Design principles for developing stream processing applications

Deepak Turaga\(^1\), Henrique Andrade\(^1,\)\(^*\),\(^ †\), Buğra Gedik\(^1\), Chitra Venkatramani\(^1\), Olivier Verscheure\(^1\), Jeffrey David Harris\(^2\), John Cox\(^2\), William Szewczyk\(^2\) and Paul Jones\(^3\)

\(^1\)IBM Research, 19 Skyline Dr, Hawthorne, NY 10532, U.S.A.
\(^3\)HM Government, London, U.K.

SUMMARY

Stream processing applications are used to ingest, process, and analyze continuous data streams from heterogeneous sources of live and stored data, generating streams of output results. These applications are, in many cases, complex, large-scale, low-latency, and distributed in nature. In this paper, we describe the design principles and architectural underpinnings for stream processing applications. These principles are distilled from our experience in building real-world applications both for internal use as well as with customers from several industrial and academic domains. We provide principles, guidelines, as well as appropriate implementation examples to highlight the different aspects of stream processing application design and development. Copyright \(2010\) John Wiley & Sons, Ltd.

Received 18 February 2010; Revised 26 May 2010; Accepted 31 May 2010

KEY WORDS: design principles; stream processing applications; System S, Spade

1. INTRODUCTION

As the world gets more interconnected and instrumented, there is a deluge of digital data coming from various software and hardware sensors in the form of continuous streams. Examples can be found in domains ranging from financial markets, manufacturing, healthcare, traffic systems, large-scale infrastructure security, to scientific and environmental domains such as radio astronomy and water management [1–5]. Consider the set of sample applications shown in Figure 1.

They include stock ticker analysis for market-making, process control for manufacturing, analysis of various sensors streams in natural/physical sciences, multi-modal surveillance for law-enforcement, fraud detection in different settings, physiological sensor monitoring for healthcare, and call detail record processing for telecommunications. The need for and scope of these applications continue to grow rapidly. In all the above diverse domains there is a growing need to gather, process, and analyze these new data streams to extract insights in real time. While the needs are very clear, there are several data processing challenges that need to be overcome for this to be enabled. First, data sources (including streaming sources as well as data repositories that contain context and history) tend to be distributed across several locations with varying levels of connectivity. Second, the generated streams are heterogeneous in format, content, rates, and information noise levels, making processing and analysis difficult. Data streams may also consist of unstructured data types such as audio, video, and text that cannot easily be handled using traditional
data management infrastructures. Third, the types of analyses that need to be performed range from simple to arbitrarily complex, including several sophisticated mining and learning techniques. Fourth, the high volume of streaming data often makes it impossible to fully store and process all the data from disk. Last but not the least, any solution using a distributed processing infrastructure must be calibrated to provide adaptation to dynamic data rates, varying data characteristics, and resource availability to scale up and meet performance objectives such as latency and throughput.

The emerging stream computing paradigm not only addresses these challenges but also enables the extraction of new insights from data in real time. Several stream computing platforms [6–11] have been developed recently. They provide support for streaming and stored data ingest, flexible and extensible analytics on the streaming data, and system support for distributed, scalable, dynamic, and high-performance computing. The availability of live streaming data and more importantly, the ability to process and analyze all of this data as it is streaming, enables more informed decision-making and extraction of new insights in real time. This directly results in value in many contexts such as customer retention [12], waste reduction [13], profit increase [14], fraud detection [15], and pattern detection, providing the insights needed to leverage new business opportunities. All these can ultimately provide a competitive edge to businesses, increased efficiency to governments, and fuel new scientific discoveries.

Despite the clear benefits of this new computing paradigm, it is non-trivial to design and develop applications that can utilize the available computing infrastructure efficiently to perform the required analytical processing on the data. Application developers need to perform intelligent application decomposition, i.e. careful mapping of application requirements onto processing flow graphs of operators, design and implement individual components of the processing, distribute and deploy the application across the processing infrastructure, and finally tune the performance over the lifetime of the application. Additionally, since applications are typically continuous and long-running, developers also need to account for dynamic adaptation in response to data and/or processing variations. These characteristics require a shift in the developer thought process and the engineering methods employed during the application design, development, and evolution.

In this paper, we explore and document various design processes and principles in developing stream processing applications, ranging from application decomposition to development, and performance tuning and adaptation. We base these principles on our experience in developing the IBM System S middleware, a stream processing runtime system; Spade, its accompanying...
distributed application composition language; as well as our hands on work in building several real-world applications from diverse domains using this computational infrastructure. We present discussions on the characteristics and classes of real-world stream processing applications that led us to develop the fundamentals of application design (understanding requirements) and implementation (guidelines and implementation patterns).

This paper is organized as follows. In Section 2, we discuss the different facets of stream processing using several real-world application examples. In Section 3, we introduce System $S$ and its programming language Spade as the specific stream computing platform that we use for describing our implementation patterns in the later sections. In Section 4, we present different guidelines and implementation patterns used to direct the stream processing application development. This section represents the core of our contribution in this work as we discuss high-level development principles that lead to the specific implementation patterns presented using Spade and System $S$. We conclude this paper with a discussion of the related work in Section 5, and a set of final remarks in Section 6.

2. FACETS OF STREAM PROCESSING APPLICATIONS

In this section, we present the key characteristics of stream processing applications that distinguish them from the traditional data analysis, and store-and-process applications. We will use these characteristics to derive the design principles and implementation patterns that we describe in Section 4.

Specifically, there are four important facets (or properties) that characterize stream processing applications—continuous data sources, continuous and long-running analysis, time-to-respond performance requirements, and failure tolerance requirements. We describe each of these in detail below.

- **Continuous Data Sources**: These data sources generate streams of data, both structured and unstructured, that flow continuously, and potentially have no end. A stream is a sequence of data items that have some notion of time or order, i.e. either timestamps associated with time of generation or ingest, or a time-to-live property, or sequence numbers or arrival order. In most cases, stream data sources generate live time series data such as discrete samples of real-world signals, transactions or different types of event streams. As most data sources are external to the stream processing system they may not be controllable either in the data rate, sampling rate or order of arrival of data items. Note that data stored in repositories may also be played back as a stream if its time property is preserved.

- **Continuous Analysis**: Stream processing applications are driven by continuous and long-running analysis requirements. This is unlike traditional database/store-and-process systems, where the data is static and queries are short-lived, running through the data to completion. Data in stream processing applications needs to be continuously processed using the appropriate analytics, to generate a continuous stream of output results. Analytics and processing need to be real time and incremental to handle the streaming data items, as data needs to be processed ‘in motion’. Additionally, the long-lived nature of applications requires analytics to dynamically adapt to time-varying resource availability, data characteristics, and user objectives.

- **Performance Constraints**: Stream processing is used in real time, critical path, business intelligence applications, leading to several performance constraints in terms of latency, throughput, and dynamics. The processing needs to keep up with data ingest rates, to provide answers as quickly as possible, with as high a ‘quality’ as possible\(^1\), and to adapt to dynamic variations in the system and data. This leads to several interesting challenges both in analytics, resource management strategies, and the interactions between these.

\(^1\)Quality of results is strongly application dependent, but can be roughly equated to the accuracy of the analysis results. Stream processing applications inherently require the fine-granular tradeoff of result quality against currently available input data and resources. As data or more resources become available result quality can often be improved.
State Management and Failure Resilience: Stream processing is often used in environments with heterogeneous and distributed processing infrastructure with varying levels of connectivity and storage. As we mentioned, many of these applications are also central and critical to an institution’s overall IT infrastructure. Given these two factors, applications have to be built to cope with issues such as data loss, corruption, and reordering that can occur due to processing infrastructure, analytics, or data source failures. Hence, there is a need for graceful degradation of results in response to failures, as it is not possible to use store-and-process methods to replay the data. Additionally, care needs to be taken to manage internal state of continuous and long-running applications to prevent failures caused by unbounded memory utilization. This requires a careful design of state management using appropriate deletion strategies or combinations of in-memory and disk persistence.

It is important to note that while the above are often dominant characteristics of stream processing applications, they do not represent either necessary or sufficient conditions for an application to be characterized as stream processing. Several key design and implementation challenges for stream processing applications are driven by combinations of these characteristics, as described in the context of several classes of applications, in the following section.

2.1. Stream processing application scenarios

In this section, we describe a non-exhaustive set of stream processing scenarios that highlight the different characteristics of stream processing applications. As part of each scenario, we also include examples of real-world applications that exhibit the related characteristics.

2.1.1. Performance-driven application scenario. The key requirements of applications in this scenario are performance-driven—specifically in terms of throughput, latency, and high availability. All data ingest, pre-processing, analysis, and dynamics are driven by these needs. There are strong requirements to keep up with ingest data rates, maximize end-to-end throughput, and minimize processing and communication latency. These applications are hence engineered for peak loads that can often be 3 to 4 times the average incoming traffic rate. These applications also often need to be fault tolerant or partially fault tolerant \[16–18\], requiring the implementation of hot-standbys, checkpointing-based state management and end-to-end consistency checking.

Data sources in these applications are often high rate with well-defined data stream attributes, and with rate variations, and growth properties usually predictable from the historical data. These applications are often used at the front-end in different business environments, and may need to ensure data consistency and completeness for regulatory and provenance reasons. The emphasis on the performance also limits the use of complex analytics with variable resource consumption, instead these applications prefer low-complexity online pre-processing, cleaning, filtering, transformation, and analytic algorithms that make decisions by applying simple models to data.

One such application involves mediation of call detail records (CDRs) in telecommunication networks \[12\]. CDRs are network-generated events that provide real-time summaries of voice, short-message service (SMS), and multimedia message service (MMS) calls between customers and include details on caller, callee, duration, and quality of service. It is critical for the telecommunication network to capture and process CDRs for billing and regulatory reasons, for fraud prevention, and for overall customer service-related applications. Mediation involves ingesting CDRs, converting them into a standardized format, and processing them with different data transformations and filtering rules to extract multiple attributes used in subsequent applications. As this is the front-end for all the additional data processing, it is required to have \(24\times7\) uptime with strong requirements on failure resilience. The large and growing volumes of CDR data being generated in current mobile phone networks require the use of large-scale stream processing on distributed compute architectures for mediation.

Several such applications are also prevalent in the financial services front-office, where increasingly large amounts of market data need to be processed with millisecond latencies to determine trading opportunities \[14\]. Streaming data feeds are obtained from stock exchanges and processed
using extremely lightweight and responsive analytics that are dynamically reconfigured at controlled intervals. An example application involves option market-making [14]. This application must calculate and present option prices to the market and requires a real-time snapshot of the market data to perform its calculations. Market data consisting of individual stock and options trade and quote transactions, and cross-exchange consolidation transactions are processed in real time to identify price differentials and arbitrage opportunities. This translates into performance requirements in terms of decoding, demultiplexing, and delivering peak rates, of millions of messages per second to pricing engines, with sub-millisecond end-to-end latency.

2.1.2. Exploration-driven application scenario. The dominant characteristic of applications in this scenario is data exploration—specifically prospecting the different data streams as well as data sources to identify new and insightful pieces of information ‘relevant’ to the exploration. Data analysis requirements drive the data ingest, processing, discard, as well as dynamic reconfiguration, failure-resilience, and resource adaptation strategies.

The set of available and relevant data sources for these applications is typically large, potentially unbounded, and time varying. Given a finite amount of computation resources, decisions on which sources to ingest data from, and what fraction of data per source to consider need to be driven dynamically by analytic results. There is often too much data to store and process, thereby requiring the design and optimization of applications such that the given set of resources can be used most effectively to produce the best results in a timely fashion. Analytics are required to be adaptive to handle the dynamic and potentially unbounded set of available data streams. There is a need for both supervised (e.g. classification, regression, signature detection) as well as unsupervised (e.g. clustering, anomaly detection) analysis. Additionally, computational limits may require approximations in the processing to tradeoff accuracy against computational needs. These applications often use hypothesis-based analysis, where the processing is dynamically changed based on the results derived so far. Hence, based on validated hypotheses, different sources, different processing, and different analytic algorithms may need to be used dynamically. Performance requirements are driven by the need to most effectively use the available resources while supporting dynamic reconfiguration and adaptation of processing. Any latency and throughput constraints are mostly implicit, and driven by the need to effectively exploit resources. Additionally, these applications are also mostly ‘best-effort’ in nature, with limited failure resilience requirements. That is, they attempt to do as best as possible in identifying useful results with the resources that are available.

There are several applications in different domains that are primarily exploration driven. These include different scientific and environmental-sensing, as well as surveillance and cybersecurity applications. The broad need for exploration is also felt in financial services, telecommunication networks, and manufacturing sectors.

Consider a cybersecurity application for botnet detection that we are currently working on. This involves identifying both bot-master domains as well as malware infected hosts that are part of the command and control infrastructure. There are several sources of available information ranging from DNS queries to netflow traces to raw packet traces that need to be analyzed to detect malicious behavior, and as a result the bot networks. These different sources contain data with different underlying data types and information across a range of network protocols and time granularities. These sources may also be supplemented with additional information from the Internet community including expert and user collated lists of suspicious domains and blogs describing different types of bot behavior, all of which make the analysis extremely challenging. Finally, this exploration also needs to account for the adversarial setting, where bot-masters attempt to obfuscate their behavior dynamically in response to the designed detection and prevention strategies. Any designed analytics and detection strategies need to be continuously adapted to capture the time-varying nature of botnets.

A different application involves cosmic-ray burst detection in radio astronomy [19, 20]. As part of this application, scientists are interested in re-aligning and re-positioning observation instruments such as radio telescopes and other sensors when certain transient signals are detected [21]. Cosmic
ray bursts are extremely rare, very short duration signals that may appear from any direction in space. Detecting cosmic rays therefore requires real-time monitoring of a very wide field of view using multiple high-frequency sampling antennas. This is computationally infeasible, and hence cosmic ray detection may be posed as an exploration problem—where the decision on which portion of the field of view to sample is dynamically driven by the analysis of the signals received so far. This leads to an iterative exploration process with continuous refinement of the models to predict direction of observation. There is a strong need to dynamically compose and reconfigure the processing based on the online learning under dynamic data characteristics, and noise and interference levels, using a constrained-resource infrastructure.

2.1.3. Decision support and control scenario. In this scenario application requirements are driven by the needs of decision-making for process control. Stream processing is used to supplement traditional processing operations to analyze huge volumes of data in real-time and provide early access to relevant information such that a control decision can be made. In many cases, the decisions require a ‘human-in-the-loop’ as these may be used to control different types of physical processes. Data sources in these applications are partially controllable, with feedback based on the result of the decision affecting some aspect of the source behavior directly or indirectly. The use of these applications for process control imposes the need to have comprehensible analytics such as rule-based or decision tree-based analysis. Decision support and control-based applications are often sensitive to data loss and failures, hence there is often a strong requirement on failure resilience.

Statistical process control (SPC) in semiconductor manufacturing is one example application of this type [13]. The automated manufacturing and test equipment require active monitoring to ensure that they are operating under proper conditions. These conditions involve appropriate control of mechanical, physical, electrical, programmatic, and environmental aspects. Examples include proper tool calibration, probe instrument cleanliness and alignment, proper test limits and test parameters application during test, and proper environment temperature settings. SPC involves real-time monitoring and analysis of the manufacturing and test process to detect outliers, anomalies, and build statistical models for predictive control. Process engineers then use this information to dynamically modify the multi-step process to improve end-to-end product quality. Such control applications are prevalent in several manufacturing environments.

Decision support streaming applications can also be found in the healthcare sector. One example of a healthcare application involves real-time data analysis of physiological sensor streams in intensive care units for nosocomial infection prediction [22]. Analytics in this case are driven by rules specified by doctors, and the results are used by doctors and other medical personnel to diagnose, and control the infection. In these applications, all data and results are usually persisted for conformance and provenance reasons.

2.1.4. Simulation-driven application scenario. In this scenario, stream processing is used primarily to analyze historical pre-recorded time-series, or artificially generated data streams, to perform back-testing of analytic models, or to provision large-scale solutions or to perform scientific exploration and simulations. In these applications, generated or pre-recorded data must be replayed while emulating the inter-arrival and stream characteristics faithfully, including statistical properties of the stream attributes. Such applications require generic mechanisms and operators for delaying and synchronizing operations, thus mimicking flows and rates observed in practice. It may also be necessary to replay the data at rates much faster than real time to verify provisioning and performance requirements. The analytics and performance requirements are variable, and depend on the resources available and the needs of the simulation itself. There is, however a strong requirement on ensuring that the simulation experiments are repeatable and any results reproducible. Additionally, as these applications are focused on back-testing, there is an emphasis on system state visualization, and active debugging.

Simulation-based application design is common to many domains ranging from financial services to cybersecurity to manufacturing to scientific analysis. Specifically in the financial services sector, new market analysis algorithms are typically first tested on replayed historical data.
Table I. Applications developed with System S.

<table>
<thead>
<tr>
<th>Application</th>
<th>Functionality</th>
<th>Size</th>
<th>Dominant Facets</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAB [13]</td>
<td>Semiconductor manufacturing process control</td>
<td>Collection of 10 applications, with approximately 1500 lines of Spade and Perl code, and 20000 lines of C++ code. Custom operators also use OpenCV [23] and VFML [24] libraries</td>
<td>Decision support, control</td>
</tr>
<tr>
<td>Financial trading [14, 25]</td>
<td>High-frequency trading infrastructure</td>
<td>Approximately 900 lines of Spade and Perl code, using about 3000 lines of user-defined operator C++ code</td>
<td>Performance-driven</td>
</tr>
<tr>
<td>DAC [15]</td>
<td>Multi-modal fraud prevention</td>
<td>Approximately 35 000 lines of mostly C++ code—this application was implemented using an early version of System S</td>
<td>Exploration-driven, decision support</td>
</tr>
<tr>
<td>t-convolve [19, 20]</td>
<td>High-speed radio astronomy imaging</td>
<td>200 lines of Spade code, plus 600 lines of C++ user-defined operator code</td>
<td>Performance-driven, exploration-driven</td>
</tr>
</tbody>
</table>

2.1.5. Summary. The characteristics summarized in this section demonstrate that stream processing applications typically require a combination of adaptive data ingest, processing, and analysis, under different types of latency, throughput, and fault-tolerance constraints. Understanding these requirements is critical to the development and deployment of stream processing applications.

In Table I, we include references to applications we have built over the last few years that embody the application facets discussed in this section as well as some of the design principles discussed later in this paper.

3. FOUNDATIONS: THE SYSTEM S PLATFORM

While databases and data warehouses, which are the cornerstone of the store-and-process paradigm, have been effective in addressing the data processing requirements of applications over the last few decades, emerging streaming workloads require a new architecture as well as new principles for application development. The underlying architecture and computational requirements for these workloads are addressed by different large-scale stream processing systems developed over the last few years. In this section, we describe the architecture and characteristics of System S [15, 26, 27], a stream processing middleware from IBM Research that supports high-performance, reconfigurable stream computing. We will use System S and its application development framework Spade to highlight the different patterns and implementation examples in the remainder of this paper.

System S supports structured as well as unstructured data stream processing and the execution of multiple applications from multiple users simultaneously. Structured data is typically characterized by a well-defined schema such as trade and quote transactions from the stock market [14], as opposed to unstructured data, where feature extraction, parsing and additional processing might be required to retrieve the relevant tidbits or feature vectors to be processed by the application [15]. These applications can be scaled to a large number of compute nodes and can interact at runtime through stream importing and exporting mechanisms. System S applications take the form of dataflow processing graphs as seen in Figure 2. A flow graph consists of a set of processing elements (PEs) connected by streams, where each stream has a fixed schema and carries a series of tuples. The PEs are containers that host operators implementing data stream analytics, and are distributed on compute nodes. Compute nodes are organized as a shared-nothing cluster of workstations or as the execution nodes in a large supercomputer such as the IBM Blue Gene. PEs communicate with each other via their input and output ports, connected by streams. The PE ports as well as streams connecting them are typed, employing application-defined type systems, which are built by combining system-supported basic types. These types range from numeric types such as 8-bit integers to complex types such as vectors of basic type elements, for example, a vector of double precision floating point numbers. PEs can be explicitly connected using hard-coded links.
or through implicit links that rely on properties of streams such as ‘streams that carry surveillance video from cameras located in the JFK airport in New York’, which can be subscribed to. The latter type of connections is dynamic. The subscriptions are evaluated dynamically, as applications are submitted or removed from the system and the resulting connections are made upon the availability of a stream that matches a subscription expression. This mechanism allows System S to support incremental application development and deployment. Besides these fundamental functionalities, System S provides several other services, such as fault-tolerance [17], scheduling and placement optimization [28], distributed job management, storage services, and security.

3.1. The streams processing language

Spade [29, 30] (Stream Processing Application Declarative Engine) is the stream processing application development framework for System S. Spade provides a rapid application development environment including design and debugging tools as seen in Figure 3. Spade includes several key functionalities:

1. A language to compose parallel and distributed stream processing applications, in the form of dataflow graphs. The Spade language provides a stream-centric, operator-level programming model. The operator logic can be implemented in a lower-level language, such as C++ or Java, whereas the Spade language is used to compose these operators into logical data flow graphs.

2. A compiler that is able to coalesce logical dataflow graphs comprising multiple operators into a collection of physical processing elements that are more appropriate for deployment on a given hardware configuration. This is achieved by fusing several operators and creating multi-operator processing elements that ‘fit’ in the available compute nodes.

3. A type-generic streaming operator model, which captures the fundamental concepts associated with streaming applications, such as windows (i.e. a collection of incoming tuples that are buffered for processing) on input streams, aggregation functions on windows, output attribute assignments, parameters that are configurable via expressions, and punctuations (i.e. stream markers denoting logically related tuples) in streams.

4. A stream relational toolkit of operators implementing relational algebra operations in the streaming context. This toolkit includes the following operators: Join (for correlating two streams based on windows), Aggregate (for aggregating tuples based on grouping attributes and a window definition), Functor (for performing selection, projections, and simple tuple transformations), and Sort (for ordering tuples within a window).
Figure 3. StreamsStudio—the rapid application development environment for System S.

Figure 4. A sample Spade application.

```
(Application)
AuctionProcessor trace

(Program)
# A Source operator to read from the auction bids file
stream Bids(bidder: String, product: String, bid: Float)
   := Source("file:///auction_bid.dat", csvFormat)

# A Source operator to read from the auctioned products file
stream Products(product: String, offer: Float)
   := Source("file:///product_auction.dat", csvFormat)

# A Join operator to match and correlate tuples from the two streams
stream MatchingProducts(schemaFor(Bids), schemaFor(Products), priceDiff: Float)
   := Join(Bids<count(30)>; Products<count(0)>)
   [ $1.product==$2.product & offer<$1.bid ]
   { priceDiff := bid - offer }

# A Sink operator to write the results to a file
nil ::= Sink(MatchingProducts)("file:///auction_result.dat", csvFormat)
```

(5) A broad range of edge adapters, which are used to ingest data from external sources and publish results to external consumers, such as network sockets, databases, file systems, as well as proprietary middleware platforms.

(6) Support for extending the language with new type-generic, highly configurable, and reusable operators. This enables third parties to create application or domain-specific toolkits of reusable operators.

3.1.1. Brief introduction to Spade syntax. In the subsequent sections, we use small Spade snippets as part of the figures that illustrate the data flow graphs for the design patterns under discussion. Although the readers are not required to know the details of the Spade syntax, a brief introduction to the syntactic structure is helpful for better interpretation of these figures. For this purpose, we will describe a simple application, named Auction Processor, shown in Figure 4.
This application consists of four basic operators, which in concert, provide a simple emulation of a trivial auction management platform. The application consists of two source operators, one join operator, and one sink operator. One source operator, which has a single output port, creates a stream called Bids. This stream is created by reading tuples from a file in comma-separated format. This source operator serves as an edge-adapter, bringing in external data into the Spade application. In a deployment setting, the source operator could be configured to read from other streaming sources, such as sockets, RSS feeds or message queues. The other source operator is configured similarly, except that it reads from a different file and creates the stream Products. Bids and Products streams have different schemas. Bids contains tuples that represent bids on the products and Products contains tuples that represent products that are being auctioned. These two streams are fed into a third operator, a stream-relational join, matching the auctioned products and bids on these products. The join operator defines a window of size 30 tuples on its first input port and an empty window on its second input port (input ports are separated by a semicolon}). Effectively, for each auction tuple received, the join operator looks at the last 30 bids and outputs the ones that satisfy the join condition. The join condition is specified as matching product names and bid price being greater than or equal to the offer price in the auction. The join operator has a single output port, which generates the stream MatchingProducts. Finally, this stream is fed into a sink operator, which has a single input port. This sink operator is also an edge adapter. It is configured to write the incoming tuples representing the actual completed sale transactions to a file.

With the System S middleware and the Spade programming model serving as a basis for illustrating our ideas in the context of a real-world stream processing system, we now shift our focus to design and implementation principles for addressing the challenges associated with the various types of stream processing applications outlined in Section 2.

4. GUIDELINES AND IMPLEMENTATION PRINCIPLES

As described in Section 2, different types of stream processing applications need to handle different types of data sources, require distinct types of analysis, and have varying constraints on adaptation and performance. Hence, designing stream processing applications requires understanding these requirements clearly, and tailoring the application design appropriately. Furthermore, the scaling needs of stream processing applications might require distributed processing that pose additional challenges. Hence, the design of distributed stream processing applications borrows from many techniques in the high-performance computing and distributed computing domains.

In this section, we describe high-level principles and implementation patterns for designing and developing stream processing applications. We group these principles and implementation patterns into five categories: data ingest and handling (Section 4.1), data stream analysis (Section 4.2), dynamic adaptation (Section 4.3), performance optimization (Section 4.4), and state management and fault tolerance (Section 4.5). These aspects correspond loosely to the different types of application facets outlined in Section 2. While these guidelines are for the most part applicable to other stream processing platforms as well as to other stream programming languages, we chose to illustrate them with actual code examples written in the Spade language to make our discussion concrete. This section is organized as follows. We first discuss the underlying issues associated with different requirements, posit a set of principles of application design, and demonstrate them using code excerpts.

4.1. Data ingest and handling

The data handling requirements of stream processing applications are driven by the input stream data rates, the stream source characteristics, and the application data loss tolerance. Note that

1Multiple streams connected to the same port are separated by commas (this is not shown in this particular example).
the data handling is also implicitly dependent on the analysis and performance requirements. In this section, we focus on the guidelines and principles related to ingesting, pre-processing, and reducing data from distributed stream sources to meet different application objectives.

4.1.1. Principles of edge adaptation. Stream processing requires ingesting data from several types of external data sources, such as distributed sensors and data repositories. We term this process ‘edge adaptation’ and it applies to both live and stored data. This involves adapting the data to a common format that is then used by the rest of the application processing graph for data shipping over the stream processing middleware.

The development of edge adapters requires an understanding of the nature of the interaction with a particular data source, e.g. whether the data is pushed or pulled, and the nature of how the raw data is organized, e.g. the data type and whether it is structured, unstructured, numeric, or categorical. Many stream processing systems, System S included, come with a set of built-in edge adapters supporting standard devices, interfaces, and data types, to simplify application development. However, given the potentially diverse set of source characteristics, data format, and network protocols, these systems also provide a set of well-defined interfaces that application writers can use to write new edge adapters, customized to their application requirements and their potentially proprietary data sources.

We can summarize principles of edge adaptation as follows:

- Match the edge adapter to its data source format and protocol.
- Consider the issues of push/pull and server client relationships between the source and the edge adapter.
- Design custom sources to match built-in APIs for ease of reuse and maintenance.
- Identify elemental units of processing and use those as the basis for mapping ingested data to tuples.
- Size tuples to avoid excessively small, i.e. only a few tens of bytes or excessively large, i.e. multi-megabyte tuples. This is to avoid large per-tuple overheads for small tuples, and high-serialization/deserialization costs for large tuples.

4.1.2. Edge adaptation implementation patterns. Most streaming middleware platforms include a range of built-in edge adapters that support multiple data formats (e.g. ASCII, binary, comma-separated), network protocols (e.g. TCP, UDP, HTTP), and pull/push requirements, whereby a source operator can act either as a server or client, matching the needs of the external data source. Specifically in Spade, the Source adapters support all the above, and can also be extended using User-Defined Functions (UDFs) to account for the specific parsing needs imposed by proprietary protocols and formats.

In this context, we will describe four implementation patterns, where specific strategies were used to enable the effective connection of a streaming application to their data sources: (1) the use of built-in operators with a mix of streaming and stored data; (2) the use of a custom extension to the Source operator to handle a specific proprietary source format; (3) a strategy for processing unstructured data streams, and (4) a strategy for tuple grouping and processing.

4.1.2.1. Edge adapters for stored and streaming data. A common need when processing live data is to analyze it in the context of historic or at-rest data stored in repositories. The stored data usually consists of aggregated and summarized historical information as well as ‘static’ configuration data such as lists of entities (e.g. people or sensor information), and locations (e.g. location of roads or terrain features). For example, a credit card processing application may be looking for suspicious credit card transactions in a live stream of transactions, based on analytic models built from previously culled data.

In the Spade example shown in Figure 5, a TCP-based Source operator, functioning as a server (note the stcp:// notation in the URI) is used to receive an incoming socket connection carrying credit card transactions. These transactions are transformed by the operator and output via a stream named input. The stream has schema corresponding to the user id (id), transaction
amount (amt), timestamp (ts), merchant name (mn), and location (loc). In this application, the credit card transaction stream is to be analyzed in the context of the credit card user’s past transaction history. Hence, we need to enrich the stream with this past history, which is statically available in a repository. The Spade language includes a toolkit of ODBC-based adapters that can connect to different types of databases.

In the example depicted by Figure 5, enriching the transaction with the user information is performed by the ODBCEnrich operator that pulls information from the repository appropriately. Note that the choice of operators and their configuration (e.g. as TCP server) is driven by the need to match the application requirements. It is important to understand the impact of these on the throughput, latency, and the overall performance.

4.1.2.2. Using custom extensions for edge adaptation. External streaming sources are very diverse in how data is packaged. Even when a streaming platform such as System S provides the adequate device support for edge adaptation, its default data encoding might not be exactly what is required by the external source. In fact, frequently the data produced by these sources might originate from physical sensors or from software platforms where messages are encoded in platform-specific formats that are difficult to change. The same problem might arise when the streaming platform is producing results for external consumption. Therefore, it becomes necessary to make adjustments as data is being ingested or output.

One approach for addressing this need is to employ custom-built logic for performing the necessary data transformation. However, it is preferable to extend a built-in operator with custom logic, when possible, instead of re-implementing a completely custom operator. Spade provided a means to extend built-in operators with logic implemented by user-defined functions (UDFs). Hence, the built-in Source operator may be used for interacting with a physical device (e.g. a socket connection or interactions with a distributed file system), while the UDF can be used for custom parsing of the input data. This marries the advantages of built-in operators such as high-performance, reliability, and generality, with the flexibility of custom processing code.

Consider an example of using a UDF in conjunction with a file Source operator to parse custom data formats, shown in Figure 6. In this example, ASCII-encoded text data is ingested from a file with a device-specific encoding (fixed length fields) for each incoming message, in this case call data records (CDRs) generated by a cell-phone switching system. As seen in the code excerpt, a user-defined function is used to customize the operator and parse the incoming messages, transforming them into System S tuples.

4.1.2.3. Handling unstructured data. Data such as audio and video are naturally packaged as streams, as are web clickstreams, software logs, chat logs, e-mail streams, Twitter streams and network traffic traces. Many of these streams are unstructured, in the sense that messages might not follow one particular schema either because there is a large variety of message formats embedded in the raw stream or because the data includes free-form messages. A key aspect in applications
that must ingest this type of data is in how to actually carry out the data ingestion task since an
individual message may require sophisticated parsing, in some cases or the individual messages
might have an intricate internal layout that might not directly map to the streaming platform’s
supported data model. This issue can be dealt with in different ways. A common approach is to
ingest an opaque byte stream and embed in the streaming application the knowledge to decode
the unstructured data. We illustrate this approach using a System S example.

In System S, in addition to typed tuples, the Spade language permits declaring and manipu-
lating low-level data in the form of a sequences of bytes. Serializing complex unstructured data
into lists of strings or bytes allows the system transport and data handling to be agnostic of the
data content. Different algorithms for decoding, parsing, and analyzing the data may then be
implemented within custom operators or functions, similar to what we demonstrated with UDFs.
This approach can be further extended by making use of pre-existing libraries that understand
the original data format. Commodity-off-the-shelf text analytics are examples of such libraries.

In Figure 7, we show an example application that receives a JPEG image stream (as a sequence
of bytes) over the network using a UDP Source operator, and then uses a user-defined operator
which makes use of the libjpeg library [31] to decode the streamed images. In the example,
a successive operator then uses a computer vision library (e.g. OpenCV) to perform complex
operations such as image segmentation and analysis.

4.1.3. Grouping tuples for processing. In many applications, while data might arrive intermittently,
segments of streamed data must be collected into windows that have to be processed together across
multiple analytic steps. There are different ways of grouping together different incoming tuples.
One possible approach to achieve this is to aggregate the appropriate tuples into one larger tuple—
by collecting the individual samples into a tuple that includes a list of the individual values held
by the original incoming tuples. Spade provides support for several different ways of aggregating
tuples using the built-in Aggregate operator that allows different aggregation window definitions.

In certain cases, this approach may not be efficient in terms of data transport or memory
requirements—especially when the aggregation needs to collect a large number of tuples. An
alternate approach involves the use of punctuation markers—a punctuation is an out-of-band marker that separates different logical windows of tuples. These punctuations create appropriate demarcations in the stream and provide means of indicating tuple groupings without the need for aggregation into large tuples.

Punctuations are supported by several streaming platforms. Specifically, in Spade programs, punctuations can be inserted into the stream based on custom processing requirements, either using a built-in Punct operator (which outputs a punctuation based on user-specific logic) or by using custom logic embedded inside user-defined operators. Punctuations may then be used by downstream operators (including Aggregate, Join, Punct, user-defined operators, etc.) to aggregate, filter, or process the tuples grouped together as desired. An example of inserting and handling punctuation is shown in Figure 8.

In this example, we use a Punct operator to insert punctuations whenever the current tuple’s id attribute does not match the previous tuple’s id, thereby creating a window containing all consecutive tuples with the same identifier. We then use an Aggregate operator to compute a summary statistic (in this case, the standard deviation) for this one group, thereby also reducing the data volume.

The choice of punctuation with small tuples versus collecting samples into one large tuple is usually driven by specific application requirements (whether individual messages need to be processed independently) and performance requirements (serialization costs versus memory requirements).

4.1.4. Achieving data reduction. In many stream processing applications, data enters the system at very high rates and in a bursty manner, as external data sources cannot always be controlled to provide a filtered/smoothed data stream. Additionally, the incoming stream is potentially at a higher granularity than, and is a superset of, what the application analytics require. For instance, the output of a radio-telescope sensor contains mostly noisy data at data rates in the order of tens of Gbps. This imposes several I/O and computational stresses on the system, especially at or close to the ingestion point [32], thereby requiring different mechanisms to keep up with the data rates, and to filter, pre-process, clean, and reduce the data volume.

The first step in coping with high data rates is to perform basic low-cost filtering as well as load balancing, spreading the work across multiple operators, potentially distributed over multiple compute nodes. This progressive filtering approach takes advantage of pipelined parallelism, where later stages in the pipeline can perform more expensive tasks, thanks to the rate reduction performed by the earlier stages. This also allows incremental information extraction to keep up with high data rates, enabling operators close to the sources to parse just enough information to allow early discard of irrelevant data. Later stages may then extract and expose more information as necessary on the lower-volume data stream. In other words, a rough, but quick, algorithm is applied to pre-classify the incoming data and route it to a more in-depth downstream analysis.

On the other hand, a load balancing strategy coupled with data and task parallel approaches may be used for effective processing and data reduction without data discard [14]. In this case, the data is routed to different chains that either (i) perform the same processing on different tuples at reduced rates (data parallelism) or (ii) perform different processing on the same or different tuples in parallel (task parallelism). Indeed, in many cases data streams are inherently parallelizable. For instance, stock exchange data streams, come pre-split into multiple channels, while some others can
be trivially parallelized since multiple logical streams are multiplexed into a single physical data source, e.g., streams of trading transactions relating to IBM in a stock market feed. It is important to note that parallelization may not be feasible in all stages of the processing [14, 32], especially if correlations need to be drawn across data streams as is the case in many high frequency or algorithmic trading applications or when data ordering must be preserved.

Further data reduction can be achieved via the use of approximation techniques. For instance, data reduction through approximation might be preferred in resource-constrained exploration settings, where not all the data can be analyzed effectively. Techniques for approximation include, but are not limited to:

- **Sampling**, which involves selecting representative data samples out of a larger group of samples, potentially discarding those samples whose processing results in minimal impact on results [33]. Several techniques for uniform, non-uniform and adaptive sampling have been developed in the signal processing and data mining community.
- **Quantization**, which involves reducing the fidelity of representing individual samples using either scalar, vector, uniform, or non-uniform techniques.
- **Summary Statistic Computation**, which involves computing different summary statistics from one or more data samples that may be used for quantitative data analysis and visual exploration. Examples of summary statistics include statistical means (e.g., arithmetic, geometric, harmonic), moments (e.g., mean, variance, skew, kurtosis, higher-order moments), densities (e.g., histogram), coefficient of variation, counts, distinct counts, quartiles, top-k values, contingency tables among others [34] that capture the characteristics of a stream.

Besides these data reduction techniques, it is also necessary to employ resource-adaptive analytic algorithms that adjust the amount of processing to the available computational cycles. For example, such analytics can be found in the signal-processing domain, where the analytic may have several operating points, corresponding to different resource-accuracy tradeoffs. Variations and spikes in load can then be dealt with by employing appropriate data reduction and dynamic analytic adaptation techniques.

We can summarize different data reduction principles as follows:

- Use pipelined parallel processing with incremental analytics to progressively reduce data rates.
- Use task and data parallelism with load-balancing, when the application supports them, to reduce streams without data discard.
- Use different lossy data reduction and approximation techniques to further reduce data rates.
- Combine data reduction with resource-adaptive analysis to optimize the computation of derived results.

### 4.1.5. Data reduction implementation patterns

Different streaming platforms make available distinct mechanisms for data reduction. In general, we have observed that most of the platforms actually rely on application developers to build these mechanisms in their applications. On the other hand, a principal design tenet of System S involves scalability and support for widely distributed applications. Hence, a number of middleware and language-level mechanisms is built in to help with data reduction. For example, the Spade language includes support for different built-in operators for parallelization, and the middleware can host a fully distributed application where these operators might be running on several cores or distinct processing nodes [20, 32]. Similarly, the language enables the implementation of operators and their use to accomplish lossy data reduction.

To illustrate the data reduction techniques, we will discuss two implementation code excerpts, one based on parallelization and the other based on lossy techniques for data reduction.

#### 4.1.5.1. Parallelization for lossless data reduction

In many cases when lossless data reduction is necessary, a simple round-robin selection scheme on incoming tuples typically works well. Consider the application excerpt in Figure 9. In the example, the Spade language built-in Split...
operator is used to split the data stream into three streams ($filt0$, $filt1$, and $filt2$) based on a sequence number $seqid$. In this case, the split function is uniform, based on the modulus function, and is used to exploit data parallelism with three parallel and independent processing paths. In general, hash functions may be applied to the data attributes to generate the appropriate splits. Splits may also be defined by pre-computed lookup tables, e.g. groups of stock ticker symbols that should be routed to the same processing chain.

An issue with parallelizing with a split operation is that there may be requirements on either synchronizing the resulting parallel streams or reordering the tuples in the merged stream, requiring additional processing logic. Similarly, other parallelization schemes may also be implemented. A detailed discussion on the different parallelization schemes is included later in Section 4.4.

4.1.5.2. Lossy data reduction. Certain applications can tolerate losses when carrying out data reduction. Consider three simple examples of lossy data reduction shown in Figure 10. The input data tuples are streamed via a TCP connection and ingested via a built-in $Source$ operator. The samples contain a value, a sequence number, and a timestamp. The stream is then processed by employing three different reduction schemes. The first scheme (top) retains every fifth sample using regular sub-sampling based on the sequence number attribute, using a $Functor$ operator. The second scheme (middle) retains only the mean summary statistic over a tumbling window $\text{[size 1 second]}$ using a built-in $Aggregate$ operator. The third scheme non-uniformly samples the $\text{[size 1 second]}$ using a built-in $Aggregate$ operator. The third scheme non-uniformly samples the

---

A tumbling window [30] is flushed every time an aggregation is performed. In contrast, a sliding window implements an eviction policy whereby certain older tuples are evicted based on events such as the arrival of additional tuples.
data based on the error between the last received sample and the current sample. The language includes a stateful Functor operator which can be used to store state throughout the lifetime of the operator. In this case, this is used to retain the value of the last transmitted sample and pass only samples with sufficiently different values. Clearly, different data reduction schemes lead to different output rates, approximation errors, and aliasing effects, as is clear from the graph displaying the data samples before and after data reduction. The choice of the appropriate reduction scheme is driven by application requirements, impact of reduction on subsequent analytic algorithms, and performance goals.

4.2. Data stream analysis

The analytic requirements of stream processing applications vary significantly based on the application objectives. In this section, we focus on guidelines related to resource-adaptive data analysis using distributed computing platforms and analytic tools.

4.2.1. Principles of stream analysis. Stream mining applications generally use a variety of analytic algorithms ranging in sophistication from simple rules for detecting known patterns, to semi-supervised learning that involve a human in the loop, to completely unsupervised learning. In many real-world applications, stream mining requires applying models learned from historical data, to live data in order to identify patterns of interest or make predictions on different attributes of data. These models involve both simple thresholding and rule-based analytics, to complex algorithms such as Decision Trees, Support Vector Machines, Neural Networks, Transform Regression, etc. In such cases, there needs to be tight integration between processing new data to extract the appropriate features, collecting and storing the appropriate historic profiles of these features, model building and learning using data mining algorithms, and online model application and scoring on the live streamed data. Consider an example of mining logs from network intrusion detection devices to identify suspicious network activity. In existing frameworks, domain experts craft rules of the following type to identify patterns of interest, for example:

Trigger an alert if 20 or more events of particular type target a single destination IP address within a 2 minute window.

These simple rules are then used to identify patterns in real-time in the incoming live data coming from network monitoring probes. There are several tools available to users to build different types of models, including commercial software such as IBM InfoSphere Warehouse, Matlab, SAS, SPSS, as well as open-source tools such as R, Weka, among others. Indeed, there needs to be an ‘analytics life-cycle’ that allows taking these models, instantiating them on the streaming data, and iterating to improve models with time as it is common for these models to evolve. For example, spam classification and fraud detection must evolve as the adversaries find new ways to defeat current methods.

There is also a need for incremental learning of models online to adapt to time-varying properties of the data as quickly as possible. Incremental learning has been studied extensively in the data mining community, and several such algorithms have been developed (e.g. for incremental Decision Tree learning [13]). When applications rely on incremental learning, several aspects must be considered. In many cases, incrementally learned online models are likely to be less accurate than offline models (given the same training data), because incremental algorithms are constrained to a single pass over the data and have to trade-off learning complexity with resource availability. Hybrid approaches where online learning only modifies model parameters for models learned offline, have also been proposed. These approaches combine the advantages of offline learning with online adaptation.

Automated stream exploration requires sophisticated orchestration of anomaly and change detection, combined with correlation extraction, and using this extracted information to build new models that drive the next analysis cycle, iteratively. Although several techniques for time-series analysis, tracking, and anomaly detection are applicable, the end-to-end stream exploration problem remains an active research area.
In summary, these are some of the principles that apply to the design of the analytic part of a streaming application:

- Understand the analytic requirements in terms of model application, model learning, and exploration.
- Understand the accuracy–complexity tradeoffs of available models—both learning and scoring.
- Select simple, comprehensible models for the mining task.
- Support stream mining by leveraging offline analysis and model building.
- Build parameterizable models that can be tuned incrementally online.
- Use pipelined parallelism to partition complex analysis into multiple steps: coarse-to-fine processing.

4.2.2. Stream analysis implementation patterns. The implementation of analytics as part of more complex applications can be accomplished in different ways. Some of the existing stream processing middleware allow the creation of user-defined operators whereby generic algorithms can be encapsulated and become part of the processing chain of an application.

The Spade language defines a language extension mechanism designed around operator toolkits, used for developing different built-in operators for implementing new analytics [35]. A scoring toolkit for supporting complex data mining algorithms was developed using this extension mechanism and is available as an add-on. This toolkit integrates algorithms from the IBM InfoSphere Warehouse [36] using the Predictive Model Markup Language (PMML) standard [37]. The supported algorithms include both supervised learning approaches, where data labels and the ground truth are available, as well as unsupervised approaches where no ground truth is available.

We now describe two implementation patterns, one for a rule-based scoring approach and one with the application of a complex model to the streaming data using the Spade scoring toolkit.

4.2.2.1. Rule scoring. Consider the network intrusion rule described in Section 4.2.1. The implementation of rules of this type involves simple attribute comparisons and temporal aggregations, followed by filtering. A Spade implementation for this rule is shown in Figure 11. In this case, a developer can rely on basic stream relational operators (i.e. multiple instances of a Functor operator and an Aggregate operator) from the Spade stream relational toolkit (see Section 3) to implement the rule. Alternately, the developer may use a set of custom operators to implement the rule. In general, rules involve complex combinations of different types of

---

[PMML is a standard XML representation that allows specifications of different mining models and their ensembles. PMML is supported by several state-of-the-art statistics and data mining software tools available from several commercial vendors (e.g. SAS, SPSS, Weka).

*Supervised learning algorithms such as decision trees require the training data to be labeled with ground truth. The decision tree is then computed during an offline training phase based on the labels. The computed decision tree is used to score the live data, providing a decision for what category the live data belongs to.*
such processing, and different implementations, e.g. built-in versus custom, may incur different development costs, support varying levels of reusability and flexibility, and lead to different performance.

4.2.2.2. Scoring with complex models. When more sophisticated data mining techniques are required, the use of operators from other toolkits becomes necessary, but this is still simple from a development standpoint, once an application architect decides which mining approach to use.

A simple example of using a pre-built operator for classification is shown in Figure 12. In this figure, the input stream with attributes \(a\) and \(b\) is fed to a decision tree configured with the parameters stored in the PMML model from the \(\text{decisiontree.pmml}\) file to generate a tuple comprising the prediction \(\text{pred}\) along with a confidence \(\text{conf}\) score. Additionally, the operator also takes as input the stream \(\text{modelupdate}\) which may be used to replace/update the model at runtime to account for changes in data characteristics, or resource availability. As we have mentioned, the \(\text{Classification}\) operator is part of \text{Spade} scoring toolkit, which also includes \text{Regression}, \text{Clustering}, and \text{Associations} scorers. In this case, developers need to account for the analytic life-cycle, including training and development of the model on historical data, deployment on live data, and training new models as necessary.

4.3. Dynamic adaptation

Stream processing applications are continuous and long-running. They generally need to account for dynamics in the available set of data sources, data characteristics, analytic models, as well as processing resources. Additionally, for complex stream exploration applications, the objectives of the analysis may also change dynamically as a result of extracted information. Hence, live, continuous, and evolving data exploration requires dynamic adaptation along several axes.

4.3.1. Principles of dynamic adaptation. Adaptation requires adding configuration knobs to an application. For example, it is possible to implement such dynamic behaviors using a set of parameterized operators whose performance may be modified using appropriate control parameters. As an example, we may vary the error threshold used for the sampling operator in Figure 10 to dynamically change the sampling characteristics. Enabling such parameter-based adaptation requires designing and implementing custom code that tweaks the operators appropriately. Hence, applications may include orchestration operators that control the overall application behavior by dynamically modifying control parameters.

In some cases, simply tuning one or more operator parameters to change end-to-end processing or resource consumption profiles can be limiting. In general, adaptation may require the processing flowgraph to be modified to bring up new models or to extract new sets of features from the incoming data. To support true dynamics, the system needs to support dynamic reconfiguration of the processing flowgraph, including the ability to bring new operators up and down to create new dynamic connections between operators, as well as parameter adjustments for tunable operators. In particular, allowing for dynamic behavioral changes of applications via the establishment of
transient dynamic connections across applications has several benefits:

- Reuse of processing across applications: Dynamic connections enable reuse of processing at runtime, lowering the overall processing demands. For example, multiple applications may make use of a data classification step, e.g. candidate fraudulent transactions are fed to two different applications one to flag the transaction for deeper inspection and the second to trigger immediate action. This can be achieved by tapping into existing intermediate sources of streaming data as well as feeding data into applications that are already running.
- Discovery of new sources of data: Dynamic connections enable applications to discover new sources of data as these sources appear and disappear. Changing application interests can be accommodated by modifying their connection properties at runtime based on predicates on stream properties. For example, a traffic monitoring application might incorporate data from additional instrumentation provided by cell-phone users driving through a particular area.
- Incremental deployment and development of applications: Dynamic connections allow the development and deployment of applications incrementally, as functionality gets implemented or refined. This also includes support for replacing or bringing down existing pieces of an application, without bringing the entire application down. In several of the applications outlined in Section 2, their continuous operation makes such capability mandatory.
- Dynamic adaptation to workload characteristics: Dynamic connections also enable launching new application segments in reaction to changes in the workload conditions. This may involve increasing the number of processing chains for a given parallel segment of the application as the input data rate increases. For example, an application screening banking or credit card transactions for fraud might require deploying additional processing capabilities during peak times, faced during special retail events such as sales.

In summary, the key principles for dynamic adaptation include:

- Understand the dynamics of an application, its data sources, the streams the application generates, and its sensitivity to resource variability.
- Design parameterized and tunable operators whose behavior can be modified at runtime.
- Allow for dynamic reconfiguration of the processing flowgraph, including allowing different applications to connect at runtime.
- Be aware of the performance implications of dynamic behavior (e.g. the additional load on reused portions).
- Design applications to monitor performance metrics and the overall state of the system to trigger dynamic adaptation, when appropriate.

4.3.2. Dynamic adaptation implementation patterns. As we have discussed, dynamic adaptation might rely on the ability to create transient stream connections on demand. System S supports this and other ways of providing dynamic adaptation. Stream processing applications implemented on System S can use both parameter adaptation as well as inter- and intra-application dynamic connections, established based on dynamic subscription matching performed on stream properties at runtime, as discussed in Section 3.

In addition, the System S runtime provides interfaces to expose and collect the performance metric counters from the middleware components as well as from the applications that are deployed and running. The list of available metrics includes the number of applications currently running, low-level I/O and computation performance metrics from operators and processing elements, traffic patterns obtained from the input and output ports for each operator, as well as a multitude of operator-specific counters used to capture the dynamic state of several built-in operators. These capabilities enable the design of applications whose behavior can be changed based on a variety of environmental and workload changes as we will describe next.

4.3.2.1. Performance monitoring for adaptation. Having access to performance counters enables an accurate assessment of the state of the application, the streaming middleware and the underlying computational infrastructure—a pre-requisite to supporting dynamic adaptation. As mentioned
earlier, System S includes interfaces for accessing performance counters from within analytics operators, by using the name of the appropriate performance resource. System S also provides support for defining custom metrics inside user-defined operators and for exposing custom performance counters for them. This allows the development of system analytic or control applications that monitor the behavior of other applications using different performance metric counters, and then dynamically optimize them. These optimizations can include triggering the launch of additional sub-applications, or modifying the operating point of one or more operators in running applications.

4.3.2.2. Dynamic parameter tuning-based adaptation. In many cases, dynamic adaptation provides the means for controlling the acceptable amount of error that an application can tolerate. Consider the example shown in Figure 13 that extends the non-uniform sampling shown in Figure 10. The Functor operator that produces the subsampled filt stream now receives the data as well as the error threshold as input, and uses the error threshold to appropriately filter out the sample. The error threshold $T$ is modified dynamically by the user-defined operator implementing the system analytics strategy for the application, which produces a control stream, carrying control directives throughout the lifetime of the application. This operator monitors the output rate of the sampling operator and recomputes a new threshold to match the output rate to the rate constraint. The monitoring is performed by the operator by periodically querying for the appropriate performance counter from the output port of the filt operator and computing the instantaneous and average rates. This is shown using a dashed line connection in the figure. The controller itself can include any arbitrary control strategy, as required by the application. For instance, the operator may implement a proportional integral derivative control algorithm \[38\] to determine the optimal threshold.

4.3.2.3. Dynamic connection-based adaptation. As previously described, the Spade language and the System S runtime provide support for dynamically connecting operators from different applications to allow dynamic interconnections across applications. There are two different ways of connecting operators dynamically. The first involves name-based stream composition. In this case, predefined stream names are advertised by producer operators and registered with the System S routing infrastructure. At runtime, a different operator can then subscribe to the appropriate stream using the advertised name. Note that both the producer operator as well as the consumer operator may be instantiated at different times, and the connection between them is established at runtime. An example of the Spade code for two different applications is shown in Figure 14.

In the figure operator $B$ from application App1 exports its output stream, whereas operator myC from application App2 imports this exported stream by its name App1.B. The connection between these two operators is made at runtime, when both applications are instantiated.
An alternate mechanism for dynamic operator connection is provided by declaring and then advertising and registering qualitative or quantitative properties of certain streams with the System S routing infrastructure. A property declaration consists of the property name and the property value. An example of stream import and export with property declarations is shown in Figure 15.

In Figure 15, stream $B$ from application $App1$ exports two different properties: $f$ that takes the string value ‘eye’ and $g$ that takes the numeric value 1. The two importing applications use different mechanisms to import the stream. $App2$ imports all streams that have property $f$ whose value is either ‘eye’ or ‘ear’, whereas $App3$ imports all streams that have property $g$ with value less than 10. Both these applications will connect to the stream $B$ as desired. In practice, this capability is used to turn on and off certain portions of an application (e.g. dynamic trading strategies in finance engineering applications based on market conditions) as well as to deploy accessory applications for in-depth data analysis, when conditions in the data warrant further data inspection.

4.4. Performance optimization

In this section, we focus on the principles and implementation patterns for application performance optimization. These techniques include different application decomposition and parallelization strategies, as well as compile-time and run-time optimizations.
Second, this allows taking advantage of parallelism that exists within a computational node and across nodes, as operators can be distributed across nodes. Third, a modular design enables an easier porting of applications to different hardware configurations, while maintaining performance. In fact, the resulting separation of the physical decomposition from the logical decomposition of the application allows the compiler and middleware runtime to handle the physical mapping of the application to the hardware platform [28, 39]. Finally, writing large, monolithic operators is an obstacle to building highly available stream processing applications, as such operators become single points of failure.

Besides modularizing the analytics into operators, it is also important to modularize the end-to-end processing flow of an application. This can be accomplished by grouping together operators that are collectively used to achieve one function. The modularization also includes deciding which parts of the processing happen on the stream processing platform and which parts happen outside. For example, certain data management tasks are a better fit for a DBMS or data warehouse, especially for managing more stable data sets where complex query or transactional capabilities must be provided. This is also the case for offline model learning. Similarly, modularization can be performed at the application level in terms of deciding how applications can be split into sub-applications that are independently deployable. Finally, modularization can be performed hierarchically, grouping together operators into composite operators [30], and grouping composites into sub-applications to abstract the complexity of the overall data processing [40].

Another dimension of the performance optimization problem relates to the specific needs of an application, in particular, when it must be designed with more strict time-to-respond requirements. As stream processing applications are continuous in nature, the definition of response time is different from what is commonly used in query/response systems. In stream processing applications, the response time for a given incoming tuple is the time taken to produce the resulting tuple (or tuples) affected by that incoming tuple (and possibly others). In many cases, this translates into a latency measure defined on the dataflow graph, with a specific start and end point [14]. Keeping the latency low requires avoiding or minimizing the major sources of processing delays whenever possible, but it also requires considerable tuning and experimentation. We have found that the following guidelines are usually useful in addressing the low latency issue:

- **Disk operations**: Accessing disk on a per-tuple basis is often prohibitive whenever the incoming tuple rate is high. In-memory storage and processing, potentially across multiple hosts, should be preferred, if possible.
- **Batch processing**: Processing tuples in batch also introduces latency and, in many cases, incremental, single-pass versions of algorithms can be used to minimize the latency.
- **Heavy analytics**: Analytics that are computationally intensive can in many cases be accelerated via the use of multi-threading [20] or custom hardware support (such as SIMD co-processors) [35].
- **Synchronization**: Waiting on multiple events that can potentially arrive from remote, nondeterministic sources typically increases latencies. It is best to avoid such synchronization, whenever possible.

When focusing on low latency, the deployment layout of an application in terms of an operator to a PE and of a PE to a host becomes a critical consideration. The primary issue is the cost of data movement. In general, while splitting processing across multiple processes and hosts increases throughput, it might also increase the latency due to data transmission and marshalling overheads, because of the crossing of a process or host boundary. Usually extensive modeling and/or experimentation might be required to evaluate throughput and latency tradeoffs [41] in the context of a specific application [25]. Latency is also affected by the process of ingesting the results from the streaming application by an external consumer. In other words, if the results are not consumed sufficiently quick, it may cause back-pressure in the streaming application and introduce latency due to queuing at the data transport level. As a result, appropriate edge adapters should be used to deliver results to the consumers, in a timely manner, in some cases with additional queueing capabilities to prevent or minimize back-pressure.
Optimizing applications for throughput involves using different modes of parallelism, supplemented with multi-threaded operator implementations [20]. It is also important to use system optimizations both at compile time as well as at runtime. Furthermore, as streaming applications grow, operator pipelines get deeper with resulting loss in throughput. It is often harder to diagnose performance bottlenecks when running large applications, especially for a fixed throughput/latency target. Hence, it is important to develop and optimize applications incrementally, i.e. start with a few operators and gradually grow the graph, while keeping track of the throughput at each step. Equally important is the use of adequate tooling for application understanding so that optimization decisions are rooted on quantitative information [40, 42].

Another consideration that must be made is akin to optimizing the performance of any computationally high-demand application, i.e. whether to employ generic or special-purpose code. In fact, there are different configurability performance tradeoffs that need to be considered when deciding between using built-in operators versus developing customized user-defined operators. For instance, built-in operators often provide much higher configurability and ease of development, at the expense of extreme high performance that may be achieved by fully specializing an operator. Similarly, complete reliance on built-in operators (which might not precisely match the specific needs of an application) as opposed to custom built operators may lead to complex dataflows. A mix of custom and built-in operators based on the needs of the application works best.

Yet another performance-related consideration is brought up by dynamic composition scenarios as described in Section 4.3.2, when applications interact via transient stream connections. In these cases, it is important to monitor inter-application interactions, as the throughput of stream processing applications is often impacted by other applications that feed data to them as well as by those that consume data from them. Performance implications of these connections must be carefully considered.

The principles of performance optimization can be summarized as:
- Decompose applications into functional units; modularize the processing.
- Design processing given time-to-respond goals. Specifically, minimize the use of disk operations, batch processing, heavy analytics, and synchronization.
- Design processing to maximize throughput. Specifically, exploit parallelism and employ multi-threaded operators.
- Use system optimizations for compile-time and runtime optimizations.
- Consider incremental application development and optimization.
- Determine when to use user-defined versus built-in operators.
- Design reusable operators.
- Examine inter-application interactions to identify potential performance bottlenecks.
- Consider the state management requirements of different applications. Use appropriate state structures and repositories (such as in-memory databases) to lower an operator’s internal processing latency.
- Design the application fault tolerance strategy to match the application’s performance requirements (additional guidelines concerning fault tolerance are given in Section 4.5).

4.4.2. Performance optimization implementation patterns. Application optimization requires in-depth understanding of the application features, the middleware characteristics, the workload and how it varies through time, and of the runtime environment. In this section we provide different Spade examples to highlight different parallelization mechanisms.

4.4.2.1. Pipelined, task, and data parallelism. Decomposing the processing into parallel execution paths generally brings performance gains, especially when exploiting data parallelism with replicated identical execution paths. In some cases, task parallelism may be exploited by decomposing the application into non-identical execution segments, with different segments performing different operations on the data in parallel [32].

In task parallel decompositions, synchronization of the split streams from these different segments is usually necessary, before downstream processing can take place. An alternate
decomposition involves breaking up the processing into serial, but potentially pipelined components. This structure provides for pipelined parallelism gains (i.e. multiple data units can be processed simultaneously at different points in the pipeline). However, such an approach usually leads to increased latency as multiple operator boundaries must be crossed, sometimes incurring additional data marshalling costs, in cases where host boundaries must also be crossed.

To highlight the tradeoffs between different decomposition structures, we examine the following illustrative example. Consider that for each input tuple we need to inspect the values of two different attributes ($A$ of type $t_A$ and $B$ of type $t_B$) using complex user-defined logic before forwarding the result for downstream processing. A serial (pipelined parallel) decomposition is shown in Figure 16 and a task parallel decomposition is shown in Figure 17.

The task parallel decomposition involves two paths. One to inspect and process attribute $A$ from the stream and the other to process attribute $B$ from the incoming stream. Clearly, this requires additional synchronization to put together the two split streams before further processing. In Spade applications, this synchronization can be accomplished by the $\text{Barrier}$ operator, that forwards a merged tuple, only when it receives one tuple on each of its input ports.

The serial decomposition involves breaking up the processing into two steps. The first where attribute $A$ is processed and the second where attribute $B$ is processed. Clearly, such a decomposition requires no additional synchronization at the expense of potentially higher latency, as we have discussed. However, these tradeoffs and the specific parallelization strategy to use must be carefully measured, often with experimentation, considering the workload characteristics and the specifics of the processing carried out on each attribute. In an earlier work [43] we demonstrated that a hierarchical tree-based structure (using serial processing) with careful optimization of the parameters, significantly outperforms a task parallel processing structures, in terms of end-to-end accuracy under resource constraints for complex semantic concept identification in video streams. Results like these tend to be specific to the characteristics of the application, requiring careful analysis and experimentation to achieve. Other factors that need to be considered while designing the appropriate parallel processing scheme include the end-to-end processing delay, the need for data filtering, the need to maintain order among data tuples, and the need to account for processing mismatches between different operators or parallel paths.

4.4.2.2. Use of system-level optimization. As we have discussed, an important aspect related to streaming applications is that, in several cases, they are tasked with processing large volumes of data, which in turn may require carefully placing different components of an application onto distributed resources to accommodate the processing load. This task is non-trivial as there are...
several system-level optimizations that can be used while mapping the logical operator graph onto an actual physical deployment across a set of distributed nodes. To the best of our knowledge, most streaming platforms lack the necessary flexibility and tooling to accomplish this. Differently, System S includes both compile-time and runtime mechanisms to address this task [28, 39, 44]. From our experience, there are two key features that developers need to optimize large-scale applications. First, the decoupling of the logical configuration of an application, i.e. its logical flowgraph, from its physical configuration, i.e. the actual operator to processing element and processing element to host allocations. Second, an extensive profiling and associated tooling to enable the middleware and its compiler to make the appropriate mapping decisions, as well as tooling to enable developers and administrators to assess how adequate the automated decisions are.

In fact, the Spade compiler includes support for a two-stage optimization approach [28, 39]. From an application tuning standpoint, in the first stage the application is profiled (using the built-in performance collection instrumentation) to collect statistics about the processing and communication characteristics of individual operators. To perform the profiling step, the application is compiled in a special mode that instruments the operators with additional code that collects statistics during runtime. These statistics include: (i) the stream rate in terms of tuples/s for each input and output port, (ii) the stream rate in terms of bytes/s each input and output port, and (iii) the fraction of CPU utilization for each operator.

In the second stage, the profiling information is used iteratively by an optimizer to determine the appropriate physical dataflow graph that is deployable across the available compute nodes. This approach not only leads to the deployment of optimized applications that are tailored to the underlying computing and networking infrastructure, but also allows re-targeting the application to a different hardware setup by simply repeating the optimization step and re-compiling the application to match the physical flow graph produced by the optimizer [28].

After the compile-time optimization phase, the System S runtime scheduler is used to place the physical flowgraph across hosts such that overloading of hosts is avoided. This is performed not only during the submission of a job by making placement decisions but also during the application execution by making dynamic migration decisions, whenever the system detects the need for processing element movement [44].

4.5. State management and fault tolerance

The final set of patterns critical to streaming applications is driven by the need for persistence and fault tolerance. The long-running and continuous nature of these applications requires the construction and maintenance of state information that may include analytical models, operating parameters, tuple and data summaries, performance snapshots, etc. It is critical to maintain this internal state appropriately without letting it grow indefinitely—leading to potential failures. For instance, stateful operators such as Joins and Aggregates can consume too much memory and grind to a halt or crash if the input data rate unexpectedly increases and the conditions for evicting tuples from the processing windows are not met. Similarly, if any container is used in a user-defined operator to maintain state, an appropriate eviction strategy must be designed to prevent the size of the data structure from spiraling out of control. The state maintenance requirements of an application and the different components within it must be considered carefully during application design.

There is a strong need for tolerance to different types of failures including processing infrastructure, analytic, and data source failure. An important aspect in these applications is that not every segment of an application is equally relevant from a reliability standpoint. For instance, tolerance to sporadic data loss is, in many cases, acceptable for portions of the application as long as the amount of error can be bounded and the accuracy of the results can be properly assessed. Other application segments, however, cannot tolerate any failures as they may contain critical persistent states. Hence, an important aspect in designing a complex application is to rely on partial fault tolerance constructs. Because of the additional overhead imposed by both tasks, a developer should take into account where and how these capabilities should be provided [18], attempting, whenever possible, to employ partial fault tolerance strategies [17].

The principles of state management and fault tolerance can be summarized as:

- Consider the state management requirements of different applications. Use appropriate state structures and repositories (such as in-memory databases) to lower an operator’s internal processing latency.
- Use fixed-size data structures where possible (e.g. count-based windows, Bloom Filters or summary statistics).
- Use a combination of state monitoring and an appropriate deletion strategies to prevent unbounded state growth.
- Consider the dynamic distribution of state across multiple hosts to support large memory requirements and to prevent single point of failure.
- Consider combinations of in-memory and disk-based persistence of state to limit memory usage.
- Design checkpointing strategy based on failure resilience needs of an application, focusing on partial fault tolerance approaches.

We now describe two different patterns, one related to maintaining and managing state and the other related to the use of flexible fault tolerance mechanisms in streaming applications.

4.5.1. Maintaining and persisting state in applications. Stream processing applications often require the maintenance and persistence of state, including both individual operator state, as well as shared state common to multiple operators. Multiple Spade built-in operators are designed to be inherently stateful, requiring the retention of different windows and moving statistics on the data.

In some cases, the type of state that needs to be retained is complex and cannot be directly represented by the built-in data types supported by a stream processing middleware. In such cases, the usual approach is to use user-defined operators to capture and manage such state using appropriate data structures. For example, in one of the applications we have built to process call detail records (see Section 2.1), i.e. transactions originated from mobile phone networks, the primary data structure to be managed and operated on is a social network graph of caller-callee interactions. The graph is such that callers and callees are nodes and calls are represented as directed edges from a caller node to a callee node. Edges have weights proportional to the total duration of all calls between the pair of users. Hence, for each new CDR, either an existing edge is updated or a new edge is created. Clearly, to have efficient inserts and searches for large-scale graphs, we must use efficient data structures. In this case, we rely on C++ constructs to accomplish this task. Specifically, we use a hash map to create the appropriate edge map, with the underlying structures used to retain the keys as well as the edge structure. This approach also lends itself to the use of a simple data partitioning scheme to distribute the state structure across multiple nodes.

In some cases, the need for efficient state management is also associated with the need to provide fault tolerance capabilities and, in some cases, to provide transactional support for managing the accumulated state. Different stream processing platforms employ different strategies for these tasks. In System S, it is also possible to persist state using both in-memory databases such as SolidDB [45] or regular databases, using operators provided by Spade toolkits. One of the operators for enriching streamed data with information retrieved from databases was shown in Figure 5.

4.5.2. Fault tolerance implementation pattern. Different streaming middleware platforms make available distinct fault tolerance mechanisms, including full reliability and replay capabilities [46]. Nevertheless, most of them do not include support to flexibly employ user-chosen fault tolerance strategies for different portions of the application processing graph.

To provide fault tolerance for critical segments of applications, the Spade language provides checkpointing support for individual operators, which can be optionally configured by application developers. An example of checkpointing the state of multiple operators is shown in Figure 18. From an implementation pattern standpoint, a developer can choose to selectively employ checkpointing throughout the application flowgraph. Specifically, as seen in Figure 18, the Aggregate operator in
the application is checkpointed every 10 s, whereas the user-defined operator is checkpointed every 5 s (as indicated by the `checkpoint=5` declaration). For user-defined operators, the compiler generates method interfaces and skeleton source code for storing as well as retrieving the relevant state should an operator be restarted. The specific state to be stored is determined by the operator developer. The interfaces are automatically invoked at the specified frequency by the System S runtime. When a failure occurs or when a processing element host movement operation is triggered at runtime (e.g. to accommodate additional workload), the latest state is retrieved and the operator state is re-populated before the application is re-started.

The Spade language also provides constructs to create replicated segments. Operators that are part of a replicated segment are marked in the source code and wrapped inside a syntactic section that specifies a replication factor. Depending on the replication factor setting, multiple replicas of the operator subgraph defined by the replicated segment are instantiated. All these replicas process the same inputs (i.e. the streams that enter into the replicated segment). In other words, these replicas are hot standbys. Note that, at any time, the results from only one of the replicas (i.e. the streams that leave the replicated segment) are consumed by the downstream operators. The replica whose results are actually being consumed is referred to as the active replica. When a failure is detected in one or more of the operators that are part of the current active replica, a new active replica is chosen by selecting a hot standby that is healthy.

Figure 19 shows an example application with two replicated segments. In this example, there are two replica flows for each replicated segment. For both of the replicated segments, the replica flow that is at the bottom is the active one, and their stream connections are shown with solid lines in the figure, as opposed to the hot standbys, where the stream connections are shown with dashed lines. Assume that at a later time, the operator `PriceModeller` that is part of the bottom replica flow in the second replicated segment fails. The system will detect this automatically and it will enable the output from the `VWAP` operator that is part of the top replica flow (now the new active replica). At the same time the system disables the bottom replica flow.
5. RELATED WORK

The area of stream processing has been very active over the last few years. Such flurry of activity has led to the development of several academic and commercial platforms. STREAM [7], Borealis [6], StreamBase [47], TelegraphCQ [9], among others, have focused on providing stream processing middleware and, in some cases, declarative language for easing the task of writing applications. Specifically, on the programming language side, examples such as StreamIt [11] and the Aspen language [48] share commonalities with the Spade language, namely the philosophy of providing high-level programming constructs, shielding users from the complexities of dealing with issues related to the development of applications requiring distributed programming. Note that the preceding references are just a small sample of the work in the stream processing area. Nevertheless, we have found that there is no work on providing software engineering principles and experienced guidance to developers in the streaming domain. There is, however, a book on models and streaming analytics edited by Aggarwal [49]. The present work is aimed at filling this gap and, hopefully, inducing other researchers and practitioners to further refine and add to our contribution.

Other more mature software engineering areas have enjoyed a considerable amount of support in the form of design principles for the development community. While we will not include an extensive survey here, we will mention a small set of representative examples. For instance, design principles for real-time systems is the focus of a book written by Kopetz [50], which includes the fundamentals on real-time processing as well as deeper aspects of system and application design. Lea [51] and Hughes and Hughes [52] have described principles and patterns for developing parallel and concurrent applications in Java and in C++, targeting mainly practitioners. Along these lines, there has also been academic work on design principles with examples in automation systems [53] and distributed data analysis middleware design [54]. In addition, popular software engineering domains such as web-based applications [55] and service-oriented architectures [56] have also seen work on design principles aimed at developers.

Finally, there has also been work by several authors on more foundational guiding principles specifically describing software development design patterns. Two very relevant contributions come from the ‘Gang of Four’ book on design patterns [57] as well as from McConnell [58], where he has collected and discussed several important principles for code development in general.

6. CONCLUSION

Stream computing is an emerging area for software development in commercial and scientific areas. The main driving force behind this is the steady increase in available data sources with fresh data and the need by business analysts and scientists to efficiently process such data and extract actionable knowledge from it. The goal in building these types of applications is to obtain improved and early insights resulting from the immediate processing of this information, separating uninteresting data from the interesting nuggets. These results can enable corporations, research, and government organizations to proactively act or quickly react to changing trends by switching and refining business strategies, ultimately providing a much better and up-to-date understanding of business and scientific phenomena.

In this paper, we provided a first look on application scenarios, discussing their characteristics and requirements as well as strategies to better develop these applications. This discussion was rooted in our own experience in designing and implementing System S as well as building numerous applications on it using the Spade language. We expect that this work will give rise to additional discussions in the research and development communities, further augmenting the design principles presented here as more work is devoted to creating innovative applications.

We continue to evolve the System S platform and its programming language [30], along with its associated integrated development and visualization environment [40, 42], focusing on providing the infrastructure and the abstractions needed by application analysts and developers as they
undertake the task of implementing ever more challenging and large-scale applications, bringing us closer to the vision of processing information via true sense-and-respond systems [59].

REFERENCES


4. Senator TE. Multi-stage classification. Proceedings of the International Conference on Data Mining (ICDM), Houston, TX, 2005; 386–393.


45. IBM SolidDB. Available at: http://www-01.ibm.com/software/data/soliddb/ [September 2009].