shares similar ideas with the practice of *warm starts* in gradient descent [36, 48]. But our work generalizes the idea and applies it to the iterative computation over evolving data. Such generalization is not trivial, since we have to deal with the complexity in distributed settings, e.g., the consistency and correctness of the computing results.

8. CONCLUSION

It's challenging to support real-time iterative analysis over evolving data. The observation that loops starting from good initial guesses usually converge fast encouraged us to develop an method to enhance the timeliness with better approximation but not incremental methods. To reduce the approximation error due the most recent inputs, we proposed a novel bounded asynchronous iteration model which can achieve fine-grained updates while ensuring correctness. The conducted experiments verified that many common iterative methods can benefit from our execution model.

There are several directions for the future development of Tornado. Currently, Tornado requires users to write their programs with procedural code. We are designing a high-level language to enhance the programmability. Ongoing research also involves the load shedding of branch loops so that given restricted resources, requests can be accomplished as many as possible.

9. ACKNOWLEDGEMENTS

This work is supported by the National Natural Science Foundation of China under Grant No. 61272155 and 61572039, 973 program under No. 2014CB340405, Beijing Natural Science Foundation (4152023), and Shenzhen Gov Research Project JCYJ2015101 4093505032.

10. REFERENCES

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *PVLDB*, 6(11):1033–1044, 2013.

[3] A. Andronidis. Hama wiki: Dynamic graphs. http://wiki.apache.org/hama/DynamicGraphs, 2013.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[5] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized pagerank. *Proc. VLDB Endow.*, 4(3):173–184, 2010.

[6] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Inc., 1989.

[7] P. Bhatotia, A. Wieder, R. Rodrigues, U. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *SOCC*, pages 7:1–7:14, 2011.

[8] B. Chandramouli, J. Goldstein, and D. Maier. On-the-fly progress detection in iterative stream queries. *PVLDB*, 2(1):241–252, 2009.

[9] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. Shah. Telegraphcq: Continuous dataflow processing. In *SIGMOD*, pages 668–668, 2003.

[10] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *TOCS*, 3(1):63–75, 1985.

[11] K. Chandy and J. Misra. The drinking philosophers problem. *TOPLAS*, 6(4):632–646, 1984.

[12] D. Chazan and W. Miranker. Chaotic relaxation. *LAA*, 2(2):199–222, 1969.

[13] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98, 2012.

[14] A. Das Sarma, S. Gollapudi, and R. Panigrahy. Estimating pagerank on graph streams. In *PODS*, pages 69–78, 2008.

[15] J. Dean, G. Corrado, R. Monga, K. Chen, D. M., Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng. Large scale distributed deep networks. In *NIPS*, pages 1223–1231, 2012.

[16] C. Demetrescu, I. Finocchi, and A. Ribichini. Trading off space for passes in graph streaming problems. *TALG*, 6(1):6:1–6:17, 2009.

[17] E. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[18] J. Duchi, E. Hazan, and Y. Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *JMLR*, 12:2121–2159, 2011.

[19] M. El Tarazi. Some convergence results for asynchronous algorithms. *Numer. Math.*, 39(3):325–340, 1982.

[20] R. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making state explicit for imperative big data processing. In *USENIX ATC*, pages 49–60, 2014.

[21] A. Frommer and H. Schwandt. Asynchronous parallel methods for enclosing solutions of nonlinear equations. *IJCAM*, 60(1Í C2):47 – 62, 1995.

[22] M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: A review. *SIGMOD Rec.*, 34(2):18–26, 2005.

[23] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[24] P. Gunda, L. Ravindranath, C. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in data centers. Technical Report MSR-TR-2010-55, 2010.

[25] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: Current status and future directions. *DMKD*, 15(1):55–86, 2007.

[26] Q. Ho, J. Cipar, H. Cui, S. Lee, J. Kim, P. Gibbons, G. Gibson, G. Ganger, and E. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, 2013.

[27] H. Hsin, C. Li, S. M., and R. Sclabassi. An adaptive training algorithm for back-propagation neural networks. In *SMC*, volume 2, pages 1049–1052, 1992.

[28] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[29] M. Li, D. Andersen, J. Park, A. Smola, A. Ahmed, V. Josifovski, J. Long, E. Shekita, and B. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[30] 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. *PVLDB*, 5(8):716–727, 2012.

[31] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[32] F. McSherry, D. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*, 2013.

[33] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, pages 439–455, 2013.

[34] S. Muthukrishnan. Data streams: Algorithms and applications. *Found. Trends Theor. Comput. Sci.*, 1(2):117–236, 2005.

[35] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDMW*, pages 170–177, 2010.

[36] B. O'donoghue and E. Candès. Adaptive restart for accelerated gradient schemes. *Found. Comput. Math.*, 15(3):715–732, 2015.

[37] B. Recht, R. Christopher, W. Stephen, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[38] D. Sarkar. Methods to speed up error back-propagation learning algorithm. *CSUR*, 27(4):519–544, 1995.

[39] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD*, pages 505–516, 2013.

[40] J. Silva, E. Faria, R. Barros, E. Hruschka, A. Carvalho, and J. Gama. Data stream clustering: A survey. *CSUR*, 46(1):13:1–13:31, 2013.

[41] A. Stutz P.and Bernstein and W. Cohen. Signal/collect: Graph algorithms for the (semantic) web. In *ISWC*, pages 764–780, 2010.

[42] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *SIGMOD*, pages 147–156, 2014.

[43] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *TKDE*, 15(3):555–568, 2003.

[44] J. Vitter. Random sampling with a reservoir. *TOMS*, 11(1):37–57, 1985.

[45] K. Vora, S. Koduru, and R. Gupta. Aspire: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. In *OOPSLA*, pages 861–878, 2014.

[46] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[47] M. D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. *CoRR*, abs/1212.5701, 2012.

[48] M. Zinkevich. Theoretical analysis of a warm start technique. In *NIPS Workshop on Parallel and Large-Scale Machine Learning*, 2011.

[49] I. Zliobaite, A. Bifet, M. Gaber, B. Gabrys, J. Gama, L. Minku, and K. Musial. Next challenges for adaptive learning systems. *SIGKDD Explor. Newsl.*, 14(1):48–55, 2012.

APPENDIX

A. CORRECTNESS PROOF

In this section, we prove that the execution model described in Section 4 is a valid bounded asynchronous iteration model.

PROOF. (1) First, we prove that the iteration delay of each update is not greater than B. Since all components cannot gather the updates in iteration $\tau + B - 1$, the maximum iteration number of components must be less than $\tau + B$. As iteration $\tau - 1$ has terminated, the iteration numbers of all vertices must be in the range $[\tau, \tau + B - 1]$. As the iteration in which a component commits its update is determined by the iteration numbers of its consumers, the maximum delay of an update hence is less than B.

(2) Second, we prove that every component can be updated in the iterations from τ to $\tau + B - 1$. As all components will eventually be updated, we only to prove that all components in iteration τ can be updated. Because our iteration model don't allow the gathering of the updates in iteration $\tau + B - 1$, we have to prove that all components in iteration τ can receive an update whose iteration number is less than $\tau + B - 1$. Here we provide a proof sketch, without considering special cases.

Given any component x whose iteration number is τ. According to the causality, it must have a producer y who is updated in the iteration τ' and $\tau' < \tau$. As we have proved that the delay of each update cannot be greater than B, so we have $\tau - B < \tau' \leq \tau - 1$.

Since iteration τ is the first iteration that does not terminate, iteration τ' must have terminated. According to the requirement of iteration termination, y must have been updated before iteration τ' terminates. In combination with the delay bound, we can know that the new iteration number of y is less than $\tau' + B$ and also is less than $\tau + B - 1$.

Consequently, for each component x in iteration τ, it can receive an update from its producer whose iteration number is less than $\tau + B - 1$. The received update can be gathered by x and allow x to be updated in an iteration whose number is greater than τ. □

B. PROGRAMMING MODEL

The primary goal of Tornado is to provide efficient real-time iterative analysis over evolving data. To ease the programming, Tornado adopts a graph-parallel programming model similar to those in popular graph processing systems [23]. The programmer can define the behavior of a vertex by implementing the following interfaces:

```
vertex::init()
vertex::gather(iteration, source, delta)
vertex::scatter(iteration)
```

The `init` function defines how the vertex is initialized when it's created. When the vertex receives an update, it will use `gather` to receive the update. When the vertex can commit its update, it may call `scatter` to emit updates to its consumers. Both *gather* and *scatter* can modify the state of the vertex, but the updates of a vertex can only be propagated by *scatter*.

As the our iteration model utilizes the dependencies to ensure correctness, a vertex should invoke the system-provided methods to explicitly maintain the dependency graph:

```
vertex::addTarget(target)
vertex::removeTarget(target)
```

Besides the vertices, there are some other auxiliary classes in Tornado. For example, `ingesser` defines how the application receives the inputs from the external sources while `progress` is used to record the progress made in the computation.

```
vertex::init()
    length = (getID() == 0 ? 0 : INT_MAX);
    sourceLengths = new map();
vertex::gather(iteration, source, delta)
    delta.isType(ADD_TARGET) => addTarget(delta.value);
    delta.isType(REMOVE_TARGET) => removeTarget(delta.value);
    delta.isType(UPDATE) => sourceLengths[source] = delta.value;
vertex::scatter(iteration)
    length = min(min(sourceLengths.values()) + 1, length);
    if(!doBatchProcessing || getLoop() != MAIN_LOOP) {
        for(target : removedTargets)
            emit(UPDATE, INT_MAX);
        for(target : addedTargets + retainedTargets)
            emit(UPDATE, length);
    }
```

Figure 10: The pseudo code for Single-Source Shortest Path.

We use the pseudo code in Figure 10 as an example to illustrate the programming model. The example program performs the Single-Source Shortest Path algorithm on a stream which is composed of the insertion and deletion of edges.

The initial length of a vertex is initialized in the `init` function. The length of the source is initialized to 0 while the length of other vertices is set to `INT_MAX`.

There are three types of updates in the program: namely `ADD_TARGET`, `REMOVE_TARGET` and `UPDATE`. The first two types come from the input stream while the last one is sent by vertices. When a vertex receives an update, it calls function `gather` to update itself.

The `scatter` function will be called when the vertex completes the gathering in the iteration and is allowed to commit its update. In this program, each vertex reports its current length to its neighbors.

The computation of the vertex in the main loop and the branch loops are defined by the same functions. That is the code of the `gather` and `scatter` function is executed in both the main loop and the branch loops. The vertex can know the loop it's updated by calling the system-provided interface `getLoop()`. For example, if we want to perform batch processing, we should not emit any update in the main loop. In such cases, a branch loop will start from the place where all vertices are assigned with the initial value.