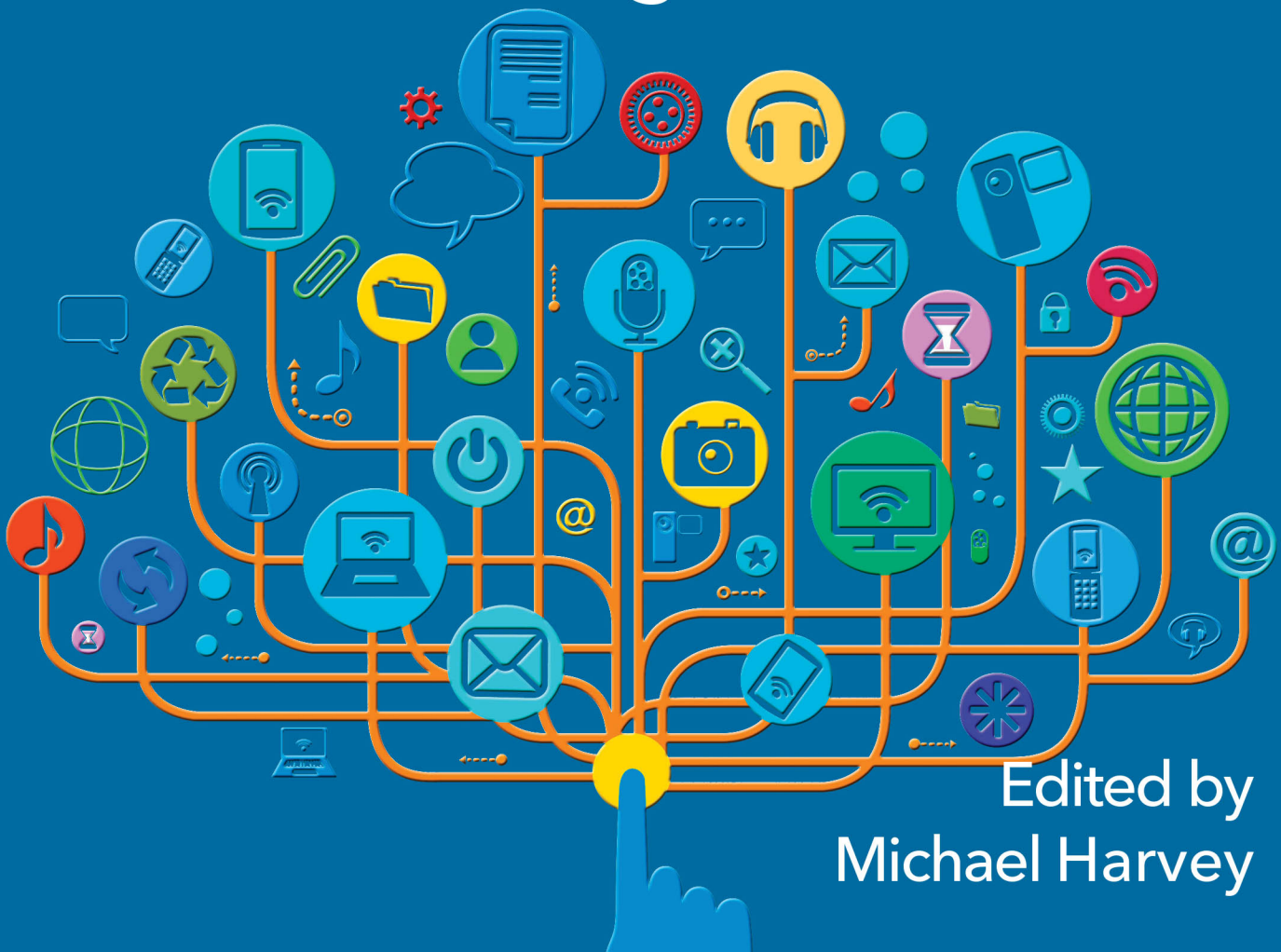
Intelligence at the Edge

Using SAS® with the Internet of Things



Edited by
Michael Harvey

The correct bibliographic citation for this manual is as follows: Harvey, Michael. 2020. *Intelligence at the Edge: Using SAS® with the Internet of Things*. Cary, NC: SAS Institute Inc.

Intelligence at the Edge: Using SAS® with the Internet of Things

ISBN 978-1-64295-780-8 (Hardcover)
ISBN 978-1-64295-776-1 (Papberback)
ISBN 978-1-64295-777-8 (PDF)
ISBN 978-1-64295-778-5 (epub)
ISBN 978-1-64295-779-2 (kindle)

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

February 2020

Contents

Preface

About the Internet of Things

Today we can hardly imagine a world without the internet. Whether we access information about a restaurant or product through our smart phone, manage our banking transactions through those same phones or our home computers, or plan a vacation through easy-to-use websites, we don't give a second thought to using it. The internet is always there, at our fingertips.

It was only sixty years ago that the preliminary research on packet switching began. Packet switching undergirds the TCP/IP protocols used by the internet. TCP/IP was designed to provide a more robust alternative network architecture than existing point-to-point computer networks. By 1971, fifteen sites were internetworking on the ARPANET, which became one of the first backbones of the internet. In 1989, Tim Berners-Lee, a computer scientist at CERN, invented the World Wide Web. He engineered a web browser to navigate this web which he released to the public in 1991. Twenty years later, 2.2 billion users (almost a third of the world's population) were using browsers to traverse the internet. They used these browsers on laptops, tablets, and smart phones (Miniwatts Marketing Group, 2019).

As you read these pages, we are experiencing a seismic shift in the way that we interact with the internet. It is no longer confined to using a web browser – it now touches almost every aspect of our day-to-day activity. The Internet of Things (IoT) takes the internetworked computer systems with which we are so familiar and attaches a plethora of devices, sensors, and objects in our world. Data is collected and processed in real time from these "things."

Those "things" are ubiquitous. Consider the following. Not long ago, my wife and I were driving back from a weekend getaway. She picked up her phone and asked her digital assistant, "How long will it take to arrive home?" Within seconds, the assistant gave a precise reply in hours and minutes. When we arrived home, the assistant's reply was off by a mere five minutes. (We encountered some congestion during the final mile of our trip.)

Think about that. While speeding down a four-lane highway, my wife used her phone, a "thing," to use the internet to ask an artificially intelligent digital assistant when we would arrive home. In a manner of seconds, that assistant used the phone's GPS to determine our current location and our current speed. Using our destination location, it calculated the distance that we wanted to travel. Accounting for current traffic conditions, it calculated the time it would take to arrive at our destination. In seconds. And the calculation was accurate within just a few minutes.

When was the Internet of Things born? Some say it was in the 1980s, when programmers at Carnegie Melon University connected a Coke machine to the internet to check whether a drink was available and was cold (Foote, 2016). Nevertheless, it was not until 1999 that the phrase "Internet of Things" was coined by Kevin Ashton, the Executive Director of Auto-ID

Labs at MIT. He proposed using Radio Frequency Identification (RFID) technology to tag devices so that computers could manage, track, and inventory them through the internet. Since then, things have been tagged with digital watermarking, bar codes, and QR codes, among other means (Foote, op cit). The number of things connected to the internet has grown at a breathtaking rate.

These days, Internet of Things involves not just collecting and tracking data, but also applying analytics to the data *as it is gathered* to intelligently manage those things. You can run analytics on systems that are physically close to the device collecting the data, that is, at the "edge," to enable faster decision making. SAS software enables organizations to collect and apply intelligence at the edge to better extract business value. Businesses translate that value into more efficient operations, lower costs, and a wider range of revenue streams. Thus, now when you collect data from sensors on trucks, trains, or jets, you can better predict when parts might fail. You then can use the results of your analysis to proactively stock the right parts and keep the vehicles operational, saving thousands of hours of downtime. You can monitor streaming data from factory equipment to more intelligently schedule maintenance. You can analyze streaming data from call center systems, news sites, and social media forums, and then integrate your analysis with issue detection processes to get a jump on corrective actions. Applying intelligence at the edge has become one of the highest priorities for businesses.

Foote, Keith D. 2016. "A Brief History of the Internet of Things," https://www.dataversity.net/brief-history-internet-things/.

Miniwatts Marketing Group, 2019. Internet Growth Statistics https://www.internetworldstats.com/emarketing.htm.

About This Book

This book is written for a general audience who wants to learn more about this rapidly changing field. Some technical knowledge of computing and statistics is useful to fully grasp the terms and concepts covered, but it is not essential to comprehend the information provided. The book describes how SAS applies analytics to derive business value from the Internet of Things.

At the heart of that endeavor is SAS Event Stream Processing. The first chapter explains how that product works, provides some simple examples, and briefly covers a scenario where an ESP server analyzes edge data and sends results to another ESP server running at the data center. It explains the distinction between static data, which sits unchanging in a database or repository, and streaming data, which continuously flows from the world into software applications.

SAS Event Stream Processing gets data from the world primarily through connectors and adapters, which are covered in Chapter 2. It explains how SAS Event Stream Processing uses connectors and adapters to communicate with the message fabrics, databases, and other protocols used to handle data transmitted from the "things" in the IoT. It includes an example that shows how connectors work.

Chapter 3 describes how to apply analytics to streaming data. It describes the IoT analytics life cycle emphasizing the analytics. It covers the algorithms that you can apply to streaming data and provides brief examples of two of those algorithms applied to data from the edge.

How can you effectively deploy and manage ESP servers at the edge? The answer is by using SAS Event Stream Manager. Chapter 4 describes how to monitor and manage your SAS Event Stream Processing environment through this web-based client. It shows how SAS Event Stream Manager works with SAS Model Manager to enable you to use the most current champion model in your environment. It also provides a step-by-step example to create a SAS Event Stream Manager deployment, associate an ESP server with it, and run, monitor, and stop the associated job.

What is an IoT reference architecture, and how does SAS Event Stream Processing fit into it? Chapter 5 covers that and the deployment considerations for the edge and cloud. It discusses the IoT life cycle emphasizing the life cycle. It provides a detailed example of using SAS Event Stream Processing with other SAS products within an IoT reference architecture.

The final chapters cover use cases of analyzing data from the Internet of Things with SAS Event Stream Processing. Chapter 6 gives an overview of applying AI to streaming data from the IoT using SAS Event Stream Processing. It also provides a detailed example of using SAS Visual Data Mining and Machine Learning to build a model and then deploying that model in SAS Event Stream Processing to score streaming data. Chapter 7 introduces the topic of geofencing and explains how the Geofence window of SAS Event Stream Processing enables it in real-time. Chapter 8 tells how SAS Event Stream Processing and machine learning algorithms can be used together to create a "digital twin" to monitor whether devices in disparate environments are operating properly and efficiently. Chapter 9 describes how SAS Event Stream Processing can detect changes in the quality of location-based data in real time and adjust it right away. Chapter 10 presents research using SAS Event Stream Processing for condition-based maintenance (CBM) of critical, high-valued machines. Chapter 11 explores how you can create intelligent computer vision systems, deploy those models on edge-devices to score streaming data, and use SAS Event Stream Processing to make decisions about what is seen in real time.

We Want to Hear from You

Do you have questions about a SAS Press book that you are reading? Contact us at saspress@sas.com.

SAS Press books are written *by* SAS Users *for* SAS Users. Please visit sas.com/books to sign up to request information on how to become a SAS Press author.

We welcome your participation in the development of new books and your feedback on SAS Press books that you are using. Please visit sas.com/books to sign up to review a book

Learn about new books and exclusive discounts. Sign up for our new books mailing list today at https://support.sas.com/en/books/subscribe-books.html.

Learn more about this author by visiting his author page at http://support.sas.com/harvey . There you can download free book excerpts, access example code and data, read the latest reviews, get updates, and more.

About the Author

Michael Harvey is a Principal Technical Writer at SAS, serving as documentation project leader for the Internet of Things (IoT). Previously, Michael worked as a manager and a writer for EMC. He also teaches Information Architecture for the Duke Continuing Studies Technical Writing professional certificate program.

Michael has a BA in English and Psychology from the University of North Carolina at Chapel Hill and an MA in Experimental Psychology from Duke University. He has served in various leadership positions for the Carolina chapter of the Society for Technical Communication (STC) and has presented at local and international STC conferences. He was honored to be named an STC Fellow in 2011. As an instructor for the Durham Technical Community College Technical Writing program in the late 1980s and early 1990s, Michael worked to overhaul the curriculum, emphasizing the importance of developing technical curiosity and acquiring technical expertise.

Learn more about this author by visiting his author page at http://support.sas.com/harvey. There you can download free book excerpts, access example code and data, read the latest reviews, get updates, and more.

Chapter 1: Using SAS Event Stream Processing to Process Real World Events

By Michael Harvey, Robert Ligtenberg, and Jerry Baulier

Introduction

As Andrew G. Psaltis, the regional CTO for Cloudera, observes, “Data is flowing everywhere around us, through phones, credit cards, sensor-equipped buildings, vending machines, thermostats, trains, buses, planes, posts to social media, digital pictures and video – and the list goes on.” Being able to harness that data presents abundant business opportunities. How can a business best capitalize on those opportunities?

The answer: SAS Event Stream Processing. It enables you to process and analyze continuously flowing real-world events in real time. Events arrive through high-throughput, low-latency data flows called event streams. These data flows are generated by occurrences such as sensor readings or market data. Each event within an event stream can be represented as a data record that consists of any number of fields. For example, an event generated by a pressure sensor could include two fields: a pressure reading and a timestamp. A more complex financial trade event could include multiple fields for transaction type, shares traded, price, broker, seller, stock symbol, timestamp, and so on. SAS Event Stream Processing can process the pressure data or the trades at any given moment. It can alert you to events of interest the instant that they occur.

Innovations in technology have enabled the reduction of the cost and size of sensors. Now sensors can be readily deployed within industrial equipment and consumer products. The number of sensors available has exploded, and a large portion of these sensors are now

connected through the internet. The deluge of resulting data streams is often called *Big Data*. The Internet of Things (IoT) attaches a plethora of devices, sensors, and objects in our world to the internet. Big Data is collected and processed in real time from these "things."

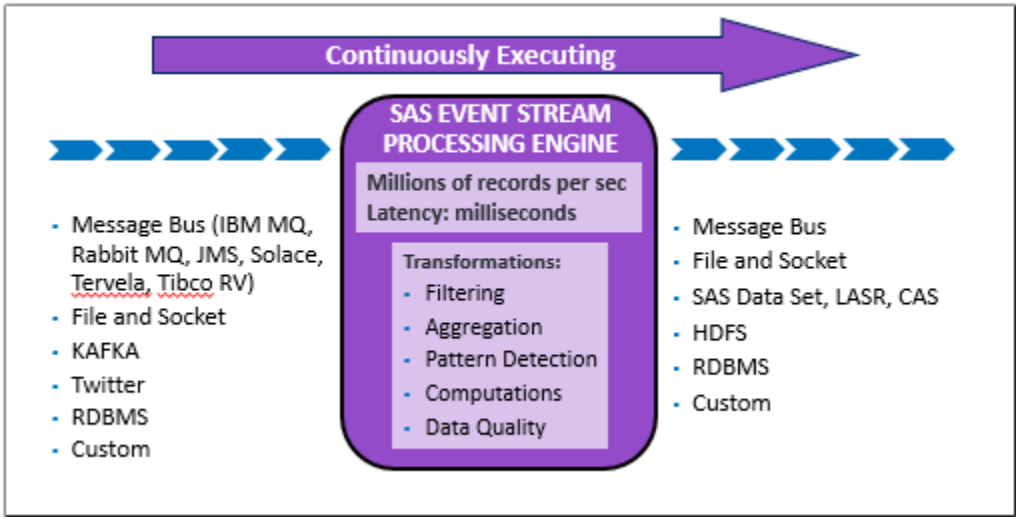
SAS Event Stream Processing processes real-world data *as it is generated*. This instantly processed data is called *streaming* data. Processing streaming data introduces a paradigm shift from the traditional approach, where data is captured and stored in a database. After an event from an event stream is processed, it can be stored or discarded. Subsequent results of event stream data processing can also be stored and explored.

When time-sensitivity is important, processing streaming data at the point of generation is critical. For example, suppose that you are using sensing devices to track a customer who is browsing products at a retail establishment or online. Based on customer or product location (in real space or cyberspace), a system processing streaming data can generate an offer in real time to entice a purchase. An application that uses data at rest is not nimble enough to make these suggestions. Another example, also involving sensing devices, is the real-time tracking of vibrations in airliner engines. When anomalous patterns are detected (perhaps as the result of a bird impact), pilots can be alerted immediately so that they can take corrective action. Catastrophic failure can be avoided.

How Does SAS Event Stream Processing Work?

SAS Event Stream Processing reads from many source formats and outputs to many target formats (Figure 1.1).

Figure 1.1: Input and Output Streams Through the SAS Event Stream Processing Engine



Many types of transformations are supported, including SQL primitives for filtering, aggregation, and pattern detection. There is built-in support for computations based on internal functions as well as on external languages like C++ and Python.

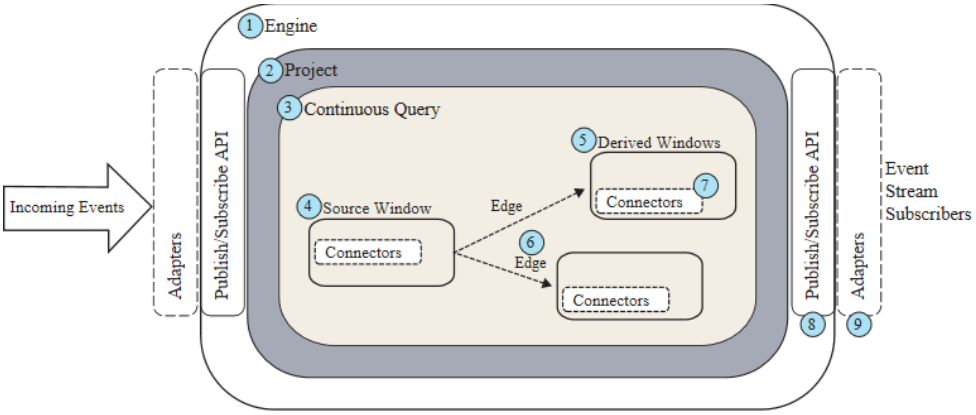
In addition, SAS Event Stream Processing provides many types of advanced analytical algorithms. These include algorithms for natural language text processing, image recognition and video image tracking, and machine learning. These analytical algorithms are covered in detail in Chapter 3.

In short, SAS Event Stream Processing provides a flexible platform with out-of-the-box capabilities to handle almost any type of event stream. You can use it to apply almost any type of business logic in real time.

What is a SAS Event Stream Processing Model?

SAS Event Stream Processing processes event streams through a *model*, which specifies how events are transformed and analyzed into meaningful results. The following figure (Figure 1.2) depicts the hierarchy of this model.

Figure 1.2: The SAS Event Stream Processing Model



1. At the top of the model hierarchy is the *engine*. Each model contains only one engine instance with a unique name.
2. The engine contains one or more *projects*, each uniquely named. You can specify a port so that projects can be spread across network interfaces for throughput scalability.
3. A project contains one or more *continuous queries*. A continuous query is represented by a directed graph. This graph is a set of connected nodes that follow a direction down one or more parallel paths. Continuous queries are data flows, which are data transformations and analysis of incoming event streams.
4. Each query has a unique name and begins with one or more *Source windows*.
5. Source windows are typically connected to one or more *derived windows*. Derived windows can detect patterns in the data, transform the data, aggregate the data, analyze the data, or perform computations based on the data. They can be connected to other derived windows.
6. Windows are connected by *edges*, which have an associated direction.

7. *Connectors* are in-process to the engine. They use the publish/subscribe API to interface directly with a variety of message buses and brokers (for example, Kafka, RabbitMQ), communication fabrics, drivers, and clients.
8. The *publish/subscribe API* can be used to subscribe to an event stream window either from the same machine or from another machine on the network. Similarly, the publish/subscribe API can be used to publish event streams into a running event stream processing project Source window.
9. *Adapters* are stand-alone executable programs that can be networked. Some adapters are executable versions of their corresponding connector. Adapters use the publish/subscribe API to publish event streams to do the following:
    - publish event streams to Source windows
    - subscribe to event streams from any window

Connectors and adapters are available for almost any streaming format or protocol, including OPC-UA, Bacnet, PI System for sensors, as well as MQTT, RabbitMQ, Kafka, Tibco, Tervela, and IBM WebSphere for messaging fabrics. A database connector gives access to Database Management (DBM) systems, and there is connector support for sockets and websockets. Connectivity to Apache Camel and Apache NiFi extends the range of streaming sources even further. For more information about how connectors and adapters work, see Chapter 2.

The most common way to specify a model is with XML, as shown in Figure 1.3.

Figure 1.3: XML Code for a SAS Event Stream Processing Model

```
<engine name="engine" port="5555">
  <projects>
    <project name="project_A">

      <contqueries>
        <contquery name="contQuery">

          <windows>

            <window-source name="sourceWindow">
              <schema>
                <fields>
                  <field name="ID" type="int64" key="true"/>
                  <field name="symbol" type="string"/>
                  <field name="quantity" type="int32"/>
                  <field name="price" type="double"/>
                </fields>
              </schema>
              <connectors>
                <connector name="input" class="fs">
                  <properties>
                    <property name="type">pub</property>
                    <property name="fsname">localhost:4556</property>
                    <property name="fstype">csv</property>
                  </properties>
                </connector>
              </connectors>
            </window-source>

            <window-filter name="filterWindow">
              <expression>quantity > 1000</expression>
              <connectors>
                <connector name="output" class="fs">
                  <properties>
                    <property name="type">sub</property>
                    <property name="fsname">localhost:4557</property>
                    <property name="fstype">csv</property>
                  </properties>
                </connector>
              </connectors>
            </window-filter>

          </windows>

          <edges>
            <edge source="sourceWindow" target="filterWindow"/>
          </edges>

        </contquery>
      </contqueries>

    </project>
  </projects>
</engine>
```

This code reflects the hierarchy of a model. An engine contains a project, which contains a continuous query. Within the continuous query, the business logic is represented by a Source window and one derived window, which in this case is a Filter window. All event streams must enter continuous queries by being published or injected into a Source window. Event streams cannot be published or injected into any other window type. The Source window and Filter window in this model are connected by an edge, as specified in the **<edge>** XML element.

The Source window is configured to ingest events from an event stream through a port on a server. The server in this example is the localhost server. The Source window also specifies a schema that defines field properties for the incoming events. In this case, there are four fields: an ID that serves as the event's key field, a stock symbol, a quantity (shares transacted), and a price.

The Filter window is configured to select events that satisfy a specific condition. In this example, the condition requires that the quantity is greater than 1000. Satisfying events are passed on while non-satisfying events are not passed on. A subscribing connector passes the output events to a port where another application might be listening.

You deploy models by executing them within a SAS Event Stream Processing engine. Upon execution, the engine establishes a connection to the specified input port and starts ingesting incoming events. The events are processed according to the configured business logic and output events are delivered to the output port.

Processing Events in Derived Windows

All continuous queries contain one or more Source windows and one or more derived windows. SAS Event Stream Processing supports a variety of derived window types, each having a specialized purpose (Table 1.1).

Table 1.1: Derived Window Types

Window Type	Description
Aggregate	Calculates aggregates like sum and average, similar to SQL group-by aggregations.
Compute	Adds calculated fields to an event stream.
Copy	Copies (replicates) an event stream and supports a retention policy. Retention is used for time-based or count-based windowing.
Counter	Counts events and calculates throughput.
Filter	Selects events based on an expression.
Functional	Executes user-defined functions. Supports regular expressions and looping to parse XML and JSON fields.
Geofence	Determines whether event locations are in or near an area of interest.
Join	Joins two event streams like an SQL join. Supports inner, left-outer, right-outer, and full outer joins.
Notification	Sends notifications. Supports SMTP, SMS, and MMS.
Object Tracking	Performs multi-object tracking (MOT) in real time.
Pattern	Detects patterns and anomalies within events and across events.
Procedural	Calls external functions (C++, SAS). Supports multiple input windows (input streams) with an input-handler function for each input.
Remove State	Facilitates the transition of a stateful part of a model to a stateless part of a model.
Text Category	Categorizes text fields.
Text Context	Extracts text and classifies terms.
Text Sentiment	Performs sentiment analysis.

Window Type	Description
Text Topic	Scores and identifies themes.
Transpose	Transposes rows to columns or columns to rows.
Union	Combines multiple event streams with the same schema into a single stream, like an SQL union.

Each of these windows is explained in detail in the "Using Source and Derived Windows" documentation, which is available at https://go.documentation.sas.com/?cdcId=espcdc&cdcVersion=6.2&docsetId=espcreatewindows&docsetTarget=titlepage.htm&locale=en.

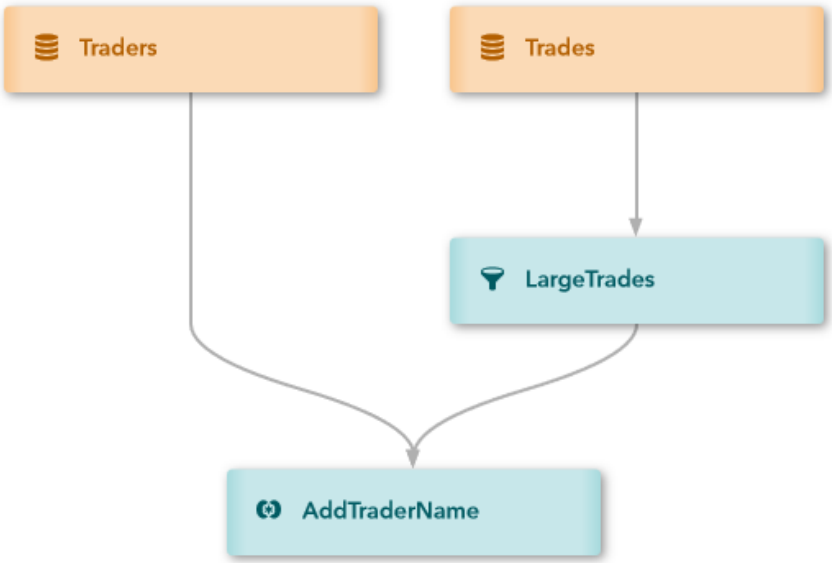
Examples of Event Transformations

The following two examples demonstrate how events are transformed within a continuous query. The first uses a Join window to combine data from two separate event streams into a single output stream. The second combines a Pattern window and Notification window to catch front-running trades.

Example: Using a Join Window

A Join window combines fields from two input event streams into a single output event stream. In the model depicted in Figure 1.4, the Join window (AddTraderName) receives events from a left input window (a Source window named Traders) and a right input window (a Filter window named LargeTrades). It produces a single output stream of joined events. Joined events are created according to a user-specified join type and user-defined join conditions.

Figure 1.4: Continuous Query That Uses a Simple Join



The Source window named Trades streams data about securities transactions. A file and socket connector is established to publish events into the Trades window from a file named trades.csv in the /data directory.

Example Code 1.1

```
<window-source name='Trades' index='pi_RBTREE'>
       <schema>
           <fields>
               <field name='tradeID' type='string' key='true'/>
               <field name='security' type='string'/>
               <field name='quantity' type='int32'/>
               <field name='price' type='double'/>
               <field name='traderID' type='int64'/>
               <field name='time' type='stamp'/>
           </fields>
       </schema>
       <connectors>
            <connector class="fs" name="publisher">
              <properties>
                <property name="type">pub</property>
                <property name="fstype">csv</property>
                <property name="fsname">/data/trades.csv</property>
                <property name="transactional">true</property>
                <property name="blocksize">1</property>
                <property name="dateformat">
                %d/%b/%Y:%H:%M:%S
                </property>
              </properties>
            </connector>
```

```
        </connectors>
</window-source>
</windows>
```

A Filter window named LargeTrades receives events from the Trades window. It filters out any event that involves fewer than 100 shares.

Example Code 1.2

```
<window-filter name='LargeTrades'>
       <expression>quantity >= 100</expression>
</window-filter>
```

A second Source window named Traders streams data about who performs those transactions. A file and socket connector is established to publish events into the Traders window from a file named traders.csv in the /data directory.

Example Code 1.3

```
<window-source name='Traders'>
      <schema>
          <fields>
              <field name='ID' type='int64' key='true'/>
              <field name='name' type='string'/>
          </fields>
      </schema>
     <connectors>
            <connector class="fs" name="publisher">
              <properties>
                <property name="type">pub</property>
                <property name="fstype">csv</property>
                <property name="fsname">/data/traders.csv</property>
                <property name="transactional">true</property>
                <property name="blocksize">1</property>
              </properties>
            </connector>
        </connectors>
</window-source>
```

The Join window named AddTraderName matches filtered transactions from the first Source window with their associated traders from the second. Specifically, the conditions tag matches the traderID values specified in the transactions Source window to the ID values specified in the traders Source window.

Example Code 1.4

```
<window-join name='AddTraderName'>
       <join type="leftouter">
           <conditions>
               <fields left='traderID' right='ID' />
           </conditions>
       </join>
       <output>
           <field-selection name='security' source='l_security'/>
           <field-selection name='quantity' source='l_quantity'/>
           <field-selection name='price' source='l_price'/>
           <field-selection name='traderID' source='l_traderID'/>
           <field-selection name='time' source='l_time'/>
```

```
            <field-selection name='name' source='r_name'/>
        </output>
</window-join>
```

By default, this join order is determined through the specification of edges. The left window is the first window that is defined as a connecting edge to the Join window. The right window is the second window that is defined as a connecting edge.

Example Code 1.5

```
                                    <edges>
        <edge source="LargeTrades" target="AddTraderName"/>
        <edge source='Traders' target='AddTraderName'/>
        <edge source='Trades' target='LargeTrades'/>
</edges>
```

Suppose that the following events stream through the Source window named Trades (Figure 1.5):

Figure 1.5: Input to the Trades Source Window

Opcode	tradeID	security	quantity	price	traderID	time
Insert	TID1234331	ibm	80	100.300000	10004	2012-07-08T08:1...
Insert	TID1234330	sap	90	32.000000	10003	2012-07-08T08:1...
Insert	TID1234329	ibm	1000	100.000000	10004	2012-07-08T08:1...
Insert	TID1234328	sap	1000	32.000000	10003	2012-07-08T08:1...
Insert	TID1234327	ibm	1000	100.300000	10002	2012-07-08T08:1...
Insert	TID1234326	sap	1000	34.300000	10003	2012-07-08T08:1...
Insert	TID1234325	ibm	1000	100.400000	10004	2012-07-08T08:1...
Insert	TID1234324	ibm	1000	100.300000	10004	2012-07-08T08:1...
Insert	TID1234323	ibm	1000	100.200000	10004	2012-07-08T08:1...
Insert	TID1234322	sap	750	34.200000	10003	2012-07-08T08:1...
Insert	TID1234321	ibm	1000	100.100000	10002	2012-07-08T08:1...

Figure 1.6 shows the events that stream from the Filter window.

Figure 1.6: Output from the LargeTrades Filter Window

Opcode	tradeID	security	quantity	price	traderID	time
Insert	TID1234329	ibm	1000	100.000000	10004	2012-07-08T08:1...
Insert	TID1234328	sap	1000	32.000000	10003	2012-07-08T08:1...
Insert	TID1234327	ibm	1000	100.300000	10002	2012-07-08T08:1...
Insert	TID1234326	sap	1000	34.300000	10003	2012-07-08T08:1...
Insert	TID1234325	ibm	1000	100.400000	10004	2012-07-08T08:1...
Insert	TID1234324	ibm	1000	100.300000	10004	2012-07-08T08:1...
Insert	TID1234323	ibm	1000	100.200000	10004	2012-07-08T08:1...
Insert	TID1234322	sap	750	34.200000	10003	2012-07-08T08:1...
Insert	TID1234321	ibm	1000	100.100000	10002	2012-07-08T08:1...

Now suppose that these events stream through the Traders Source window (Figure 1.7).

Figure 1.7: Input into the Traders Source Window

Opcode	ID	name
Insert	10004	dan penteado
Insert	10003	bernie
Insert	10002	steals a. lot

Figure 1.8 is the result of matching filtered transactions from the first Source window with their associated traders from the second.

Figure 1.8: Output from the AddTraderName Join Window

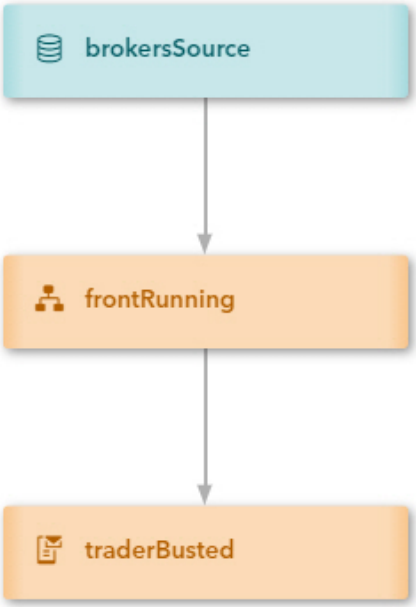
Opcode	tradeID	security	quantity	price	traderID	time	name
Insert	TID1234329	ibm	1000	100.000000	10004	2012-07-08T0…	dan penteado
Insert	TID1234328	sap	1000	32.000000	10003	2012-07-08T0…	bernie
Insert	TID1234327	ibm	1000	100.300000	10002	2012-07-08T0…	steals a. lot
Insert	TID1234326	sap	1000	34.300000	10003	2012-07-08T0…	bernie
Insert	TID1234325	ibm	1000	100.400000	10004	2012-07-08T0…	dan penteado
Insert	TID1234324	ibm	1000	100.300000	10004	2012-07-08T0…	dan penteado
Insert	TID1234323	ibm	1000	100.200000	10004	2012-07-08T0…	dan penteado
Insert	TID1234322	sap	750	34.200000	10003	2012-07-08T0…	bernie
Insert	TID1234321	ibm	1000	100.100000	10002	2012-07-08T0…	steals a. lot

Example: Using a Pattern Window and a Notification Window

A Pattern window enables you to detect events of interest (EOIs) as they stream through. To create a Pattern window, specify a list of EOIs and assemble them into an expression that uses logical operators. A Notification window enables you to send notifications through email using the Simple Mail Transfer Protocol (SMTP), text using the Short Message Service (SMS), or send multimedia messages using the Multimedia Messaging Service (MMS).

The following example uses a Pattern window to catch stock traders when they attempt front-running buys. A broker caught in the act is sent an email, an SMS text message, and an MMS message produced by a Notification window (Figure 1.9).

Figure 1.9: Continuous Query to Detect Front-Running Buys



The message sent includes details of the trades involved in the violation, and for the channels that permit graphics, the message also contains an image of someone in a jail cell. All relevant message routing information is included in the broker dimension data streamed into the Source window.

Example Code 1.6

```
i,n,1012112,Frodo,ESP,940 Orion Suite 201 Cary NC
27513,,frodo.doe@orion.com,5556466705,txt.att.net,mms.att.net
i,n,1012223,Sam,ESP,940 Orion Suite 201 Cary NC
27513,,sam.doe@orion.com,5556466706,txt.att.net,mms.att.net
i,n,1012445,Pippin,ESP,940 Orion Suite 201 Cary NC
27513,pippin.doe@orion.com,5556466707,txt.att.net,mms.att.net
i,n,1012334,Merry,ESP,940 Orion Suite 201 Cary NC
27513,merry.doe@orion.com,5556466708,txt.att.net,mms.att.net
i,n,101667,Gandalf,ESP,940 Orion Suite 201 Cary NC
27513,gandalf.doe@orion.com,5556466709,txt.att.net,mms.att.net
i,n,1012001,Aragorn,ESP,940 Orion Suite 201 Cary NC
27513,aragorn.doe@orion.com,5556466710,txt.att.net,mms.att.net
```

Note that the last four fields contain the email, phone number, and SMS and MMS gateways for each broker.

First, data streams into the model through a Source window named brokersSource.

Example Code 1.7

```
<window-source name='brokersSource' insert-only='true'>
   <schema-string>broker*:int32,brokerName:string,brokerage:string,
```

```
brokerAddress:string,brokerPhone:string,email:string,
                  smsGateway:string,mmsGateway:string</schema-string>
   <connectors>
      <connector class='fs'>
         <properties>
            <property name='type'>pub</property>
            <property name='fstype'>csv</property>
            <property name='fsname'>data/brokers.csv</property>
         </properties>
      </connector>
   </connectors>
</window-source>
```

The Pattern window is constructed to detect front-running violations. The window deals with up to three events (trades) at a time (e1, e2, and e3). Each trade contains broker and customer information as well as the trade data. All data must be available in order to format a notification message.

The Pattern window looks like Example Code 1.8.

Example Code 1.8

```
<window-pattern name='frontRunning'>
   <schema>
      <fields>
         <field name='id' type='int64' key='true' />
         <field name='broker' type='int32' />
         <field name='brokerName' type='string' />
         <field name='email' type='string' />
         <field name='phone' type='string' />
         <field name='sms' type='string' />
         <field name='mms' type='string' />
         <field name='customer' type='int32' />
         <field name='symbol' type='string' />
         <field name='tstamp1' type='string' />
         <field name='tstamp2' type='string' />
         <field name='tstamp3' type='string' />
         <field name='tradeId1' type='int32' />
         <field name='tradeId2' type='int32' />
         <field name='tradeId3' type='int32' />
         <field name='price1' type='double' />
         <field name='price2' type='double' />
         <field name='price3' type='double' />
         <field name='quant1' type='int32' />
         <field name='quant2' type='int32' />
         <field name='quant3' type='int32' />
         <field name='slot' type='int32' />
      </fields>
   </schema>
   <splitter-expr>
      <expression>slot</expression>
   </splitter-expr>
   <patterns>
      <pattern index='broker,symbol'>
         <events>
            <event name='e1'>((buysellflg == 1)
             and (broker == buyer)
```

```
            and (s == symbol)
            and (b == broker)
            and (p == price))</event>
         <event name='e2'>((buysellflg == 1)
            and (broker != buyer)
            and (s == symbol)
            and (b == broker))</event>
         <event name='e3'><![CDATA[((buysellflg == 0)
            and (broker == seller)
            and (s == symbol)
            and (b == broker)
            and (p < price))]]></event>
      </events>
      <logic>fby{1 hour}(fby{1 hour}(e1,e2),e3)</logic>
      <output>
         <field-selection name='broker' node='e1'/>
         <field-selection name='brokerName' node='e1'/>
         <field-selection name='brokerEmail' node='e1'/>
         <field-selection name='brokerPhone' node='e1'/>
         <field-selection name='brokerSms' node='e1'/>
         <field-selection name='brokerMms' node='e1'/>
         <field-selection name='buyer' node='e2'/>
         <field-selection name='symbol' node='e1'/>
         <field-selection name='date' node='e1'/>
         <field-selection name='date' node='e2'/>
         <field-selection name='date' node='e3'/>
         <field-selection name='id' node='e1'/>
         <field-selection name='id' node='e2'/>
         <field-selection name='id' node='e3'/>
         <field-selection name='price' node='e1'/>
         <field-selection name='price' node='e2'/>
         <field-selection name='price' node='e3'/>
         <field-selection name='quant' node='e1'/>
         <field-selection name='quant' node='e2'/>
         <field-selection name='quant' node='e3'/>
         <field-expr>1</field-expr>
      </output>
   </pattern>
   <pattern index='broker,symbol'>
      <events>
         <event name='e1'>((buysellflg == 0)
            and (broker == seller)
            and (s == symbol)
            and (b == broker))</event>
         <event name='e2'>((buysellflg == 0)
            and (broker != seller)
            and (s == symbol)
            and (b == broker))</event>
      </events>
      <logic>fby{10 minutes}(e1,e2)</logic>
      <output>
         <field-selection name='broker' node='e1'/>
         <field-selection name='brokerName' node='e1'/>
         <field-selection name='brokerEmail' node='e1'/>
         <field-selection name='brokerPhone' node='e1'/>
         <field-selection name='brokerSms' node='e1'/>
         <field-selection name='brokerMms' node='e1'/>
         <field-selection name='seller' node='e2'/>
```

```
            <field-selection name='symbol' node='e1'/>
            <field-selection name='date' node='e1'/>
            <field-selection name='date' node='e2'/>
            <field-expr> </field-expr>
            <field-selection name='id' node='e1'/>
            <field-selection name='id' node='e2'/>
            <field-expr>0</field-expr>
            <field-selection name='price' node='e1'/>
            <field-selection name='price' node='e2'/>
            <field-expr>0</field-expr>
            <field-selection name='quant' node='e1'/>
            <field-selection name='quant' node='e2'/>
            <field-expr>0</field-expr>
            <field-expr>2</field-expr>
         </output>
      </pattern>
   </patterns>
</window-pattern>
```

Events stream from the Pattern window into the Notification window.

Example Code 1.9

```
<window-notification name='traderBusted'>
   <smtp host='smtp-server.ec.rr.com'
         user='esptest@ec.rr.com'
         password='esptest1' port='587' />
   <schema>
      <fields>
         <field name='id' type='int64' key='true' />
         <field name='broker' type='int32' />
         <field name='brokerName' type='string' />
         <field name='email' type='string' />
         <field name='phone' type='string' />
         <field name='sms' type='string' />
         <field name='mms' type='string' />
         <field name='customer' type='int32' />
         <field name='symbol' type='string' />
         <field name='tstamp1' type='string' />
         <field name='tstamp2' type='string' />
         <field name='tstamp3' type='string' />
         <field name='tradeId1' type='int32' />
         <field name='tradeId2' type='int32' />
         <field name='tradeId3' type='int32' />
         <field name='price1' type='double' />
         <field name='price2' type='double' />
         <field name='price3' type='double' />
         <field name='quant1' type='int32' />
         <field name='quant2' type='int32' />
         <field name='quant3' type='int32' />
         <field name='slot' type='int32' />
         <field name='day' type='string' />
         <field name='price1' type='double' />
         <field name='price2' type='double' />
         <field name='price3' type='double' />
         <field name='time1' type='string' />
         <field name='time2' type='string' />
         <field name='time3' type='string' />
```

```
      <field name='profit' type='double' />
   </fields>
</schema>
<function-context>
   <properties>
      <property-list name='time1' delimiter=' '>$tstamp1
      </property-list>
      <property-list name='time2' delimiter=' '>$tstamp2
      </property-list>
      <property-list name='time3' delimiter=' '>$tstamp3
      </property-list>
   </properties>
   <functions>
      <function name='profit'>
       product($quant3,diff($price3,$price1))</function>
      <function name='day'>listItem(#time1,0)</function>
      <function name='time1'>listItem(#time1,1)</function>
      <function name='time2'>listItem(#time2,1)</function>
      <function name='time3'>listItem(#time3,1)</function>
      <function name='price1'>precision($price1,2)</function>
      <function name='price2'>precision($price2,2)</function>
      <function name='price3'>precision($price3,2)</function>
   </functions>
</function-context>
<delivery-channels>
   <email test='true' throttle-interval='1 day'>
      <deliver>contains(toLower($brokerName),'@BROKER@')</deliver>
      <email-info>
         <sender>esptest@ec.rr.com</sender>
         <recipients>$email</recipients>
         <from>ESP Broker Surveillance</from>
         <to>$brokerName</to>
         <subject>You have been caught cheating,
           $brokerName</subject>
      </email-info>
      <email-contents>
         <html-content><![CDATA[
         <body>You bought <b>$quant1</b> shares of <b>$symbol</b>
               for $<b>$price1</b> on <b>$day</b> at <b>$time1</b>.
               You then bought <b>$symbol</b> for customer
               <b>$customer</b>
               at <b>$time2</b>, after which you sold
               <b>$quant3</b> shares of
               <b>$symbol</b> at <b>$time3</b> for $<b>$price3</b>,
               thus making you a profit of $<b>$profit</b>.
               <br/><br/></body>
         ]]></html-content>
         <image-content type='image'>
          http://esp-base:18080/esp/stuff/jail.jpg
         </image-content>
      </email-contents>
   </email>
   <mms test='true' throttle-interval='1 day'>
      <deliver>contains(toLower($brokerName),'@BROKER@')</deliver>
      <mms-info>
         <sender>esptest@ec.rr.com</sender>
         <subject>You have been caught cheating,
           $brokerName</subject>
```

```
            <gateway>$mms</gateway>
            <phone>$phone</phone>
         </mms-info>
         <mms-contents>
            <text-content>You bought $quant1 shares of $symbol for
            $$price1 on $day at $time1. You then bought $symbol for
            customer $customer at $time2, after which you sold $quant3
            shares of $symbol at $time3 for $$price3, thus making you
            a profit of $$profit.
            </text-content>
            <image-content type='image'>
             http://esp-base:18080/esp/stuff/x.jpg
            </image-content>
         </mms-contents>
      </mms>
      <sms test='true' throttle-interval='1 day'>
         <deliver>contains(toLower($brokerName),'@BROKER@')</deliver>
         <sms-info>
            <sender>esptest@ec.rr.com</sender>
            <subject>You have been caught, $brokerName</subject>
            <from>ESP Broker Surveillance</from>
            <gateway>$sms</gateway>
            <phone>$phone</phone>
         </sms-info>
         <sms-contents>
            <text-content>You bought $quant1 shares of $symbol
            for $$price1 on $day at $time1. You then bought $symbol
            for customer $customer at $time2, after which you sold
            $quant3 shares of $symbol at $time3 for $$price3,
            thus making you a profit of $$profit.</text-content>
         </sms-contents>
      </sms>
   </delivery-channels>
</window-notification>
```

Because this example uses MMS, you need to define a different SMTP server. Any email account referenced by that server must be specified in your SMTP configuration. The window calculates fields to use when formatting notification messages to the broker. A schema and a function context are defined.

When an event comes in, functions are run on the input event and schema data is created. You can use values from either the input event or the schema data in the message content. For example, see Example Code 1.10.

Example Code 1.10

```
We noticed you bought <b>$quant1</b> shares of <b>$symbol</b> for
$<b>$price1</b>
on <b>$day</b> at <b>$time1</b>. You then bought <b>$symbol</b> for
customer <b>$customer</b> at <b>$time2</b>, after which
you sold <b>$quant3</b> shares of <b>$symbol</b> at <b>$time3</b>
for $<b>$price3</b>, thus making you a profit of $<b>$profit</b>.
```

Note the number of variable references. Some of the variable references are to the schema data (`quant1`, `price1`, `price3`, ...), and some to the input data (`symbol`). Variable references are also used to resolve the routing information for the notification.

Example Code 1.11

```
<recipients>$email</recipients>
<gateway>$sms</gateway>
<phone>$phone</phone>
```

A function is used to determine when to send the notification. The same deliver function is used for all channels.

Example Code 1.12

```
<deliver>contains(toLower($brokerName),'@BROKER@')</deliver>
```

Whenever you see the notation `@TOKEN@` in an XML model, this means that the token is resolved when the project is loaded. These tokens can be resolved in one of three ways:

- on the command line, for example, `dfesp_xml_server -BROKER pippin`
- in your environment, for example, `$ export BROKER=pippin`
- in the properties for a project, for example, `<property name='BROKER'>pippin</property>`

In this case, you can specify which broker to use to send a notification.

Streaming Analytics

SAS provides an extensive toolset of analytical algorithms and machine learning techniques, which can be directly applied to real-time streams in SAS Event Stream Processing models. You can use algorithms and techniques to address the following common challenges with data from the Internet of Things (IoT):

- lots of disparate variables
- noisy or missing data
- redundancy in the data
- prediction of rare events

Common use cases for streaming analytics include the following:

- preprocessing, transforming, or filtering data – determining how much and what data to send from the edge to the data center
- detecting anomalies
- monitoring system stability or degradation
- processing unstructured text, audio, video, or image data in order to discern patterns or trends

You can take one of two approaches with streaming analytics:

1. Use collected and stored data to develop analytical models off-line and then deploy them to SAS Event Stream Processing models to analyze and score incoming streams in real time.
2. Train analytical models in SAS Event Stream Processing on incoming streams and then update the models online based on the latest training results.

Additional derived window types are provided to handle analytical algorithms and machine learning techniques (Table 1.2).

Table 1.2: Analytics Window Types

Analytics Window Type	Description
Calculate	Applies a variety of analytical algorithms to event streams.
Model Reader	Accepts analytical models that were developed off-line.
Model Supervisor	Manages executing analytical models based on update requests.
Train	Uses incoming events to refine analytical model parameters.
Score	Scores incoming events based on the latest values of the analytical model parameters.

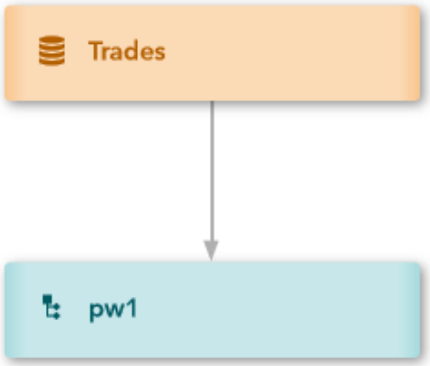
Chapter 3 reviews the innovations in advanced analytical models that are trained on data at rest and scored on streaming data, along with those that are directly applied to streaming data.

Using SAS Micro Analytic Service Modules with Streaming Analytics

A SAS Micro Analytic Service (MAS) module is essentially a named block of code that you execute within a model. This block, which you define at the project level, can contain one or more functions. You define a MAS map in a Calculate window to bind a function to any of its input windows. This binding acts as the input handler for the Calculate window.

Consider the following continuous query (Figure 1.10):

Figure 1.10: Continuous Query Containing a Source Window Streaming into a Calculate Window



The Source window Trades contains a schema that specifies the fields that define the structure of incoming events. Events stream into the Source window through a file and socket publisher connector. The input data is contained in a file named input.csv in the current working directory.

Example Code 1.13

```
<window-source name='Trades' index='pi_RBTREE'>
          <schema>
            <fields>
              <field name='tradeID' type='string' key='true'/>
              <field name='security' type='string'/>
              <field name='quantity' type='int32'/>
              <field name='price' type='double'/>
              <field name='traderID' type='int64'/>
              <field name='time' type='string'/>
            </fields>
          </schema>
          <connectors>
            <connector class='fs' name='pub'>
              <properties>
                <property name='type'>pub</property>
                <property name='fstype'>csv</property>
                <property name='blocksize'>1</property>
              </properties>
            </connector>
          </connectors>
        </window-source>
```

A MAS module named module_1 is defined at the project level. It contains a function named compute_total written in Python. The Python code for compute_total is specified within the <code> element of the module. This function acts as the input handler for all events that are passed from the input window to the Calculate window.

Example Code 1.14

```
<mas-modules>
    <mas-module language="python" module="module_1" func-
names='compute_total' >
      <code>
        <![CDATA[
         def compute_total(quantity, price):
             "Output: total"
             total = quantity * price
         return total
        ]]>
      </code>
</mas-module>
  </mas-modules>
```

Data relevant to the security being traded, the quantity of shares, the current prices, and the person who performed the trade are streamed into the Calculate window named pw1. (Before SAS Event Stream Processing 5.2, support for MAS modules was provided through the Procedural window.) At the map level of the Calculate window, the window map binds the compute_total function of module_1 to the input window.

Example Code 1.15

```
 <window-calculate name='pw1' algorithm='MAS'>
          <schema>
            <fields>
              <field name='tradeID' type='string' key='true'/>
              <field name='security' type='string'/>
              <field name='quantity' type='int32'/>
              <field name='price' type='double'/>
              <field name='traderID' type='int64'/>
              <field name='time' type='string'/>
              <field name='total' type='double'/>
            </fields>
          </schema>
          <mas-map>
            <window-map module="module_1" revision="0" source="Trades"
function="compute_total"/>
          </mas-map>
 </window-calculate>
```

Edges connect the Source window to the Calculate window.

Example Code 1.16

```
<edges>
        <edge source='Trades' target='pw1' role='data'/>
</edges>
```

Suppose that you stream the following events through the Source window (Figure 1.11):

Figure 1.11: Events That Stream Through the Trades Window

Opcode	tradeID	security	quantity	price	traderID	time
Insert	TID1234329	ibm	1000	100.000000	10004	08/Jul/2012:08:1...
Insert	TID1234328	sap	1000	32.000000	10003	08/Jul/2012:08:1...
Insert	TID1234327	ibm	1000	100.300000	10002	08/Jul/2012:08:1...
Insert	TID1234326	sap	1000	34.300000	10003	08/Jul/2012:08:1...
Insert	TID1234325	ibm	1000	100.400000	10004	08/Jul/2012:08:1...
Insert	TID1234324	ibm	1000	100.300000	10004	08/Jul/2012:08:1...
Insert	TID1234323	ibm	1000	100.200000	10004	08/Jul/2012:08:1...
Insert	TID1234322	sap	750	34.200000	10003	08/Jul/2012:08:1...
Insert	TID1234321	ibm	1000	100.100000	10002	08/Jul/2012:08:1...

Figure 1.12 shows the resulting events from the Calculate window.

Figure 1.12: Results From the Calculate Window

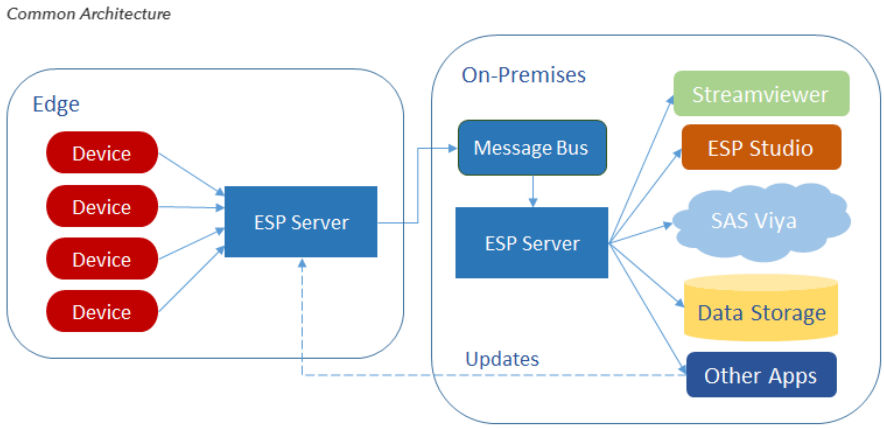
Opcode	tradeID	security	quantity	price	traderID	time	total
Insert	TID1234329	ibm	1000	100.000000	10004	08/Jul/2012:0...	100,000.000000
Insert	TID1234328	sap	1000	32.000000	10003	08/Jul/2012:0...	32,000.000000
Insert	TID1234327	ibm	1000	100.300000	10002	08/Jul/2012:0...	100,300.000000
Insert	TID1234326	sap	1000	34.300000	10003	08/Jul/2012:0...	34,300.000000
Insert	TID1234325	ibm	1000	100.400000	10004	08/Jul/2012:0...	100,400.000000
Insert	TID1234324	ibm	1000	100.300000	10004	08/Jul/2012:0...	100,300.000000
Insert	TID1234323	ibm	1000	100.200000	10004	08/Jul/2012:0...	100,200.000000
Insert	TID1234322	sap	750	34.200000	10003	08/Jul/2012:0...	25,650.000000
Insert	TID1234321	ibm	1000	100.100000	10002	08/Jul/2012:0...	100,100.000000

For more information about using MAS modules in Calculate windows, see "Working with SAS Micro Analytic Service Modules" documentation at https://go.documentation.sas.com/?cdcId=espcdc&cdcVersion=6.2&docsetId=espan&docsetTarget=p1qv3axlms1ckmn1gv3hub74xmue.htm&locale=en.

Addressing Big Data and the Internet of Things

SAS Event Stream Processing enables solutions to meet IoT challenges through reference architectures, which are discussed in detail in Chapter 5. Figure 1.13 shows a common reference architecture.

Figure 1.13: Internet of Things Reference Architecture



Here, event streams from sensor devices are ingested by an ESP server running on an edge device. The edge device could be on an airliner, a tanker, or a locomotive. There could be any number of these edge devices in operation. The edge ESP servers can be running ESP models that are configured to detect worrisome combinations of sensor values. Specific events of interest trigger alerts to the operator of the edge device, which are then communicated through a message bus to the on-premises ESP server.

The on-premises ESP server aggregates events that are received from the edge devices and performs further analytical model refinement as well as analysis, monitoring, and reporting functions. This ESP server can send analyzed events to a data storage device or to other applications for further processing. SAS Event Stream Processing Streamviewer can be used to visualize events as they stream through the ESP server. SAS Event Stream Processing Studio can interact with the ESP server to create, edit, upload, publish, and test event stream processing models. This enables ongoing streaming project development and analytical model development. The on-premises ESP server can also interact with a SAS Cloud Analytic Services (CAS) server, which is shipped with SAS Viya.

This common reference architecture can be applied to any number of real-world use cases. Consider the analysis of power transmission on a grid that is monitored with Phasor Measurement Units (PMUs). PMUs take high-frequency measurements of power frequency, voltage, current, and phase angle at different locations along the grid. They use GPS signaling to ensure the time accuracy of measurements taken at different locations. High-frequency and time-accurate measurements enable operators and engineers to make informed decisions as they monitor and control the grid.

To maintain stability on a power grid, operators and engineers can use this reference architecture to do the following:

- Understand the steady state operation of the grid using streaming analytics that calculate descriptive statistics from real-time and historical PMU measurement data.
- Detect anomalous events on the grid in real time by monitoring PMUs and comparing measurements with steady state descriptive statistics.

- Categorize events by type, count, intensity, time, location, and equipment type.
- Respond appropriately to detected events.
- Capture data for post-event analysis using a high-volume data storage platform.

Edge Model to Process Measurements from a Power Substation

In the power-grid use case, the edge model ingests events that consist of PMU measurement data from various locations (power substations). The event schema includes the following:

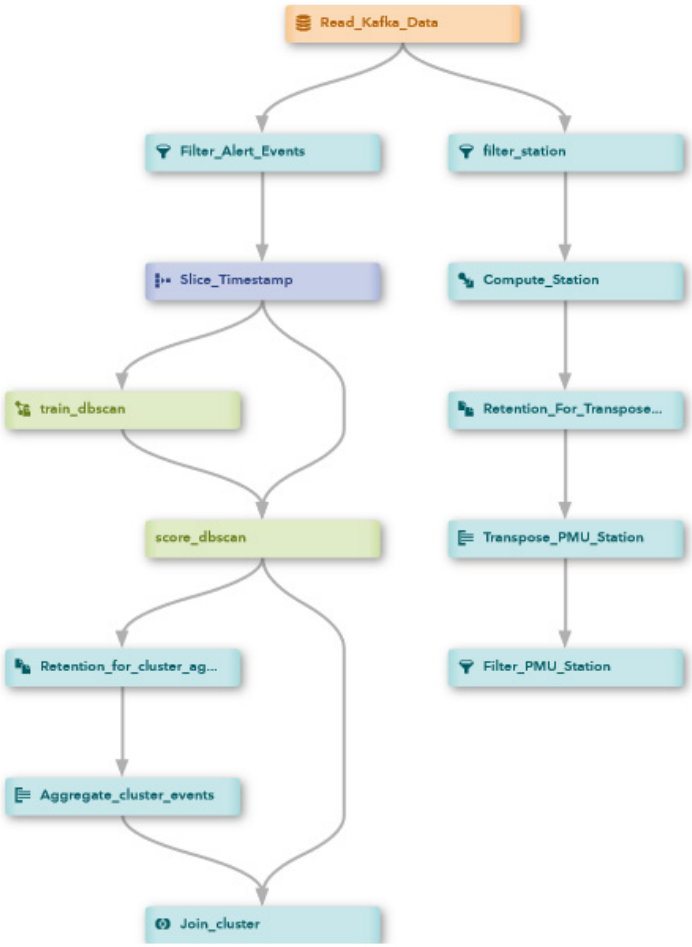
- A field for the measurement type, such as voltage, current, frequency, and angle measurement.
- A field for the time of the event's observation.
- A field that specifies the measurement's value.

The edge model includes a Source window and a series of derived windows to detect events of interest and perform analytical calculations on the PMU data. The model calculates and compares forecasted data with observed data with a series of Aggregate, Compute, Copy, Counter, Functional, and Join windows. A downstream Compute window calculates the control limits for each PMU based on derived statistics. After the control limit data is calculated, it is published from the Compute window to a Kafka broker through a Kafka subscriber connector.

On-Premises Model for Further Processing

The on-premises model (Figure 1.14) reads data from the Kafka broker that receives data from the ESP servers on the edge.

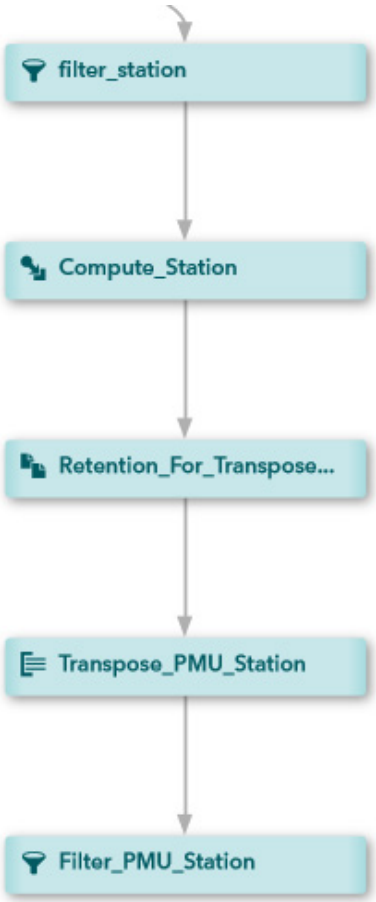
Figure 1.14: On-Premises Model



A Source window in the on-premises model receives data from the Kafka broker using a Kafka publisher connector. The schema of the Source window on-premises is similar to the schema of the Compute window that published to the Kafka broker from the edge. All the fields are the same, but here, the Source window has a key field that is unique to the location (substation) and measurement type rather than a key field unique to just the measurement type. From the Source window, the model splits the data stream into two branches:

The first branch combines incoming data events with unique measurement types and locations for a given timestamp into a single event for each timestamp with fields for measurement values of each type and location (Figure 1.15).

Figure 1.15: First Branch of On-Premises Model

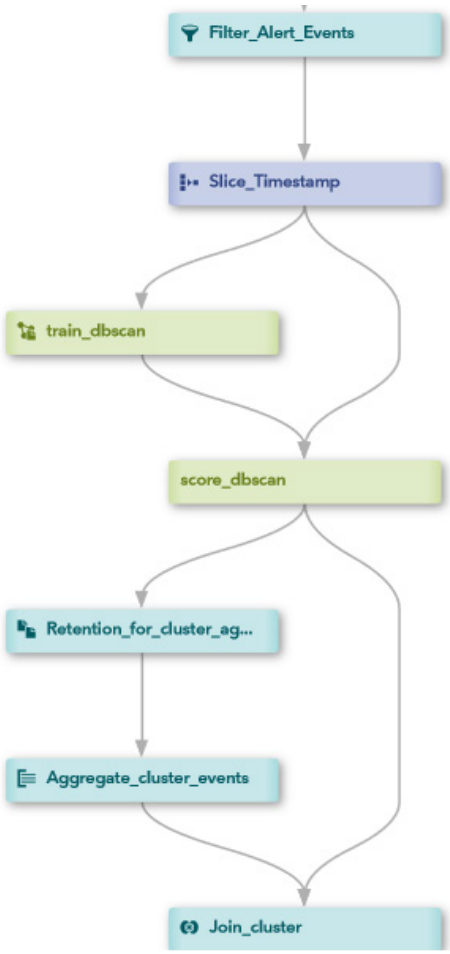


A Compute window creates an event where the measurement value field is the value for the location and measurement type corresponding to the station field. The window places a null value in the measurement value fields of all locations and measurement types that do not correspond to the station field.

An Aggregate window downstream of the Compute window populates null event fields with the last non-null value affecting that field within each 30-second interval. A CAS adapter subscribes to the events streaming through the Aggregate window in order to store the event data for further processing.

The second branch sends data events with values outside the upper and lower control limits to a training and scoring clustering model that is used to detect events of interest across the power grid (Figure 1.16).

Figure 1.16: Second Branch of On-Premises Model



A Filter window filters events with values outside the upper and lower control limits. Values outside the control limits are considered alerts.

A Procedural window calculates geographical information for the stations in the event and creates new fields for the ZIP code, longitude, and latitude of each event's station.

A Train window downstream of the Procedural window clusters the events based on the timestamp, using the DBSCAN clustering algorithm. A Score window receives data events from the Procedural window and model events from the Train window. The Score window scores the data events using the trained DBSCAN mode. It assigns a cluster ID to each event based on its timestamp, along with the minimum distance of the event to the cluster centroid.

An Aggregate window downstream of the Score window counts the events in each cluster for each minute interval. A Join window combines the events that stream through the Aggregate window with the events from the Score window by cluster ID and ZIP code.

Two SAS Cloud Analytic Server (CAS) subscriber adapters read in data events from the two branches of the model. After that data has been loaded to CAS tables, you can interact with snapshots of the data streams with SAS Visual Analytics.

Conclusion

SAS Event Stream Processing is an enterprise-class application designed to address the Big Data and IoT challenges faced by most modern businesses. It provides the flexibility to build event stream processing models that ingest event streams from any source and apply business logic of any type and complexity. You can use it to analyze structured and unstructured data sources, including video, text, and image classification and identification, using advanced analytics with embedded AI and machine learning capabilities. As a component of the wider SAS software suite of applications, it can tap directly into the proven capabilities around data integration, data quality, and advanced analytics.

Returning to the observations of Andrew Psaltis, "[T]he digital universe is doubling in size every two years. ...A great way of putting that in perspective [is to realize that] if a byte of data were a gallon of water, in only 10 seconds there would be enough data to fill an average home. In 2020, it will only take 2 seconds." SAS Event Stream Processing has the capability to process and analyze those streams no matter how quickly the digital universe grows. You will never be underwater when you use it to capitalize on the opportunities presented by the Internet of Things.

About the Contributors

As Principal Technical Training Consultant in SAS Education, **Robert Ligtenberg** develops and delivers customer training courses in the Data Management space. Robert has a PhD in physics from North Carolina State University and an undergraduate degree from the University of Twente, the Netherlands.

As the Vice President of Research & Development for all Internet of Things (IOT) offerings at SAS, **Jerry Baulier** works closely with customers, partners, and industry analysts to help research and development teams at SAS develop IOT vertical solutions on the Event Stream Processing product suite, Analytics for IOT product suite, and the Quality Analytics product suite. Jerry holds a master's degree from Stevens Technical Institute and a bachelor's degree from UMAS Dartmouth.