Architectural Patterns for Real-time Visual Analytics on Streaming Data

Eser Kandogan[*], Danny Soroker[*], Steven Rohall[*], Peter Bak[*], Frank van Ham[*], Jie Lu[*], Harold-Jeffrey Ship[*], Chun-Fu Wang[+], Jennifer Lai[*]

[*]IBM Research, [+]University of California, Davis

ABSTRACT

Monitoring and analysis of streaming data, such as social media, sensors, and news feeds, has become increasingly important. 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. In this poster we discuss architectural patterns for real-time visual analytics based on building a real-time Twitter monitoring application.

Keywords: Visual analytics architecture, streaming data, design patterns, web-scale.

Index Terms: H.5.2 [Information Interfaces and Presentation]: User Interfaces – Graphical user interfaces (GUI).

1 INTRODUCTION

Visual analytics involves a combination of algorithmic and visual techniques to provide insight into data. With the proliferation of data sources, we see a separation of concerns between accessing and processing data for analysis. For example, for Twitter data several parties are involved in storage, integration, and distribution through web-based APIs. Our expectation is that the visual analytic tools of the future will operate in much the same way as the data providers of today: they will offer online services that are accessible to many users simultaneously and serve requests for different analytics.

Moving from monolithic applications to online open analytic platforms require changes to the core visual analytics architecture. We see several requirements (extending [1][2]): 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 overall user experience, 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. In this poster we present a set of architectural patterns for visual analytics over streaming data, satisfying the above requirements.

* Contact: Eser Kandogan (eser@us.ibm.com)

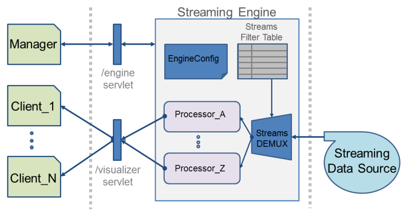

Figure 1: High-level view of proposed reference architecture

2 ARCHITECTURAL PATTERNS

2.1 Pattern: Federated Consumers

Figure 1 shows the streaming engine, which is at the highest level of the reference architecture and 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 through web-based APIs. As such, streaming engines should be cognizant of data provider rules and capabilities related to availability, performance, and data use. Providers might put caps on the volume of data, number of connections and number of queries, or require consumers to enforce data storage and deletion restrictions.

In the *Federated Consumers* pattern a central component mediates all interaction between the provider and consumers. Queries from consumers are combined into an aggregate query, and data from the provider in response to the aggregate query is distributed to consumers based on their respective queries. This way only a single connection is maintained per streaming data source, over which users can define many processors that query the source data. A major benefit of this is compliance with provider requirements is the responsibility of a single component and not a concern of consumers. The weakness of this pattern is the potential for a performance bottleneck, though implementation may replicate the central component to scale better. Implementation should be lean and keeps the data moving at requested rates. An example implementation is the *Streams DEMUX* component (Fig 1.)

2.2 Pattern: Queue of Observers

Designing how computations are performed and structured among several analytics and visualization components is critical for achieving a good performance and user experience. Key challenges are to maximize utility such that 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 (also see [3]), which 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. In addition, there is a queue of observers, defining a workflow of analytic and

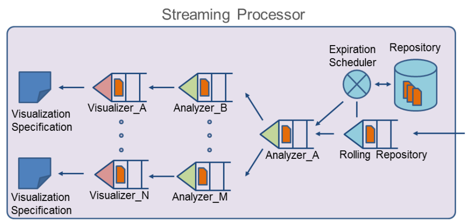

Figure 2. Stream Processor and Data Processing Pipeline

visual computations. Data is transformed through each of the components producing either 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. 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, which can be addressed by replicating critical computations. An example implementation is *Streaming Processor* (Figure 2.)

2.3 Pattern: Sliding-Window Repository

Analysis may require access to data beyond what is currently streaming. In such cases, past data from the provider needs to be temporarily accessed. Legal constraints of the provider, which may limit the persistent storage of data, need to be adhered to. To address these challenges we propose a *Sliding-Window Repository* pattern in our reference architecture, as in [4].

In this pattern, data is only transiently stored that allow querying and expiration 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. The sliding window is essentially a FIFO queue, where data comes in, slides down the window, and eventually expires. The advantage of this pattern is that provider-imposed constraints can be implemented in a single component providing the right level of concurrency. The disadvantage is that this pattern may limit concurrent write/update access from many components. An example implementation is the *Rolling Repository* shown in Figure 3. In this implementation expiration criteria can either be time based, volume based, or dynamically changed based on an algorithm that trades off performance and quality.

2.4 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.

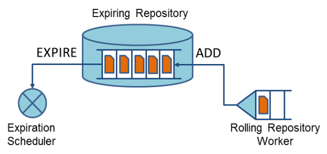

Figure 3. New records are added to or removed from the rolling repository based on an expiration criteria

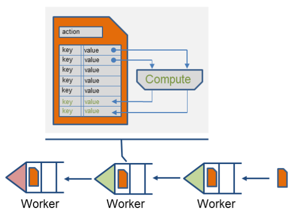

Figure 4. A pipeline of workers each adding/updating new key-value pairs to the work record based on computation performed on data

In the original Blackboard pattern, there is a repository of global variables that can be accessed and modified by separate autonomous processes, each performing some part of the computation. Similarly, in the Moving Blackboard, the current state of the solution is stored in the blackboard as a set of <*key, value*> pairs. Unlike the original pattern, variables are not global rather they are passed from component to component. Each component receives a blackboard, performs some additional computations, and writes the result back to the blackboard before passing it on. The advantage of this pattern is that components can independently work on any number of blackboards representing all state data that they need at a given point in time. As such, if any component 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 and associating it with data. A disadvantage may be memory utilization, but that can be relieved by putting references (instead of deep copying, if applicable), and handling concurrency on objects representing complex data.

An example implementation is workers, which run on their own thread, performing computation based on the action and data defined in a work record (the work record is the moving blackboard) (Fig 4.) We implemented several analytics workers such as part-of-speech tagging, geo location identification, histogram analysis, and topic clustering and visualization workers such as phrase and word clouds visualizers.

3 Conclusion

Using these set of architectural patterns we created *TwitterViz*, a flexible suite of visualizations and analytics that monitor streaming data from Twitter, supporting two use-cases on both ends of the scalability spectrum: 1) a highly-scalable application supporting many users, but with limited interaction, 2) a highly-interactive application with complex analytics that can be shared by several users. Both use cases are equally well supported from a single engine instance, which can host any number of public streams and a smaller number of dedicated interactive streams.

References

[1] C. Rohrdantz, D. Oelka, M. Krstajić, F. Fischer. Real-time visualization of streaming text data: tasks and challenges. In Proc. of IEEE Workshop on Interactive Visual Text Analytics, VisWeek 2011.

[2] 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.

[3] E. G. Hetzler, V. L. Crow, D. A. Payne, A. E. Turner. Turning the Bucket of Text into a Pipe. In *Proc. of* InfoVis '05, pp.89,94, 2005.

[4] Wong, P., Foote, H., Adams, D., Cowley, W., Thomas, J. Dynamic visualization of transient data streams. In Proc. of InfoVis '03, pp. 97-104, 2003.