2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

# Scalable Infrastructures for Data in Motion

David Ediger Rob McColl Jason Poovey Dan Campbell
Georgia Tech Research Institute
Atlanta, Georgia

***Abstract*—Analytics applications for reporting and human interaction with big data rely upon scalable frameworks for data ingest, storage, and computation. Batch processing of analytic workloads increases latency of results and can perform redundant computation. In real-world applications, new data points are continuously arriving and a suite of algorithms must be updated to reflect the changes. Reducing the latency of re-computation by keeping algorithms online and up-to-date enables fast query, experimentation, and drill-down. In this paper, we share our experiences designing and implementing scalable infrastructure around NoSQL databases for social media analytics applications. We propose a new heterogeneous architecture and execution model for streaming data applications that focuses on throughput and modularity.**

## I. Overview and Background

Among the many use cases of "big data", the characteristics of streaming social media have motivated a paradigm shift in algorithms and system design for large-scale computing. Aggregate data set sizes range on the order of petabytes. New data arrives continuously. Twitter and Facebook report sustained data volumes of 5,000 to 50,000 events per second with momentary peaks over 100,000 events per second [1], [2].

Computing in a batch model over the full data set might require hours or days—too long for an Internet in which conversations can become trending topics in a matter of minutes. With high-velocity data in motion, the objective is to minimize the latency between when data is generated and when a suite of analytic applications is updated to reflect the new information.

Data arising from social media contain rich semantic information. Aside from the content (which itself can vary by language and format restrictions), we can obtain information about the poster and any other actors that are mentioned, including geographical and profile data. It is not uncommon to have presences on multiple services. Aggregation and correlation of data by actor across systems is an ongoing requirement.

The questions that analysts may ask are similarly varied. A simple query might look like "What are the most common words and phrases right now?" Applying social network analysis techniques to the graph of who is speaking to or referring to whom, one might ask "Who are the influential posters and how do they form communities?" Looking at the content, we can extract topics of discussion through classification and measure sentiment of text for positive and negative response. These methods can be combined in many variations to drill down into the data.

Having developed state-of-the-art parallel algorithms to solve many of these queries on dynamic data sets [3], [4], [5], [6], this paper describes and justifies a scalable infrastructure for managing streaming data and analytic tools that supports real-time applications with complex queries. In the remainder of this section, we describe existing approaches to large-scale analytics, as well as give a brief overview of the algorithms of interest. In Section II, we describe a feed-forward streaming architecture that supports multiple data ingest streams, a multithreaded in-memory graph database, and a scheduler for analytic modules that publish and subscribe to result stores. Section III analyzes the data ingest portion of the system, including a new template-based text parser that converts structured data objects (such as Tweets in JSON format) into graph edges. Section IV describes the multithreaded in-memory server process that manages the graph data store and schedules algorithms. Last, we demonstrate how to implement RESTful and stateful HTTP interfaces to this data store.

### *Prior Work*

Research and development of large-scale data mining frameworks for complex data have focused largely on distributed memory, homogeneous clusters. A variety of execution models are employed: MapReduce, bulk synchronous parallel, asynchronous message passing, on-disk and in-memory. Graph algorithms, particularly when applied to social network analysis, have garnered much attention in the research community because of data scale and their inherent complexities.

Pegasus [7] is a graph mining system that runs on Apache Hadoop MapReduce and the Hadoop Distributed File System (HDFS). Algorithms are expressed in a data-parallel model that operates on all graph edges (or vertices) simultaneously.

Google's Pregel [8] implements a distributed graph processing framework with a bulk synchronous parallel (BSP) execution model. Vertices are mapped to cluster nodes with their incident edges. At each time step, vertices receive messages from other vertices, compute and modify their local state, and send messages to other vertices to be received in the next time step. Graph algorithms are easily expressed in this model, although communication volume can be high [9].

The GraphLab [10] execution model extends BSP with asynchronous execution. Each vertex is computed in parallel. During execution, it can read the state of neighboring vertices,

978-1-4799-2784-5/14 $31.00 © 2014 IEEE
DOI 10.1109/CCGrid.2014.91

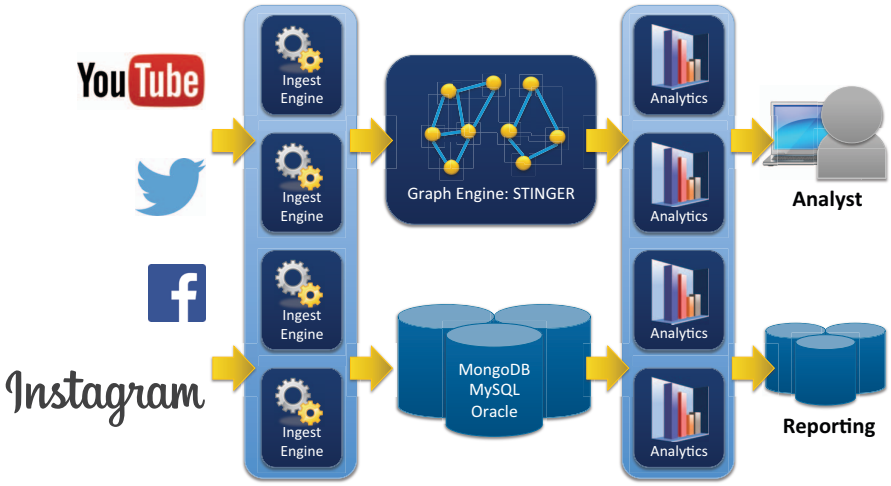



Fig. 1. Conceptual overview of the STINGER streaming framework. The framework converts raw social media data to representations amenable to graph databases, NoSQL, and relational databases. Analytics use the data source most appropriate for their queries.

perform computation, and signal neighboring vertices. The computation section is decomposed into three phases: gather, apply, and scatter. This decomposition enables the framework to further parallelize the gather and scatter portions of the computation.

Spark [11] is an in-memory MapReduce framework for distributed computing that runs on top of Hadoop 2 YARN. A suite of tools for SQL, machine learning, graph analytics, and streaming computations are built on top of Spark.

Our work on STINGER [12] focuses on the streaming graph model. In this model, an unending stream of graph edge insertions and deletions arrives continuously. Graph algorithms minimize re-computation by isolating only the changes to the graph data structure and metric result. STINGER is a multithreaded, in-memory graph data structure that supports edge and vertex types, edge and vertex weights, and edge timestamps.

For a comparison and performance evaluation of these and other open source graph databases, please refer to [13].

A number of efforts are underway that implement distributed event-processing frameworks. The Apache Incubator is supporting development of Storm, S4, and Samza. Amazon Web Services recently announced the commercial availability of Kinesis. IBM offers InfoSphere Streams as a distributed, event-driven system. With streaming graph algorithms, the state of the algorithm can be too large to forward from node to node for each event in a distributed cluster. This work expands the horizons of distributed event-processing frameworks, incorporating complex graph analysis and leveraging large shared memory servers to create a more general execution model for stream computing in heterogeneous applications.

## II. System Design

In a real-time, streaming social network analytics application, to turn data into actionable knowledge, our objective is to increase throughput and decrease latency. The pipelined, feed-forward architecture in Figure 1 ingests raw social media data from a variety of services. Parsing data is usually embarrassingly parallel and can be accomplished with a scale-out cluster. Data objects are transformed into representations understood by the databases.

Multiple databases are used to answer different types of queries. Graph-based information is sent to a graph database where streaming social network analysis algorithms continuously update metrics of interest. Topic classification and sentiment analysis compute in real-time as the text content flows through the system. Raw data can be stored in a NoSQL database / document store such as MongoDB, or a traditional relational database such as MySQL or MS SQL Server for *post-hoc* analysis. Consistency between databases is not necessary because online algorithms do not access historical data. In a pipelined system, the throughput of the system is bound by the slowest stage. Using multiple types of data stores ensures that queries are handled by the database most capable of responding quickly.

Having observed many graph algorithms in a streaming data scenario, several core design principles emerge. First, all analytics need a static view of the portion of the graph currently being examined and the list of updates to be made. The alternative is to design algorithms to operate on a data structure that could be changing while they are reading. Designing for this scenario requires handling of complex corner cases that are unlikely to occur. In our framework, algorithms always receive a snapshot view of the entire graph before and after sets of updates.

A second guiding design principle is that analytics should be able to utilize the results of other analytics. While many research papers consider algorithms in isolation, it is good practice to be able to reuse results. For example, a sampling-based algorithm would need to consider the connected components of the graph to ensure that each component is represented according to the distribution of component sizes. Likewise, a community detection algorithm that is agnostic to scoring function could subscribe to another running algorithm to provide its scoring input.

Last, in a streaming context with no beginning or end, it is important that the server continue to run and algorithms can come and go during execution. To meet this requirement, algorithms must run in separate processes from the server, registering via a known protocol. Running algorithms in separate processes adds a layer of isolation. With appropriate watchdog timeouts, algorithms cannot stall the server from making forward progress on the incoming data stream.

In the next three sections, we will describe in detail how to design and implement a streaming data framework capable of processing hundreds of thousands to millions of events per second on commodity hardware. Our motivating application is social media analysis that combines real-time network analysis, text analysis, and historical queries in an interactive web-based visualization.

## III. Data Ingest

To produce data sets from dynamic sources, researchers often crawl data sources over a short window of time to create a static snapshot or capture the stream of changes that can be replayed at a later time. Offline, these snapshots and stream samples may be carefully manipulated into easily parsed formats in order to be ingested by the system. This data munging and ingest process usually requires a non-trivial amount of work from an expert and can take significantly longer than the actually running analytics. In order to truly be able to analyze streaming data, this process must be an automated and efficient part of the infrastructure.

In the workflow presented in Figure 2, the stream processes on the left-hand side handle the ingest task. It is the responsibility of a stream to produce batches containing some combination of edge updates, vertex updates, and metadata to send to the server. The data in the batch is obtained through either consuming an input data source or generating synthetic data within the stream process. Input data sources can include static or replayed sources such as files or databases, or dynamic sources such as streaming web endpoints. The stream process must understand how to infer the relationships in the data to produce edges and vertices. Although the raw data itself is not stored in STINGER, the stream could send the raw data through and include a reference in each edge and vertex update to indicate which data object created the update. This allows streaming algorithms to analyze content along with the relational data.

The provided system uses the concept of templates to enable flexibility and ease-of-use to provide generic JSON (JavaScript Object Notation) and CSV (Comma-Separated Values) parsing streams. Each assumes that its input will be a template read from a file and a stream of entire valid JSON objects or CSV rows, one per line, read from standard input. The template is expected to follow a format similar to the input stream with one or more valid JSON objects or CSV rows, but should contain reserved keywords in place of values to indicate where vertices, weights, timestamps, types, etc. should be found. The stream data is checked against each template and extra data is ignored. Data that results in at least one match will be added as metadata. This concept could easily be extended to a variety of formats including XML, log files, and even plain text.

In order for the system to process the data, it must be stored and transmitted in a common and serializable format. In the workflow presented in Figure 2, Google's Protocol Buffer [14] serialization library is used. Protocol Buffers were selected based on a balance of flexibility, compactness, language support, and speed of serialization and deserialization. In addition to the run-time library, a compiler is provided that uses message format definitions in *.proto* files to produce implementation code in the desired language (C++, Python, and Java supported by Google, additional language support provided by the community). The Batch message used by streams contains four variable-sized arrays: edge insertions, edge deletions, vertex updates, and metadata. In addition to being used by streams to send data to the server, three other message types, two of which contain a nested Batch message, are used by the server to communicate with the other process types.

Streams send batches to the server via TCP. The server multiplexes incoming connections and data from one or more streams accross a handful of threads. Incoming data is atomically enqueued to be processed by the system on a first-come, first-served basis. This design means that stream processing can easily be distributed to one or more separate machines and that content analysis could be performed as a pre-processing step, if needed.

## IV. Server and Algorithm Scheduling

The STINGER data structure and the streaming graph algorithms are designed and tuned for multicore, shared memory systems. Today it is possible to buy commercial off-the-shelf (COTS) servers with up to 2 TB of DRAM. Systems from SGI and Cray can be built with up to 64 TB of shared memory. Shared memory computing leverages the relatively high internal interconnect bandwidth and low latency inside the chassis. Communicating over an Infiniband or 10Gb Ethernet network generally incurs at least a factor of 10 decrease in bandwidth and increase in latency compared to the capability of the DRAM. For data-intensive algorithms, this gap can make distributed computation expensive. In high-performance

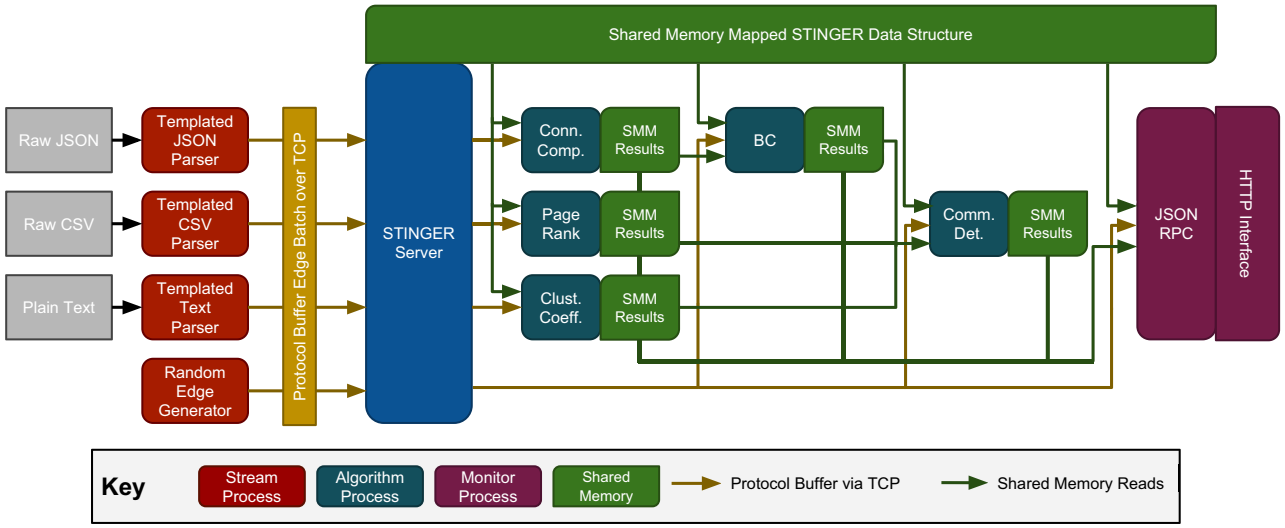



Fig. 2. Data flows from left to right. All stages aligned vertically execute on the data in parallel. All processes in direct contact with shared memory have write access to that memory. Execution is synchronized by the server. Streams, algorithms, and monitors can join or leave the workflow at any time.

contexts, the fine-grained, light-weight synchronization provided by atomic operations within a single memory space can be highly beneficial when cooperation between threads and processes is needed [15].

Leveraging shared memory simplifies algorithms. No data partitioning or sharding is necessary. Massive data sets in social media are notoriously difficult to partition in a balanced manner [16]. We will use the shared memory to isolate analytic processes from the server process, increasing resilience to failure and security. Using shared memory to share data among processes increases the efficiency of multicore as single-threaded analytics can be run simultaneously.

The STINGER server in Figure 2 is a multithreaded server process that is responsible for accepting incoming edges (in the form of Protocol Buffers), applying the updates to the graph, managing the in-memory data store, and registering and scheduling analytic algorithms.

Graph edges arising from social media are tagged with screen names or user handles in the form of unique identifier strings. To conserve space and simplify indexing, the STINGER server maintains a look-up table that converts unique identifier strings to unique integers and vice versa. The performance of this component of the system is critical as all external requests will use strings to identify vertices.

The mapper is implemented as a pre-allocated hash table that is double the number of vertices in size (must be a power of 2) to map from strings to vertex identifiers, a look-up table to map from vertex identifiers back to strings, and a pre-allocated stack to store string data. Although safe parallel deletion from the table is possible following semantics similar to insertion, created mappings are permanent at this time due to the use of stack-based allocation for string storage.

Strings are hashed by XORing each 64-bit word (zero padded), then applying a 64-bit mix function and masking the result. The hash function is optimized for speed over uniformity. Atomic in-place locking operations are used on the hash table when creating new entries following the same full-empty emulation used in [15]. This fine-grained locking approach allows for safe parallel insertion and guarantees that two threads creating the same string mapping will receive the same vertex identifier. Functions for looking up mappings in each direction are also provided.

The scalability of this design is shown in Figure 3 in which 100,000 words of 12 or more letters were randomly selected and inserted in parallel. The test is performed with a) an initially empty map and b) with a map with the set already inserted. The number of random insertions is tested at 1x, 2x, and 10x the size of the set. An equivalent mapping using a C++ STL map and vector protected with a lock is also tested for comparison. Results show that, given a data source with multiple relationships forming around individuals (i.e. edges are new, but the vertices involved are likely to have been seen), such as a social network, the data structure performance does scale with parallelism and drastically outperforms the simple lock.

### A. *Shared Memory Mapping*

Analytics or algorithm clients are compiled into standalone executables and linked with the STINGER dynamic library. Algorithms run in isolated processes on the same system as the STINGER server process. At the beginning of execution, a client algorithm registers with the server on a known port. It declares a unique name, known dependencies, and requests a pool of local storage (more on these later in the section).

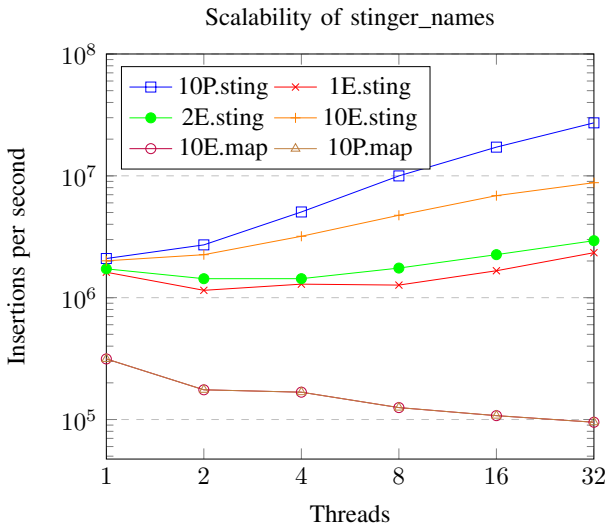



Fig. 3. The scaling performance of the stinger_names mapping compared with a C++ STL map and STL vector protected with a lock. 'E' indicates an empty mapping at start and 'P' a pre-filled map. 1,2, and 10 indicate 100k, 200k, or 1M insertions were randomly selected from a set of 100k words. Experiment conducted on a dual Intel Xeon E5-2670 with 64 GB main memory. Note: 10P.map and 10E.map are nearly indistinguishable.

The client receives from the server a shared memory mapping of the in-memory graph data structure that can be mapped private (to allow for local modification). The advantage of this shared mapping is that the client algorithm computes on a static version of the graph while the server process can update incoming edges in parallel.

The server process is responsible for scheduling client algorithms so as to avoid data races. After a client first registers and receives a private mapping of the data structure, it enters the initialization phase. In this phase, the algorithm computes its results from scratch for the current state of the data. After completing initialization, the algorithm is scheduled alongside other client analytics.

The processing loop consists of three phases. Phase one is the Pre-Update phase in which clients view the graph in a state before any updates are made. Not all algorithms require a pre-update phase, but some, such as clustering coefficients, will be comparing the state of the graph before and after updates are made.

Phase two of the update cycle is when the server applies the batch of new edges to the graph. During this phase, client algorithms must pause because the graph is in an inconsistent state. When updates are complete, the server notifies client algorithms to enter the Post-Update phase. In this phase, each client maps the new graph data structure and receives the batch of updates. Using this information, it is able to update or recompute its metrics. The finite state machine that regulates algorithm clients is depicted in Figure 4.

The STINGER client library is responsible for negotiating

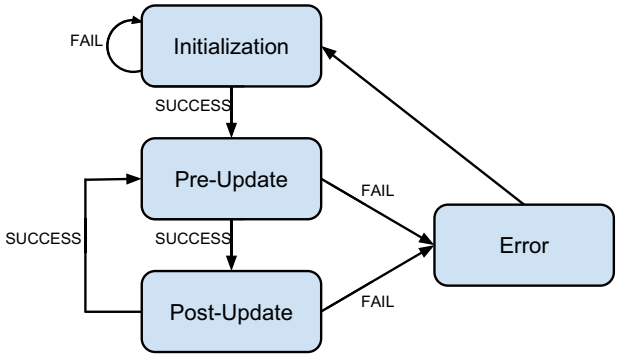



Fig. 4. Finite state machine for client algorithms

with the server and re-mapping the data structure for each new phase. The developer need only supply initialization, pre-update, and post-update functions that take a pointer to the data structure. The client library is available in C/C++ with a Python wrapper implemented using CTypes. The Python wrapper has proven useful in our work to quickly connect content to powerful analysis libraries such as scikit-learn and the Natural Language Toolkit which can in turn provide additional information to graph algorithms. The wrappers also enable faster prototyping of algorithms that are meaningfully connected into the workflow.

### *B. Pub/Sub Vertex Model*

The client algorithm connects to and registers with the STINGER server, it has the opportunity to request a pool of storage. This pool of storage is allocated by the server in shared memory and advertised to all other algorithms. This feature enables an algorithm to publish its result set so that it can be read by other clients.

The storage pool consists of one row per vertex in the graph. The client specifies the number of columns per row and the data type of each column. This specification is given in a description string passed from client to server. Supported data types are: `f` for single-precision floating point, `d` for double-precision floating point, `i` for 32-bit signed integer, `l` for 64-bit signed integer, and `b` for 8-bit binary.

In the description string, data types are listed first followed by a name for each column. For example, a client implementing PageRank might specify "`d pagerank`" using a single double-precision floating point value per vertex called "pagerank". A client algorithm implementing community detection might require an integer per vertex to store the community ID and a double per vertex to store the score. Its data description field would specify "`ld communityId score`".

The server allocates a contiguous chunk of memory according to the description string specifications and gives a handle to the client. The client maps this portion of memory read-write. It is the only process that will write to this space. Other clients can subscribe to this result set, re-mapping it privately with

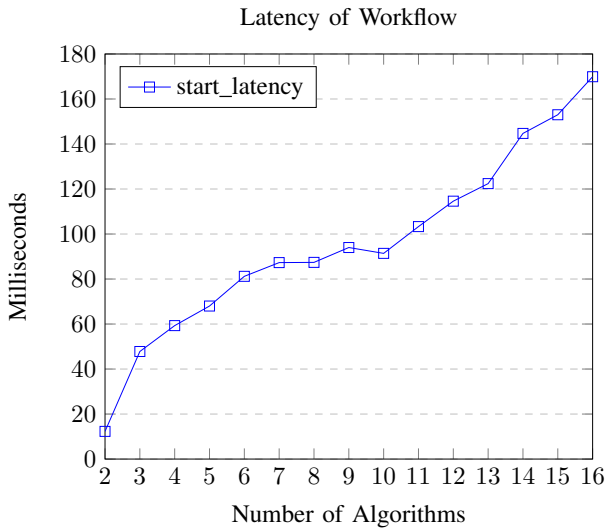



Fig. 5. This chart gives the average latency between the minimum and maximum start time for algorithms running in parallel in a single stage of the workflow as the number of algorithms is increased from 2 to 16.

each update phase. The server and client libraries coordinate re-mapping transparently.

### C. Roles & Dependencies

The STINGER execution model with pub/sub enables one algorithm to use the results of another algorithm's computation. An example use case might be a community detection algorithm that uses connected components information as well as the PageRank scores of each vertex as input to the computation. To fully implement this feature, the STINGER server coordinates algorithm dependencies and scheduling, without the need to double buffer result sets.

When a client registers with the server, it expresses its dependencies by giving the string identifiers of algorithms that it subscribes to. The server constructs a directed, acyclic graph (DAG) of algorithms and schedules their execution in a parallel, level-synchronous manner. Each algorithm re-maps the data structure and any other client result sets at the beginning of execution, ensuring that the data subscribed to is most current.

The overhead of running multiple single-threaded algorithms in parallel within the workflow is plotted in Figure 5. In the figure, the start latency, computed as the difference between the start of the pre-processing phase of the earliest algorithm and the same point for the latest algorithm, is measured as the number of processes is increased from 2 to 16. The figure demonstrates a small cost to adding algorithms, likely due to sequentially sending the start messages, which could be improved through parallelization or switching from TCP to UDP. On average, the latency between the end of the pre-processing phase of an algorithm and the beginning of the pre-processing phase of an algorithm dependent on it was approximately 29 milliseconds.

To avoid freezing the system, algorithms must respond before a watchdog timer runs out, typically about five seconds. If an algorithm cannot keep up with the forward progress of the system, it is placed back in initialization state and can re-join the computation after it has reset its internal state.

An example timeline in Figure 6 shows the interaction between client algorithms and the server.

Today, STINGER supports only required dependencies. However, optional dependencies may aid system flexibility. In our example of community detection using PageRank scores, if PageRank was running, the community detection algorithm could utilize the results through the pub/sub mechanism. However, if PageRank was not available, the community detection algorithm would compute the score itself or provide an alternate means of scoring.

The corollary to optional dependencies are role-based algorithms. Calculating optimal results of many graph queries is NP-Complete or worse. As a result, many heuristic approximations have been proposed. In a system like STINGER, several clients could run, each computing a different heuristic approximation. A role-based system would enable each to advertise that they are computing the same metric, but in a different way. Other clients subscribing to their results could utilize one, several, or all of the result sets as appropriate.

## V. External Interfaces

Each client algorithm can produce a very large, dense result vector during each update cycle. A client producing a single double-precision floating point value for each vertex in the graph consumes 8 GB, assuming one billion vertices in the graph. It is prohibitive to send 8 GB to an analyst workstation every minute. The memory bandwidth of the server is much higher than the network bandwidth between the server and the workstation.

The analyst usually does not need all of this data, but rather is looking for specific data points. As a result, we can minimize client bandwidth by performing as much of the operation on the server, where the data is already in memory, before sending the results to the client at the very last moment.

For this reason, the STINGER execution model employs a special class of algorithm client known as the "monitor" process. It is a standalone process that is scheduled like a client algorithm, but cannot publish data and is dependent on all other algorithms in the system. By virtue of this attribute, it is always scheduled in the last update slot and has access to all other algorithms' result stores. In Figure 6, the HTTP Endpoint is a monitor process that we will now describe.

Our primary monitor process is an HTTP endpoint that serves a RESTful interface to the algorithm result stores and graph data. Using the embedded Mongoose HTTP server [17], we accept JSON-RPC [18] POST requests. The JSON-RPC protocol is a simple remote procedure call that utilizes JSON

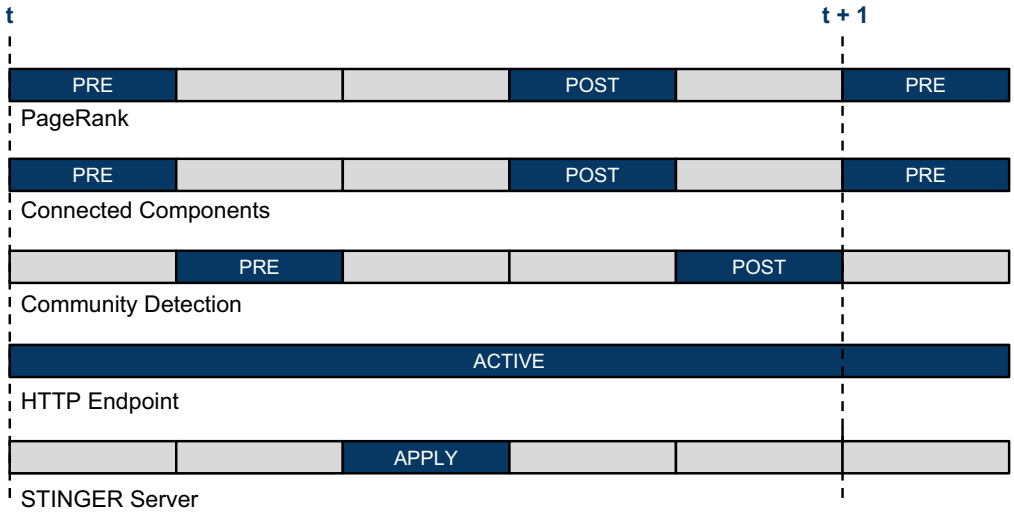
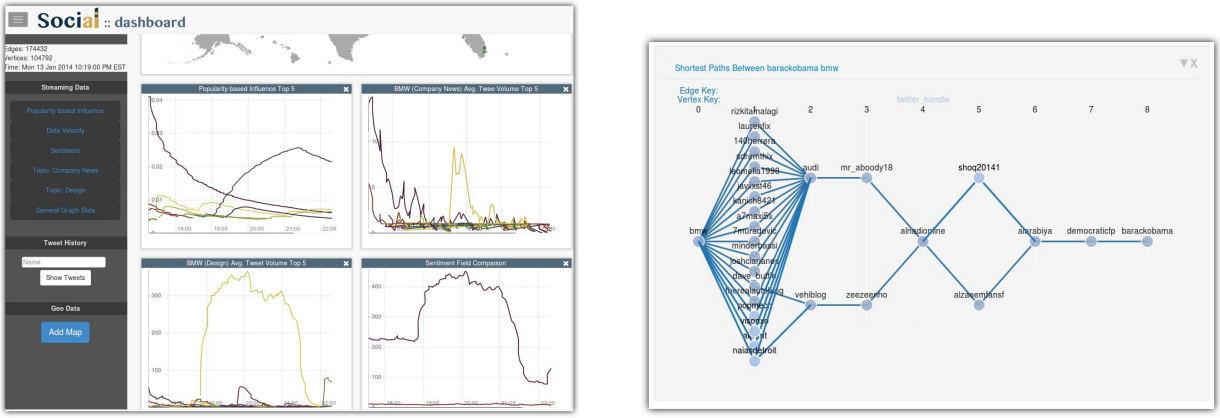



Fig. 6. This timeline shows the interaction between client algorithms, monitor processes, and the STINGER server as a batch of edge insertions and deletions are applied to the graph. In this example, the Community Detection algorithm is registered as dependent on PageRank and Connected Components algorithms.




(a) JavaScript GUI for sentiment and topic tracking

(b) JavaScript tool for subgraph connectivity visualization

Fig. 7. Examples of two analytics visualizations built on the STINGER streaming framework

formatting to encode a method and parameters. JSON is a convenient choice because it is ubiquitous in web-based, client-side computing. RapidJSON [19] is an open source, C++ library for parsing and formatting JSON strings, and was chosen for performance.

Basic RPC methods include querying what algorithms are running and the data description of each algorithm. More advanced methods perform queries on the client data stores. For example, we provide a method to return the top/bottom $n$ values from a data array. This method sorts all of the data in-memory and extracts only the values of interest. An optional parameter to all methods returns the vertex identifier strings with the result.

Sampling over a data array is achieved by an optional parameter that specifies the number of samples or the bucket size, and includes exponential sampling. Combining sampling with sorting effectively computes a histogram on the server, returning kilobytes instead of gigabytes.

A similar RPC method returns the results for a specific set of vertices. Another method performs a reduction over the data array, such as a sum, min, or max. Each time a query is made, the requester chooses to receive the current results immediately or wait for an update. If waiting for an update, the HTTP server will not respond to the request until the next batch of edges is processed.

In cases where a streaming algorithm does not exist or the results are input-dependent, such as a connectivity query, we extend our RESTful API by implementing sessions. For example, if the query is "What are the shortest paths between vertex $A$ and $B$?", we would register our query with the

server, which would create a session, compute the current shortest paths, and return the edge list to the requester with a session identifier. In subsequent requests, the requester need only reference the session identifier and receive a list of edge insertions and deletions since the last query that can be used to keep the requester in sync with the server. The visualization corresponding to such a query can be seen in Fig. 7(b).

We have also used monitor processes to periodically checkpoint the state of the graph to disk for an additional layer of resilience. Monitor processes can perform tasks such as checking the consistency of metadata within the data structure to identify errors. We provide a monitor process that periodically snapshots a client result store to a MongoDB instance.

## VI. Conclusion

As data volumes continue to grow, it will become impossible to store all raw data that is collected. At each time step, data will be computed and analyzed and only those elements perceived to be most important will be stored. This environment will create the need for new infrastructures and execution models for computation.

In this paper, we have described the STINGER framework for streaming data analytics. The system utilizes heterogeneous clusters to ingest data from multiple social media services, form in-memory graph representations of relational data, store raw data in traditional databases for historical search, and give analysts up-to-the-minute insight into a suite of streaming analytics. The system design is focused on performance and modularity, minimizing latency and increasing throughput.

The infrastructure that we have described is free and open source software. It is BSD licensed and available at `www.stingergraph.com`. In Figure 7, you will find screenshots of several web-based tools that have been implemented on the STINGER architecture. These tools combine complex graph analytics, topic classification, text sentiment analysis, and geolocation for real-time, streaming data analysis.

## References


[1] Twitter, "#Goal," April 2012, http://blog.uk.twitter.com/2012/04/goal.html.

[2] Facebook, "Key facts," May 2012, http://newsroom.fb.com/content/default.aspx?NewsAreaId=22.

[3] D. Ediger, K. Jiang, J. Riedy, and D. A. Bader, "Massive streaming data analytics: A case study with clustering coefficients," in *Workshop on Multithreaded Architectures and Applications (MTAAP)*, Atlanta, Georgia, Apr. 2010.

[4] D. Ediger, E. J. Riedy, D. A. Bader, and H. Meyerhenke, "Tracking structure of streaming social networks," in *5th Workshop on Multithreaded Architectures and Applications (MTAAP)*, May 2011.

[5] J. Riedy and D. Bader, "Multithreaded community monitoring for massive streaming graph data," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 1646–1655.

[6] R. McColl, O. Green, and D. Bader, "A new parallel algorithm for connected components in dynamic graphs," in *Conference on High Performance Computing (HiPC), 2013 IEEE 20th International*, December 2013.

[7] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ser. ICDM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 229–238.

[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146.

[9] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 65:1–65:12.

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 17–30.

[11] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[12] D. A. Bader, J. Berry, A. Amos-Binks, D. Chavarría-Miranda, C. Hastings, K. Madduri, and S. C. Poulos, "STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation," Georgia Institute of Technology, Tech. Rep., 2009.

[13] R. McColl, D. Ediger, J. Poovey, C. D., and D. Bader, "A performance evaluation of open source graph databases," in *Parallel Programming for Analytics Applications (PPAA), 1st Workshop on*, Sept 2, pp. 1–5.

[14] Google, "Protocol buffers," http://code.google.com/apis/protocolbuffers/.

[15] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "STINGER: High performance data structure for streaming graphs," in *The IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, Sep. 2012, best paper award.

[16] K. Lang, "Finding good nearly balanced cuts in power law graphs," Yahoo! Research, Tech. Rep., 2004.

[17] S. Lyubka, "Mongoose," https://code.google.com/p/mongoose/.

[18] JSON-RPC Working Group, "JSON-RPC 2.0 specification," http://www.jsonrpc.org/specification.

[19] "RapidJSON," https://code.google.com/p/rapidjson/.