Stream Processing in Action

Henrique Andrade  
JP Morgan  
hcma@hcma.info

Buğra Gedik  
Bilkent University  
bgedik@cs.bilkent.edu.tr

Deepak Turaga  
IBM Research  
turaga@us.ibm.com

The world is now information-driven. Many business and government institutions are fully automated, relying on a variety of sensors and infrastructure to collect, store, and analyze incoming data on a continuous basis. Clearly, the need to monitor and act on information from sensors in the field, to drive rapid decisions, to tweak production processes, to optimize logistics choices, and, ultimately, to better manage physical systems is now fundamental.

Data stream processing was born from increasingly stringent data management, processing, and analysis requirements of this new breed of business and scientific applications, coupled with the confluence of two major scientific and technological shifts: first, the advances in data management technologies including the software and hardware infrastructure for supporting large distributed deployments; second, the advances in basic techniques in online data mining and real-time data analytics.

Stream Processing Applications (SPAs) are used in sense-and-respond systems that continuously receive external signals in the form of one or more data streams from multiple sources and employ analytics aimed at detecting and, sometimes, predicting actionable information.

Examples of streaming applications and systems abound. Early instances include different Supervisory Control And Data Acquisition (SCADA) systems deployed for monitoring and controlling manufacturing, power distribution, and communication networks, as well as environmental monitoring systems. More recently, SPAs hosted by Stream Processing Systems (SPSs) supporting modern business applications have become common, in areas such as finance engineering, surveillance, and telecommunications.

While SPAs are becoming central to a variety of new application domains, designing and developing them is challenging. Not only is it essential to ensure that the application performs its required analytical task correctly, but it is also important to ensure that this is accomplished efficiently, scalably, and reliably, utilizing the available computing infrastructure parsimoniously, while meeting its performance requirements and adapting to unexpected variations in load and in resource availability.

To meet these goals, application developers, with the help of the SPS, must carefully decompose the application, mapping it onto a set of operators organized in a Data Flow Graph (DFG), design and implement individual components to perform the desired analytic tasks, distribute the application across the computational infrastructure, and, finally, continuously tune the application so it evolves appropriately. Additionally, developers also need to account for fault tolerance and dynamic adaptation in response to data and processing variations that might arise at runtime.

In our experience, these characteristics require a shift in the developer’s thought process to incorporate new analytic and engineering methods during application design, development, and its continuous evolution. The stream processing paradigm represents this shift, and SPAs serve as the software platform technology that enables the implementation of applications that adhere to this new computing paradigm.

A Brief History

Research on data stream processing emerged in earnest in the late 1990s. However, the use of data stream processing technologies relies on the availability of SPSs, allowing developers to focus on the analytical side of their applications, rather than on the complexities of distributed programming, such as data transport, distribution of application logic, and managing the runtime execution. The spawning of this type of middleware marks the beginning of the data stream processing era.

In academia, systems such as STREAM [4], Borealis [1], TelegraphCQ [8], and in industrial research labs, systems such as Gigascope [9] and System S [15], have focused on providing stream processing middleware and, in some cases, declarative languages for easing the task of writing applications.

After nearly a decade of combined distributed systems, data management, and analytics research, data stream processing has come to age with commercially available and open-source middleware, tooling, and analytic libraries that supply developers with a foundation to build sophisticated SPAs. InfoSphere Streams [13] and StreamBase [21] are two examples of well known SPSs available commercially. In the open-source world, newer SPAs have recently become available, including Storm [20] and S4 [16], enabling a broader swath of developers to make use of this technology.

Characteristics of SPAs

SPAs have specific features that, in conjunction, set them apart from traditional interactive and batch processing applications. We discuss the most fundamental of these features.

1Numerous technologies have been known as stream processing. We emphasize data to refer to systems that deal with stateful, selective, general-purpose data manipulations and analytic operators.
Edge adaptation

SPAs implement systems. Data cleaning and transformation as well as protocol conversion—referred to as edge adaptation. This activity might involve data cleaning and transformation as well as protocol conversion and rate adaptation to match the needs of the external systems.

Data-in-motion and long-running analytics

SPAs implement data-in-motion analytics. These analytics ingest data from live, streaming sources and perform their processing as the data flows through the system. This on-the-fly model of computation provides the ability to ingest and analyze large volumes of data at rates that are not possible to achieve with traditional data-at-rest analytics, imparting the ability to adapt to changes and to derive insights in real-time.

Furthermore, SPAs are usually long-running applications. As a result, they are usually designed to function autonomously as well as to be maintainable (with component hot-swap capabilities) and tolerant to transient failures, ensuring resilience in the face of workload fluctuations, changes in resource availability, application requirements, and availability of data sources and sinks. Hence, the use of adaptive resource management techniques including load balancing and load shedding as well as partial fault tolerance techniques are both prevalent.

Reactive and pro-active analytics

SPAs are reactive, built to detect patterns of interest in streaming data and to generate actions as a result. Hence, these applications maintain state typically held in a window, whose contents are updated as the data flows in. Many SPAs are also proactive, particularly when implementing exploration tasks where the current and past history of one or more incoming data streams are used to build models for predicting future events.

Both reactive and proactive analytics must adapt the internal processing logic of the application based on newly detected or predicted events. To implement this, it is necessary to propagate control information from later stages of a DFG to earlier ones in the form of feedback loops. In general, the feedback control information can also trigger topological modifications to an application’s DFG. Examples of such changes include altering the set of data sources and sinks used by an application, adding new analytical pieces, or removing existing portions from a running application.

Multi-modal analytics

SPAs use multiple modes of data input and output, a requirement driven by the need to correlate information across heterogeneous data sources. Structured information such as relational database records, unstructured information such as audio, video, and text, and semi-structured information such as XML- and JSON-encapsulated data are a few examples of the types of data processed by a SPA. Hence, SPAs must have support of a rich type system and programming model so an application’s analytical engine can be properly expressed and engineered.

High performance and scalable analytics

SPAs often demand high throughput and low latency, as a consequence scalability to handle the vast volume of data they analyze is paramount. Meeting throughput and latency requirements implies the use of both vertical scalability (scale-up) and horizontal scalability (scale-out) techniques. Vertical scalability demands scaling the performance of an application as the computational capacities of the hosts where it runs increases. Thus, a SPA is usually designed to take advantage of more CPU cores, additional bandwidth, and additional memory when these resources become available. Horizontal scalability demands scaling the performance of applications as the number of hosts used to execute them increases. Thus, a SPA is usually designed to be elastic.

A Structural Example

A typical application embodying the characteristics above contains three main high-level components: data ingress, analytical processing, and data egress. Figure 1 presents a simple application that has these three components, in pseudo-code written in the SPL language.

The canonical application, implemented by the composite operator CanonicalApp in the sample namespace CanonicalApp { graph stream<InputT> Input = DataIngress() { param ... } | stream<OutputT> Output = Analytics(Input) { param ... } | () as Sink = DataEgress(Output) { param ... } }

Figure 1: The canonical application implemented in pseudo-SPL.
Typically, in a real-world application, one or more source operators will continuously ingest data produced by one or more external sources, for instance, a TCP socket connection, an HTTP feed, a device driver interface, or a dynamic set of input files. Indeed, the application whose DFG is depicted by Figure 2 employs two source operators (the two leftmost boxes), whose data streams are then fed to multiple data processing operators downstream.

In a typical application, the analytical processing component is usually expressed by a combination of several operators, potentially alongside with additional systems (e.g., a data warehouse), that work in concert to perform a complex streaming analytical task. Again, we refer to Figure 2, where the application’s DFG mid-section contains a combination of composite and primitive operators, performing numerous data processing steps, that collectively implement the application’s analytical core.

Data egress can range from simply writing tuples from an incoming stream to the standard output as text to performing format transformations, buffering, and communications, providing whatever adaptations that are needed by an external sink. In a realistic application such as the one in Figure 2, multiple sink operators are used to route the application’s results to external sinks through a mechanism that might include interfacing with a TCP socket, a message queue, or a DBMS.

Examples of Real-World SPAs

We discuss two real-world applications, one for large-scale systems operations monitoring and the other for clinical care patient monitoring. Each highlights different aspects of stream processing in action.

**Operations Monitoring**

Consider an environment where a large number of streams of log messages, originating from different software components that are part of a corporate IT infrastructure, must be monitored and managed. Such an environment is present, for instance, in network service providers where voice, SMS, and data services are made available directly to users or to other applications over a cell phone network.

In such an environment, the software components that support these services generate log messages that capture instantaneous snapshots of their internal operation, including information about their resource utilization. Maintaining such a multi-component software environment in good working order is critically important to deliver reliable service to users.

To provide this capability we designed a mediation SPA that continuously collects streaming log messages and performs appropriate filtering and transformation as a front-end to other analytic applications. The analytic applications monitor the overall infrastructure continuously, providing system state monitoring, resource usage enforcement, and system outage detection and prediction. The overall solution is structured around three requirements:

- **Edge adaptation**: to isolate the analytic applications from specific protocols and data formats used to ingest the logs produced by the monitored software infrastructure.
- **Data-in-motion and long-running analytics**: to provide filtering and transformation of the data to enable different analytic applications to subscribe to the content they are individually interested in, minimizing the transmission of irrelevant information.
- **High-performance and scalable**: to deliver sufficient performance, meeting latency, throughput, load-balancing, and fault tolerance requirements to handle a large number of log messages and be flexible to scale as the data rates grow and more compute resources become available.

In terms of edge adaptation a typical mediation application in a large enterprise must ingest data from up to 1000 different sources of log messages where each source is connected via a TCP connection, extracting an equivalent number of streams, each one with a particular topic of interest. In this case, the data-in-motion analytics are focused on parsing the text messages and transforming them into a well-defined structure defined by the topics, which can then be dynamically subscribed to and used by other applications.

The performance requirements are what make the implementation of this SPA challenging. It supports (potentially) millions of connections that interconnect producers and consumers, providing continuous operation with fault tolerance and flexible distribution of messages with a throughput in the order of millions of messages per second with sub-second end-to-end latency.

Alternate implementations of this design employing batch processing and using disk storage, leveraging either a map-reduce toolkit or a DBMS, were shown not to support the throughput and latency requirements defined by the customer. Moreover, a custom implementation using MPI is
harder to develop, extend, and reconfigure and is harder to flexibly reconfigure to exploit tradeoff required for adequate load balancing.

In contrast, our implementation using a SPS has been shown to process 14 million messages per second with sub-second latency using 4 nodes with a total of 32 cores, interconnected with a 10GBit Ethernet switch. More importantly, the same implementation can be automatically reconfigured to use more nodes as well as to service more sources and consumers.

**Patient Monitoring**

Our second application, used in clinical setting with patients in a hospital who need to be continuously monitored for different health ailments, highlights the sophistication in the data analysis that can be carried out in continuous fashion.

The data to be analyzed includes different physiological sensor readings from each patient, which are available as continuous streams, coupled with other types of medical records including laboratory test results and patient history. The application’s analytical goal is to apply signal processing to this data to identify and predict complications as early as possible. Among the available signals is electrocardiogram (ECG) data, which can be used to predict the onset of different clinical issues. Yet, before applying more sophisticated analytics, the raw ECG signal must be pre-processed to first discard what is called normal or ectopic beats.

In this case, we designed a SPA whose aim is to ingest the raw ECG data, segment it into heartbeats, and then identify and discard the ectopic beats. It is structured as follows:

- **Edge adaptation:** to connect, in a fault-tolerant manner, different on-patient sensors and bed-side data gathering devices to the data analysis application.
- **Data-in-motion and long-running analytics:** to perform signal processing to identify individual heartbeats from raw ECG and to extract spectral features to characterize them.
- **Reactive and proactive analytics:** to dynamically build and apply models for ectopic beat detection to adapt to time-varying characteristics of each patient’s heartbeats.
- **High-performance and scalable:** to deliver sufficient performance, adhering to strict latency, throughput, and fault tolerance quality boundaries and to be able to handle a large number of signals from multiple patients.

The ECG signal is typically sampled at around 500Hz and contains three separate channels from different electrodes. In an Intensive Care Unit (ICU) with 50 beds, this volume adds up to close to 5Mbps. In an ICU where patient state is continuously monitored, analytical results must be available quickly such that appropriate interventions can be performed when they are most effective.

One of the key challenges in this environment is “alarm fatigue” as many alarms are often triggered due to a change in the patient state that may or may not require intervention. For instance, certain medications are known to affect patient physiology in a certain way, and, if the models used by the analytics are not incrementally adaptive to automatically account for this, they can trigger unnecessary alarms.

Again, an implementation relying on map-reduce software or on DBMSs may be a reasonable technical solution in small ICUs, however scaling up as a result of the need to handle larger numbers of patients or by integrating additional signals becomes increasingly more difficult. More importantly, the need for adaptive processing with incremental learning of data mining models as well as of dynamic reconfiguration of the application with feedback loops in its DFG are much more naturally implemented using a streaming system.

Our SPS-based implementation [6] has been shown to reduce the number of false alarms by half when compared with a statically trained model.

**Building Streaming Applications**

An SPS typically provides an IDE that integrates both language and application design elements specifically suited for creating SPAs. The two most fundamental capabilities that are usually provided include stream flow composition and flow manipulation. Flow composition is about putting together an application’s DFG, connecting its operators to form a processing topology. Flow manipulation is about selecting and configuring operators used to process the data transported by the application’s streams.

**Data flow graphs**

There are different approaches for putting together an application’s DFG. One is graphical composition, which relies on the boxes-and-arrrows metaphor, often supported by a drag-and-drop editor. Another approach is domain-specific languages, such as extensions of SQL (e.g., StreamSQL [22], EPL [12]) or data-flow oriented languages (e.g., SPL). Yet another approach is programmatic composition (e.g., Storm’s Java APIs [20]). Some systems support one or more of these at the same time, allowing the round-tripping between the language and the visual representation [13, 21] or providing a domain-specific language that targets the programmatic composition API [19].

In general, DFGs are arbitrary graphs, but some systems constrain them to Directed Acyclic Graphs (DAGs) to avoid deadlocks in the data flow. However, in many applications, cycles are needed to support information feedback loops, where a downstream operator feeds control data back to an upstream operator, supplying information that can be used to dynamically switch the application behavior. Deadlocks can be avoided by restricting the streams forming feedback loops to connect only to control ports, i.e., input ports that do not generate output tuples, but instead simply update the internal state of an operator.

**Composite operators**

Another important aspect of flow composition is scalability, from a software engineering perspective. As an SPA grows in size, its DFG becomes large and complex. This problem is even more pronounced when multiple team members are collaborating to build a large application, which must be segmented into smaller components.

This complexity can be managed by imposing a modular hierarchy on the graph, via the use of composite operators. Composites encapsulate a subgraph, potentially embedding additional composites. Through parameterization, they also...
enable the reuse of the same subgraph in multiple sites in an application’s DFG.

For instance, in the patient monitoring application, the composite that classifies the heartbeats can be instantiated more than once, each using a different value for the parameter that specifies the classification model to be loaded and scored. The results from each instance can then be merged creating a more powerful classifier.

### Dynamic composition

The flow composition mechanisms discussed so far are all static, used to explicitly construct an application’s DFG at development time. A complementary technique is dynamic composition where the exact endpoints of a stream connection are established at run-time, subject to development-time constraints.

Dynamic composition enables two important use cases. First, there is the incremental deployment where an application is laid out in the execution environment in piecemeal fashion, tapping additional components as needed. Second, there is the dynamic discovery of sources and sinks, where an application is designed to consume data from, or to deliver data to, a variable set of producers and consumers. Such producers and consumers can potentially be other applications sharing the same execution environment.

For instance, in the operations monitoring application, the mediation layer exports several log streams, each with a different topic. The analytics applications import some of these streams based on their interests on the different topics. Essentially, a stream-level topic-based pub/sub system is used for establishing dynamic connections. This can be further extended to content-based matching, where analytic applications import some of the streams partially, based on a content filter (such as filtering logs belonging to a certain topic stream based on the source of the log).

### Extensibility

SPAs are expected to make use of both cross-domain as well as domain-specific operators. Cross-domain operators implement common data manipulations. Most SPAs provide operations such as selection, projection, aggregation, joins, and sort algorithms available as stream relational operators. Other cross-domain operations such as splitting, merging, and de-duplication are also commonly available. Nevertheless, it is often necessary in the course of developing a new SPA to add custom-made operators and, for this reason, most SPAs are extensible.

Extensibility mechanisms vary, but are most useful when they make it possible to build user-defined operators that are also first-class operators, working similarly to built-in operators. In general, these general-purpose operators should be (1) type generic, working with any stream irrespective of its schema; (2) parameterizable, employing thanks (expression parameters that are evaluated lazily as part of the operator’s processing logic; e.g., a match condition in a join operator) and allowing (user-defined) functions in expressions; (3) able to use processing windows, accumulating information given some chunking boundaries; and (4) able to define custom aggregations, representing computations applied on a group of tuples rather than on a single tuple.

### Running Streaming Applications

Beyond supplying a development environment, SPAs also provide a runtime environment for managing the execution of SPAs. As such, the fundamental task of the runtime environment is to make available an execution substrate and to perform common systems tasks such as placement and scheduling of an application’s execution components, data transport for streams, fault tolerance as well as management and debugging capabilities for users, system administrators, and developers.

### Placement and scheduling

An application’s DFG composed by a developer is simply a logical representation of the SPA. This graph comprises primitive and composite operators and the runtime execution environment must map this logical DFG to a physical representation and deploy it onto the resources available to the system.

In a distributed SPA, the runtime resources might consist of a cluster of hosts where each one can potentially have more than one processing core. Hence, to perform the physical mapping, the runtime execution environment must create a physical DFG. This task is typically accomplished by assigning operators to Processing Elements (PEs), which are sometimes called worker processes.

The physical DFG might fuse together multiple operators into a single PE, which in turn will be hosted by an OS-owned process. In other words, each PE is a container for application subgraphs. Subsequently, the SPA places the PEs onto hosts, usually with the goal of optimizing the application’s throughput and/or its latency.

Placement is a challenging problem requiring knowledge about i) the application characteristics, including operator costs and selectivities; ii) the resource characteristics of the system, including the data transport costs within a PE, across PEs, and across hosts as well as CPU and bandwidth availability; and iii) the characteristics of the workload, including incoming stream rates.

Further complicating this problem, all of this information can change at run-time requiring potentially disruptive placement modifications as an application runs. As a result, a substantial amount of research has been devoted to placement and scheduling algorithms with various degrees of support and sophistication provided by different SPAs.

### Data parallelism

The throughput of a SPAs might be limited by one (or more) bottleneck operators in its DFGs. The buffer used to manage the data flowing into an input port of a bottleneck operator will eventually fill up, causing the upstream operators to block, so no data is lost. This effect can propagate all the way to the external data sources, a phenomenon known as backpressure.

When a bottleneck operator is stateless or when its state can be cleanly partitioned, an opportunity arises for removing the bottleneck via the use of data parallelism, where the operator is replicated and the internal operator state, if it exists, is partitioned and segmented. While the use of data parallelism can be manually programmed by a developer, it is tedious and it can also be error-prone. Hence, some SPAs support developer-guided parallelization, in which case the developer indicates where and how much data parallelism is needed and the runtime automatically i) segments the in-
The first is processing live streaming data. The difficulties stem from the inherent non-determinism associated with SPAs. ii) instantiates, wires, and places the replicas, and iii) merges the results as seen in Figure 3.

Fault tolerance

SPAs are typically long-running applications and, in many cases, perform critical operations in a corporate IT environment. Therefore, fault tolerance is an important concern. In a streaming context, this problem has two sides: the SPS middleware itself and the applications. A fault-tolerant middleware usually makes use of a transactional store for maintaining its critical internal state and for ensuring that the services it provides to the applications it hosts remain accessible even in the presence of sporadic transient failures.

Designing and developing fault-tolerant applications, even with the help of a fault-tolerant SPS, is potentially a much more challenging problem. The difficulties stem from the increase in complexity and from the additional performance costs, as well as from the existence of (distributed) application state and the inherent non-determinism associated with processing live streaming data.

These problems are usually addressed using one of two approaches. The first is active replication where two or more copies of the application (or of some of its critical components) are run on disjoint sets of hosts, switching from a faulty to a normally operating replica when necessary. The second is checkpointing where (some of) the application state is periodically checkpointed and, on failure, the state is restored from the latest available snapshot, possibly, incurring some data loss in the interim. Each approach has different engineering trade-offs, which are usually looked at and evaluated during the design phase of a new SPA.

Debugging and visualization

Debugging distributed and asynchronous applications is challenging. SPAs are no exception, yet their DFG structure reduces the complexity of the debugging task compared to, for instance, what is found in MPI-based applications, which can have more complex interactions and less clear boundaries in terms of the data exchanges between its computational components.

In general, there are four categories of debugging tasks associated with SPAs: i) semantic debugging where the goal is to assert that streams contain the intended data; ii) source code debugging where the internal workings of an operator can be examined, usually leveraging the integration of a streaming debugger and a traditional debugger (e.g., jdb and gdb); iii) deployment debugging where the physical deployment of an application can be checked usually with the help of a visualization infrastructure; and, finally, iv) performance debugging where an application can be examined for compliance to its throughput and latency requirements. Naturally, SPAs vary in how well these tasks are supported, but elements of each of these categories are usually available.

Closely related to debugging, visualization infrastructure is usually provided by SPSs as part of either their IDEs or system management tooling. The visualization of SPAs is, in most cases, not limited to the mere exploration of their logical and physical DFGs as seen in Figure 2, but include also the presentation of the streaming data, typically, the results to be output by the sink operators. Indeed, the capabilities of some platforms include the ability to automatically provide rich graphical dashboards.

Analytics

Different types of analysis are typically performed on streaming data to extract appropriate insights and actionable intelligence. It involves applying techniques drawn from data mining, machine learning, statistics, signal processing, and artificial intelligence in the context of continuous processing. Streaming analysis requires analytics for the following four key stages in the knowledge discovery process:

- Data Preprocessing: data cleaning, interpolation, normalization, temporal alignment, formatting, and reduction.
- Data Transformation: feature extraction, summarization, changing representation.
- Modeling: identifying interesting patterns, similarity and groupings, correlations, and anomalies.
- Evaluation: applying or evaluating a mining model and the interpretation of the results.

Streaming analysis poses several novel requirements, such as continuous and long-running analysis, limited compute and memory usage, one-pass processing, incremental update and adaptation, and finally distributed and parallel analysis. Hence, several traditional “at-rest” data analysis techniques need to be adapted to meet the requirements of streaming analysis.

In many real-world scenarios, it is common to use a large repository of previously collected data for model learning and then use these models for evaluation as new data flows into an application. In this case, we need to combine offline modeling with streaming evaluation. Model sharing across these analytic implementations is enabled by Data Mining Group (DMG)’s Predictive Model Markup Language (PMML) standard representation for different types of models. Most commercial data mining tools support the export and import of PMML models.

However, there are several real-world settings where offline learning (even with a periodic retraining) is insufficient, as was the case for the patient monitoring application. In these cases there is a need to have streaming techniques for both modeling as well as evaluation. A list of some popular...
streaming techniques \[2\] for these different stages are shown in Table \[1\].

While several of these algorithms are used in different application settings, the design and development of streaming analytics is still an active area of research.

**What Lies Ahead**

Newer and more innovative business and scientific problems drive application designers to take on more intricate analytical tasks. Often, developers must also contend with the need for processing larger and more complex workloads with ever more sophisticated analytical demands. These trends create new challenges for SPAs and, hence, require additional engineering and research.

**Auto-parallelization**

Auto-parallelization is the set of language, compiler, and middleware techniques used to transparently parallelize SPAs to better take advantage of hardware resources. This is an area whose importance is closely aligned with the changes in the computing landscape where multi-core chips, GPUs, and the use of cluster computing are now the norm.

Despite these trends in the hardware landscape, the efficient use of these computing resources still hinges on substantial manual intervention to parallelize code. A big challenge is to determine the right level of parallelism that maximizes the performance as well as the arbitration between various forms of parallelism inherently found in or extracted from SPAs (data, task, and pipeline parallelism), while making sure that application semantics are preserved.

**Distributed state**

In-memory state, maintained locally on a per-operator basis, is the norm in SPAs. However, the scaling up of applications, algorithms, and analytical models requires effective ways of maintaining large amounts of distributed state and the support of operations such as indexing, searching, and updating on this state’s underlying data structures.

Management of large-scale in-memory distributed state is a major challenge for SPAs because many applications have requirements on consistency, balanced distribution, and resilience to sporadic faults with respect to their distributed state. Providing these capabilities require seamless integration of SPAs with distributed memory technologies, such as key/value stores, a feature that is lacking in today’s SPAs.

**Adaptive learning and exploration**

The emerging area of big data analytics creates yet another challenge, related to the strong synergy between offline learning and online scoring. Adaptation of the model, via another round of offline learning, needs to be triggered by an online flow, whenever the data mining model starts to degrade, i.e., when concept drift is detected. Development of online concept drift algorithms that are generic, effective, and applicable to a wide range of models is key to efficient resource usage and improved accuracy of the learning.

Finally, supporting the automatic composition, deployment and orchestration of analytics across stream processing and offline platforms is essential for large-scale data exploration – where it is apriori unclear what is “interesting” in the data. Such problems are key to building cybersecurity (and other) applications where the set of network behaviors corresponding to malicious activities are often unclear, and rapidly time-evolving.

**Conclusion**

Stream processing emerged from the confluence of advances in multiple computer science areas to provide an intuitive computing paradigm where data is consumed as it is generated, computation is performed on-the-fly, and results are immediately produced, all within a continuous cycle.

When stream processing is put to action, a new crop of applications, focused on extracting intelligence from large quantities of continuously generated data to provide faster, online, and real-time response can be designed and developed in domains that include environment and infrastructure monitoring, manufacturing, finance, healthcare, telecommunications, cyber-security, as well as in large-scale scientific and experimental research.

In this introduction we provided an abbreviated overview of stream processing in action. In our upcoming book \[3\], we revisit many of these topics, providing an expanded, in-depth discussion with plenty of design and implementation examples. We invite the interested readers to peruse it and share in the exciting possibilities brought about by stream processing.

**Acknowledgments**

This paper is the product of numerous discussions and refinements with a much larger team. Therefore, we acknowledge the collaboration of colleagues from multiple IBM sites, including Rochester (MN), Toronto (Canada), Silicon Valley (CA), Raleigh (NC), the China Research Lab, and from the Thomas J. Watson Research Center (NY).

<table>
<thead>
<tr>
<th>Stages</th>
<th>Category</th>
<th>Illustrative Streaming Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocessing and Transformation</td>
<td>Descriptive Statistics</td>
<td>BasicCounting: Approximate sliding window summaries (min/max, average, distinct count).</td>
</tr>
<tr>
<td></td>
<td>Sampling</td>
<td>Reservoir Sampling: Perform random sampling from a stream, with a fixed sample size.</td>
</tr>
<tr>
<td></td>
<td>Sketches</td>
<td>Count-Min Sketch: Approximate occurrence frequencies of items in a stream.</td>
</tr>
<tr>
<td></td>
<td>Quantization</td>
<td>Moment Preserving Quantization: Reduce bit precision while keeping temporal shape.</td>
</tr>
<tr>
<td></td>
<td>Dimensionality Reduction</td>
<td>SPIRIT: Compute a time-varying linear transform to reduce dimensions.</td>
</tr>
<tr>
<td></td>
<td>Transforms</td>
<td>Wavelet transform: Provide a time-space multi-resolution representation.</td>
</tr>
<tr>
<td>Modeling and Evaluation</td>
<td>Classification</td>
<td>Very Fast Decision Tree: Perform incremental decision tree learning.</td>
</tr>
<tr>
<td></td>
<td>Clustering</td>
<td>Microclustering: Build online summary data structures for clustering.</td>
</tr>
<tr>
<td></td>
<td>Regression</td>
<td>Stochastic Gradient Descent: Update a linear regression model incrementally.</td>
</tr>
<tr>
<td></td>
<td>Frequent Pattern Mining</td>
<td>Lossy Counting: Retain approximate counts of frequent items in a stream.</td>
</tr>
</tbody>
</table>

Table 1: Some streaming algorithms for knowledge discovery.


