Time-Evolving Graph Processing at Scale

Anand Padmanabha Iyer
UC Berkeley
api@cs.berkeley.edu

Li Erran Li
Uber Technologies
erranlli@uber.com

Tathagata Das
Databricks
tdas@databricks.com

Ion Stoica
UC Berkeley
istoica@cs.berkeley.edu

ABSTRACT

Time-evolving graph-structured big data arises naturally in many application domains such as social networks and communication networks. However, existing graph processing systems lack support for efficient computations on dynamic graphs.

In this paper, we represent most computations on time evolving graphs into (1) a stream of consistent and resilient graph snapshots, and (2) a small set of operators that manipulate such streams of snapshots. We then introduce GraphTau, a time-evolving graph processing framework built on top of Apache Spark, a widely used distributed dataflow system. GraphTau quickly builds fault-tolerant graph snapshots as each small batch of new data arrives. GraphTau achieves high performance and fault tolerant graph stream processing via a number of optimizations. GraphTau also unifies data streaming and graph streaming processing. Our preliminary evaluations on two representative datasets show promising results. Besides performance benefit, GraphTau API relieves programmers from handling graph snapshot generation, windowing operators and sophisticated differential computation mechanisms.

1 Introduction

Graph-structured data is on the rise, in size, complexity and the dynamism they exhibit. From social networks (e.g., Facebook, Twitter) to telecommunication networks (e.g., cellular networks), applications that generate graph-structured data are ubiquitous. With the increasing interest in the Internet-of-Things (IoT), the trend is likely to continue in the future. Unlike unstructured datasets, the dynamic nature of these datasets give them a unique characteristic—the graph-structure underlying the data evolves over time. Unbounded, real-time data is fast becoming the norm [2], and thus it is important to process these *time-evolving* graph-structured datasets efficiently.

Mining time-evolving graphs can reveal insights that are beneficial for businesses. To extract maximum insights, frameworks for time-evolving graph processing must be able to support a variety of analysis tasks. First, they must be able to execute iterative graph algorithms in real-time. For example, social networks such as Twitter can recommend products based on up-to-date TunkRank (similar to PageRank) of people in an attention-graph [8], and cellular network operators can fix traffic hotspots in their networks as they are detected [12]. Second, analytics tasks typically often involve combining graph-structured data with unstructured and tabular data. For example, social networks may join a user's node attributes (e.g., recent tweets) with her browsing data or purchase data for better real-time ad placement. Third, it is also necessary to run these analytics over windows of input data. For instance, Twitter may be interested in keeping track of influential users over sliding windows.

Existing solutions do not satisfy all these requirements. Stream processing systems (e.g., Storm [15], Spark Streaming [17], Millwheel [1], Flink [3]) provide support for efficient window operations on unbounded datastreams, but lack support for graph processing. Graph processing systems (e.g., Giraph [4], GraphX [9], PowerGraph [10]) on the other hand support iterative graph algorithms, but assume a static underlying graph. Specialized systems have been proposed for evolving graph processing [8, 11]. These solutions do not offer a convenient way to mix structured and unstructured data nor do they provide window operations on multiple graphs. Naiad [13] is a timely dataflow framework that uses differential dataflow to execute iterative, fully incremental algorithms on dynamic datasets. However, it does not offer windowing operations on graph snapshots. Additionally, its dependence on checkpointing for fault-tolerance makes it less desirable for some use-cases.

Building a time-evolving graph processing system with all desired properties is challenging and requires managing many tasks. These tasks include consistent and fault tolerant snapshot generation, tight coordination between snapshot generation and computation stages and operations across multiple graph snapshots. In more details, algorithms and computation models on time-evolving graphs typically operate on consistent graph snapshots. To achieve low latency, computation on a graph snapshot starts as soon as the snapshot is available. These two interlocking stages of snapshot generation and computation need to be carefully coordinated. The algorithms can run continuously as new data becomes available on tumbling or sliding windows. Certain algorithms can run in parallel among graph snapshots in a window (e.g., top users), while others have to be run sequentially on the time ordered graph snapshots (e.g., event processing). States are updated for each graph snapshots (e.g., any temporal properties of the graph). We distill these patterns of time-evolving graph processing into a few dataflow operators. By doing so, we identify the critical path for system optimization.

In this paper, we present GraphTau, a system for time-evolving graph processing that is built on a dataflow framework. GraphTau enables efficient streaming graph processing using two simple but powerful techniques: first, it provides a way to create and manipulate consistent graph snapshots in user defined windows. Second, it presents an incremental computation model that allows graph computations to "shift" from a stale snapshot to a new snapshot even in between iterations of the underlying algorithm. In summary, we make the following contributions:

GRADES 2016, June 24 2016, Redwood Shores, CA, USA

ISBN 978-1-4503-4780-8/16/06…$15.00

- We present GraphTau, the first time-evolving graph processing system, to our knowledge, built on a general purpose dataflow framework. This enables a single computation engine to blend in different datasets and computation models.
- GraphTau offers a graph windowing model that enables creation, management and manipulation of consistent snapshots of the time-evolving graph. GraphTau also presents an incremental computation model on graphs that enables switching of computation from one snapshot to another in between iterations. This enables GraphTau the flexibility to provide approximate results unlike previous solutions where algorithms must converge before perusing the latest updates.
- GraphTau achieves high performance and fault tolerant graph stream processing via a number of optimizations. These optimizations include incremental maintenance of internal graph data structures such as routing tables and triplets and bulk updates of lineage graphs for graph snapshots.

We evaluate GraphTau on two real-world datasets, one which represents a slowly evolving graph (Twitter follower graph) and another which represents a highly dynamic graph (cellular network dataset). Our preliminary evaluations show promising results.

2 Background

In this section, we briefly review the relevant Apache Spark subsystems that GraphTau builds upon.

2.1 Resilient Distributed Datasets (RDDs)

Resilient Distributed Datasets (RDDs) is the main abstraction provided by Apache Spark [16], a data-parallel computation engine that supports general DAG computations. RDDs are immutable, partitioned collections that are fairly generic—they can be collections residing in external systems (e.g, disk or HDFS) or they could be a derived dataset obtained from other RDDs by applying a set of deterministic operations (e.g., map, join). that are distributed across the cluster; and can be created using various operators. Since each RDD tracks the lineage graph of operations used to build it, these operations can be replayed to recompute the RDD if some partitions of it are lost thus enabling fault tolerance.

2.2 Spark Streaming

Spark Streaming [17] is the streaming component in Spark, which is fundamentally a batch system. The main idea in Spark Streaming is to treat streaming computations as a series of deterministic batch computations, commonly referred to as mini-batches. To achieve this, Spark Streaming introduces the abstraction of Discretized Stream (D-Stream), a sequence of immutable, partitioned datasets (RDDs). Just as in Spark, users interact with DStreams using operations on them. In contrast to traditional streaming approach of record-at-a-time, Spark Streaming's approach makes computation and the associated state fully deterministic, thus enabling efficient parallel fault recovery as it can leverage Spark's lineage based fault tolerance.

2.3 GraphX

Xin et. al. observed that operations performed by many specialized graph processing engines can be distilled down to a specific group of *join-map-groupby* dataflow patterns. Based on this observation, they propose GraphX [9], a graph processing engine built on top of Spark which is a dataflow engine. GraphX represents property graphs internally as a pair of RDDs—the vertex collection and the edge collection. A key stage in graph computation is constructing and maintaining the triplets view, which consists of a three-way join between the source and destination vertex properties and edge properties. To implement the triplets view efficiently, GraphX implements several optimizations, including vertex mirroring, multicast joins and incremental view maintenance.

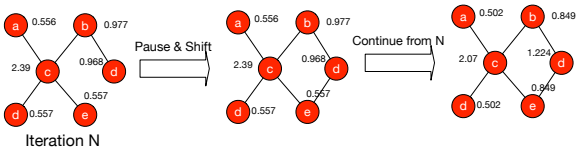


Figure 1: Pause-Shift-Resume computational model.

3 Abstraction & Computational Model

We now present the key ideas in GraphTau.

3.1 Discretized Graph Streams

The main idea in GraphTau is to treat time-evolving graphs as a series of consistent graph snapshots, and dynamic graph computations as a series of deterministic batch computations on discrete time intervals. A graph snapshot is simply a regular graph, stored as two RDDs, the vertex RDD and the edge RDD. We define GraphStream as a sequence of immutable, partitioned datasets (graphs represented as two RDDs) that can be acted on by deterministic operators. User programs manipulate GraphStreams to produce new GraphStream, as well as intermediate state in the form of RDDs or graphs.

3.2 Computational Models

GraphTau enables two computational models on graph snapshots. Both these models are implemented using a differential computation approach described in § 4.3.

3.2.1 *Pause-Shift-Resume*

Certain class of graph algorithms are robust towards graph modifications before the algorithm has had a chance to converge. For instance, consider PageRank. If the underlying graph on which the PageRank is run changes before the initial run has converged, the algorithm would still converge but to a different answer. Studies have shown that these answers are within a reasonable error to actual answer had the algorithm started running from scratch on the modified graph. GraphTau exploits this observation and offers a *Pause-Shift-Resume* computational model. In this model, GraphTau starts running a graph algorithm as soon as the first snapshot of a graph is available. Upon the availability of a new snapshot, it *pause*s the computation on the current graph, *shift*s the algorithm specific meta-data to the new snapshot and *resume*s computation on the new graph. This is illustrated in figure 1.

3.2.2 *Online Rectification*

While the PSR computational model works for certain classes of algorithms, many other graph algorithms are not resilient to graph changes. For such algorithms, GraphTau proposes the online rectification model. In this model, GraphTau rectifies the errors caused by the underlying graph modifications in an online fashion using minimal state. For instance, consider the illustration in figure 2 which shows a connected component algorithm being run on a graph. If vertex *a* gets removed while the algorithm is still running, it is possible to go back to a state where the effect of the vertex has not yet influenced the algorithm. In the connected component case, this can simply be achieved by having every vertex keep track of its component ID over time. This approach works on algorithms that are based on label propagation. While this method saves a lot of computation, it requires keeping algorithm specific state.

3.3 Consistency

Fine-grained streaming graph updates and computation on fine-grained snapshots are difficult to achieve at the same time. Graph

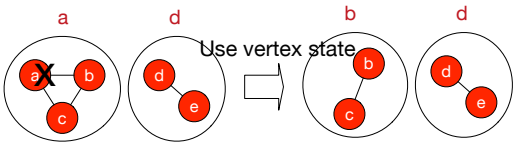


Figure 2: Online rectification of errors.

algorithms typically require global snapshots periodically. Treating graph data as mutable state will delay updates until the computation has finished processing a snapshot. Fine-grained updates are not robust to input update reordering. Suppose there are two updates. The first is to add node v. The second is to add an edge between u and v. u exists before the updates. If the second update arrives first, an edge will connect u to a non-existent node. Running graph algorithms on such snapshots may not produce consistent results.

With GraphStream, the semantics are clear. Each interval's updates reflects all of the input received until then (see § 3.5). This is despite the fact that the DeltaRDD and its updated graph snapshot are distributed across nodes. As long as we process the whole batch consistently (e.g., ordered by timestamps), we will get a consistent snapshot. This makes distributed state much easier to reason about and is the same as "exactly once" processing of the graph updates even with faults or stragglers.

3.4 Fault and Straggler Tolerance

Similar to a DStream, the fully deterministic nature of GraphStream enables two powerful recovery mechanisms difficult to apply in traditional graph stream processing systems: parallel recovery of lost state and speculative execution. Since GraphTau implements graph streams as GraphX graphs, each of which is implemented as two RDDs: vertex RDD and edge RDD, it inherits the two mechanisms automatically. In addition, GraphTau periodically *checkpoints* state GraphStreams by asynchronously replicating them to other worker nodes. When the system detects node failure, it finds out all missing RDD (vertex and edge) partitions and launches tasks to recompute them from the latest checkpoint. GraphTau mitigates stragglers by running speculative backup copies of slow tasks.

3.5 Timing Considerations

GraphStream splits the time in non-overlapping intervals, and then stores all the inputs received during these intervals in batches. This simplifies the problem in terms of when a new batch should start. Since worker nodes are synchronized using NTP (same as in DStream), a new batch can start as soon as the current time interval passes. This works well for applications where the records are generated at the same location as the streaming program. For data generated by external applications, e.g. Twitter feeds or session records in cellular networks, developers can specify *external timestamps* of when an event happened. Similar to DStream, there are two ways to handle the problem. First, the system waits for a limited "slack time" to ensure data of the current batch is arrived. This introduces a fixed latency to all results. Second, user programs can correct for late records at the application level. For details, please refer to [17]. Note that timing concerns are inherent to graph stream processing since any such system must either handle external delays or tolerate approximate results.

4 API

In this section, we present GraphTau APIs that handle input data and enable common computation models on time-evolving graphs.

4.1 DeltaDStream API

Like Spark streaming, GraphTau receivers process input data from external sources such as Twitter feeds or sessions records in telecommunication networks. Depending on the external sources, the input data may or may not come in as a collection of vertex updates and edge updates. Furthermore, in special systems such as cellular networks [12] and for some analytics tasks, input data can be used to build a graph snapshot solely using the batch in each time interval. To support generation of graph updates, we define DeltaRDD as an RDD whose elements are updates that need to be applied to a graph. These updates consist of vertex updates and edge updates. Each update can be an addition, deletion and update. A DeltaDStream is a stream of DeltaRDDs.

With the initial graph, and a vertex update function and an edge update function, we can create a GraphStream from a DeltaDStream. To implement DeltaDStream, we extend DStreams by incorporating an implicit convert function. We also support GraphStream construction directly from a vertex DStream and an edge DStream.[1]

4.2 Operators

```
abstract class GraphStream[VD, ED] = {
  def transform[VD1, ED1](
        transformFunction: (Graph[VD, ED], Time)
           => Graph[VD1, ED1]): GraphStream[VD1, ED1]

  def transformWith[VD1, ED1, VD2, ED2](
        other: GraphStream[VD1, ED1],
        tranformFunction: (Graph[VD, ED],
        Graph[VD1, ED1]) => Graph[VD2, ED2])

  // Incrementally run the algorithm through snapshots.
  def mergeByWindow(mergeVertexFunction: (VD, VD) => VD,
        invMergeVertexFunction: (VD, VD) => VD,
        mergeEdgeFunction: (ED, ED) => ED,
        invMergeEdgeFunction: (ED, ED) => ED,
        windowDuration: Duration,
        slideDuration: Duration)

  //Streaming BSP for differential computation
  def StreamingBSP(initalMsg: M, maxIter: Int,
      activeDir: EdgeDirection, periodResult: Time)
      (gs: GraphStream[V,E],
       updateActiveSet: (
            Collection[(Id, VD)], Graph[VD, ED],
            Graph[VD, ED]) => Collection[(Id, VD)],
      vprog: (VertexId, VD, M) => V,
      sendMsg: EdgeTriplet[VD, ED] =>
                    Iterator[(VertexId, M)],
      mergeMsg: (M, M) => M): Graph[VD, ED]

  //return a new ``state'' by applying a update function
  def updateLocalState[S](updateFunction: (S, Graph[VD, ED]) => S,
        initialState: S): LocalState[S]

  //Output operation
  def foreachGraph(foreachFunction: (Graph[VD, ED], Time) => Unit)

  //Reuse Spark Streaming operators for DStream views
  def transform[T](transformFunction: (Graph[VD, ED], Time)
                  => RDD[T]): DStream[T]
}
```

Listing 1: GraphStream Core API

To use GraphStream in GraphTau, users write a driver program that defines one or more GraphStreams using our functional API as shown in Listing 1. The program can register one or more GraphStreams from outside.

Reuse GraphX operators GraphTau core API has one *transform* function. It applies a graph-to-graph function to every t graph snapshot of the source GraphStream and returns a new GraphStream. It enables the use of GraphX operators for graph-to-graph transformation. For example, one can apply the *subgraph* or *connectedComponent* operator to obtain a new GraphStream. GraphTau

[1] *VD, ED* denote the vertex and edge property type respectively.

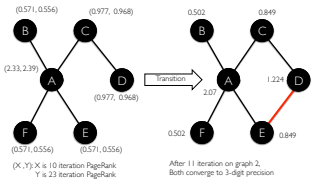


Figure 3: Benefits of differential computation for PageRank

also provides many convenient functions (not shown) such as *mapV, mapE, subgraph*, etc. These functions are equivalent to the transform operator with the supplied graph operator.

Operators on two GraphStreams GraphTau can use *transformWith* to implement various *join*, *cogroup* operators to combine two GraphStreams. These operators support applications who want to operate on multiple GraphStream jointly.

Sliding window operators The *mergeByWindow* operator merges all graphs from a sliding window of past time intervals into one graph. A function is supplied to aggregate the vertex and edge collections respectively. The aggregation functions must be associative and commutative. If the aggregation functions are also *invertible*, a more efficient version also takes a function for "subtracting" graphs and maintains the state incrementally. This operator can implement the *graphReduceByWindow* operator in [12]. In case an invertible function exists, GraphTau will maintain a cumulative graph representation so that sliding window operation will be simply updating the cumulative graph and "subtracting" from the graph that has just passed the sliding window.

State tracking The *updateLocalState* operator enables event processing and state tracking. We describe this operator in § 4.4.

Output operator The *foreachGraph* operator applies a function to each graph generated from the GraphStream. This function should push the data in each graph to a external system, like saving the graph to a file, or writing it over the network to a graph database.

Interface with graph database GraphTau supports loading data periodically from a distributed graph database, e.g. Titan. GraphTau supports *saveAsGraphDB* operation to output each graph or the updates in a GraphStream to a graph database. For example, gs.saveASGraphDB(path, "Titan", "10s") saves the GraphStream into a Titan database. If the dbType string is "Neo4j", then the GraphStream will be saved to Neo4j graph database.

Reuse Spark streaming operators Beside the core *transform* operator, GraphTau also provides another *transform* operator that applies a graph-to-RDD function to every t graph snapshot of the source GraphStream and returns a new DStream. With this, GraphTau can reuse Spark streaming operators. Furthermore, GraphTau provides two convenience functions (not shown) to return the component vertex DStream and edge DStream.

StreamingBSP operator For many iterative algorithms that compute a fixed point such as PageRank or require many iterations, we need to provide better support for efficient computation. Specifically, users may want to output top users in terms of PageRank periodically for a window of graph snapshots. We do not want to wait for the last graph snapshot in a time window to become available to start computation. Instead, we would like to start PageRank computation in the first time interval of the current time window. We also want to skip remaining iterations and resume computation as soon as a new graph snapshot is available. We refer to this type of computation as *differential computation* which we will discuss in more details next.

4.3 Differential Computation

GraphTau's *transform* and *foreachGraph* operators are very powerful and can be used to construct differential computation primitives on evolving graphs. Listing 2 shows how to implement StreamingBSP operator. This operator enables efficient implementation of a large class of incremental algorithms on time-evolving graphs. We signal the availability of the new graph snapshot using a variable in the driver program. In each iteration of Pregel, we check whether a new graph is available. If so, we do not proceed to the next iteration on the current graph. Instead, we resume computation on the new graph reusing the result, where only have vertices in the new active set continue message passing. The new active set is a function of the old active set and the changes between the new graph and the old graph. For a large class of algorithms (e.g. incremental PageRank [7]), the new active set includes vertices from the old set, any new vertices and vertices with edge additions and deletions.

```
var isNewGraphAvailable = false
def StreamingBSP(initalMsg: M, maxIter: Int,
      activeDir: EdgeDirection, period: Time)
      (gs: GraphStream[VD,ED],
       updateActiveSet: (
              Collection[(Id, VD)], Graph[VD, ED],
              Graph[VD, ED]) => Collection[(Id, VD)],
       vprog: (VertexId, VD, M) => V,
       sendMsg: EdgeTriplet[VD, ED]
                => Iterator[(VertexId, M)],
       mergeMsg: (M, M) => M
       ): Graph[VD, ED] = {
        gs.transform { (graph, time) =>
        isNewGraphAvailable = true
        graph
      }.foreachGraph { (graph, time) =>
        isNewGraphAvailable = false
        var transformedGraph: Graph[X, Y] = _
        var i = 0
        var activeSet: Collection[VertexID] = _
        var resultG: Graph[VD, ED] = _

        val newTransformedGraph = iteratorFunc(graph)
        newTransformedGraph.persist()
        newTransformedGraph.materialize()

        //update activeSet
        activeSet = updateActiveSet(activeSet,
        transformedGraph, newTransformedGraph)

        //unpersist previous one
        if (transformedGraph != null)
            transformedGraph.unpersist()
        transformedGraph = newTransformedGraph

        while(!isNewGraphAvailable || i<maxIter
              currentTime % period==0) {
           i += 1
           transformedGraph.Pregel.oneIter(
               resultG, activeDir, activeSet)
               (vprog, sendMsg, mergeMsg)
     }
     if (currentTime % period==0)
        resultG //force an output of resultG
   }
```

Listing 2: Differential Computation on Time-Evolving Graphs

With the StreamingBSP operator, we now show we can easily implement the incremental PageRank algorithm (see listing 3). After a graph is updated each time, we need to update the page rank. Instead of rerunning the page rank algorithm from scratch, we can reuse the page rank computed for the previous graph snapshot. To achieve this, we can simply call StreamingBSP API and supply the PageRank specific vertex program, *sendMsg* function and *mergeMsg* function. Figure 3 shows a concrete example. For the first graph, it takes 23 iterations to converge to 3-digit precision. If we reuse this page rank for the second updated graph on the right, it will take another 11 iterations to converge to 3-digit precision on the new graph. On the other hand, if we only finishes 10 iterations on the first graph, then transition to the updated graph. It will take the same 11 iterations to converge to 3-digit precision on the new graph. Essentially, we saved 13 iterations of PageRank computation.

```
def pageRankEvolGraph(gs: GraphStream) = {
    def vprog(v: VertexId, msgSum: double) = 0.15+0.85*msgSum
    return gs.StreamingBSP(1, 100, EdgeDirection.Out, "10s")
             (vprog,
             triplet => triplet.src.pr/triplet.src.outDeg,
             (msgA, msgB) => msgA+msgB)
}
```

Listing 3: Page Rank Computation on Time-Evolving Graphs

4.4 Live Graph State Tracking

Streaming graph applications may want to keep track of live graph state. For example, social network applications may keep track of users with live sessions. In cellular networks, the system needs to track users in the connected state [12].

GraphTau provides the *updateLocalState* operator to keep track of state over time. There are two main differences from Spark streaming's *updateStatebyKey* operator. First, Spark streaming stores state as RDDs. In contrast, we do not store them as RDDs since the states are typically small. Storing them in the driver program is more efficient. Since the driver needs to be reliable, reliability of local state will by default be reliable (e.g. synchronously replicated to a slave or use Zookeeper for consistency and reliability). Second, the new batch data can be a graph. GraphTau allows users to supply an initial state and a user defined function to operate on old state and the graph of the current time interval.

GraphTau's *updateLocalState* operator enables many temporal graph property computations on time-evolving graphs. For example, when high degree users make the transition to have high betweeness [14], quantiles (e.g. 95%) of PageRank may be used for analysis. We can easily incorporate a decaying function such as exponential moving average to discount old information. For arbitrary decay function, we have to keep track of the past relevant states.

4.5 Unifying Data & Graph Streams

GraphTau unifies data and graph stream processing. We illustrate the idea with a program that computes the top users in terms of triangle counts from Twitter attention-graph. A triangle is a clique of three nodes and is extensively used in social networks for various algorithms such as community detection. In Listing 4, the code first

```
val ds = TwitterUtils.createStream(ssc)
//create a DeltaDStream from a DStream,
//then turn it into a GraphStream
val gs = ds.createDeltaDStream(convFunc: T=>Update[VD, ED])
           .createGraphStream(deltaFunc: (Graph[VD, ED],
                              deltaRDD, fv, fe), null)
//tricnt is a DStream, each element is vertice ID and count pair
val tricnt = gs.transform(graph => graph.triangleCount.vertices)
               .mapValues(x => if (x > 1000) 1 else 0)
//compute the top popular users each 1sec interval of a 10s window
//reduceByKeyAndWindow is a DStream operator
val topuser = tricnt.reduceByKeyAndWindow(_+_, "10s", "1s")
```

Listing 4: An example that computes persistent top users by triangle counts over a sliding window.

creates a DStream called *ds* from an external source (e.g., Twitter feeds). We then create a GraphStream from it. To do this, we first convert each component RDD into a DeltaRDD that can be used to update graph snapshots. We then call the *createGraphStream* function of the *DeltaDStream* to create the GraphStream *gs*. After applying *triangleCount* graph computation to each graph snapshot and filtering out users whose triangle count is not greater than 1000 (i.e. by setting the count field to 0), the result is a DStream, where each component RDD is the vertex collection (key is vertex ID and value is the number of triangle counts the node involved). Finally, we compute the number of times a user is a top user over a sliding window of 10 seconds, outputting results every 1 second.

Dataset	Edges	Vertices
twitter [5, 6]	1,468,365,182	41,652,230
live LTE network	Varies	2,000,000

Table 1: Datasets used for evaluation. These represent two categories of time-evolving graphs, one that varies slowly, and one that is highly dynamic.

5 Preliminary Evaluation

We now demonstrate the performance of GraphTau using results from our preliminary evaluations.

Evaluation Setup: All our experiments were conducted on Amazon EC2 using 16 `r3.2xlarge` machines unless mentioned otherwise. Each machine consists of 8 virtual CPUs, 61GB or memory and one 160GB SSD storage. The cluster runs a recent 64-bit version of Linux. GraphTau was built on Spark version 1.2.

Datasets: The datasets used in the evaluation of GraphTau is shown in Table 1. The first dataset is the follower-relationship graph in Twitter [5, 6], which was also used in the evaluation of GraphX. This dataset consists of about 41 million users and approximately 1.5 billion edges. The second dataset contains LTE control plane data from a top tier cellular network operator in the United States. The data is from a live network which serves around 2 million subscribers.

Algorithms: We consider two standard algorithms, PageRank and connected components, that can encompass many tasks.

5.1 Streaming PageRank

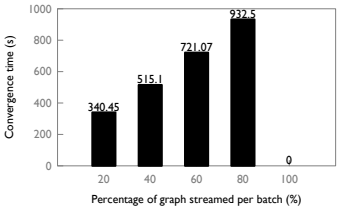
In this experiment, we compare the performance of running PageRank on the Twitter follower graph that evolves over time. Since the Twitter dataset does not provide information on when edges were added or deleted, we revert to simulating the growth of the selected graph over time as follows: we break up the graph into several partitions, ranging from 1 partition (whole graph) to 5 partitions (20% of the graph). Then at fixed intervals, we apply the updates one by one to an initial graph (which begins as an empty one). Intuitively, this represents the evolution of Twitter graph over time.

We then ran PageRank subgraph using both GraphX and GraphTau. Since GraphTau supports streaming version of Pregel, we leverage it to the full extent. Hence, GraphTau starts the PageRank computation whenever a partial graph is available. It iterates over the partial graph until PageRank converges, or a new update is available. When a new update is available, GraphTau pauses the PageRank computation, carries over the state to the updated graph and resumes computation. In contrast, GraphX cannot carry over state whenever an update is available[2]. Figure 4a presents the time taken by both systems to run PageRank to convergence.

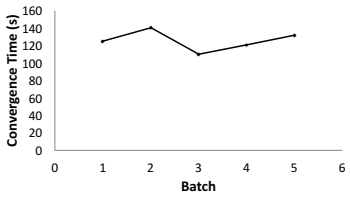
We see that GraphX is not able to run the PageRank to completion (depicted as 0 in the figure). On the other hand, GraphTau is able to complete PageRank when the batch sizes are small. The convergence time for GraphTau also decreases when the batch size is smaller. This is intuitive, since a smaller batch size means GraphTau is able to shift the PageRank computation to the updated graph more frequently. Thus, it can reduce the amount of staleness. On the other hand, GraphX cannot leverage such optimizations. With smaller batches, GraphTau can provide upto 3× improvement in PageRank convergence time compared to larger windows. This is a desirable property in time-evolving graph processing systems, where it is better to process the updates as soon as they are available.

In several cases, it is desirable to run iterative algorithms in a sliding window. Consider PageRank itself—it is easy to envision the need to update the PageRank periodically by updating it with latest

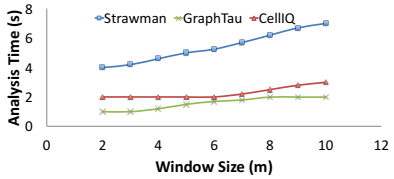
[2]It is possible to implement a clever version of PageRank in GraphX that save results from a previously converged run, but it still cannot apply the results to a new graph until the current run has converged.



(a) Differential computation enables significant gains when done over smaller windows.



(b) Sliding window computation on PageRank shows almost constant convergence time.



(c) Performance matches or exceeds that of specialized systems.

Figure 4: GraphTau's performance.

modifications to the graph while simultaneously removing the edges and vertices that are too old. To simulate this environment, we ran a modified version of the algorithm. Like the previous experiment, we divided the Twitter graph into several small parts. We start with a graph that is formed by the majority of the small parts so that underlying graph on which the PageRank executes is an almost complete graph. Periodically, we remove one subpart from the graph and add a new part and record the time for the algorithm to converge. The results are shown in figure 4b. We see that convergence time almost remains the same between windows, showing the efficacy of GraphTau's incremental computation model.

5.2 Streaming Connected Components

For this experiment, we consider the streaming version of the connected components algorithm. CellIQ [12] has shown that incremental connected component can be very useful in domain specific analytics and proposes a specialized system that can support efficient analysis. Our access to a similar cellular dataset enables us to implement similar analysis tasks in GraphTau. It is to be noted that GraphTau is intended to be a generic streaming graph processing system, and hence may not match the performance of a specialized system. Our intention here is to show that even specialized systems could be implemented using our APIs.

We implemented the persistent hotspot detection algorithm as explained in CellIQ. The purpose of this algorithm is to constantly monitor hotspot components. We compare this against a strawman implementation of the algorithm that simply buffers the graph in the window, combines it and runs connected components. In contrast, GraphTau uses *mergeByWindow* to achieve efficient merging of graphs. The results are shown in figure 4c. GraphTau provides significant benefits over the strawman implementation. While we do not incorporate any domain specific optimizations, it is easy to incorporate them using our API. Thus, we conclude that GraphTau's API is generic to support the implementation of specialized systems.

6 Conclusion and Future Work

There is a growing demand for processing dynamic graph-structured big data in real time. In this paper, we presented GraphTau, a time-evolving graph processing system built on a data flow framework that addresses this demand. GraphTau represents time-evolving graphs as a series of consistent graph snapshots. On these snapshots, GraphTau enables two computational model, the Pause-Shift-Resume model and the Online Rectification model which allows the application of differential computation on a wide variety of graph algorithms. These techniques enable GraphTau to achieve significant performance improvements. For future work, we plan to explore a cost based optimizer that would let GraphTau decide whether it is necessary to rerun an algorithm upon a graph modification, and if so choose which is optimal—incremental computation or restarting the algorithm.

Acknowledgments

We thank the anonymous reviewers for their feedback. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Apple Inc., Arimo, Blue Goji, Bosch, Cisco, Cray, Cloudera, Ericsson, Facebook, Fujitsu, HP, Huawei, Intel, Microsoft, Pivotal, Samsung, Schlumberger, Splunk, State Farm and VMware.

References

[1] 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. *VLDB*, 2013.

[2] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *VLDB*, 2015.

[3] Apache Flink. https://flink.apache.org.

[4] Apache Giraph. http://giraph.apache.org.

[5] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*, 2011.

[6] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW*, 2004.

[7] Z. Cai, D. Logothetis, and G. Siganos. Facilitating real-time graph mining. In *CloudDB*, 2012.

[8] 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*, 2012.

[9] J. Gonzalez, R. Xin, A. Dave, D. Crankshaw, and I. Franklin, Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[11] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: A graph engine for temporal graph analysis. In *Eurosys*, 2014.

[12] A. Iyer, L. E. Li, and I. Stoica. Celliq : Real-time cellular network analytics at scale. NSDI, 2015.

[13] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[14] R. A. Rossi, B. Gallagher, J. Neville, and K. Henderson. Modeling dynamic behavior in large evolving graphs. In *WSDM*, 2013.

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

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

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