Kineograph: Taking the Pulse of a Fast-Changing and Connected World

Raymond Cheng[*†] Ji Hong[*‡] Aapo Kyrola[*◇] Youshan Miao[*§] Xuetian Weng[*♮]
Ming Wu[*] Fan Yang[*] Lidong Zhou[*] Feng Zhao[*] Enhong Chen[§]

[*]Microsoft Research Asia [†]University of Washington [‡]Fudan University [◇]Carnegie Mellon University
[§]University of Science and Technology of China [♮]Peking University

ryscheng@cs.washington.edu ji_hong@fudan.edu.cn akyrola@cs.cmu.edu youshan.miao@gmail.com
{v-xuweng,miw,fanyang,lidongz,zhao}@microsoft.com cheneh@ustc.edu.cn

Abstract

Kineograph is a distributed system that takes a stream of incoming data to construct a continuously changing graph, which captures the relationships that exist in the data feed. As a computing platform, Kineograph further supports graph-mining algorithms to extract timely insights from the fast-changing graph structure. To accommodate graph-mining algorithms that assume a static underlying graph, Kineograph creates a series of consistent snapshots, using a novel and efficient *epoch commit* protocol. To keep up with continuous updates on the graph, Kineograph includes an incremental graph-computation engine. We have developed three applications on top of Kineograph to analyze Twitter data: user ranking, approximate shortest paths, and controversial topic detection. For these applications, Kineograph takes a live Twitter data feed and maintains a graph of edges between all users and hashtags. Our evaluation shows that with 40 machines processing 100K tweets per second, Kineograph is able to continuously compute global properties, such as user ranks, with less than 2.5-minute timeliness guarantees. This rate of traffic is more than 10 times the reported peak rate of Twitter as of October 2011.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design; D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Design, Performance

Keywords Graph processing, Distributed storage

1. Introduction

Increasingly popular services such as Twitter, Facebook, and Foursquare represent a significant departure from web-search and web-mining applications that have been driving much of the distributed systems research in the last decade. Information available on those emerging services has two defining characteristics. First, new information (e.g., tweets) is continuously generated and is far more time-sensitive than mostly-static web pages. Breaking news appears and propagates quickly, with new popular activities and trending topics arising constantly from real-time events in the physical world. Second, while each piece of information may be small and contains limited textual content, rich connections between entities such as users, topics, and tweets can be powerful in revealing important social phenomena. Information search and retrieval on micro-blogs has started to receive a lot of attention [27].

Kineograph is a distributed system designed for the need to extract *timely* insights from such a *continuous* influx of information with *rich* structure and connections. Kineograph has to address a set of new challenges. First, Kineograph must handle continuous updates, and its computation must produce timely results. Ideally, new updates should be reflected in the computed results within a short budget of 1-2 minutes. The widely adopted batch-processing paradigm (e.g., MapReduce [9]) optimizes for throughput and cannot provide the needed timeliness guarantees. Second, Kineograph must maintain a graph structure that captures the relationships among various entities. This is particularly challenging because the graph is often large and must be maintained *consistently* while being stored in a distributed fashion. Third, Kineograph must support graph-mining algorithms that extract insights from the graph structure. A continuously changing graph poses a challenge for many graph-mining algorithms. For example, most of the graph-mining algorithms assume a static underlying graph and their results may no longer offer the same expected meaning when operating on a constantly changing graph.

Kineograph addresses those challenges by designing a distributed in-memory graph storage system, along with a graph engine that supports incremental iterative propagation-based graph mining. The distributed graph store produces reliable and consistent snapshots periodically, so that existing graph-mining algorithms can be applied on a static snapshot. This design also decouples graph mining from graph updates to avoid any unnecessary interference, as graph mining works on existing snapshots while new updates are used to create new ones. Leveraging the nature of graph updates, a simple and novel *epoch commit* protocol with quorum-based replication is used to handle graph updates to achieve consistency and reliability efficiently, without global locking or significant cross-server coordination. With reliable consistent snapshots, computation does not need to be made deterministic or replicated. Kineograph resorts to re-execution in face of failures during graph mining. The final computation results can be replicated using traditional primary-backup schemes.

We have designed Kineograph to be a flexible platform, upon which developers can build scalable graph applications using Kineograph's APIs. We have developed three representative applications on Kineograph with real Twitter feeds for experimentation: TunkRank [31] for user ranking, SP [28] for approximate shortest path, and K-exposure [27] for controversial topic detection. We have conducted experiments on a real Twitter data set that generated a graph with more than 8 million vertices and 29 million edges, at a rate of more than 100,000 tweets per second. Our results show that Kineograph produces timely mining results, such that on average the computed results reflected all tweets updated within 2.5 minutes.

The rest of the paper is organized as follows. Section 2 presents an overview of Kineograph, with details described in the following sections. Section 3 explains how Kineograph maintains and creates distributed consistent graph snapshots. Section 4 introduces Kineograph's graph-computation model. Section 5 illustrates how to build applications in Kineograph, along with the description of three representative applications we built. Section 6 describes Kineograph's support of fault tolerance, incremental expansion, and decaying. Evaluations of the representative applications are reported in Section 7, followed by discussions of related work in Section 8. We conclude in Section 9.

2. Overview

Figure 1 shows an overview of Kineograph. Raw data feeds (e.g., tweets) come into Kineograph through a set of *ingest nodes* (step 1). An ingest node analyzes each incoming record (e.g., a tweet and its associated context), creates a "transaction" of graph-update operations, assigns a sequence number to the transaction, and distributes operations with the sequence number to *graph nodes* (step 2). Graph nodes essentially form a reliable distributed in-memory key/value store, with enhanced graph support. Rather than an opaque value field, the storage engine on each graph node maintains with each vertex, an adjacency list as its *graph structure metadata* and stores separately each application's associated data. In addition, the storage engine supports snapshots. Graph nodes first store the graph updates from ingest nodes. Afterwards, each ingest node reports the graph update progress in a global *progress table* maintained by a central service (step 3). Periodically, a *snapshooter* instructs all graph nodes to take a *snapshot* based on the current vector of sequence numbers in the progress table (step 4). This vector is used as a global logical clock to define the end of an *epoch*. Graph nodes are then instructed to execute and commit all stored local graph updates in this epoch following a pre-determined order. The end result of this *epoch commit* produces a graph-structure snapshot (step 5). Updates in the graph structure due to epoch commit further trigger incremental graph computation on the new snapshot to update associated values of interest (step 6).

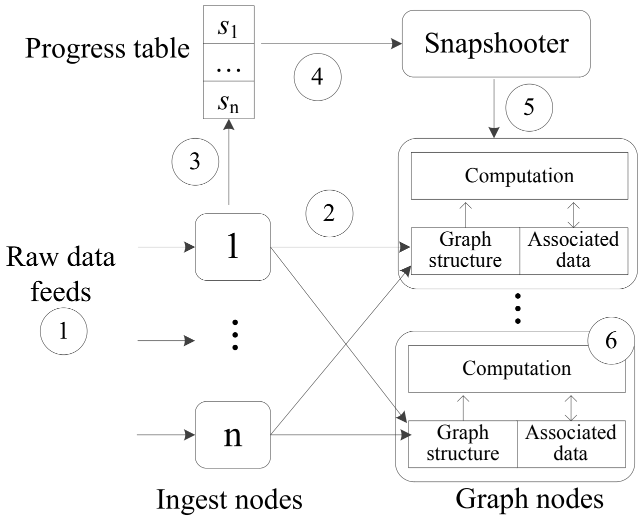

Figure 1. System overview.

A key decision that differentiates Kineograph from existing systems is the *separation* of graph updates and graph computation. This key insight leads to a simple, yet effective system architecture. To enable the separation, Kineograph stores the graph-structure metadata separately from the application data associated with the graph. Graph updates modify only the metadata that defines the graph structure and are therefore simple (e.g., adding a vertex and adding an edge). The separation also gives rise to the epoch commit protocol where graph nodes first store updates and then execute them in an epoch-granularity in order to create globally consistent snapshots on graph structures without global locks. Kineograph further uses the snapshots to decouple graph computation from graph updates in a staged manner: graph computation is performed on static snapshots, greatly simplifying the graph algorithm design. Finally, the separation of graph updates and computation enables the development of separate and simple fault tolerance mechanisms in different components of Kineograph (as will be described in Section 6.1).

3. Creating Consistent Distributed Snapshots

Graph nodes in Kineograph consist of two layers: a *storage layer* that is responsible for maintaining graph data and a *computation layer* that is responsible for graph computation. We describe the storage layer in this section and leave the computation layer to the next.

The storage layer of graph nodes implements a distributed key/value store, enhanced with primitive graph features. A graph is split into a fixed number (say 512) of logical partitions, which are further assigned to physical machines. Currently, graph partitioning is based on the hashing of vertex ids, without any locality considerations. This scheme is simple and generally good for load balance. Each logical partition consists of a set of vertices, each with a set of directed weighted edges stored in a sorted list. Edges are considered part of the graph structure, and are added and modified by the snapshot mechanism in the storage layer. Each vertex also has a set of named *vertex-fields* that store the *associated data* for each configured graph mining algorithm. The type of values stored in vertex-fields is arbitrary, as long as it can be serialized.

A key function provided by the storage layer is to provide consistent snapshots of graph structures. The snapshot mechanism is implemented through cooperation among ingest nodes, graph nodes, and a global progress table. The ingest nodes in Kineograph do not just serve as simple front-ends, but play an important role in the system. An ingest node is responsible for turning each incoming record into a transaction consisting of a set of graph-update operations that might span multiple partitions (e.g., creating vertex v_2, adding an outgoing edge to vertex v_1, and adding an incoming edge to vertex v_2). Each of those operations can be executed entirely on the data structure associated with a vertex. In addition, each ingest node creates a sequence of transactions, each with a continuously increasing sequence number. Those sequence numbers are used to construct a *global logical timestamp* to decide which transactions should be included in a snapshot and also used as the identifier for that snapshot.

Kineograph's snapshot mechanism implements an *epoch commit* protocol that defers applying updates until an epoch is defined, as in the following process. An ingest node sends graph update operations to graph nodes, along with the sequence number of the transaction they belong to. A global progress table keeps track of the progress made by each ingest node by recording a sequence number for each ingest node. An ingest node i updates its entry to sequence number s_i if it has received acknowledgments from all relevant graph nodes that graph-update operations for all transactions up to s_i have been received and stored. Periodically (say every 10

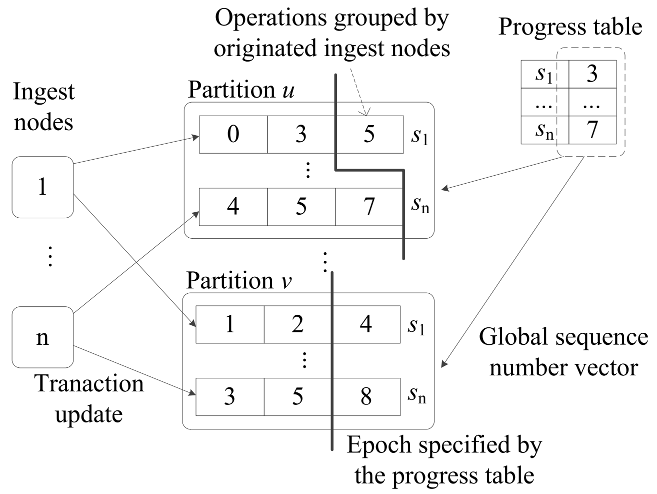
Operations grouped by originated ingest nodes
Progress table
Ingest nodes
Partition u
1
n
Tranaction update
Partition v
Global sequence number vector
Epoch specified by the progress table
s1
sn
0 3 5
4 5 7
1 2 4
3 5 8
3
7

Figure 2. An example of creating a consistent snapshot across partition u and v.

seconds), the *snapshooter* takes the vector of sequence numbers from the current global progress table, $\langle s_1, s_2, \ldots, s_n \rangle$, where s_i is the sequence number associated with ingest node i, and uses it as a global logical timestamp to define the end of the current epoch. The decision is broadcasted to all graph nodes, where all graph updates belonging to this epoch are processed in the same deterministic, but artificial, order in all logical partitions. A graph-update from ingest node i with sequence number s is included in epoch $\langle s_1, s_2, \ldots, s_n \rangle$ if and only if $s \leq s_i$ holds. Even when operations on a logical partition are processed in serial, there are usually enough logical partitions on each graph node, leading to sufficient concurrency at the server level. Figure 2 shows an example where partition u and v are instructed to create a consistent snapshot by a global logical timestamp from the progress table.

The process of creating a snapshot does not stop incoming updates. Ingest nodes continuously send new graph updates into the system with higher sequence numbers. The process of (ingest nodes) dispatching and (graph nodes) storing graph-update operations overlaps with the process of creating snapshots by applying those updates. This property ensures that the deferred execution does not affect throughput over a sufficiently long period of time, even though it might introduce extra latency. Kineograph effectively batches operations in a small epoch window to strike a balance between reasonable timeliness and being able to handle high incoming rate of updates. At a higher rate, batching becomes more effective.

Consistency. The epoch commit protocol provides a non-traditional concurrency control solution that avoids blocking among the ingest nodes. The protocol does not require global serialization when ingest nodes are sending transactional operations, due to the simplicity of graph updates. Global serialization is deferred and implicitly achieved by the snapshooter that retrieves global logical timestamps (a form of vector clock) from the progress table, keeping it off the system's critical path. This process is fundamentally different from existing schemes such as two-phase locking or timestamp ordering [30].

Kineograph guarantees atomicity in that either all operations in a transaction are included in a snapshot or none of them are. This ensures that we cannot have a snapshot that includes one vertex with an outgoing edge, but with no matching incoming edge to the destination vertex. Kineograph further ensures that all transactions from the same ingest node are processed in the same sequence number order. It is worth pointing out that due to the separation of graph updates and graph computation, Kineograph has to deal with only simple graph updates when creating consistent snapshots and leverages the fact that each transaction consists of a set of graph-structure updates that can each be applied on a single vertex structure. Only in the computation phase (for graph mining) have we seen cases where updates depend on the states of other vertices.

In essence, the snapshot mechanism in Kineograph ensures consensus on the set of transactions to be included in a snapshot and can even impose an artificial order within that set, so that all the transactions are processed in the same order. However, the order is artificial. For example, graph nodes can be instructed to process all updates from the first ingest node before processing those from the second, and so on. This externally imposed order does not take into account any causal relationship. It reflects neither the physical-time order nor any causal order. We find it sufficient in our case, partly because Kineograph separates graph updates from graph mining—graph updates are usually simple and straightforward. In most cases, order does not matter at all. Even if updates were applied in a different order, the resulting graphs would stay the same, provided the same order on all graph nodes. One important property Kineograph ensures is deterministic vertex creation. For example, if there is a vertex created for each Twitter user ID, that vertex has an internal ID that depends on that Twitter user ID deterministically. This way, we can create an edge from or to that vertex even before that vertex is created, thereby eliminating cross-operation dependencies.

4. Supporting Incremental Graph-Mining Computation

The computation layer of graph nodes in Kineograph is responsible for executing incremental graph-mining. Computation results are updated based on recent changes in the graph, reflected in new snapshots. Graph-mining algorithms operate on a set of user-defined vertex-fields that store the associated data for those algorithms.

Kineograph adopts a vertex-based computation model [17, 18]. In this model, the data of interest is stored along with vertices, and computation proceeds by processing across vertices.

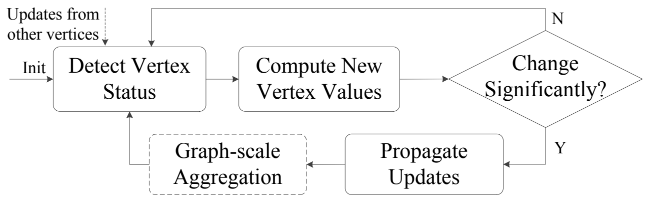

Figure 3. Computation overview.

4.1 Overview

Figure 3 illustrates the overall graph mining process from a vertex's point of view. Initially, Kineograph uses user-defined rules to check the vertex status compared to the previous snapshot. If the vertex has been modified (e.g., edges added, values changed), Kineograph invokes user-specified function(s) to compute the new values associated with the vertex. If the value changes significantly, Kineograph will propagate the changes to a user-defined set of vertices (usually in the neighborhood). There is an optional aggregation phase in the computation within which a vertice might be involved in graph-scale reductions to compute global values. These can be arbitrary complex values, such as top X influential users, or the number of vertices of a certain type. The next iteration of the computation on a vertex is also driven by the status change, but this time it is triggered by the propagation received from other vertices. During a typical computation process, changes in user-defined vertex-fields propagate in a sub-graph, sparked by some change in the structure of the graph (such as adding an edge). The propagation proceeds until no status changes happen across all vertices in the graph, which designates the termination of the computation.

To better support various graph mining algorithms that might require different inter-vertex communication patterns, Kineograph support the *push* and the *pull* models in the computation [17, 26]. To check the status of a vertex and perform computation, the push model allows other vertices to push updates to a specific vertex and a vertex in the pull model proactively pulls data from neighboring vertices. Kineograph further enhances the two models to support incremental computation and efficient distributed execution.

The following subsections elaborate the details of the computation.

4.2 Push model

In the push model, each vertex can send a partial update to another vertex's user-defined vertex-field. For example, the PageRank [22] of a vertex is a weighted sum of the PageRanks of its neighboring vertices. In the push-model, each vertex sends its PageRank to its out-neighbors and the system adds them together to form the total pagerank. In *incremental* algorithms, each vertex sends its incremental change. In the PageRank example, each vertex only needs to send the difference of its current and previous PageRank.

One key feature of the push-model is the ability to perform sender-side aggregation. For each vertex-field, programmers can define a local aggregation function that combines updates sent by several vertices to one single update (see `accumulator` in Section 5). In our implementation, we observe that sender-side aggregation could reduce more than 90% of the total RPC calls during the computation, which shortens the overall computation time significantly.

To further support incremental computation, Kineograph keeps track of *dirty* (i.e., modified) vertices for a new snapshot and during computation (see `trigger` in Section 5). When a field is declared dirty, its *update function* is invoked. The role of an update function is to calculate and *push* the difference of a new value and its previous value to other vertices (see `updateFunction` in Section 5). Kineograph keeps track of the value that is sent to each of the neighboring vertices and performs incremental computation.

4.3 Pull model

A typical vertex update function in a pull model reads the values of its neighbor-vertices and produces a new value for itself. If it determines the change was significant, the function will ask the system to notify its neighbors, and the computation propagates in the graph dynamically.

In Kineograph, to reduce the communication cost an update function could read values from a specified subset of neighboring vertices. For example, some application only needs neighbors of a certain type or an individual vertex (such as a newly created vertex). Likewise, the requested data can be specified as a subset of the data associated with the vertices. In addition, different update functions might need different types of data: perhaps most functions require only the value in a particular vertex-field of a neighboring vertex, but some functions require more data, e.g., a list of edges of a neighbor.

Kineograph schedules updates to vertices in a way that minimizes network communications. In particular, it combines requests to the same vertices (if several update functions request for the same vertex) and executes the updates only when all requested data is available. This is in contrast to the synchronous model where the program issues synchronous calls to vertices while it is being executed. Requests are aggressively batched so there are more chances to merge requests and to reduce the number of RPC-calls.

Kineograph pull-model supports incremental computation by starting the computation from only the changed part of the graph, i.e., new or updated vertices/edges.

4.4 Initializations

Users can define functions that are invoked when there exist new vertices or new in/out-edges in a snapshot (see `initialize` in Section 5). Those are used to initialize the incremental graph computation. In the push model, it is typical to set the corresponding vertex-field to dirty, which will subsequently lead to invoking the update function on the vertex. Likewise, in the pull model, an initialization phase involves asking the system to prepare the data needed to execute an update function.

4.5 Global aggregates

In addition to vertex-based computation, Kineograph provides a mechanism to compute global values using an *aggregator* functions that perform a distributed reduction over all vertices. This mechanism is identical to the Aggregators in Pregel [18] or Sync-mechanism of GraphLab [17], and we do not discuss it in details in this paper.

4.6 Execution schedule

Kineograph is designed for frequent incremental computation steps. It adopts a scheduling mechanism similar to the *partitioned scheduler* introduced in GraphLab [17]. Computation proceeds by executing consecutive *super-steps* on which scheduled vertices are executed across partitions.

The execution model of Kineograph can be seen as a hybrid of the BSP used in Pregel and the dynamic scheduling championed by GraphLab. Unlike GraphLab, Kineograph does not enforce computational consistency: neighboring vertices can be updated in parallel. However, as Kineograph does not allow direct writes to neighbors, write-write races are not possible. In our experiments, we did not notice the need for a stronger consistency guarantee (sequential consistency) for the computation.

Kineograph executes a defined maximum number of super-steps at each snapshot unless the task-queues are empty and there are no vertices to compute, which usually implies the computation has converged. Global aggregators are updated after each super-step.

5. Building Applications on Kineograph

Kineograph is designed to be a platform, where applications can be built on top by having a set of functions instantiated and customized appropriately. First, each ingest node can be instantiated with a function that parses a record in an input stream and produces a transaction consisting of a set of graph-update operations. This function defines the graph structure for an application. In addition to platform-supported graph operations like add edge/vertex, it is possible for an application to define a customized graph-update operation (e.g., increasing the weight of an edge by 10%): the application simply provides a callback function to be invoked when that operation is applied on a graph node in generating a snapshot. An application can further control the configuration of Kineograph in terms of the mapping of vertex IDs to logical partitions, as well as the assignment of logical partitions and their replicas to servers.

Second, an application can define a list of vertex-fields as associated values for a vertex. An application can further define a set of functions to implement a graph-mining algorithm. For the push model, an application defines vertex-fields that can be pushed to as push-fields. They have the following attributes:

- τ defines the type of the field.
- $\texttt{value}_0$ is the initial value of the field.
- `initialize` marks the changed vertices to initiate pushes.
- `updateFunction(vertex)`: the function that will be invoked by the `trigger` function.
- `trigger(oldval : `τ`, newval :`τ`) : boolean`: the function that detects whether the field has changed enough (dirty) to trigger an `updateFunction`.
- `accumulator(accumValue : `τ`, update : `τ`) : `τ accumulates two push updates into one.

For the pull model, an application defines two functions.

- `initialize` provides an update function to process changed vertices and generates a list of `vertex-request` for other vertices (i.e., pull). The `vertex-request` specifies the data fields required from a vertex.
- `updateFunction(vertex,List[readonly-vertex])` modifies the data field of a `vertex`. It is passed a list of *read-only* vertices that correspond to the list of `vertex-request` generated in the `initialize` function. The type is *read-only*, because Kineograph does not allow update functions to directly modify other vertices. These vertices only contain the data fields that were specified in the `initialize` function.

We have implemented three applications on Kineograph. We describe them in the rest of this section.

5.1 TunkRank: computing user influences

One of the most common applications for social network analysis is to estimate the influence of certain users. For our experiments, we use the TunkRank algorithm [31], which is similar to PageRank [22]. In this model, influence of a user X is defined as:

$$\text{Influence(X)} = \sum_{Y \in \text{Followers}(X)} \frac{1 + p * \text{Influence(Y)}}{|\text{Following(Y)}|},$$

where p is a constant retweet probability.

In our experiments, instead of measuring the influence based on "followers", we use a stronger connection between users based on who mentions who. In Twitter, if a tweet contains "@username", it means that the submitter of the micro-blog mentions user *username* (i.e., pays attention to *username*). According to [27], the resulting attention-graph is a more reliable metric of the actual influence than the follower-graph.

To compute the TunkRank, we use the push model as follows:

```
ProcessTweet(tweet) {
  foreach(word in tweet.text) {
    if (word starts "@") {
      mentionedUser = word[1:]
      EmitOperations(createEdge,
        from: tweet.user, to: mentionedUser)
    }
  }
}
```

Figure 4. Pseudo-code of the function for each ingest node that is used to create the attention graph.

```
UpdateTunkRank(v) {
  val newRankToPush =
     (1+p*v["tunkrank"])  /v.numOutEdges()

  foreach(e in vertex.outEdges()) {
    val prevSent = v.("tunkrank", e.target)
    val delta = newRankToPush - prevSent
    if (|delta| > threshold)
      v.pushDeltaTo("tunkrank", e.target,
                    delta)
  }
}
```

Figure 5. Pseudo-code for the update-function for TunkRank algorithm.

- `graph`: a graph of user-vertices with edges connecting users who have mentioned each other. Figure 4 shows the function that ingest nodes use to construct the graph. Each `EmitOperations` emits two `createEdge` operations: one for the source to add an outgoing edge and the other for the destination to add an incoming edge.
- `initialize`: for new out-edges, mark the vertex.
- `updateFunction(vertex)`: sends the difference of the new and the previous weighted TunkRank to its neighbors. Its pseudo-code is shown in Figure 5.
- `accumulator`: sum-operation.
- `trigger(oldval,newval)`: `abs(oldval-newval)>`ϵ.

By adjusting the ϵ in the `trigger`, we can adjust the accuracy/computation time trade-off. In addition, we use an global aggregator object to maintain a list of K most influential users. In the experiment, we set ϵ to 0.001, a value sufficient to find top influential users.

5.2 SP: approximating shortest paths

Computing shortest paths between two vertices in a graph is a classic problem that is interesting in the context of social-network graphs and so on. We implement a landmark-based algorithm introduced by [28]. The algorithm uses a set S of vertices as landmarks (*seeds*). For each vertex, we maintain the shortest-path information from and to S. The shortest path between two arbitrary vertices v_1 and v_2 can then be approximated by the concatenation of shortest paths between v_1 to s and between s and v_2 for some $s \in S$. It has been shown that this approximation is satisfactory with a reasonable set of landmarks. In our experiment, we use the results of TunkRank and selects the top-ranked users as landmarks.

To maintain the shortest-path information from v to a landmark s, we use a *relaxation*-based algorithm derived from Bellman Ford algorithm [8]. The algorithm involves iteratively performing the following operations until no changes occur: for any vertex v, for each of its $inNeighbor$ u, check whether u can get a shorter path toward s with v as its first step (i.e., checking whether $dist(v) + 1 < dist(u)$). If so, reset the distance of u as the smaller value and then schedule a same procedure from u later. We implement this algorithm in the push model.

- `graph`: same mention-graph as in TunkRank.
- `initialize:` for new in-edges, mark the vertex.
- `updateFunction(vertex)`: sends its own length of shortest path plus one as candidates to its in-neighbors.
- `accumulator`: minimize-operation.
- `trigger(oldval,newval)`: `oldval.length` > `newval.length`.

5.3 K-exposure: detecting controversial topics

Our third algorithm was recently proposed in [27] as a way of identifying hashtags ("#tag") that are *controversial*. [27] discovered in particular that political hashtags of controversial subjects had a clearly different spreading pattern than light topics such as celebrity-related hashtags. To study these patterns, an *exposure histogram* is computed for each hashtag. Exposure is computed as follows: let S be a micro-blog post by user U that contains hashtag H at time t. Then $k(S)$ is defined as follows:

$$k(S) = |\{\text{neighbors of U}\} \cap \\ \{\text{ users who posted a message with } H \text{ at time } < t\}|.$$

Note that t is defined by the timestamp information attached to every tweet.

By computing $k(\cdot)$ for posts that contain H, we can compute the *k-exposure histogram* for each hashtag. Our implementation of this algorithm uses the pull model, is incremental, and does not propagate.

- `graph:` for each unique hashtag and user we create a vertex and assign an edge from the hashtag to the user. In addition, we utilize the same mention-graph created for TunkRank.
- `initialize:` for each out-edge added to a hashtag, schedule the update function and request all edges of the corresponding user (target vertex of the edge).
- `updateFunction(hashtagVertex, [userVertex]):` compute the intersection of the out-edges of `userVertex` and `hashtagVertex` and update the k-exposure value of `hashtagVertex`.

6. Fault Tolerance, Incremental Expansion, and Decaying

As a distributed system, Kineograph must tolerate failures and allow incremental expansion to cope with increasing update rates and computation needs. Unique to Kineograph, because of the time-sensitive nature of the applications it targets, Kineograph should ideally support decaying, so that newer information has a higher weight in the results we produce. The design of Kineograph makes it easy to support fault tolerance, incremental expansion, and decaying, as we describe in detail here.

6.1 Fault tolerance

Kineograph has servers taking different roles in the system; each of them needs to be designed to cope with failures. A Paxos-based [15] solution (e.g., Chubby [4] or ZooKeeper [12]) can be used to implement Kineograph's centralized functionalities, such as maintaining the global progress table, coordinating graph-mining computation, monitoring machines, and tracking replicas. As the mechanism is well-explained in existing literature under similar settings, we do not describe it in detail here. Our current implementation uses a single server.

Ingest nodes. Because ingest nodes in Kineograph are more than just stateless front-ends, care must be taken in handling their failures. Kineograph's epoch commit protocol assumes that each ingest node produces monotonically increasing sequence numbers for transactions of graph-structure updates. This property must be preserved despite machine failures. Note that it is possible for an ingest node to fail in the middle of sending updates to multiple graph nodes.

Kineograph introduces *incarnation numbers* and leverages the global progress table to address this problem. Each ingest node has an incarnation number. We replace sequence numbers with pairs $\langle c, s \rangle$, where c is an incarnation number and s is a sequence number. They are used in graph-structure updates sent to graph nodes and recorded in the global progress table. When an ingest node fails and recovers, or when a new machine takes the role of a failed ingest node, that resurrected ingest node i consults the global progress table for the pair $\langle c_i, s_i \rangle$ associated with ingest node i. It seals c_i at s_i and uses $c_i + 1$ as the new incarnation number. It can reset the sequence number to 0 or continue at $s_i + 1$.

By sealing c_i at s_i, all requests with $\langle c_i, s\rangle$ where $s > s_i$ are considered invalid and discarded. To avoid any loss of transactions, all incoming data feeds must be stored reliably and can only be garbage collected after they have been reflected in the progress table. Here, we are taking advantage of epoch commit to "undo" operations for free.

Replication at the storage layer. The separation of graph updates and graph computation is crucial in simplifying fault tolerance in Kineograph, as Kineograph uses two different mechanisms to handle failures at the storage layer and at the computation layer.

At the storage layer, graph-update operations and the resulting graph data need to be stored reliably on graph nodes. We leverage ingest nodes and use a simple quorum-based replication mechanism: each logical partition is replicated on k (say 3) different machines and can tolerate f (say 1) failure, where $k \geq 2f + 1$ holds. Graph-update operations are then sent to all replicas and an ingest node considers the operation reliably stored as long as $f + 1$ replicas have responded.

Some replicas might miss some operations for its logical partition. An ingest node keeps a counter for the number of operations for each logical partition and attaches the counter with each operation. A replica can use the counter to identify holes and ask the missing information from other replicas. (Note that some transactions might not touch certain logical partitions. Therefore, we cannot use sequence numbers to identify missing operations.) All replicas will create the same snapshots as they apply the same set of operations in the same order. We rely on the fact that graph update operations are deterministic.

A replica G loses all the in-memory data in case of machine failures. Kineograph will replace G with a new node G'. In order to retrieve the lost data and catch up with the other replicas in the same replica group R, G' first asks each ingest node s to send all future operations hosted on R to G' starting from sequence number t_i ($1 \leq i \leq n$). Once G' learns from the snapshooter that a snapshot P with a vector clock $\langle s_1, s_2, \ldots, s_n\rangle$ satisfying $s_i \geq t_i$ for each $1 \leq i \leq n$ is created, G' retrieves snapshot P from other replicas in R and hence has all the information needed to take over G. During the recovery process, other replicas in R continue to serve graph updates and produce snapshots, hence the service will not be interrupted.

Replication at the computation layer. Kineograph triggers incremental graph-mining computation on consistent snapshots. Each invocation of computation takes a relatively small amount of time (up to the order of minutes in our experiments). Because snapshots are reliably stored with replication at the storage layer, Kineograph simply rolls back and re-executes if it encounters any failures in a computation phase. The result of a computation can be replicated to tolerate failures. We do not perform replicated graph-mining computation on replicas since certain graph computation is non-deterministic. Instead, we use a simple primary/backup replication scheme, where the primary does the computation and copies the results to the secondaries.

6.2 Incremental expansion

The scale of Kineograph depends on many factors, including the rate of incoming data feeds, the size of the resulting graphs, and the complexity of graph-mining computation. There are cases where Kineograph needs to recruit more machines into the system in order to handle higher load, larger amount of data, and/or heavier computation. In our experiments, we have seen continuously increasing memory footprint as the system takes in more and more data, partly because we have not implemented any decaying mechanism. It would be ideal to be able to spread the graphs onto a larger set of machines when needed.

In our design, we create a large number of logical partitions up front. Incremental expansion can then be achieved by moving certain logical partitions to new machines, rather than splitting logical partitions, although technically splitting logical partitions can easily be achieved as well.

Live migration of a partition can be challenging in general, but is made much easier thanks to our snapshot mechanism. For simplicity, we ignore replication in the description, as adding replication into the protocol is straightforward. The overall procedure is similar to the failure recovery mechanism at the Kineograph storage layer. Suppose Kineograph wants to migrate a logical partition from S to T. It communicates with each ingest node s about the migration and a promise to send all future operations on that logical partition to both S and T starting from sequence number t_i. Once a snapshot with a logical clock $\langle s_1, s_2, \ldots, s_n\rangle$ satisfying $s_i \geq t_i$ for each $1 \leq i \leq n$ is created, Kineograph instructs a copy of that snapshot from S to T. Once T receives the snapshot, it has all the information needed to take over the logical partition from S. Because computation overlaps with incoming updates, T can usually catch up with S quickly without causing any performance degradation. We have not implemented incremental expansion at the time of this writing.

6.3 Decaying

In our experiments, we have seen a continuous increase in the graph size as Kineograph contunuously takes in more data. In practice, the value of information decays over time and outdated information should gradually have decreasing impact on results. Although we have not implemented this mechanism, we outline here how Kineograph could support decaying by leveraging global logical clocks based on sequence numbers.

Suppose we care only about the information in the last n days and that the information within those n days has a different weight depending on which day it is. Kineograph can essentially create $n + 1$ parallel graphs to track the last n days and plus the current day. The window slides as a day

Kineograph	LoC	Applications	LoC
Storage	6180	TunkRank	310
Computation	6714	K-Exposure	137
RpcLib	1177	SP	487
Log	2123	GraphUpdate	527
Total: 17655			

Table 1. Line of code count breakdown.

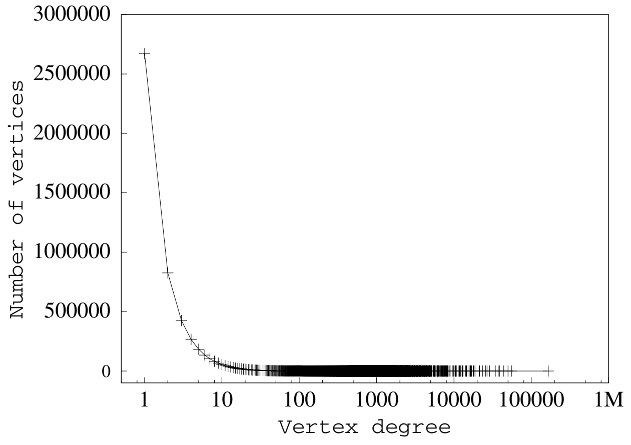

Figure 6. In-edge degree distribution across vertices.

passes. Instead of using real time, which could lead to inconsistencies in a graph due to different interpretations of real time on different servers, Kineograph align those decaying time boundaries with the epochs defined by logical clocks of sequence numbers. When a day passes in the real time, Kineograph can look at the current epoch number and use this as the boundary. The real graph used for computation can be constructed by taking a weighted average of those parallel graphs.

7. Evaluation

We have implemented Kineograph using C# with more than 17,000 lines of code, excluding test code. Table 1 summarizes the lines of code for different components in the Kineograph system and its applications. Note that we have developed our own RPC library that allows optimized data marshalling.

We evaluated Kineograph on a cluster with up to 51 machines, each connected with Gigabit Ethernet. 25 of the machines contained an Intel Xeon X3360 CPU (quad-core, 2.83GHz) and 8GB memory. The remaining had an Intel Xeon X5550 CPU (quad-core, 2.67GHz) and 12GB memory. All the machines ran the 64-bit version of Windows Server 2008R2 with .NET framework 4.0.

In our experiments, we simulated a live Twitter stream by feeding our system from a bank of 100 million archived tweets. This data set forms a graph of over 8 million vertices (users and hashtags) and 29 million edges. Figure 6 shows the edge distribution in the graph Kineograph constructs. The distribution exhibits *power law*, where very few vertices

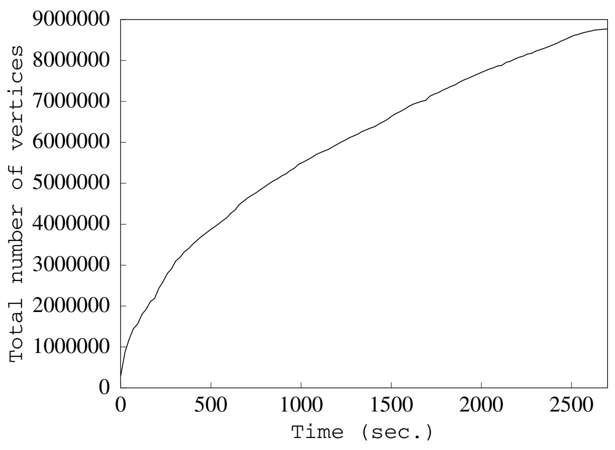

Figure 7. Number of vertices increases over time with 2 ingest nodes and 32 graph nodes.

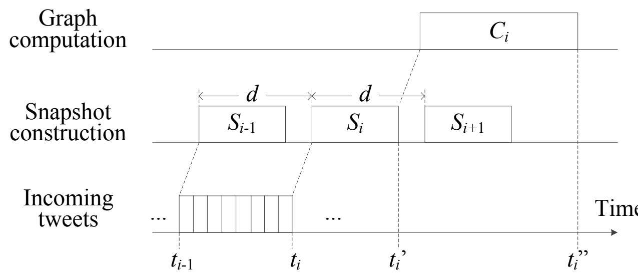

Figure 8. A streamlined view of graph updates and graph computation.

have extremely high degrees (more than 200k). Figure 7 shows how the graph size changes when we feed the data into the system with two ingest nodes.

To understand the end-to-end performance of the system, we ran evaluations across the three applications described in Section 5 on top of Kineograph.

Kineograph is designed to capture and mine a changing data set (graph) in a timely manner. There are two key system properties that are of interest to potential Kineograph users. i) *Update throughput*: whether Kineograph is able to support high update rates to the graph. ii) *Data timeliness*: whether Kineograph can help applications compute timely results out of the changing graph. We will report the experimental results and our findings in the rest of this section.

7.1 Graph update throughput

Kineograph should be able to support high update throughput that matches existing popular on-line services like Twitter. As of October 2011, the peak amount of Twitter traffic was 8.9K tweets per second [29]. Ideally, the system should further take the future growth of the Twitter service into account.

In Kineograph, throughput is measured by counting the number of tweets that have been processed after Kineograph finishes constructing a snapshot for each epoch. As illustrated in Figure 8, snapshot construction is performed at a

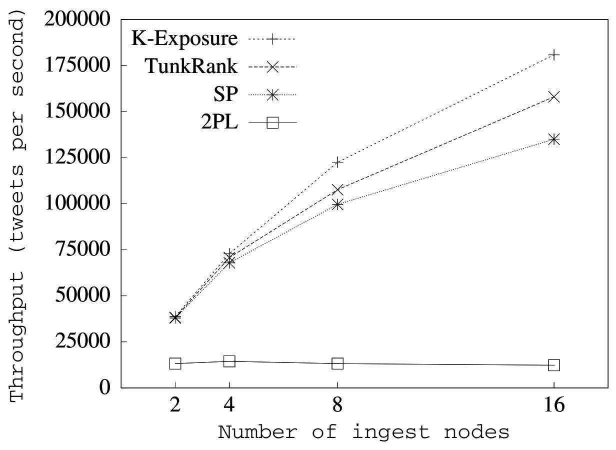

Figure 9. Graph-update throughput on 32 graph nodes with varying numbers of ingest nodes and with different applications. Snapshot interval is set to 10 seconds.

regular interval d. Snapshot S_{i-1} is constructed out of the tweets accumulated before time t_{i-1}. Snapshot S_i is constructed with those tweets logged between time t_{i-1} and t_i (denoted as $N[t_{i-1}, t_i]$). Therefore, the average throughput is calculated by $N[t_{i-1}, t_i]/d$.

To evaluate the peak graph-update throughput, we use a preprocessor to parse the retrieved tweets and keep only the information of interest to the applications (e.g., tweet text, user name, and time stamp). The pre-processing reduces the data-processing burden on an ingest node so that we can generate more loads with a small number of ingest nodes to the system for higher update throughput. By default, Kineograph enables batch update (batch size set to 512) to improve communication efficiency.

In our first experiment, we use different numbers of ingest nodes to inject input streams at different rates and measure the average graph-update throughput of Kineograph. All experiments use 32 graph nodes with the snapshot interval set to 10 seconds. The results are shown in Figure 9: the update throughput increases with the number of ingest nodes. With 16 ingest nodes, the sustained average update throughput can be more than 180k tweets per second, 20 times more than the recorded Twitter traffic peak as of Oct. 2011 [29].

Figure 9 also shows that the update throughput varies under different applications. This is because the update procedure competes for computation resources (e.g., CPU cycles and network bandwidth) with applications running on top of Kineograph. Since different applications consume different amounts of computation power, update throughputs suffer from different levels of interference. For example, SP and TunkRank require more computation power than K-Exposure. Thus the update throughput under K-Exposure is higher than those under the other two applications.

To understand further the benefit of the epoch commit protocol for graph updates, we implement a simplified two-phase locking (2PL) scheme [30], where ingest nodes obtain locks from graph nodes in a fixed-order (to avoid deadlocks) and release locks when all locks are obtained. We omit the actual execution of the operations and enable batching (with a batch size of 512) to improve throughput. Batching reduces the number of round-trips, but at the risk of introducing more contention due to coarse granularity. Our experiments do show better throughput at that batch size. Figure 9 shows that the throughput of a 2PL-based scheme does not increase with the number of ingest nodes. A closer look reveals significant contention in the system, mostly due to well-connected vertices in the graph. Due to the power-law distribution shown in Figure 6, most concurrent updates compete for the access to very few vertices, which results in significant lock contention.

Snapshot Interval(s)	SCT Max/Avg	Avg SCT(s)	Throughput(t/s)
10	3.1	1.9	137.6k
30	2.2	4.4	143.0k
60	1.9	8.4	150.8k

Table 2. The impact of transient imbalance on throughput under K-Exposure, with 8 ingest nodes, 32 graph nodes. SCT Max/Avg: The ratio of maximum over average Snapshot Construction Time.

Figure 9 also shows that the update throughput increases sub-linearly with the number of ingest nodes. Our investigation attributes this to *transient load imbalance* during updates. In particular, we find that some graph nodes take more time to construct a snapshot locally than others. Table 2 shows an example in which the graph node with the maximum snapshot construction time (SCT) can spend 3 times as much as the average time of snapshot construction (denoted as SCT Max/Avg ratio).

We further observe that the imbalance is *transient*. Smaller snapshot windows, translates to more severe imbalances, and correspondingly, lower aggregated throughput. For example, if we increase the snapshot interval from 10 to 60, the SCT Max/Avg ratio decreases from 3.0 to 1.9. Consequently, the total update throughput increases by 10%. (Note that larger interval also makes batching more effective.) A larger snapshot interval improves throughput, at the expense of timeliness.

In summary, Kineograph can support the current peak throughput and leaves plenty of capacity for future growth of online services.

7.2 Data timeliness

The next set of experiments focus on data timeliness. As shown in Figure 8, the computation for snapshot S_i completes at time t''_i. The computation result C_i reflects the input data between t_{i-1} and t_i. We define timeliness for the input window $[t_{i-1}, t_i]$ to be between $t''_i - t_i$ and $t''_i - t_{i-1}$.

As shown in Figure 8, data timeliness depends on snapshot interval d, snapshot construction time (SCT) for snap-

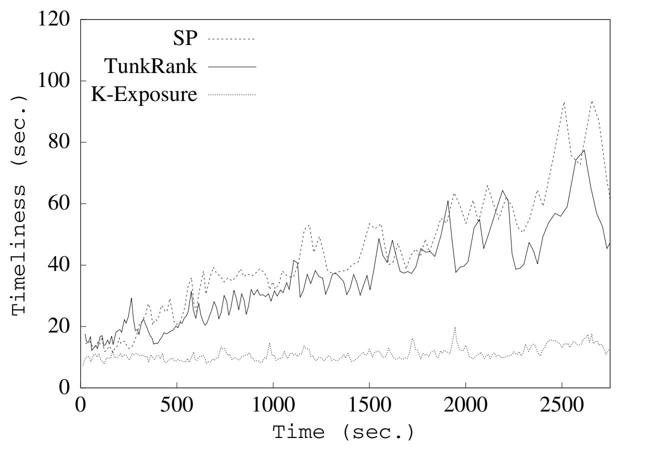

Figure 10. Data timeliness for different applications with 2 ingest nodes and 32 graph nodes.

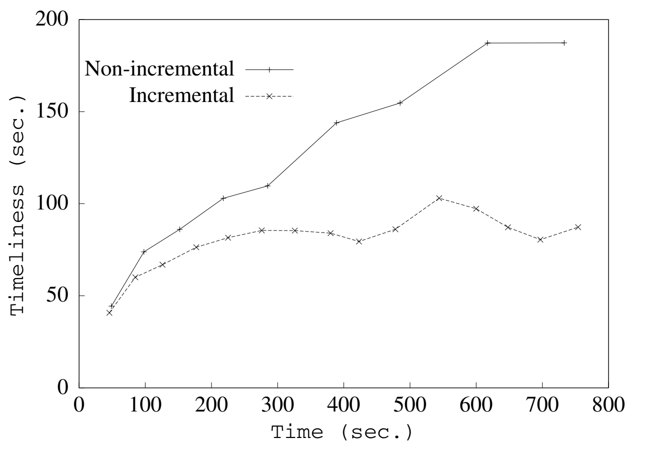

Figure 11. Timeliness changes over time for incremental and non-incremental graph computation with TunkRank, 4 ingest nodes, and 32 graph nodes.

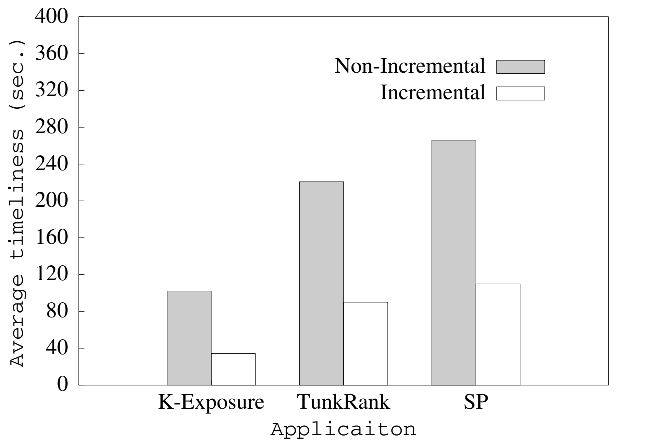

Figure 12. Average timeliness improvement of incremental applications under 4 ingest nodes and 32 graph nodes.

shot S_i, and computation time for the corresponding result C_i. Since computation complexity of different applications varies, the actual data timeliness differs on different applications, even when Kineograph has stable system behavior.

Figure 10 shows data timeliness of different applications and confirms the impact of application complexity. Compared to K-Exposure, the TunkRank and SP algorithms are more complex, require multiple rounds, and suffers from worse data timeliness.

Figure 10 also shows that data timeliness becomes worse over time. A closer look shows that this property is mainly due to increases of the graph size over time, as shown in Figure 7. We did not use decaying in our experiments.

The overall data timeliness is within minutes. In the case of TunkRank, Figures 9 and 14 show that Kineograph is able to provide data timeliness of less than 3 minutes, with the update throughput more than 100k tweets per second, more than 10 times the peak throughput of Twitter (as of Oct. 2011). Two factors contribute to such good timeliness results: support of incremental graph computation and the use of a distributed system. We evaluate the effect of these two factors in our subsequent experiments.

Figure 11 shows the benefit of incremental TunkRank computation over the non-incremental version. Because the incremental version reuses the previous results as the starting points, we actually see that the benefit grows over time. As the graph size grows, the non-incremental version has to compute TunkRank from scratch at an increasing cost. Figure 12 shows similar benefits of incremental computation for all three applications.

Figure 13 demonstrates the scalability of TunkRank running on Kineograph. With more graph nodes involved in the computation, the data timeliness becomes better, an indication of reduced computation time. This data shows that Kineograph scales well with the increase of graph nodes, for this particular application. The 32-node case has 84% better timeliness than the 8-node case. Note that the 32-node case has more than 4 times speedup over the 8-node case, mainly due to the extra memory management cost (e.g., garbage collection) in the 8-node case.

Another factor that affects timeliness is the incoming data rate on the system. Figure 14 shows that with added ingest nodes and increased incoming data rate, the data timeliness becomes worse. The results arise from three factors. First, for the given input stream at the same time instance, the graph size becomes larger at a higher rate; computation over a larger graph is expected to take more time. Second, at a higher rate, the *change* of the graph between two snapshots becomes larger. Consequently, it takes more time to finish the computation, even with incremental computation. Finally, a higher incoming rate leads to higher throughput and more resource consumption, taking resources away from computation and causing it to slow down.

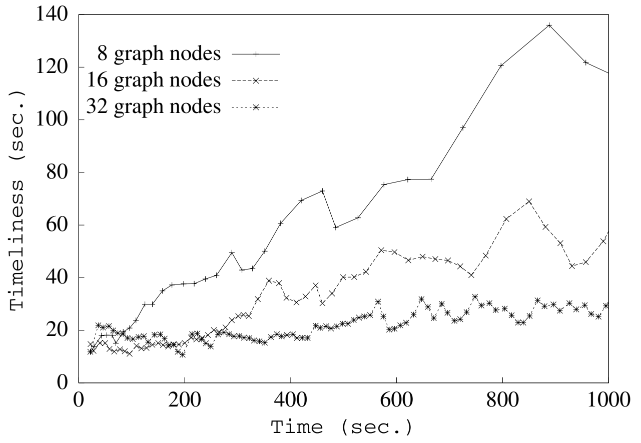

Figure 13. Scalability of TunkRank with different numbers of graph nodes and 2 ingest nodes.

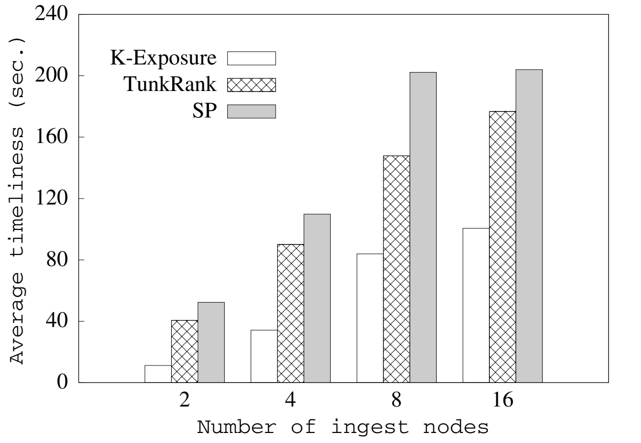

Figure 14. Average data timeliness with different number of ingest nodes and 32 graph nodes.

7.3 Fault tolerance

We demonstrate the system behavior under machine failure using 2 ingest nodes and 48 graph nodes, running TunkRank. The graph nodes host 16 graph partitions in total, i.e., every partition has three replicas. To fully utilize computation resources, for every graph partition, each of the three replicas is responsible for the computation of 1/3 of the graph partition.

Figure 15 shows how the system performance changes over time when we kill one graph node during the experiment. At the time around 324 seconds (t_0) after the experiment begins (after the construction of snapshot 31), we terminate one graph node. The system detects the failure and initiates the failure recovery process for the graph node as described in Section 6.1.

As shown in Figure 15, since the storage layer of Kineograph uses a quorum-based replication scheme, the graph update throughput does not suffer from the machine failure. The computation layer, however, is more vulnerable to failures since Kineograph does not replicate computation. The TunkRank computation stops right after the failure occurs. Therefore we observe no timeliness data during the failure time (between t_0 and t_1). The computation restarts after the resurrected graph node catches up with other replicas at snapshot 35 (around 360 seconds). As expected, the data timeliness right after the recovery increases greatly because the graph data has been accumulated over 30 seconds and the benefit of incremental computation decreases due to larger graph changes. Actually, Kineograph produces the first computation result at t_1, more than 40 seconds after the completion of the failure recovery procedure. The data timeliness becomes normal after several rounds of computation since the TunkRank incremental computation gradually catches up with the changes in the graph.

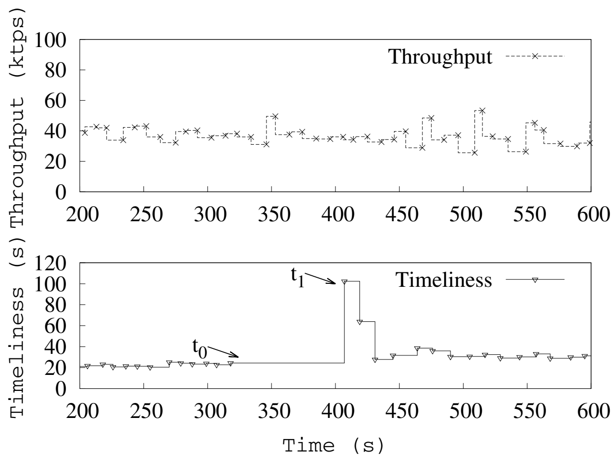

Figure 15. TunkRank data timeliness (s) and graph update throughput (1k tweet/s) with 2 ingest nodes and 48 graph nodes under 1 graph node failure.

In summary, the failure behavior of Kineograph matches our design goal well.

8. Related Work

Kineograph builds on a large body of existing literature in distributed systems and database systems. We focus on three most related areas: distributed in-memory storage (key/value) systems, incremental data processing, and graph computation.

Distributed in-memory storage systems. Distributed in-memory key/value stores have received a lot of attention, both in the research community and in the industry [13, 19, 21]. Kineograph leverages this technology, adds basic graph support, and more importantly supports snapshots.

Incremental data processing. Recently, many research efforts have focused on improving computation efficiency through augmenting existing scalable batch-processing engines with incremental computation capability. Systems like

Incoop [2], Haloop [3], DryadInc [25], Yahoo continuous bulk processing (CBP) [16], Comet [11], and Nectar [10] achieve this by allowing their applications to reuse existing computation results. However, those systems are not designed for scenarios with fast continuous data updates and timely computation results. Most of the work focuses on variations of relational or MapReduce models, rather than a graph model.

Google Percolator [24] provides a trigger-based event-driven programming model for incremental web index construction. It provides a lock-based mechanism to support update transactions with snapshot isolation. Their design targets the scenarios where the conflict rate of transactions is low. This is unfortunately not the case in highly-connected graph structures. Instead of using locking, Kineograph uses epoch commit to construct consistent snapshots.

There are also extensive works in the database community on incremental computation [5–7]. Stream processing databases continuously accept new incoming updates, incrementally maintain database view, and adopt window-based relational operators to process the incoming data and generate results in real-time. Kineograph differentiates itself from them on the following aspects. First, rather than incremental computation only on a window of incoming updates, Kineograph supports computation on a new global snapshot and needs to merge efficiently new updates with the existing snapshot to construct a new one. Secondly, Kineograph targets graph computations, which might not be well supported in a relational data model [1, 20].

Graph computation. In recent years there has been a lot of interest in the research and industry towards graph computation, which has been driven by the rapid growth of graph data, such as social networks and the web. In addition, the scientific computation community has been studying planar and grid-graphs for decades. Recent influential works on *vertex-based* computation include Google's Pregel [18] and GraphLab [17]. Pearce et al [23] propose an asynchronous graph computation model for multi-core that is based on an extended version of the graph visitor pattern. To process graphs that do not fit in memory, [23] employs efficient mechanisms to store part of the graph in flash-memory, while we provide a distributed computation model. Pegasus [14] is a collection of highly scalable batch graph-mining algorithms written for Hadoop. Unlike the existing offline graph engines that perform graph computation on static graph structures, Kineograph extends them with snapshot-awareness on a fast and continuously changing graph structure, and provides the ability to perform incremental graph-mining to produce timely computation results.

9. Concluding Remarks

Kineograph reflects our belief that there is a potential paradigm shift in distributed-system research. It departs from the now "traditional" areas of high-throughput and scalable batch systems, as represented by systems such as GFS and MapReduce. The new paradigm is inspired by increasingly popular social networking, micro-blogging, and mobile Internet applications. These services are more centered around graph-based storage and computation, while striking a different and delicate balance among timeliness, consistency, and throughput. While this paper focuses on an overall architectural design with novel constructs, we expect to see new abstractions and building blocks emerging in the near future.

Acknowledgments

We are in debt to Emre Kiciman, Sean McDirmid, and Vijayan Prabhakaran for the discussions and feedback on Kineograph. We would also like to thank the reviewers and our shepherd Niels Provos for their valuable comments and suggestions.

References

[1] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.

[2] P. Bhatotia, A. Wieder, R. Rodrigues, U. Acar, and R. Pasquini. Incoop: MapReduce for incremental computations. In *ACM SoCC*, 2011.

[3] Y. Bu, B. Howe, M. Balazinska, and M. Ernst. HaLoop: Efficient iterative data processing on large clusters. In *VLDB*, 2010.

[4] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[5] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams – a new class of data management applications. In *VLDB*, 2002.

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[7] J. Chen, D. J. Dewitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.

[8] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd. edition, 2001.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1): 107–113, 2008.

[10] P. Gunda, L. Ravindranath, C. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, 2010.

[11] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: Batched stream processing in data intensive distributed computing. In *ACM SoCC*, 2010.

[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[13] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A high-performance, distributed main memory transaction processing system. In *VLDB*, 2008.

[14] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. In *IEEE International Conference on Data Mining*, 2009.

[15] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[16] D. Logothetis, C. Olston, B. Reed, K. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *ACM SoCC*, 2010.

[17] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence(UAI)*, 2010.

[18] 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*, 2010.

[19] memcached. Memcached: A distributed memory object caching system, 2011. `http://memcached.org`.

[20] Neo4j. Neo4j: The graph database, 2011. `http://neo4j.org`. Accessed October, 2011.

[21] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *SOSP*, 2011.

[22] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. *Stanford Technical Report*, 1999.

[23] R. Pearce, M. Gokhale, and N. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[24] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[25] L. Popa, M. Budiu, Y. Yu, and M. Isard. Dryadinc: Reusing work in large-scale computations. In *HotCloud*, 2009.

[26] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[27] D. Romero, B. Meeder, and J. Kleinberg. Differences in the mechanics of information diffusion across topics: Idioms, political hashtags, and complex contagion on twitter. In *WWW*, 2011.

[28] A. Sarma, S. Gollapudi, M. Najork, and R. Panigrahy. A sketch-based distance oracle for web-scale graphs. In *WSDM*, 2010.

[29] D. Sullivan. Tweets about steve jobs spike but don't break twitter peak record, 2011. `http://searchengineland.com/tweets-about-steve-jobs-spike-but-dont-break-twitter-record-96048`.

[30] A. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[31] D. Tunkelang. A twitter analog to pagerank. *Retrieved from http://thenoisychannel. com/2009/01/13/a-twitter-analog-to-pagerank*, 2009.