A Reference Web Architecture and Patterns for Real-time Visual Analytics on Large Streaming Data

Eser Kandogan[a], Danny Soroker[b], Steven Rohall[a], Peter Bak[a], Frank van Ham[a], Jie Lu[a], Harold-Jeffrey Ship[a], Chun-Fu Wang[b], Jennifer Lai[a]
[a]IBM 650 Harry Rd. San Jose, CA, USA 95120; [b]University of California, Davis, 2063 Kemper Hall, 1 Shields Avenue, Davis CA 95616

ABSTRACT

Monitoring and analysis of streaming data, such as social media, sensors, and news feeds, has become increasingly important for business and government. Volume and velocity of incoming data are key challenges. To effectively support monitoring and analysis, statistical and visual analytics techniques need to be seamlessly integrated; analytic techniques for a variety of data types (e.g. text, numerical) and scope (e.g. incremental, rolling-window, and global) must be properly accommodated; interaction and coordination among several visualizations must be supported in an efficient manner; and the system should support the use of different analytics techniques in a pluggable and collaborative manner. Especially in web based environments, these requirements pose restrictions on the basic architecture for such systems. In this paper we report on our experience of building a reference web architecture for real-time visual analytics of streaming data, identify and discuss architectural patterns that address these challenges, and report on applying the reference architecture for real-time Twitter monitoring.

Keywords: Streaming data, visual analytics architecture, design patterns, web-scale.

1. INTRODUCTION

Visual analytics involves a combination of statistical and visual techniques to provide users with insight into data. With the proliferation of data sources and the explosive growth in data volume, variety, and velocity, we increasingly see a separation of concerns for accessing, processing, and visualizing data for analysis. While in the past, the data under analysis used to be tightly under the control of the visual analytics tool this is no longer the case for most data sources. For example, for Twitter data several entities are involved in storage, integration, and distribution through web-based APIs. There are commercial entities that originate the data (e.g. Twitter, Facebook), governmental-academic entities that curate the data (e.g. Sloan Digital Sky Survey or data.gov), 3rd party commercial entities that provide additional value over original data by offering data refinement, extraction, and integration with data from other sources, and yet others that provide analytic solutions to users. While this separation of concerns has allowed users to collect, store, and share increasingly large amounts of dynamic information, it has reduced the ease with which users can access, process, and visualize this information.

Just as data providers offer the same data sources to a potentially large user base, providers of analytic applications should be able to provide their services to multiple users at the same time. Our expectation is that the analytic tools of the future will operate in much the same way as the data providers of today. They will offer an online service that is accessible to large amounts of users simultaneously and will need to serve different requests for analytics. Moving from monolithic applications that have tight control over their data source and serve a single user, to online open analytic platforms, which offer users various data, analytics, and visualizations from several providers, requires changes to the core architecture.

There are two challenges involved in accomplishing this change. First, we need to create a processing architecture that can comply with the limitations of streaming data providers, and still provide enough data enrichment to be useful. Interactivity is a major bottleneck here, as data providers are not always suited for highly interactive queries, and analytic applications do not have direct access to all of the detailed data needed for deep interactivity. Second, architectures need to be incrementally and elastically scalable, such that one can easily add computing resources without having to change the architecture to provide a good user experience.

In this paper, our contributions include (1) identification of challenges and requirements of online open visual analytics platforms for real-time data, based on our review of related work on visual analytics applications and architectures, (2) a reference architecture and a set of patterns, when used together, satisfying identified requirements, and (3) discussion of our experiences building an online visual analysis system for Twitter data, examining advantages and disadvantages of proposed patterns.

We begin by first discussing related work on analyzing streaming data and streaming data architectures. Then, we describe details of our proposed reference architecture and patterns that are able to deal with many challenges of real-time analytics. We then describe how we applied these patterns in developing a prototype system that supports interactive visual analysis of Twitter streams. We conclude by discussing a number of observations and recommendations that will be useful for others implementing similar systems.

2. RELATED WORK

We consider related work from two different perspectives: 1) research focused directly on identifying requirements for streaming data processing systems, especially for visual analytics, 2) work on visual analytics applications, architectures, and architectural patterns, particularly for streaming data.

2.1 Requirements

Babcock, et al. [1] recognized early on that current DBMS architectures were not suitable for a range of new applications, where data arrives in multiple, continuous, rapid, and time-varying streams, because of their inability to handle continuous queries. They identified approximation and adaptivity as key ingredients for new data processing architectures. Stonebraker, et al. [2] later defined a number of requirements for real-time stream processing for data-intensive applications, which included rules such as keeping the data moving to avoid costly storage, handling stream imperfections such as missing and delayed data, integrating stored and streamed data, and processing and responding instantaneously.

In the visual analytics community, Keim, et al. [3] identified data streaming as one of the major challenges for visual analytics, citing challenges such as coping with very large amounts of data arriving in bursts or continuously, tackling difficulties of indexing and aggregation in real time, identifying trends and detecting unexpected behavior when the dataset is changing dynamically. Thomas, et al [4] identified the three most significant challenges as providing appropriate situational awareness, showing changes, and fusing different data sources. In [5] Rohrdantz, et al. discuss tasks and challenges in the real-time visualization of streaming text data, suggesting tasks such as monitoring, change detection, trend identification, event tracking, and historical exploration, among others. Mannsman, et al. [6], particularly considering the role of the user, coined the term dynamic visual analytics pipeline as the process of integrating knowledge discovery and interactive exploration, as people interact with dynamic models and visualizations.

While most of these requirements also apply to online open visual analytics platforms, we believe there are further requirements which need to be taken into account for open platforms: 1) Provider Requirements: such as compliance with data provider limitations; and ability to mix and match computational analytics and visualizations from different providers, 2) User Requirements: scaling to many users; supporting effective collaboration among users; and supporting diverse analytic tasks while delivering a good real-time user experience overall, and 3) Data Requirements: ability to integrate streaming data with other types of data, coming from several sources; and scaling to volume, velocity, and variety of data.

2.2 Applications and architectures

In the visual analytics field there has been several applications of streaming data, such as stock markets, systems management [7][8], software development and monitoring [9], network monitoring [10] [11], telecommunications [12], sensor networks, social media [13][14], emergency management and response, news [15], email [16][17], and blogs [18][19], each presenting their own domain-specific requirements. For example, Dörk, et al. [14] introduced the concept of a visual backchannel to support timely exchange of opinions during events, such as political speeches, sports competitions, and natural disasters, and identified problems that make it easy to lose focus and miss what just happened. They also identified design goals such as ability to summarize the conversation, integrate now and recent, extend presence of the present, and support topical and social exploration.

While most visual analytics applications are custom built, with their own application-specific implementations of data structures, storage, processing, and visualization techniques, there are still common components or patterns that are broadly applicable. Fekete [20] reviews some of these architectural models of visualization, data management, analysis, dissemination, and communication components and outlines the inherent challenges.

Early work focused on extending existing visual analytics systems to real-time data. In [21] Hetzler, concerned about how to extend IN-SPIRE [22] (originally designed to work on static document collections) created a completely new dataset at each increment by removing aged-off records, adding newly arrived records, and then processing the result-set of records as a static dataset. As in [21], the requirement to expire documents affects all data structures that analytic and visualization components maintain. In other early work, LiveRAC [7] used a streaming pipeline model [12] to process time-series data from systems.

More recently, Rohrdantz [5] recognized several challenges, including dealing with constantly growing amounts of data, showing new data in the context of older data, handling fluctuations in data arrival intervals. They identified on-the-fly processing as a critical challenge--data cannot be stored indefinitely, and needs to be buffered only temporarily--and choosing the buffer size is an important decision. Dubinko, et al [23] report different results of the analysis based on the size of the buffer and argued that choosing the right buffer size is difficult and subject to the task. Buffer size can be user controlled as in TextPool [24], which uses a temporal pooling approach to collect stream content that arrived during a user-controlled period of time and then visualizes it, thereby showing only the most recent portion of the stream. Temporal pooling approach was first proposed by Ishizaki [17], whose perForm system allows users to visualize stream data using an agent-based infrastructure. There are also automated approaches such as in [25] where Wong et al. suggest a volume/rate dependent analysis, trading between accuracy and performance. They present MDS-based visualizations for a moving window of data, but do not deal with interactive exploration. They argue that unpredictable and unbounded characteristics of streaming data can potentially overwhelm algorithms that require full re-computation. They offer an adaptive technique based on the stratification of incoming data into several streams that intelligently reduces data intake size and processing time. Alsakran [26] also buffers documents during peak times, and processes them in idling periods but uses two time frames with different levels of detail for in-depth and context analysis.

Event Visualizer [27], uses a generic messaging service (Java Message Service – JMS) among modules such as event analyzers and visualizers. In essence, an event service module listens on given data streams and processes them to create event objects, which are forwarded to the message broker's incoming queue. To keep up with stream flow rates they make use of distributed and multi-threaded analyzers, to which events are equally distributed to ensure load balancing.

Based on our review of literature, clearly there are gaps between requirements and architectures/patterns, particularly when considering online open visual analytics platforms. None of the architectures considered an open platform, as such requirements from the various players, from data providers to analytics providers, and matters of compliance and interoperability. While some of the architectures considered data requirements, only few considered user requirements, particularly collaboration.

3. REFERENCE ARCHITECTURE AND PATTERNS

In this section we outline a number of architectural patterns, and describe a reference architecture, in which all of these patterns come together providing a complete system based on our experience of building a real-time visual analytics platform for streaming data. Patterns describe a reusable template by providing a context, problem, solution, strengths and weaknesses, and related patterns. Architectural patterns have a specific focus on the organizations of the components of a system, particular responsibilities of individual components, relationship among components, connections, interactions, and communication mechanisms, and behavioral aspects of the system as a whole [28][29][30].

3.1 Streaming engine

The streaming engine is at the highest level in the reference architecture. The engine is responsible for connecting to streaming data sources, creating data processors, which perform analytic and visual computations, and managing the distribution of streaming data to processors. Streaming engine has several web-based APIs for clients to connect to and interact with visualizers, and perform engine and processor administration tasks (Fig. 1).

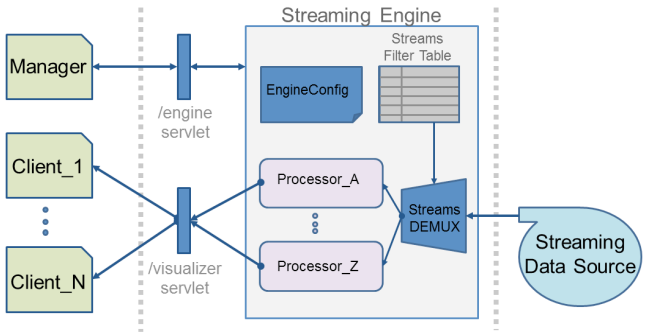

Figure 1. High-level view of proposed reference architecture

3.1.1 Architectural pattern: Federated Consumers

A core task of a streaming engine is to channel data from providers to processors that consume data to create interactive visualizations. As such, streaming engines should be cognizant of data provider rules and capabilities, particularly related to availability, performance, and legal constraints. Providers might put caps on the volume of data, number of connections and number of queries, and require consumers to enforce data storage and deletion restrictions.

In our reference architecture, we propose the Federated Consumers pattern to address provider constraints. In this pattern a central component exists that mediates all interaction between a single provider and multiple consumers. Queries from individual consumers are combined into a single aggregate query, and matching data from the provider in response to the aggregate query is distributed to consumers based on their respective queries. This central component makes sure all provider constraints are satisfied. The strength of this pattern is that external compliance is the responsibility of a single component in the system and not a concern for multitudes of data consumers. The weakness of this pattern is the potential for a performance bottleneck. Care must be given to the specifics of the implementation such that it is very lean, and keeps the data moving at a rate demanded by both the data provider and consumers. Additionally, the implementation may replicate the central component to scale better. A related pattern is Mediator, which abstracts all inter-object interaction details into a separate mediating object with knowledge about the interacting group of objects and services they provide. In Mediator, objects are heterogeneous and interact with each other, whereas in Federated Consumer there is no interaction between consumers, but only between several consumers and the provider.

3.1.1.1 Example implementation: Streams DEMUX

The first component in our reference architecture through which data from the real-time streaming data source comes in is the streams demultiplexer (Streams DEMUX) component, which implements the Federated Consumers pattern. It aggregates queries from all processors, and sends it to the data source, and distributes incoming data to a processor if that data matches the processors query.

Streams DEMUX maintains a table, which has one entry per processor defining the query and filter conditions for that processor. Using this table it combines individual queries into an aggregate query, in disjunctive normal form. For example, a processor might define a stream with the query "A or B", another processor might define "B or (C and D) ", and another as "A and D", where A, B, C, D might be keywords, tags, user ids, or some other filtering condition such as location. The computed aggregate query would then be "A or B or (C and D)". Whenever new streaming data is received, it is distributed to multiple processors according to the query of each stream/processor. The aggregate query is recalculated whenever a new stream is created, deleted, or updated from an engine at runtime.

In this architecture only a single connection is maintained per streaming data source, over which users can define many processors that query the source data with different filtering criteria. A major benefit of this is compliance with streaming data source requirements, which often limit number of connections by IP and client credentials. For example, the Twitter public streaming API only allows a single user connection per IP address. By aggregating all queries into a single query, the system represents itself as one stream, one connection to the data source. By splitting the aggregate stream into multiples and distributing them accordingly to multiple processors, the system represents itself to the clients as multiple streams, as expected. One potential problem is that as the number of streams multiplexed increases, it gets easier to reach the total max rate on the provider side, depending on the query selectivity of each stream.

3.2 Stream processing

In processing of streams to perform analytics and visual computations there are three questions that impact the architecture: (1) How is computation performed and structured among components of the system? (2) What is the unit of computation and associated data? (3) How is data stored, accessed, and shared? Below, we will first provide a set of architectural patterns and later implementation examples that puts them together.

3.2.1 Architectural pattern: Queue of Observers

Visual analytics requires several specialized components that process different aspects of the data and compute output, which may in turn be used as input to another component. As such, there is a high degree of dependency between components. Designing how computations are performed and structured among several components of data is critical for achieving a good performance and user experience. Key challenges of the design are to maximize utility such that a computation is not repeated and to maximize throughput while handling components with different performance characteristics. To address these challenges we propose the Queue of Observers pattern.

Queue of Observers is similar to the basic Observer pattern in that a subject component maintains a list of observers (dependents) and notifies them automatically when its state changes. When a notification is received, all observers perform computation to update their internal states. The additional factor here is that there is a queue of observers, defining a workflow of analytic and visual computations. Data is transformed through each of the components producing new data or transforming old data. Like the Observer pattern, this pattern is useful when there is a one-to-many dependency between components. Defining work as a queue of observers is further suitable for complex computations requiring several components to build on each other's computations. One advantage is that computation is easily reusable, and performance variations among components are accommodated through the use of queues. A potential disadvantage is lack of resiliency: if a component fails, dependent components in the workflow will fail. One remedy for this problem though is replicating components such that computation is performed whenever any one of the subject component notifies.

3.2.1.1 Example implementation: Streaming Processor

In our reference architecture, streaming processors are dedicated to each data stream, as defined by a query. A streaming processor is responsible for building and executing a workflow of analyzer and visualizer workers, implementing the Queue of Observers pattern. This workflow is essentially a queuing system, where each worker has a separate queue and workers are connected to each other in a way that allows them to observe and notify each other. Each of the workers receives work from observed workers and performs some sort of computation, e.g. transform data into some visualization, do text analytics on some text field, etc. (Fig. 2.)

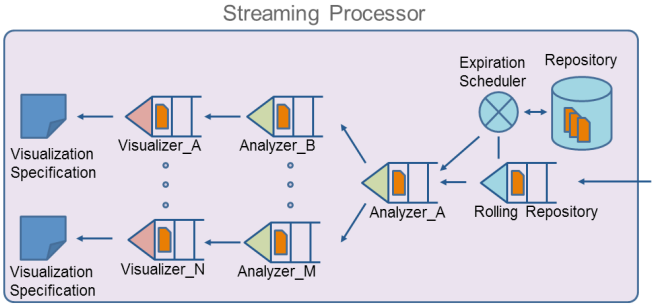

Figure 2. Stream Processor and Data Processing Pipeline

3.2.2 Architectural pattern: Sliding-Window Repository

Visual analytics may require access to data beyond what is currently streaming. In such cases, past data from the provider needs to be temporarily accessed such that it can be used by components retroactively. Legal constraints of the provider, which may limit the persistent storage of data, would be additional constraints for such repositories. To address these challenges we used a Sliding-Window Repository pattern in our reference architecture, as in [25].

In this pattern, data is only transiently stored in structures that allow querying and expiration of data based on defined criteria (user-defined or automatic). Expiration should trigger notification to components that consumed the expired data so that they can update their state. Data in this pattern is essentially on a sliding-window, like a FIFO queue, where data

comes in, slides down the window, and eventually expires based on expiration criteria. The advantage of Sliding-Window Repository is that it is a central repository that can be queried by components, and provider-imposed constraints can be implemented in a single component providing the right level of concurrency. The disadvantage of this pattern is that it may limit concurrent write/update access from many components.

3.2.2.1 Example implementation: Rolling Repository

In each processor, there is a sliding-window repository that stores the records temporarily, based on an expiration criterion set forth when the processor was defined (Fig. 3). The expiration criterion can be changed dynamically at run time and existing records are automatically checked for expiration based on the newly set criterion. Expiration criteria can either be time based, for example last 5 minutes, or last hour; or count based, for example last 100 or 1000 records. Expiration criteria can also be dynamically changed based on an algorithm that trades off performance and quality.

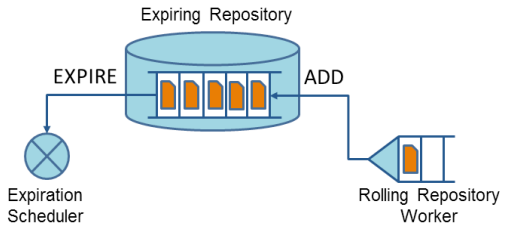

Figure 3. New data are added to the rolling repository as they arrive and data is removed based on an expiration criteria.

For time-based criteria, a timer is used to check time of records on a frequency that is about 10 times more than the set criterion, for example every 30 seconds in the case of 5-minute expiration. At every counter cycle, repository records are processed from oldest to newest, expiring a record if its timestamp is older than current time minus the specified expiration time, and stopping when the condition is not met, to be re-checked in the next cycle. In the count-based criterion nothing is done until the max count is reached, at which point each new record causes the oldest record to expire. As such, no periodic checks are needed in this case.

We have also implemented a drop policy that works with a maximum data rate. If the data rate goes above the limit, data will be dropped even before entering the repository (in an alternating fashion), until it goes below a percentage of the max limit. This allows some flexibility when the data rate fluctuates over time.

Data in the repository can also be queried by components whose computation requires global analysis. One such case is topic analysis, where the analyzer can query over the last N records to compute emerging topics over a period of time.

3.2.3 Architectural pattern: Moving Blackboard

While Queue of Observers provides sufficient flexibility in how a variety of components can be connected to each other, we need to match that flexibility in how computation and associated data are shared among components. Key issues here are flexibility in naming interfaces and data types. Rather than having predefined method names as in the Accessor pattern, we propose a Moving Blackboard pattern that moves the blackboard from one component to the other.

In the Blackboard pattern, there is a repository of "global variables" which can be accessed and modified by separate autonomous processes, each performing some part of the computation and updating variables. In the Moving Blackboard, like in the original Blackboard pattern, the current state of the solution is stored in the blackboard, in this case as a set of <key, value> pairs. Unlike the original pattern, the blackboard is not a central repository that captures all state of computation, but rather captures the state of processing per datum, and it is moved from component to component. Basically, each component receives a blackboard, which contains state of the computations up until now from components preceding it, performs some computations, writes the result back to the blackboard, and passes it on and starts waiting for the next one. As such at any point there are as many blackboards throughout the system as there are work items to perform computation on.

The advantage of the Moving Blackboard pattern is that components can independently work on a number of blackboards representing all the state that a component needs at that point in time. As such, if any component in the workflow requires more computational resources, performance can be improved by simply replicating them. Another

advantage is flexibility in naming input and output data. Any component can add a new entry to the blackboard by providing a key or label (in text) and associating it with data of any type. A disadvantage of Moving Blackboard may be memory utilization, as all state is replicated, but that can be relieved by putting references (instead of deep copying, if applicable) and handling concurrency on the object representing complex data. Compared to the Accessor pattern, which defines specific methods for each attribute separately, the proposed pattern provides more flexibility at the expense of potential type errors.

3.2.3.1 Example implementation: Work

In our reference implementation a work record implements the Moving Blackboard pattern. Essentially, a work record contains a set of key-value pairs along with an action, such as ADD, DELETE, etc. (Fig. 4). Each worker essentially examines the data in the work record and updates existing data or adds new data based on its computation.

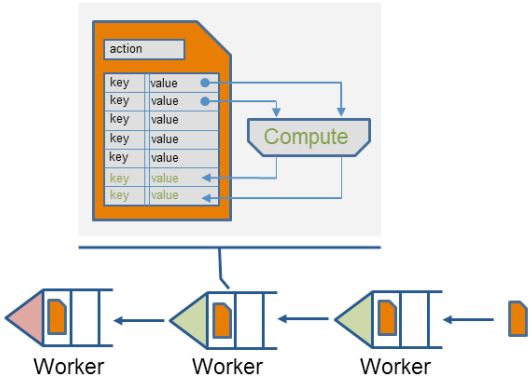

Figure 4. A pipeline of workers each adding new key-value pairs to the work record, or updating existing pairs based on computation performed on data in the record.

In the Streams DEMUX when new data is received, before passing it onto processors, a work record is created and attributes from original data are extracted and put into the work record. When distributing a work record to processors, each matching processor gets its own copy of the work record.

Work can be performed either on individual records or in batch mode, applying an action on a set of records, each containing a set of key-value pairs, representing a set of streaming data. The reference architecture has the ability to work with each individual data item as it comes in, or work in batch mode, where data is collated and processed together. Custom actions can be defined dependent on the nature of the streaming data source and also to meet data provider limitations.

Likewise, batch work actions can be dependent upon the streaming data source, but at a minimum they include BATCH_ADD and BATCH_REMOVE. These actions can obviously improve the performance of the overall processing, as computation can be performed on a set of data, rather than incrementally. This is especially useful for computation that has a large constant initialization and post-processing step that doesn't depend directly on the number of data elements. Batch work actions are also useful when a certain action is defined that requires re-processing of all or portion of streaming data in the repository at once. For example, a FILTER action based on some user interaction that needs to reset the state of each worker and re-compute with the set of records that fit the filtering condition in the sliding-window repository.

3.3 Analytics and visualization workers

Workers are the main components of the architecture that perform any sort of computation. In our reference architecture, they are linked to each other through an observer interface, as defined in the workflow by the stream processor. When new work is observed, first, the work object is added to its queue. A worker, which runs on its own thread, fetches an item from the top of the queue and performs computation based on the action defined in the work record, on data that is in the work record. After the computation is performed it puts the result back to the work record, before passing on the work to its observers (Fig 3.)

Actions on work items include ADD, REMOVE, BATCH_ADD, BATCH_REMOVE, and, optionally RESET, PAUSE, and RESUME. ADD is used to report that a new data item came in and needs to be processed, and each worker typically performs some computation and updates its own state. REMOVE is used to report that a past data item has either expired or been pulled out of the queue for some reason (sometimes providers correct themselves and remove data out of the stream retrospectively). Here workers often undo their ADD computation and reflect it on their own state. Some workers do nothing in the case of REMOVE, when their sole purpose is data transformation, such as computing the longitude and latitude from a text description of the location. In this case there are no side effects on the workers internal state. RESET is an action designated to clear any internal state of the workers. PAUSE and RESUME temporarily halt and restart the flow of streaming data.

Additional actions may be defined based on the data source and interactions designed. For example, FILTER sets an active filter on the incoming streaming data, besides filtering an existing data item already in the rolling repository. In some cases an action can be transformed before passing it on to listeners. For example, when a rolling repository (itself a worker) receives a FILTER work item, it first selects existing data in the repository based on the filter condition and sends RESET and BATCH_ADD work items to its listeners (with the selected data already in the repository) rather than the FILTER work item. Only then does it update its own state to actively select future streaming data based on the filter condition.

There are two main types of workers in the reference architecture: analyzer workers and visualizer workers described more fully in the next sections.

3.3.1 Analyzer workers: data analytics

Analyzer workers typically perform some sort of analysis on the incoming data with the goal that the computed analysis would be used in some visualization or in another analysis down the line. Examples of analyzer workers are part-of-speech tagging, geo location identification, histogram analysis, and topic analysis.

The nature of these analyzers may require access to different amounts of data. For example, part-of-speech tagging on a textual attribute of the streaming data requires only one unit of data, whereas topic analysis requires much more data to operate as well as having to keep some extracted attributes globally, even beyond the sliding-window of data in the repository.

We have encountered four types of analyzers depending on the volume of data and style of processing:

- *Local Analytics*: the analyzer is stateless and computation is performed on one or more units of streaming data.
- *Incremental Analytics*: the analyzer has internal state that is carried over from one work item to another, yet computation is performed on only one unit of streaming data, updating internal state incrementally.
- *Sliding Window Analytics*: the analyzer has internal state, and computation is performed locally on a rolling window of data. Analytics are performed on a set frequency, at regular intervals (based on time, volume of data, or some other criteria)
- *Global Analytics*: the analyzer has state that goes beyond data in the repository, and computation is performed globally, i.e. on all streaming data or, preferably, on its extracted attributes.

Local and incremental analytics are supported in a straightforward manner: an analyzer receives a new streaming data item, performs some computation and puts the result back to the work record, in the latter case also updating its internal state. In the latter case it is important that some form of that internal state is reflected back in the data put in the record. This way, other workers down the line work with correct state of computation, i.e. state at the time the work is completed and passed on. This is important since it might take a while for others to take their turn and during that time, new data might alter the state of computation.

Supporting sliding window analyzers is also simple, using BATCH_ADD. The analyzer worker performs its computation on the set of data passed down in the work record. Additionally, querying of the repository is also supported from within the workers, so analyzer workers can query for additional information based on their needs (e.g. different aggregations, selections, etc.)

Global analyzers are also supported but with some additional work. Particularly, the analyzer needs to store results of its global computation in its own internal state and maintain it over time. One challenge here is scaling for large volumes where attention must be paid to the storage characteristics of the extracted information.

3.3.2 Visualizer workers: data visualization

Visualizer workers are typically (but not necessarily) leaf nodes in the workflow. They rarely put any data back to the work record but rather produce a visualization specification that is used when rendering the visualization on the client-side. The latest rendered visualization is kept as a complete specification in the worker so requesting clients can be served the same visualization state without requiring re-computation. Clients, when receiving the visualization state simply render the transformed data according to the specification.

Most visualizer workers keep an internal current state, while others query the repository when re-rendering their visualization. Like analyzers, we have four types of visualizer workers depending on the extent and scope of the information rendered:

- *Local Visualization*: Each data item is shown independently. Rendering is updated on each new event, or at some frequency.
- *Incremental Visualization*: the visualizer has internal state, and visual representation of a data item can be calculated in an incremental fashion, affecting only a small portion of the overall visualization. Rendering is again updated on each new event, or at some frequency.
- *Sliding Window Visualization*: Rendering of the visualization requires access to a set of data items, and computation to update representation occurs on a set frequency.
- *Global Visualization*: Visualization spans more data than exists in the repository.

One design decision is how much computation to perform in the visualizer worker vs. splitting up the work into one or more analyzer workers. The choice depends on the reusability of the computation.

3.4 Interaction support

Interaction support can be facilitated by defining new actions that can be pushed from the client to the backend and through the various analyzer and visualization workers in the workflow. In our reference implementation, the visualization specification also specifies interaction as metadata. The client then observes defined interaction events and matching actions (as specified in the specification). When such an event occurs, the client makes a backend call with the actions and the underlying data associated with the event. For example, the specification might associate a "click" event with the FILTER action. When the user clicks on a specific object in the visualization (representing either individual or aggregate data), underlying data for that object is identified and along with the FILTER action sent to the backend. Underlying data for the action can be either actual data in the stream, or a parameter for the action such as a range of time, location, keyword, or topic.

When such an action event is received from the client, the backend handles that work item similarly to other work types. The rolling repository first receives the work item along with its parameters, performs work associated with the action, and passes it on. For example, in response to the FILTER action, the repository sets a filter condition for future streaming data, filters existing data in the repository, and sends a RESET and BATCH_ADD work down the line, with the set of data matching the filter condition. All analyzers and visualizers re-compute their state, and a set of new visualizations are produced reflecting the filter action, thus the various visualizations remain coordinated, where an action in one of them may result in re-rendering them all.

3.5 Collaboration support

Considering two high-level analytics tasks, monitoring and exploration, the reference architecture supports both tasks collaboratively. Multiple clients can connect to the same data processor and receive the same visualizations from the server, and thus can collaboratively monitor the real-time stream. If a user identifies a particular event to explore further they can switch temporarily to a dedicated data processor, which supports full interaction with the stream, allowing her to drill-down (e.g. by time, location, etc.) on any aspect of the data. When such a session is initiated, the public data processor/stream is replicated and a dedicated processor can be created on the fly, copying all content from the repository. As a result of this examination, the user might find it useful to share her dedicated processor/stream with others and other users can effectively join the interactive session where any one of them can interact directly with the visualizations, and create backend work actions. Additionally, the repository supports recording of stream data, thus enabling post-mortem collaborative exploration.

3.6 Plugin support: data, visualization, analytics

The proposed reference architecture is very flexible in that the workers can be considered as loosely coupled components that can be connected and reconnected on the fly. For example, a new data stream can be added to the workflow as a producer worker and push data from another stream, effectively allowing the analyzers and visualizers to fuse data from multiple sources. Likewise, additional analytic workers can be inserted to either improve performance or to examine the quality of different analytics algorithms on the flow. Last, but not least, in a similar fashion new visualizations can be added to the workflow, and feed upon streaming data as soon as they are connected. All of these are possible particularly due to the Queue of Observers pattern, which allows workers to be flexibly added and removed on the fly, and the Moving Blackboard pattern, which carries state from one worker to another, while at the same time providing an open and flexible data sharing capability.

4. APPLICATION: TWITTERVIZ

Social media data sources such as Twitter are increasingly vital for companies and governments to listen to public opinion about issues and events, to identify new markets for products and services, to improve customers' brand perception, and to address issues before they turn negative on a large scale. Opinions can quickly reach millions of potential customers and impact an institution before it is aware of the problem. Monitoring social media in real-time, and making timely sense of this fire hose of data is important yet it is a daunting task due to the data volume, speed, and the unstructured nature of the data. These characteristics make it an ideal application to examine the viability of our reference architecture for real-time visual analytics.

We created TwitterViz, a flexible suite of visualizations that monitor the data stream coming from Twitter. The Twitter real-time data stream is processed, augmented and visualized online, providing a window into the conversations on Twitter around a particular subject. A combination of visualizations we have created provides a multifaceted view of the data, including rates, sentiments, locations, common words, topics and phrases. Figure 5 shows a screen-shot of TwitterViz around the precise time the new pope was elected, and provides immediate insights into the public conversation at that time.

In this section we discuss these visualizations and their requirements and implications, as well as that of Twitter as a streaming data provider and of this particular use case on our reference architecture.

4.1 Twitter streaming API

The Twitter Streaming API, like most other publicly available streaming sources, has certain restrictions to prevent overloading of their platform. In Twitter's case, these restrictions apply to volume/rate, number of client connections, and query size.

Essentially, Twitter has a cap on the maximum volume a client can receive. While the max volume is fairly high, the rate is also adjusted according to the client's capacity for consuming status messages (tweets). As such, this requires careful design of how to (temporarily) store incoming data, and process them down the line. In our reference architecture, a new incoming tweet is first stored in the queue of DEMUX, before any processing starts, thus providing some cushion for any processing delays. Then, a tweet is transformed into a work object and passed onto respective rolling repositories where they are temporarily stored based on the expiration criteria. There, they can either be sent down the pipeline of queues of various workers (analytic/visualization) immediately as a single unit of work, or collated to create a batch work, where a number of tweets are processed together as a group. This provides a good balance between real-time perception, where as little as a single tweet can update the visualization, and the system load, where grouping as many tweets as possible improves performance.

In terms of the number of client connections and query size, while Twitter limits only a single client connection (based on IP address and Twitter credentials) the number of keywords in a query is fairly high (in the hundreds). Our reference architecture, particularly the DEMUX, in combining several streams into a single query for Twitter API and distributing received tweets into multiple streams on our end, complies with this restriction.

4.2 Use case: real-time Twitter monitoring

Our architecture supports building applications for both ends of the real-time monitoring and exploration use case: at one end, a small set of users performing deep analysis on a Twitter stream, thus requiring a highly interactive system; at the other end, a highly-scalable application supporting many users, but with interaction limited to monitoring only.

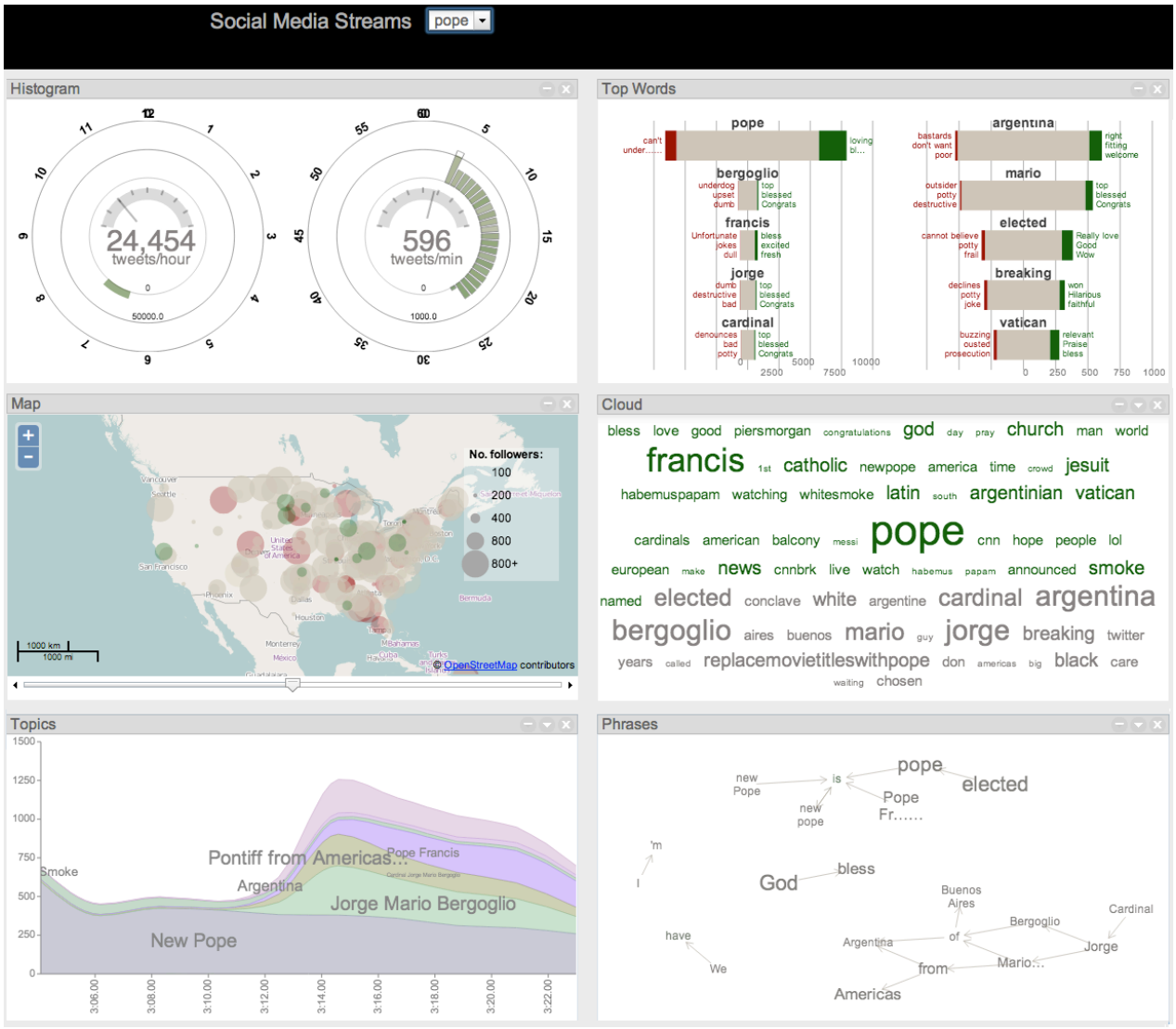

Figure 5. TwitterViz: A dashboard of visualizations showing real-time tweets containing the word "pope", showing the important topics, top phrases, locations, time and volume of the conversation.

The first use case requires that users effectively monitor events (e.g. a news event, a customer experience report, etc.) and be able to drill-down to understand the conversation more deeply. All our visualizations are coordinated so a drill-down action based on location, time, topic, phrase, key words, updates all visualizations. Thus, users can drill-down on a specific region on the map, choose a specific topic, limit to a specific time-period and examine sentiment, top words, and phrases easily. To support this, users can create a stream with a dedicated processor, where multiple users can create interaction events, such as drill-downs.

In the second use case, large numbers of users monitor broadly interesting public events (e.g. elections, pope selection process, etc.) with the goal of getting a sense or a first impression of the public opinion around the event. To support this, one can create a public stream, with a processor providing service for all users, monitoring the same set of visualizations.

Both use cases are equally well supported from a single engine instance, which can host a number of public streams and a smaller number of dedicated streams.

4.3 Map: location analytics and geo visualization

In order to display tweets on a map, we need to identify the geo-location (longitude/latitude) of the tweets. While only a small percentage of tweets come with that information, some users put location in a text field, with varying granularity (e.g. city, state, country). Even so, many of the values of the location fields contain information other than actual location. In any case, we perform a location analysis. This analysis is "local" in that the scope of analysis is only a single particular tweet. In our current implementation we perform a dictionary-based analysis, which examines the location field and tries to match an entry in the dictionary with a specified geo-location. The dictionary is built in a hierarchical manner, going from country to state, to district and then to city levels. If a location cannot be identified, we sample an arbitrary geo-location. To avoid clutter, we added an additional 20 miles random noise to the tweets.

For putting data on the map, we designed a synchronized two-layer component. One layer contains the map tiles from OpenStreetMaps (openstreetmaps.org). This layer is also responsible to catch map-related user interaction events, such as ZOOM and PAN. Another layer holds the data visualization. This visualization layer encodes every tweet as circle, having size corresponding to number of the tweeter's followers, and color encoding the sentiment. For the encoding of the temporal sequence, we let the circles fade out over time by reducing their opacity, enabling a smooth transition when the view is refreshed. To synchronize the layers we apply the map-layer's projection also to the visualization layer.

The Map visualization widget, like any other visualization, supports user-initiated filtering, in this case via drill-down on a particular region of the map (Fig. 6.) As a result of selecting a region of the map, the coordinates of the region are sent to the back-end and a FILTER work is created, which is received by the sliding-window repository worker of the corresponding processor. The repository worker filters matching tweets in the current rolling window and sends a RESET and BATCH_ADD work down the pipeline, where all analyzers and visualizers first reset their state and perform a batch add, eventually updating all visualization with the filtered data.

4.4 Phrase: phrase detection and graph visualization

Phrase analysis and visualization is split into four analyzer and visualizer workers. First, the Part of Speech (POS) analyzer worker identifies POS tags associated with each word in the tweet body. Then, a phrase analyzer worker identifies a set of phrases occurring in each tweet based on some heuristics that leverage language structure based on POS tags. Both of these analyzer workers are local analyzers. Next, the top phrases worker kicks in at a set frequency to calculate top phrases from all of the phrase sets obtained in the current sliding window. Last, the phrase visualizer worker creates a set of nodes for each phrase element in the top (20) phrases and links them appropriately. Thus, the scope of the top phrases worker and phrase visualizer workers are the sliding window.

In the phrase visualizer worker, the node/link diagram is then fed into a layout algorithm that continuously tries to optimize the layout, updating it at a set frequency, so that the users can see intermediate layouts. To prevent a subsequent update of the next set of top phrases completely erasing any spatial reference to existing nodes (phrase elements), only new nodes are fed to the graph layout algorithm thus preserving locations of any previous nodes in the layout.

The Phrase visualizer also supports drill-down of phrases and phrase elements. Users can select a phrase or a set of phrase elements and all visualizations are updated to reflect only those tweets containing the selected phrase elements.

4.5 Topic: topic analysis and timeline visualization

The topics visualization we built is an example of a sliding-window visualization. This visualization shows the distribution of topics within the tweets as a stacked bar chart, showing the evolution of topics similar to [16]. Unlike the other visualizers, the topics visualization needs a set of tweets on which to perform the topic analysis. A challenge is to collect enough data to perform a meaningful topic analysis while still ensuring that the user sees something relatively quickly. In the current implementation, the topics visualizer waits until it has access to at least 20 tweets before it performs its analysis; after that, it recalculates the topics after 20 new tweets have been received or a timeout has expired. The cache of tweets will grow until older tweets expire; thus, the performance improves over time.

In order to support topic analysis over generic Twitter streams, we use an unsupervised clustering system. That is, we do not have any training period for our clustering analysis. Cluster labels are generated on-the-fly using phrases extracted from the tweets in a given cluster. To ensure visual stability we used the cluster label as an index into a set of colors used in the stacked area graph. We sorted the clusters label for each cluster before the color selection process. So, even though the entire stacked area graph is recalculated as new tweets arrive or FILTERs are processed, the visualization appears stable to the user.

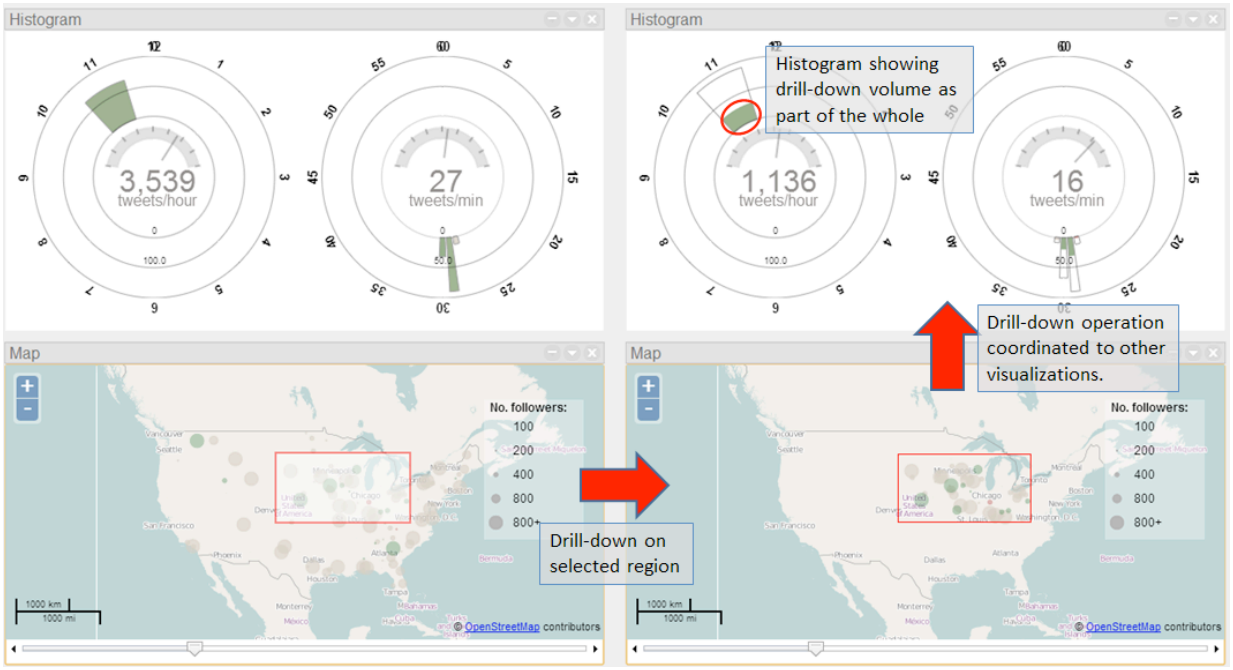

Figure 6. User selecting a region on the map causes a drill down on all visualizations on the dashboard.

4.6 Word and cloud: sentiment and text visualization

One of our back-end analytics is a sentiment analyzer, which labels individual words and tweets as positive, negative or neutral. TwitterViz contains two visualizations that focus on individual words that occur most commonly in the tweets. Top Words focuses on showing the volume of tweets and their sentiment for a relatively small set of words. Each top word is represented by a bar partitioned into the three sentiment types by color – gray for neutral, green positive, and red negative. Through this visualization one can get a clear sense of the relative volumes of tweets and the sentiment around them. Cloud, on the other hand, provides fuzzier information on a much larger set of words. Here the size also corresponds to volume, but color is calculated using a log scale (and thus has a more muted dynamic range).

5. DISCUSSION

Based on our experience with building and using the TwitterViz we would like to discuss the patterns and the reference architecture along four key points: 1) scale 2) client vs. server 3) interaction, and 4) flexibility.

There are at least two factors to consider regarding scale. One is scaling to the volume of data, while the other is scaling to the number of users. Based on our experience, both are well supported with our architecture. We believe several of the patterns discussed in the reference architecture contributed to this. For example, Federated Consumers supported reuse of the same data source by several users while complying with the provider limitations. The flexibility provided by the Queue of Observer pattern facilitated replicating workers as well as whole queues to scale up to the larger volumes of data. The reference architecture also lends itself well to approaches that can automatically adjust stream rate. We implemented a simple scheme but certainly more sophisticated approaches as in [25] are possible in our architecture. Likewise the ability to serve the same visualizations to multiple users facilitates scaling up with respect to the number of users.

Regarding client vs. server, the fundamental question in client-server web applications is how to split functionality between the server and client, which has significant bearing on the kinds of interactions available to the end user. In our reference architecture the server does the "heavy lifting:" analyzing and filtering the streaming data as well as preparing the visualizations. The client, on the other hand, is thin, only responsible for rendering the visualizations and relaying user interaction back to the server. In our experience, this partition of labor is advantageous in that the server can be scaled out (e.g., on a cluster) to support larger streams, more analytics, and more visualizers, and that multiple clients can benefit from common work done on the server. However, a disadvantage is that certain user interactions require a server round-trip.

Several considerations affect the design of interaction capabilities for an application built on our reference architecture. First, can the result of the interaction be supported within our architecture solely on the client (or with minimal support by the server)? If the answer is "yes", then a highly scalable application can support this interaction. We have built several types of interactions of this type. For example, pause/resume control, where the data keeps streaming to the server, but the paused client stops sending update requests to the server. Another example is visual augmentations, such as tool tips and highlighting. These can be done within the client if the supporting information is conveyed in the data sent over by the server as part of the visualization. Yet another example is client-control of update frequency. A variation of this is taking snapshots of visualizations, which is also well-supported in our architecture. There are other examples, where the interaction primarily involves only a single visualization, such as zooming and panning a geographic map. This is done without involving our server. For computation-intensive operations, which affect the content of the streams, the answer is "no." Examples of this type of interaction we built are creating a dedicated stream for exploration, and filtering a stream based on keywords, topics, phrases, sentiment, or location on a map.

A second consideration is how to arrange the visualization content to support a desired interaction effect. When the interaction is related to the contents of a visualization (such as showing a tool-tip or selecting a piece of a visualization for specifying a filter), the associated metadata needs to be packaged with the visualization specification. When a user interface event occurs, the appropriate metadata is picked up and used to produce the desired interaction. For example, a mouse hover over a visual element may pick up part of the metadata to display as a tool tip. In this case, the interaction is performed purely client-side. Interactions that necessitate server support can be facilitated by defining new actions that can be pushed from the client to the server and through the various analyzer and visualization workers in the workflow. The client picks up the relevant metadata (that encodes an action and associated data), and makes a backend call with the action and the data associated with the event. For example, we specify mouse-click events to be matched to FILTER actions. When the user clicks on a specific element in a visualization (representing either individual or aggregate data), underlying data for that element is identified, and is sent to the server along with the FILTER action.

Related to interaction is support for collaboration. The architecture lends itself nicely to support collaboration during at least two types of analytic work, monitoring and exploration. Collaborative monitoring is supported by the ability of each processor to share the resultant visualizations at no or little cost. Collaborative exploration is facilitated by allowing users to share a processor and then redirecting them to a dedicated data processor with full interactive capabilities, where each user can interact with the same visualizations, seeing each other's interactions.

Finally, we believe our reference architecture supports flexibility that can facilitate easy changes to the workflow, dynamically adding new workers to analyze and visualize data. Particularly, Queue of Observers pattern allows connecting and re-wiring workers flexibly while the Moving Blackboard pattern facilitates easy sharing of data among workers.

6. CONCLUSION

In this paper we presented a reference architecture and proposed a number of architectural patterns that are aimed at designing flexible, open, real-time, web-based visual analytics platforms to support a variety of applications. We shared our experience building an application based on our reference architecture and discussed the advantages and disadvantages of the patterns, as they have been applied to visualizing real-time twitter data. We believe a discussion of the architectural patterns is critical and timely, as business demands push for more open and shared analytics and visualization platforms.

REFERENCES

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom. Models and issues in data stream systems. In Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems, PODS '02, pp. 1-16, 2002.

[2] M. Stonebraker, U. Çetintemel, S. B. Zdonik. The 8 requirements of real-time stream processing. SIGMOD Record 34(4): 42-47 (2005)

[3] D. A. Keim, J., Kohlhammer, G. Ellis, F. Mansmann. Mastering The Information Age – Solving Problems with Visual Analytics. Eurographics 2010.

[4] J. Thomas, K. Cook. Illuminating the path: The research and devlopment agenda for visual analytics. IEEE Computer Society, 2005.

[5] C. Rohrdantz, D. Oelka, M. Krstajić, F. Fischer. Real-time visualization of streaming text data: tasks and challenges. In Proceedings of IEEE Workshop on Interactive Visual Text Analytics for Decision Making at VisWeek 2011.
[6] F. Mansmann, F. Fischer, D. Keim. Dynamic visual analytics -- facing the real-time challenge. In Expanding the Frontiers of Visual Analytics and Visualization 2012, pp. 69-80.
[7] P. McLachlan, T. Munzner, E. Koutsoos, S. North. LiveRAC: interactive visual exploration of system management time-series data. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '08, pp. 1483-1492, 2008.
[8] M. C. Hao, D. A. Keim, U. Dayal, O. Oelke, C. Tremblay. Density Displays for Data Stream Monitoring. Computer Graphics Forum, 27(3):895-902 (2008).
[9] W De Pauw, H. Andrade. Visualizing Large-Scale Streaming Applications. Information Visualization 8(2): 87-106 (2009).
[10] D. Best, S. Bohn, D. Love, A. Wynne, W. Pike. Real-time visualization of network behaviors for situational awareness. In Proceedings of the Seventh International Symposium on Visualization for Cyber Security, pp. 79-90, 2010.
[11] S. Foresti, J. Augtter, Y. Livnat, S. Moon, R. Erbacher. Visual correlation of network alerts. IEEE Computer Graphics and Applications. 2648-59, 2006.
[12] E. Koutsoos, S. North, R. Truscott, D. Keim. Visualizing large-scale telecommunication networks and services. In Proceedings of Visualization '99, pp. 457-461, 1999.
[13] Marcus, M. S. Bernstein, O. Badar, D. R. Karger, S. Madden, R. C. Miller. Twitinfo: aggregating and visualizing microblogs for event exploration. In Proceedings of the 2011 annual conference on human factors in computing systems, CHI '11, pp. 227-236, 2011.
[14] M. Dörk, D Gruen, C Williamson, S. Carpendale. A Visual Backchannel for large scale events. IEEE Transactions on Visualization and Computer Graphics, 16(6):1129-38 (2010).
[15] D. Luo, J. Yang, M. Krstajic, W. Ribarsky, D. A. Keim: EventRiver: Visually Exploring Text Collections with Temporal References. IEEE Transactions on Visualization and Computer Graphics, 2010.
[16] F. Wei, S. Liu, Y. Song, S. Pan, M. X. Zhou, W. Qian, L. Shi, L. Tan, Q Zhang. TIARA: a visual exploratory text analytic system. In Proceedings of the 16th ACM SIGKDD international conference on knowledge discovery and data mining, KDD '10, pp. 153-162, 2010.
[17] S. Ishizaki, Multi-agent model of dynamic design: visualization as an emergent behavior of active design agents. In Proceedings of the SIGCHI conference on Human factors in computing systems, CHI '96, pp.347-354, 1996.
[18] S. Havre, E. Hetzler, P. Whitney, L. Nowell. ThemeRiver: Visualizing thematic changes in large document collections. TVCG: Transactions on Visualization and Computer Graphics, 8(1):9–20, 2002.
[19] D. Fisher, A. Hoff, G. Robertson, M. Hurst. Narratives: A visualization to track narrative events as they develop. In VAST 2008: IEEE Symposium On Visual Analytics Science And Technology, pp. 115-122, 2008.
[20] Fekete, J. D. Infrastructure (Chapter 6). In Mastering The Information Age – Solving Problems with Visual Analytics, D. A. Keim, J., Kohlhammer, G. Ellis, F. Mansmann (eds.), pp. 87-108, 1999, EuroGraphics, 2010.
[21] E. G. Hetzler, V. L. Crow, D. A. Payne, A. E. Turner. Turning the Bucket of Text into a Pipe. In *Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization*, InfoVis '05, pp.89,94, 2005.
[22] E. Hetzler, A. Turner. Analysis Experiences Using Information Visualization. IEEE Computer Graphics and Applications, 24(5): 22-26, 2004.
[23] M. Dubinko, R. Kumar, J. Magnani, J. Novak, P. Raghavan, A. Tomkins. Visualizing tags overtime. ACM Transactions of the Web (TWEB), 1, August 2007.
[24] Albrecht-Buehler, B. Watson, D. Shamma. Visualizing live text streams using motion and temporal pooling. Computer Graphics and Applications, IEEE, 25(3):52 – 59, 2005.
[25] Wong, P., Foote, H., Adams, D., Cowley, W., Thomas, J. Dynamic visualization of transient data streams. In Proc. IEEE Symposium on Information Visualization, pp. 97-104, 2003.
[26] J. Alsakran, Y. Chen, Y. Zhao, J. Yang, D. Luo. STREAMIT: Dynamic Visualization and interactive exploration of text streams. IEEE Pacific Visualization Symposium, pp. 131-138, 2011.
[27] Fischer, F. Mansmann, D. Keim. Real-time visual analytics for event data streams. In Proceedings of 27th Symposium on Applied Computing (SAC), 2012.
[28] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Pattern: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1996.
[29] M. Shaw and D. Garlan. Software Architecture: Perspectives on a emerging discipline. Prentice Hall, Englewood Cliffs, NJ, 1996.
[30] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture. A System of Patterns. John Wiley & Sons Ltd., Chichester, UK, 1996.