# SQL_R2_Logo.jpg

SQL Server Technical Article

**Using SQL Server Integration Services and StreamInsight Together**

**Writer:** Ping Wang, Wee Hyong Tok, Alexei Khalyako, Xiaoyan Zhao, Renhe Li, Igor Grinev

**Technical Reviewer:** Mark Simms, Denny Lee, Prem Mehra, Peter Carlin, Douglas Laudenschlager, Carla Sabotta

**Technical Editor:** Beth Inghram

**Published:** December 2010

**Applies to:** SQL Server 2008 R2

**Summary:** The purpose of this paper is to provide guidance for enriching data integration scenarios by integrating StreamInsight with SQL Server Integration Services. Specifically, we look at the technical challenges and solutions for such integration, by using a case study based on a customer scenario in the telecommunications sector.

Copyright

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This white paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

© 2010 Microsoft Corporation. All rights reserved.

Contents

[Introduction 5](#_Toc280776558)

[Using StreamInsight and Integration Services Together 6](#_Toc280776559)

[Using Multiple Processes 7](#_Toc280776560)

[StreamInsight Embedded Inside Integration Services 8](#_Toc280776561)

[Integration Services Embedded Inside StreamInsight 9](#_Toc280776562)

[Telecommunications Operators Scenarios 10](#_Toc280776563)

[Overview of Telecommunications Operation 10](#_Toc280776564)

[CDR Data Characteristics 11](#_Toc280776565)

[Common Telecommunications Scenarios 12](#_Toc280776566)

[Callback 12](#_Toc280776567)

[Social Network Detection 13](#_Toc280776568)

[Analysis of Spam Attacks in SMS Messages 13](#_Toc280776569)

[SIM Card Clones 13](#_Toc280776570)

[Revenue Assurance 14](#_Toc280776571)

[Fraud Detection for International Roaming 14](#_Toc280776572)

[Case Studies 14](#_Toc280776573)

[Callback Detection 14](#_Toc280776574)

[StreamInsight Pipeline Component Description 16](#_Toc280776575)

[Callback Scenario Query 18](#_Toc280776576)

[How to Parallelize the Query 18](#_Toc280776577)

[Performance Results 21](#_Toc280776578)

[Lessons Learned 22](#_Toc280776579)

[Revenue Assurance 23](#_Toc280776580)

[Lessons Learned 26](#_Toc280776581)

[Fraud Detection for International Roaming 27](#_Toc280776582)

[Discussion on the Different Design Options 29](#_Toc280776583)

[Integration Services and StreamInsight Working Together 30](#_Toc280776584)

[Processing Out-of-Order Data 31](#_Toc280776585)

[Development Experience 31](#_Toc280776586)

[Scale-Out and Scale-Up 32](#_Toc280776587)

[Summary 32](#_Toc280776588)

# Introduction

Microsoft SQL Server Integration Services (SSIS) is a platform for building enterprise-level data integration and data transformations solutions. Integration Services can extract and transform data from a wide variety of heterogeneous and disparate sources (for example, relational, flat file, XML, and so on), apply a rich and powerful set of data transformations (for example, fuzzy lookup, data cleansing, aggregation, joins, and so on) and then load the data into one or more destinations. Integration Services customers can also visually design their data integration solutions using Business Intelligence Development Studio (BIDS).

Integration Services has been widely used today in different industries and scenarios, such as telecommunications, retail, commercial real estate, finance, and supply chain, as a powerful tool for performing data integration. Integration Services is commonly used to integrate data from different data sources, normalize the data, cleanse the data (removing duplication, validating the data, and so on), transform the data, and then load the result into a data warehouse. Customers build reports and OLAP cubes using the data that has been loaded into the data warehouse.

SQL Server 2008 R2 introduced a new platform called Microsoft StreamInsight. StreamInsight is a complex event processing engine that provides the ability to build rich temporal data processing applications over real-time event streams. StreamInsight offers the following strengths:

* Well-defined query algebra
* Rich temporal semantics
* Event-driven, near real-time processing
* Support for both online and historic data processing
* Flexible interfaces to bring data into and out of the system
* Ability to be easily embedded inside other applications

Both Integration Services and StreamInsight provide you with the ability to perform powerful data integration tasks at high volumes of data for enterprise scale applications. In this white paper, we discuss how you can build data integration scenarios that leverage the combined strengths of Integration Services and StreamInsight. All the solution architectures that are covered in this paper can be realized using Integration Services and StreamInsight capabilities that are part of SQL Server 2008 R2.

Specifically, we look at how the combination of Integration Services and StreamInsight can help you perform the following tasks.

|  |  |  |
| --- | --- | --- |
| **Task** | **Uses StreamInsight features** | **Uses Integration Services features** |
| Perform analysis of data based on time (temporal analysis), when a large volume of data is loaded into a data warehouse. | Y | Y |
| Perform real-time analysis of data by performing aggregation based on time intervals before the data is loaded into a data warehouse. | Y | Y |
| Process critical fast-moving data, extract the results, and load it into the data warehouse. Critical data is usually the information that needs to be extracted earlier, faster so that business decisions are made earlier, faster. | Y | Y |
| Implement a custom eventing engine.  The eventing engine must be able to handle continuous flow of events, and have the ability to deal different temporal windows of data. | Y | N |

The paper is organized as follows:

* In Section 2, we explore the different solution architectures, which leverage the capabilities of Integration Services and StreamInsight in SQL Server 2008 R2.
* In Section 3, we examine different real-life scenarios in the telecommunications industry, where the combination of Integration Services and StreamInsight can complement each other, and provide customers with the ability to perform real-time analytics while loading data into a data warehouse.
* In Section 4, we explore the key learnings from a proof-of-concept (POC) solution that shows how Integration Services and StreamInsight can be used for callback fraud detection for a real telecommunications customer. In the POC, we built a custom StreamInsight transform that can be used in existing SSIS packages for performing real-time analysis of the data that passes through a data integration process. We present smaller case studies (with sample code snippets) to show the other kind of projects in which StreamInsight and/or Integration Services can be used. We also present the performance results from running the POC in the SQLCAT customer lab.
* In Section 5, we describe the key challenges from the POC, and then we conclude.

# Using StreamInsight and Integration Services Together

In this section, we discuss various architecture options for integrating StreamInsight with SQL Server Integration Services. We show how we can use multiple StreamInsight and SSIS processes and make use of inter-process communication to allow them to work together. We also show how to use a single process that embeds one inside the other. The solution architectures described in this section can be built using Integration Services and StreamInsight capabilities that are part of SQL Server 2008 R2.

## Using Multiple Processes

The diagram shows StreamInsight and Integration Services, in separate processes.

Data Sources, Assets, Feeds, Sensors, Devices

StreamInsight Engine

**SSIS**

**Engine**

Data Warehouse

StreamInsight Input Adaptor

SSIS Output Adaptor

SSIS Process

StreamInsight Process

StreamInsight output as Integration Services input

Data Sources

StreamInsight Engine

**SSIS   
Engine**

SSIS Input Adaptor

Operations

SSIS Process

StreamInsight Process

Output Adaptor

Integration Services output as StreamInsight input

An example scenario is credit card data fraud detection, where StreamInsight and Integration Services can be used to perform real-time credit card fraud detection. All the detected fraud records are produced by the StreamInsight engine are sent to Integration Services. Integration Services performs cleansing, normalization, and transformation of the data before loading it into a data warehouse.

In order to communicate between the two processes, inter-process communication (IPC) mechanisms like Named Pipes, Shared Memory, or TCP/IP are used to exchange data between different processes.

The advantages of using this approach are:

* The two products are used in a loosely coupled way. There are no dependencies between the two products as long as data or events can be passed between the two products.
* The solution can be scaled to multiple machines. For example, multiple StreamInsight instances can be created on multiple machines. These instances can be used to process large amounts of data, and then which can be passed to Integration Services on another machine to perform data loading (or vice versa).

The disadvantages of this approach are:

* The IPC mechanism needs to be built between the two processes.
* A reliable synchronization mechanism needs to be identified between the two processes for the different stages of execution.
* Two different products must be used, which have different ways of acquiring, processing, and outputting data, different programming languages, and different management tasks.

## StreamInsight Embedded Inside Integration Services

StreamInsight is designed to be easily embedded within an application. Because both Integration Services and StreamInsight are in the same process, they can share the same available memory. The event/data transfer between Integration Services and StreamInsight can be optimized for high throughput. There are several approaches for using StreamInsight in the SSIS pipeline.

One of the approaches is to use StreamInsight as a source component in the SSIS pipeline. In this approach, the StreamInsight appears as a data source component inside Integration Services. It uses its own input adapter to process external data; this removes the restriction of batch processing of Integration Services, enabling continuous, real-time data processing of large volumes of data. The output of StreamInsight can then be fed into other Integration Services components to continue other types of data transformations. (In the diagram, SI stands for StreamInsight.)

**Integration Services**

SSIS Output Adaptor

SI

Input Adaptor

Stream

Insight

SSIS

Transforms

SSIS

Transforms

**StreamInsight as an Integration Services Source**

Another approach is to place StreamInsight in the middle of the SSIS pipeline. With this approach, the StreamInsight data processing is restricted by the batch processing of Integration Services, so it may not be able to provide real-time insight into the data; however, StreamInsight can still provide the temporal data processing capability that Integration Services lacks.

**StreamInsight as One SSIS Pipeline Component**

**Integration Services**

SSIS Input Adaptor

SSIS

Transforms

SSIS Output Adaptor

Stream

Insight

SSIS

Transforms

The advantages of using this approach are:

* Temporal query semantics can be used in the ETL pipeline
* Large volumes of data can be processed in a continuous fashion
* Low-latency processing offers real-time insights into the data

The disadvantages of this approach are:

* StreamInsight and Integration Services compete for memory and CPU resources in the same process on the same machine.
* Because Integration Services and StreamInsight use the same process and run on one machine, it is hard to scale out to multiple machines. Integration Services needs to be scaled out manually. The solution needs to be manually partitioned and scaled out using multiple machines, which incurs additional management costs.

## Integration Services Embedded Inside StreamInsight

All SSIS pipeline components are embeddable pieces that can be leveraged by other products to do data transformations. If StreamInsight can reuse existing Integration Services components during event processing, it can take advantage of the rich set of Integration Services components that are available. This is illustrated in the following diagram, where Integration Services components are wrapped using the StreamInsight operator interface, and used during stream processing.

StreamInsight

Data

Cleansing

Fuzzy

Lookup

The benefits of this approach are:

* It leverages functionalities of both Integration Services and StreamInsight.
* It uses StreamInsight to achieve scale-out and high availability (HA).
* Integration Services and StreamInsight share StreamInsight’s scheduling, so the two do not compete for scheduling.

The challenge of this approach is that in order to allow Integration Services components to be used as StreamInsight operators, significant work needs to be done with existing Integration Services components and the StreamInsight query processing model.

As previously stated, when StreamInsight and Integration Services processes are executing on the same machine, they compete for resources. Integration Services and StreamInsight cannot globally balance the resources from both processes. You can use Windows System Resource Manager to manage the resources allocated for each of the processes.

# Telecommunications Operators Scenarios

In this section, we discuss the technologies to use to perform data integration in telecommunications scenarios. We start with an overview of the way the data we work with in our example is generated and some of the common issues encountered, and we provide some basic information about two companies we worked with. We then discuss other common issues in telecommunications and show ways in which Integration Services and StreamInsight can work together to address them. Finally, we walk through some case studies that show how Integration Services and StreamInsight can be used together to leverage the strengths of each technology.

## Overview of Telecommunications Operation

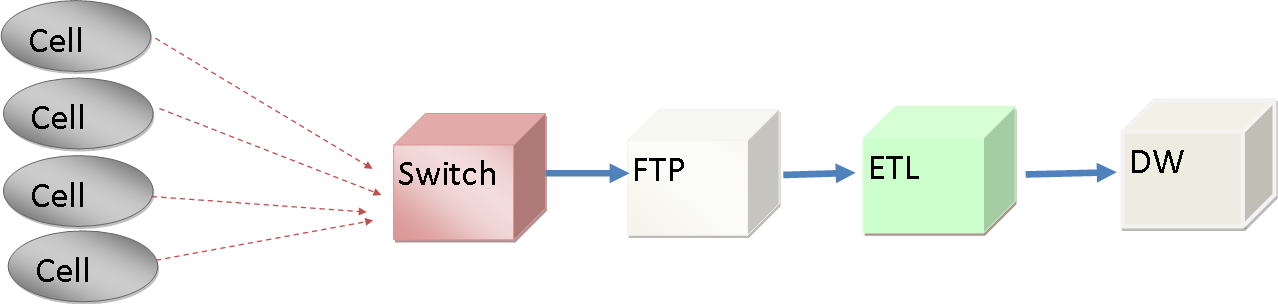
Many telecommunications operators handle both cellular and landline calls. The information for each of these calls is captured in a *call detail record (CDR)*. CDRs capture the information for call setup and completion. The format of a CDR varies among different telecommunications operators, but they all serve more or less the same purpose (that is, to track how a subscriber uses telecommunications services). These CDRs are then dumped on a periodic basis into CDR files. Extract-Transform-Load (ETL) processes are then used to load the CDR files into a corporate data warehouse.

Cellular and landline calls have several distinct differences. The most important difference for this discussion in the role of switches and their impact on how CDRs are created, because the issues these differences create affect data collection and the ETL process. Cellular calls pass through several switches during the duration of the call. As part of the post-call assembly process, the records from the various switches are assembled into a single-call, summary CDR. If data from a switch is delayed, a complete call CDR can’t be assembled until the data arrives. Meanwhile, follow-up call involving other switches may have completed. Consequently, the subsequent CDR will arrive first.

Landline CDRs are not produced by switches. Instead, the switches provide a stream of signaling information that is assembled into CDRs, reflecting a complete call to a billing system. Landline CDRs capture information from two signaling streams: one from the originating switch, and the other from the receiving switch for a call. A CDR record, reflecting a complete call, is assembled by downstream system after processing the signaling information. Hence, a landline CDR requires a call to be completed before the CDR can be assembled.

Data processing follows a standard format in telecommunications companies. The switches capture the call details and periodically dump the data to CDR files. Depending on how the cell switches are configured, the cell switches can perform a dump when the switches’ memory is full or perform a dump at specific time intervals.

The CDR files from the cell switches are transferred via FTP to a staging machine. ETL process is then invoked to load the data from the CDR files to a data warehouse. As part of ETL, different types of transformations can be used as part of the data processing pipeline (such as data validation, filtering, cleansing, and lookups).



**Data Integration in Telecommunications Scenarios**

## CDR Data Characteristics

Due to the method in which the data is collected at the various cell switches, CDR data has the following characteristics:

* **Large amount of switches** - Most telecommunications operators maintain a large number of switches.
* **Dirty data** - Due to various hardware issues, the CDR data can be dirty. For example, some CDRs might have a call end time that is earlier than the call start time, and for dropped calls, the information in the CDR can be incomplete.
* **Out-of-order data arrivals** - Each call record file is dumped out by one switch. The call entries in one log file are not sorted by call time. Instead, the order the data is dumped to the log file depends on the internal implementation of the switch. Some switches use buffer threshold to dump out data; some switches might be configured to dump data on a schedule. Due to the huge number of switches, working in parallel with multiple files delivered by the switches introduces even more late-arriving data.

The following table captures the data characteristics of cellular CDR data for two different telecommunications operators in Asia and North America.

|  |  |  |
| --- | --- | --- |
| Data Point | Telco A | Telco B |
| Data volume | 1.5 GB/hour | 300 GB/hour |
| Size of files that are loaded | 10 MB | Ranges from 500 MB to 2.5 GB |
| Number of switches producing data | 70 or more | 500 |
| Time in which the entire ETL process must finish | Keep up with incoming rate | 45 minutes |
| Number of servers running Integration Services to perform ETL | 1 (8-core x64 box) | 64 |
| Size of data warehouse | 6 terabytes (compressed) | 50 terabytes |
| Length of time files are kept | Kept on FTP for 1 week Kept on local ETL server for 2 weeks | 2 days |

## Common Telecommunications Scenarios

In this section, we discuss several common telecommunications scenarios where Integration Services and StreamInsight can be used. These include the following:

* Callback
* Social network detection
* Analysis of spam attacks
* Detection of SIM card clone fraud
* Revenue assurance
* Fraud detection for international roaming
* Monitoring network traffic

### Callback

In this scenario, telecommunications operators want to analyze the calling patterns for the operation of a commercial callback service. These patterns, known as *callback patterns*, enable operators to identify errant noncommercial subscribers who are operating callback sites. A callback pattern is defined as follows:

*A user (with number A) calls number B for a very short period of time (for example, less than 1 second), and within a short period of time (for example, 20 seconds), the user receives a call from either number B or C. This is referred to as a callback.*

One possible approach to identifying callback trends is to first load the CDR data into a data warehouse, and then to devise a self-join query to identify the records that exhibit a callback pattern. This approach has the following limitations:

* **Large amount of data** - The process of loading the CDR files can be modeled as a data stream, where each event in the data stream is a single CDR record. After each batch of CDR files is loaded into the database, the data analysts at the telecommunications company will have to run the same query. Most importantly, the data that needs to be processed continues to grow as new CDR files are loaded. Data partitioning can be used to reduce the amount of data that is used in the callback trend computation.
* **Latency** – In this approach, the data must first be loaded into the data warehouse because it can be used for callback trend computation. This increases the latency in which telecommunications operators can obtain the results.

### Social Network Detection

Social network detection tries to use information regarding the interactions between people through different social activities like phone calls, instant messages, and email to analyze social network groups. This is valuable for service companies like Telco to better understand customer behavior and target relevant offerings to new and existing customers.

Because this detection is usually based on a large amount of data and the computation cost is usually very high, it is challenging to come up with a good algorithm to efficiently and incrementally detect the social community as information comes and goes. Also as time passes and the situation changes, the social structure may change, so it is valuable to continuously monitor the trend and modify the social network groups as necessary.

### Analysis of Spam Attacks in SMS Messages

*SMS attack* is a form of spamming directed at the text messaging service of a mobile phone. The users of a text messaging service see unsolicited commercial advertisements being sent to their phones through text messaging. This is annoying and costly for the cell phone users, and it is challenging for the message service provider to detect and filter out these messages.

Messages involved in this type of attack normally have following characteristics: the SMS is sent from a single number with a high sending ratio (for example, several hundred per minute). Because telecommunications companies have identified these characteristics, they can use StreamInsight to detect an SMS attack and react accordingly by taking action such as blocking the number, marking the number as suspicious, or charging more.

### SIM Card Clones

A SIM card is used to uniquely identify the caller (which pays for the call). By cloning SIM cards illegally, malicious users can easily steal other people’s SIM cards and make calls that are paid for by the victims. Normally each phone call makes the following entries in the CDR file:

* MSISDN –Mobile Station International Subscriber Directory Number. This is the mobile phone number.
* IMEI -- International Mobile Equipment Identity. This is the phone device identification.
* IMSI -- International Mobile Subscriber Identity. This is the SIM card identification number.
* RegionalID – The region where the call is initiated from.

There are different ways of detecting SIM card clones. For example, you can track the abnormal cases where consecutive phone calls made from the same SIM card are coming from different regions, and it is impossible for the same SIM card to physically transfer between these locations in the time between the two calls.

### Revenue Assurance

In order to guarantee revenue, telecommunications organizations need to make sure they don’t charge less than what their users consume. One of the mechanisms is to compare the data in the data warehouse against the data that are stored in the billing system.

The billing system contains data that is already prepared for billing customers. The data from the billing system is already aggregated, while the data warehouse contains raw data. You can use Integration Services to perform loading of the data by reading from several switch flat files and then correlate with the data in billing system to see whether the data matches. For example, for a particular user, the service database may show that the SMS service was used 100 times, and the billing system may show that the user sent 100 messages, which are billed at a per-message rate. The two data sources match.

### Fraud Detection for International Roaming

According to the GSM Association, minimizing losses on international roaming is a common task requiring close and reliable cooperation of the parties involved. If it is received in a timely fashion, roaming call data can be used by the telecommunications companies involved to detect roaming fraud before it becomes a problem.

When users travel internationally, their phones are registered in the telecom networks of the areas in which they use their phones. This is known as *roaming*. Billing rules depend on the call. A local call uses the service of the local provider, and an international call may use the service of the user’s home provider or another international network. However, because the user is not a subscriber of the local network, this provider may initiate crosscharges with the user’s cell network provider to ensure that the services the users used are appropriately paid for. This ensures that the local provider is paid for the service it provides to the roaming user.

# Case Studies

## Callback Detection

In this section, we present a case study on how Integration Services and StreamInsight can be used in a data integration solution. In order to make use of Integration Services and StreamInsight for callback detection, a custom SSIS transform needs to be built. The custom SSIS transform embeds the StreamInsight engine, which is then used for callback pattern detection during ETL processing.

A callback scenario is defined as follows:

*A user (with number A) calls number B for a very short period of time (for example, less than 1 second), and within a short period of time (for example, 20 seconds), the user receives a call from either number B or C. This is referred to as a callback.*

We make use of a real dataset from a telecommunications company, called Telco A. The data has been post-processed to remove sensitive information. Each record consists of the following columns:

* Calling number
* Called number
* Time when the call is made
* Duration of the call

The table shows sample data from the Telco A dataset.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SwitchCode | CallingNum | CalledNum | Date | Time(s) | Period |
| 288 | 0920-112-xxx | 898683038-832-xxx | 1/26/2009 | 82883 | 580 |
| 288 | 0920-112-xxx | 898683038-832-xxx | 1/26/2009 | 82883 | 580 |
| 288 | 0998-129-xxx | 000-558-xxx | 1/26/2009 | 82929 |  |
| 288 | 0998-231-xxx | 0992-638-xxx | 1/26/2009 | 82928 | 86 |
| 288 | 0998-231-xxx | 0992-638-xxx | 1/26/2009 | 82928 | 86 |
| 288 | 0929-836-xxx | 886932-220-xxx | 1/26/2009 | 82906 |  |

From the snapshot, we observed that dirty data is present in the dataset. For example, the duration of a call might be missing, or the time captured is not in the correct format.

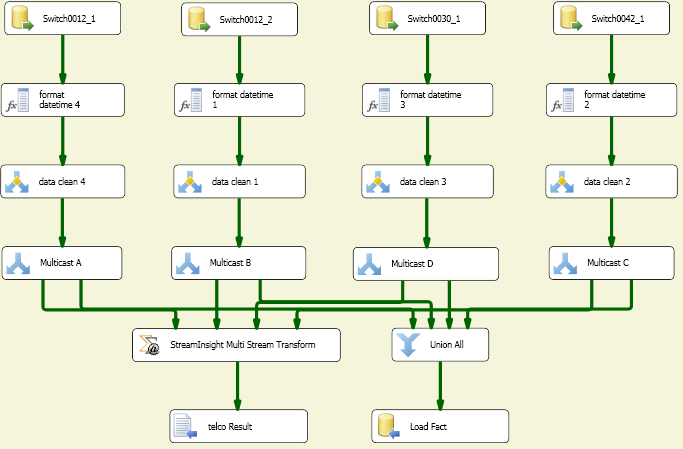
When the dataset is processed, the following challenges need to be addressed:

* How to deal with out-of-order data
* How to deal with many input switch streams
* How to deal with the large volume of data that needs to be processed
* How to reduce the latency for producing results

In this paper, we describe one of the solutions for using StreamInsight and Integration Services for processing the data. In this solution, the StreamInsight engine is embedded in a SSIS transform. An SSIS package is designed to perform the following:

* Data cleansing: Remove the dirty data during data processing and standardize the date time format.
* Detect callback scenarios: We implement a special SSIS transform, StreamInsightMultiStreamTransform, that contains an embedded StreamInsight engine. It is used to process the data from SSIS pipeline and output the result.
* Load the data into data warehouse: The cleansed data should also be stored into data warehouse for further analysis.

These steps are represented in a single SSIS data flow task. This allows the SSIS package (which makes use of StreamInsight) to be easily deployed and scheduled using existing tools. The following diagram shows an overview of the SSIS package (with the StreamInsight transform).

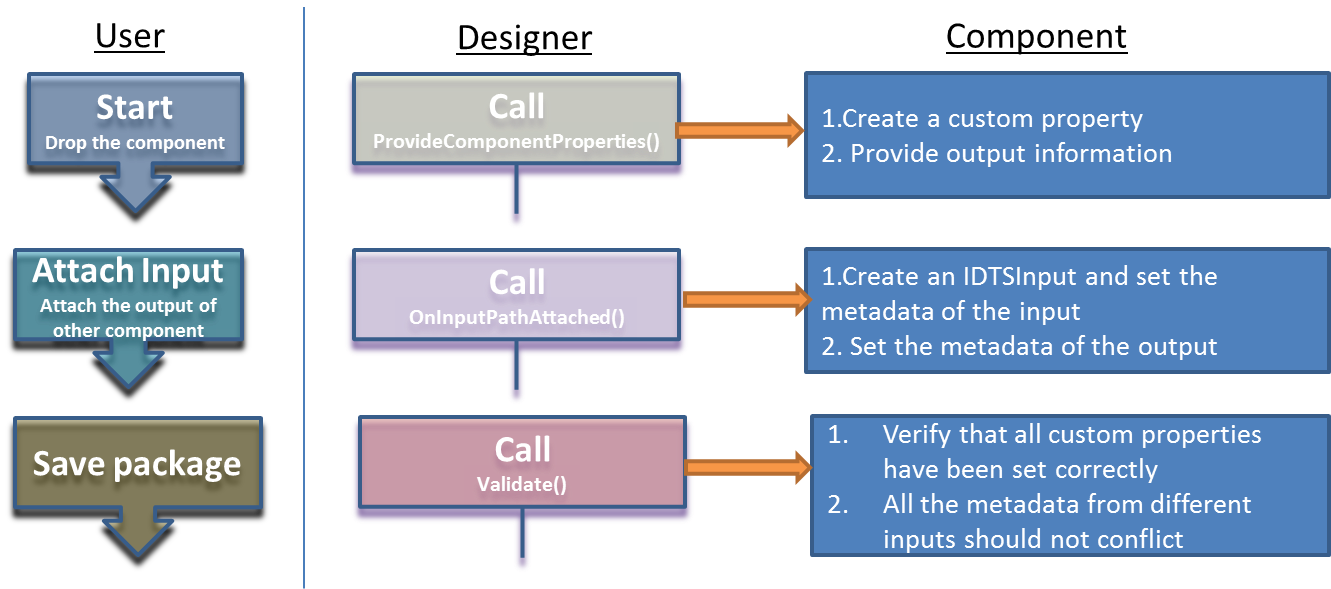


### StreamInsight Pipeline Component Description

The core of this package is the StreamInsight Multi Stream Transform component. The transform obtains data from different input streams and processes that data based on the StreamInsight query. The output of this transform is sent to downstream Integration Services components for processing.

In order to implement the component, the design-time and run-time functions need to be implemented. The design-time functions provide the behaviors when the component is added, deleted, validated, and saved. The run-time functions define how the component is used to process the data that is passed to it.

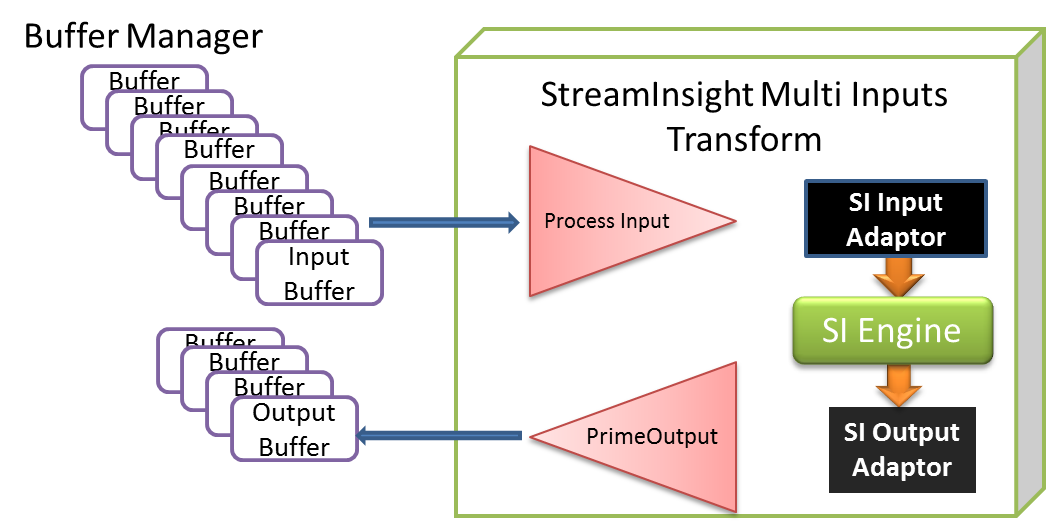
In our transform, we implemented the following design-time functions: SetUsageType(), OnInputPathAttached(), and ProvideComponentProperties().



During run time, StreamInsight Multi Stream Transform creates a CEP server in memory to process all the incoming data. The following run-time functions are implemented:

* PreExecute(): This method is called before the execution of this transform. It starts to create the StreamInsight Server, generates the input and output adaptor, registers the query into the server, and starts the server.
* ProcessInput(): This method gets the data from the input and passes it to the StreamInsight input adaptor for analysis. This is called for every buffer in the input until the end of the dataset.
* PrimeOutput(): This method gets the data back from the output and puts it into the SSIS pipeline buffer.

Here is a picture to indicate how the data is transferred within that component. (In the picture, SI stands for StreamInsight.)



### Callback Scenario Query

The StreamInsight query in this callback scenario involves a self-join that joins the incoming stream with itself and finds all the conditions that satisfy the callback pattern.

// Change event lifetime of each event from the call data stream to one minute so that events

// can be joined with other events that arrive less than one minute after the call begins.

// This is needed because the StreamInsight Join is a join over time – events can only join

// when their lifetimes overlap.

CepStream<PhoneDetail> streamOneMinute = inputStream.AlterEventDuration(e1 => TimeSpan.FromMinutes(1));

// Do a self-join to find the callback pattern

CepStream<PhoneOrder> callbackStream =

from e1 in streamOneMinute // self join

from e2 in streamOneMinute

where (e1.CallingNum == e2.CalledNum) && // A == Y

(e1.CalledNum != e2.CallingNum) && // B != X

(e1.mydate <= e2.mydate) && // first call happens earlier

(e1.Duration < 1) && // first call duration is short

(e2.Duration > 10) // second call duration is long

select new PhoneOrder

{

CallingNum = e1.CallingNum,

CalledNum = e1.CalledNum,

CalledNumB = e2.CalledNum,

CallingNumB = e2.CallingNum,

duration1 = e1.Duration,

duration2 = e2.Duration,

start1 = e1.mydate,

start2 = e2.mydate,

};

What the query tries to do is to join the call data stream with itself to find patterns that would lead to a callback situation. Event e1 represents the first call (A => B) and event e2 represents the second call (X => Y). The join conditions are:

1. A == Y -- second call’s callee is first call’s caller
2. B != X -- it is not a direct callback
3. e1.mydate <= e2.mydate -- e2 happens after e1
4. e1.Duration < 1 -- e1’s call duration is less than 1 second
5. e2.Duration > 10 -- e2’s call duration is greater than 10 seconds

The output of the query tries to capture all interesting information from the two events – all call numbers involved, both call durations, and call start time.

### How to Parallelize the Query

The query in the previous section works fine, but it won’t scale; the most CPU-consuming work is the join, but the join is not currently parallelized to leverage multiple cores of a computer. In order to make the query parallelizable, you could rewrite the query by leveraging the equi-join fact of the query so that the input stream can be partitioned by the calling number(A) of first stream and called number(Y) of second stream. StreamInsight has a built-in operator called Group&Apply that you can use to group the input streams and parallelize the join operation inside each apply branch.

For example, you can use the following query, which contains Group&Apply, to enable parallel processing of incoming stream.

// Change event lifetime of each event from the call data stream to one minute so that events

// can be joined with other events that arrive less than one minute after the call begins.

// This is needed because the StreamInsight Join is a join over time – events can only join

// when their lifetimes overlap.

CepStream<PhoneDetail> streamOneMinute = inputStream.AlterEventDuration(e1 => TimeSpan.FromMinutes(1));

// Create original call stream where each event is treated as an original call

var originalCallStream =

from e1 in streamOneMinute

where e1.Duration < 1

select new

{

Id = 1,

CallingNum = e1.CallingNum,

CalledNum = e1.CalledNum,

Duration = e1.Duration,

mydate = e1.mydate

};

// Create CallBack stream where each event is treated as a callback

var callBackStream =

from e2 in streamOneMinute

where e2.Duration > 10

select new

{

Id = 2,

CallingNum = e2.CallingNum,

CalledNum = e2.CalledNum,

Duration = e2.Duration,

mydate = e2.mydate

};

// Union as one stream because Group&Apply only works with one stream

// The way to differentiate them is the Id field – Id = 1 means original

// call stream; Id = 2 means callback stream

var union = originalCallStream.Union(callBackStream);

// Group by join key, which is caller number for the first stream and callee number in

// the second stream

var partition =

from e in union

group e by new { match = (e.Id == 1) ? e.CallingNum : e.CalledNum };

// Join inside a Group&Apply to enable parallel processing

var callBackStream = partition.ApplyWithUnion(applyInput =>

from e1 in applyInput.Where(e => e.Id == 1)

from e2 in applyInput.Where(e => e.Id == 2)

where e1.mydate <= e2.mydate && e1.CalledNum != e2.CallingNum

select new PhoneOrder

{

CallingNum = e1.CallingNum,

CalledNum = e1.CalledNum,

CalledNumB = e2.CalledNum,

CallingNumB = e2.CallingNum,

duration1 = e1.Duration,

duration2 = e2.Duration,

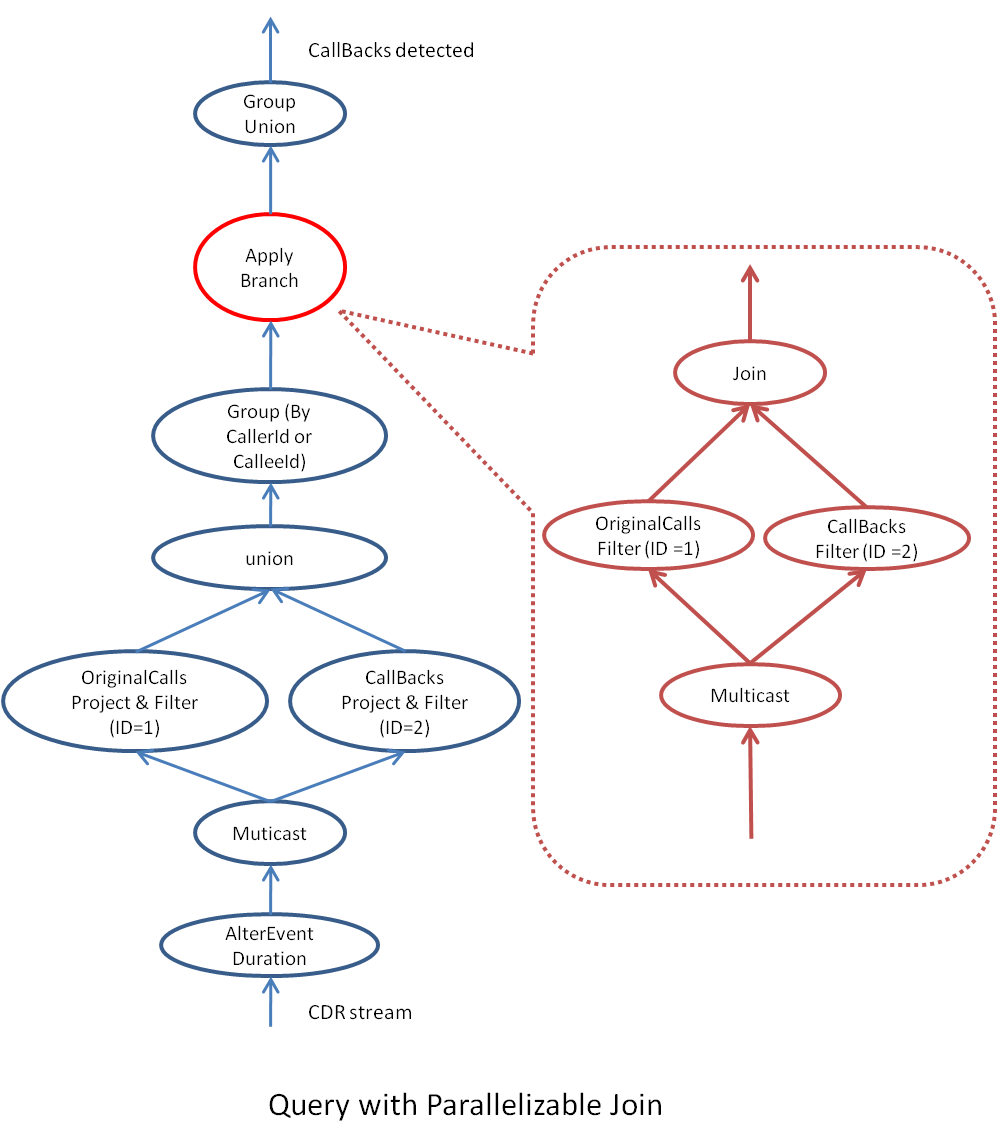
start1 = e1.mydate,

start2 = e2.mydate,

});

In this query, CDR calls are partitioned in two ways – one is partitioned by callerNum, the other is partitioned by calleeNum. The join can then run in parallel against the two partitioned streams. Due to the limitation that Group&Apply can only work on one input stream, the two streams are temporarily merged into one stream with an additional field ID to differentiate them (Id = 1 represents event from the caller stream, and Id = 2 represents event from the callback stream). Inside Group&Apply, the stream is separated again by using a multi-cast and filter before streams can be joined.

The following diagram illustrates the query.



Inside the red dotted frame, the logic of one apply branch is shown – it is essentially splits the input stream into two streams (CallingStream and CallBackStream) and then joins them to find the callback pattern you want to detect. The rest of the code is just preparation for making the join parallelizable.

### Performance Results

We performed the tests using SQL Server 2008 R2 Enterprise running on a 32-core (8 Quad-Core) HP DL785 Server, with 128 GB RAM. The 64-bit edition of Windows Server 2008 R2 Enterprise is installed on the server using the parallelized version query described earlier.

|  |  |
| --- | --- |
| **HP DL785 Server Details** | |
| Processors | 8 Quad-Core |
| Cores | 32 |
| Memory | 128 GB |

Summary of HP DL785 Server configuration

The following table shows the tests for various dataset sizes. In the Package column, package\_*x*\_*y*, *x* refers to the number of data sets used in each test, and *y* refers to the number of data sources in a package. For example, package\_20\_30 refers to a package that reads 20 sets of data. Each set of data reads 30 CDR files.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Package | **Number of**  **input rows** | **Number of output rows** | **Peak**  **memory** |  | **Dataset on disk (GB)** | **Data rate (GB/hour)** | ***Process time***  ***(hour)*** |
| package\_20\_30 | 30,000,000 | 42,018,591 | 20.05 |  | 3.52 | 5.10 | 0.69 |
| package\_20\_40 | 40,000,000 | 71,950,425 | 31.02 |  | 4.69 | 5.86 | 0.8 |
| package\_20\_50 | 50,000,000 | 109,835,550 | 39.00 |  | 5.86 | 5.05 | 1.16 |
| package\_20\_60 | 60,000,000 | 155,664,134 | 46.77 |  | 7.03 | 4.26 | 1.65 |
| package\_20\_70 | 70,000,000 | 209,453,693 | 55.14 |  | 8.20 | 3.81 | 2.15 |
| package\_20\_80 | 80,000,000 | 258,536,931 | 66.75 |  | 9.38 | 3.64 | 2.58 |
| package\_20\_100 | 100,000,000 | 418,606,597 | 84.17 |  | 11.72 | 2.90 | 4.04 |
| package\_100\_20 | 100,000,000 | 100,309,632 | 47.24 |  | 11.72 | 7.15 | 1.64 |

The data in the table shows that the average data processing rate is about 4.72 GB per hour. Using the hardware (presented earlier), we can process approximately a daily call volume of 113.28 GB.

For a 100 million rows dataset (that is, package\_20\_100), peak memory usage is about 84 GB. In package\_100\_20, the 100 million rows spans across 48 hours. In this case, the current time increment (CTI) is able to purge the in memory data every 2 hours. This allows the peak memory usage to stay around 47 GB.

### Lessons Learned

By building the prototype for the telecommunications callback scenario and running performance tests, we found that in order to have an efficient StreamInsight pipeline component, we need to tackle the following items. This section shows how we used Integration Services and StreamInsight to accomplish that.

**Efficiently synchronous data transfer between Integration Services and StreamInsight.**

In order to pass data from Integration Services to StreamInsight, the pipeline component needs to be able to work with data pushing patterns from both worlds. The StreamInsight input adapter needs to take the batch of events from Integration Services and push events into StreamInsight engine using the protocol StreamInsight requires. The adapter needs to avoid allocating new memory to buffer events in between. It is very easy to keep a .NET queue in between to smooth the difference in pace between StreamInsight and Integration Services. But if the amount of memory consumed is significant, the .NET Garbage Collection activity halts the processing most of the time, which makes it really slow to process the incoming call records.

**Integration Services needs to support parallel passing of multiple input streams.**

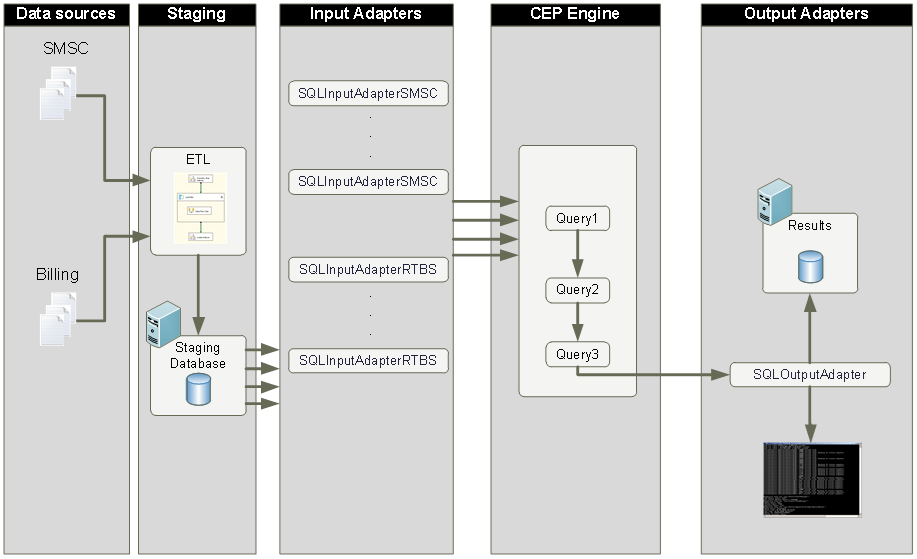
When there are multiple input streams to the same SSIS pipeline component, the handoff of buffers from the SSIS scheduler is serialized. This causes CPU to be wasted while the system waits for all events from all input streams to be available. We observed in our testing that the average CPU consumption in such cases is less than 50 percent because the handoff of data buffer is serialized. But multiple streams feeding into multiple pipeline components can be pushed in parallel. So to work around this problem, you can start multiple pipeline components and try to partition the query in Integration Services instead of in StreamInsight. However, in our prototype, we chose not to use this approach because it would have increased the complexity of the Integration Services workflow. We hope that Integration Services will support pushing of multiple streams into the same pipeline components in the future, which could solve this issue automatically.

## Revenue Assurance

In the revenue assurance scenario, we compared the data from billing system and the data warehouse to ensure that a customer is billed for all the services used. The data warehouse contains raw, unaggregated data. We used Integration Services to perform loading of the data by reading from several flat files that contain data from switches. The billing system contains data that is already prepared for billing customers. The data from the billing system is already aggregated, as follows.

|  |  |  |  |
| --- | --- | --- | --- |
| **Service** | **QTY** | **Price per service used** | **Total** |
| SMS | 100 | $0.50 | $50 |

This example shows how to check that no SMSs are missed and to verify that the customer’s bill accurately reports the number of SMSs. The following diagram illustrates the general architecture of the system.



Data from two sources resides on the file shares. The data consists of comma-delimited text files. SMS Centre (SMSC) data is written to the share every 15 minutes (about 30,000 files per day). The billing system extracts data once per day (about 100 files). Every night, an SSIS package is executed to load data from the text files to the staging database.

Then data is loaded from the database to the CEP engine by means of input adapters (one adapter type per source system). Due to the possibility of unordered events in the source files, tables in the staging database have clustered indexes on the EventDate field. That means that the data in the input sorted is sorted by EventDate key, which guarantees that events in the StreamInsight input adapter will always be ordered by date. Therefore before a specific event, no “late arriving data” occurs, and the time window is significantly reduced, causing less memory to be reserved for storing events. Because the input is sorted by the EventDate, we can issue CTI events often (for example, every 1,000 records). We also used a staging database as a StreamInsight data source, enabling data to be loaded in parallel through the use of partitioning (for example, by the last number of MSISDN).

The CEP engine filters and aggregates data using logic defined by the LINQ queries and writes results to the SQL Server database for the future analysis.

The following diagram illustrates the logic for the LINQ queries.



The following list describes the process:

1. Two input data streams are loaded:
   1. SMSC – raw data from SMS Centre
   2. Billing – data about charged events in the billing system
2. Each stream is filtered to exclude nonrelevant data (for example, in the Billing stream there is no need for incoming SMS events).
3. The streams are unioned.
4. Lookups are performed in the database tables to enrich data in the streams:
   1. Find region by MSC ID
   2. Find market code by subscriber’s MSISDN
   3. Find subscriber type (Pre-paid/Post-paid) by subscriber’s MSISDN
5. The data is aggregated over the time window and grouped by:
   1. Subscriber type (Pre-paid/Post-paid)
   2. Region
   3. Market Code
   4. DEF Code
6. The difference in the number of events is calculated between streams.
7. Groups where difference greater than a predefined threshold are selected.
8. The results are saved to a SQL Server database.

This logic is expressed using the following LINQ queries.

// Union and filter streams

for (int i = 0; i < streams1.Length; i++)

{

evt1 = evt1.Union

(from e in streams1[i]

where e.UsageAmount == "1"

select e);

}

for (int i = 0; i < streams2.Length; i++)

{

evt1 = evt1.Union

(from e in streams2[i]

where

e.State == "I“ && (

e.Type == "M" ||

(e.Type == "T" && e.SMSType == (int)SMSType.Outcoming)

)

select e);

}

// Group and aggregate

var evt2 = from e in evt1

group e by new { e.SubscriberType, e.SMSType, e.Region, e.MarketCode, e.DEFCode}

into grouped

from window in grouped.TumblingWindow(TimeSpan.FromMinutes(hoppingWindow))

select new EventPayloadOutput

{

SubscriberType = grouped.Key.SubscriberType,

SMSType = grouped.Key.SMSType,

Region = grouped.Key.Region,

MarketCode = grouped.Key.MarketCode,

DEFCode = grouped.Key.DEFCode,

RecCount = window.Sum(e => e.RecCount),

RecTotal = window.Count()

};

// Apply threshold

var evt3 = from e in evt2

where Math.Abs(e.RecCount) > threshold

select e;

### Lessons Learned

The key lessons learned include the following:

1. If you are dealing with unordered date events in incoming streams, you will need to keep periods between CTIs quite large. This may lead to excessive memory consumption. Consider breaking the process into two phases:
   1. Use an SSIS package to save events into a staging table with a clustered index on event date column.
   2. Use the StreamInsight SQL input adapter for reading events from a staging table and then push the events to the CEP engine.

Having a clustered index on the event date column in the StreamInsight input adapter means that events are ordered by date; therefore before a specific event, no “late arriving data” occurs, and the time window is significantly reduced, causing less memory to be reserved for storing events. Because the input is sorted by the EventDate, we can issue CTI events often (for example, every 1,000 records).

1. In order to process data from a staging table in parallel, add columns with event data distribution to the clustered index. For example, you can use last digit of subscriber's MSISDN as partitioning column. Then create ten instances of SQL Server input adapters, each of them querying staging table using query like:

'select \* from stage\_table where MSISDNLastDigit = N order by EventDate' where N = (0 … 9)

for each adapter's instance

1. If you don’t need to aggregate data over partitioning columns (for example, by customer’s region), you can scale out CEP processing by running input adapter instances for different regions on separate servers.

## Fraud Detection for International Roaming

When users travel internationally, their phones are registered in the telecom networks of the areas in which they use their phones. Billing rules depend on the call. A local call uses the service of the local provider, and an international call may use the service of the user’s home provider or another international network. However, because the user is not a subscriber of the local network, this provider may initiate crosscharges with the user’s cell network provider to ensure that the services the users used are appropriately paid for. This ensures that the local provider is paid for the service it provides to the roaming user.

Fraud Forum's estimates state that NRTRDE provides 90 percent reduction in losses from roaming fraud and saves hundreds of thousands of dollars annually. The new technology is a substitute for High Usage Report (HUR), an out-of-date and ineffective technology requiring roaming call data to be sent to partners not less than once each 4 hours. For comparison, NRTRDE's rules set this periodicity to every 5 to 15 minutes. Timely received roaming call data can be timely analyzed by roaming partners to detect and eliminate fraud before it can affect revenue.

The solution described here uses Integration Services in conjunction with StreamInsight to analyze roaming data in the near real-time. This solution can be implemented using Integration Services and StreamInsight capabilities that are part of SQL Server 2008 R2.



Binary files in the NRTRDE format are uploaded to the FTP server from roaming partners. Next, newly arrived data is loaded by an SSIS package that runs periodically and scans the FTP directory. Then data is parsed by the script component and sent to the StreamInsight transformation embedded in the SSIS pipeline. The transformation processes input over a set of LINQ queries and then passes results to the OLE DB destination. The saved results can be analyzed by a client tool to detect fraud activities.

In the standard scenario for StreamInsight-based system, all data processing logic (LINQ queries) is coded by the developer, compiled, and then deployed to the CEP server. Therefore any changes in data processing logic leads to changes in source code. This process requires an experienced developer with a solid background in .NET and LINQ knowledge.

The key component of this solution is the Fraud Rules Designer, which is a user-friendly tool for defining data processing logic in StreamInsight transformation. The main idea is to provide UI where a user can construct rules for data processing using graphical designer and a set of wizards.

The following code shows how the rules that are constructed using the custom developed Fraud Rules Designer can be transformed into equivalent C# code. The C# code consists of classes with LINQ queries definition.

string sourceCode = "

using System;

using Microsoft.ComplexEventProcessing.Linq;

using Payloads;

namespace QueryLibrary

{

public class Query1

{

public object GetQuery()

{

var stream = CepStream<EventPayloadInput>.Create("SampleInput");

var evt = from e in stream

where e.callEventDuration > 60

select e;

return evt;

}

}

}

"

Then the code is compiled into the assembly and stored in the form of a .dll file.

CSharpCodeProvider compiler = new CSharpCodeProvider(

new Dictionary<string, string>() { { "CompilerVersion", "v3.5" } });

CompilerParameters parameters = new CompilerParameters();

parameters.ReferencedAssemblies.Add("Microsoft.ComplexEventProcessing.dll");

parameters.ReferencedAssemblies.Add("FMS.StreamInsight.Payloads.dll");

parameters.ReferencedAssemblies.Add("System.Core.dll");

parameters.GenerateInMemory = false;

parameters.GenerateExecutable = false;

parameters.OutputAssembly = Path.Combine(destPath, "Query1.dll");

CompilerResults compiledResult = compiler.CompileAssemblyFromSource(parameters, sourceCode);

if (compiledResult.Errors.HasErrors)

{

string errorMsg = "";

errorMsg = compiledResult.Errors.Count.ToString() + " Errors:";

for (int i = 0; i < compiledResult.Errors.Count; i++)

errorMsg = errorMsg + "\r\nLine: " +

compiledResult.Errors[i].Line.ToString() + " - " +

compiledResult.Errors[i].ErrorText;

MessageBox.Show(errorMsg, "Compiler Demo");

return;

}

The StreamInsight transformation component uses reflection API to load the compiled assembly and run LINQ queries defined in the assembly on the local in-memory CEP server.

Assembly assembly = Assembly.LoadFrom(queryAssemblies[i].FullName);

object stream = null;

// Walk through each type in the assembly looking for the class

foreach (Type type in assembly.GetTypes())

{

if (type.IsClass)

{

if (type.FullName.Contains(".Query1"))

{

// Create an instance of the object

object ClassObj = Activator.CreateInstance(type);

// Dynamically invoke the method

stream = type.InvokeMember("GetQuery",

BindingFlags.Default | BindingFlags.InvokeMethod,

null,

ClassObj,

null);

}

}

}

CepStream<EventPayloadInput> evt = CepStream<EventPayloadInput>.Create("Input1");

evt = (CepStream<EventPayloadInput>)stream;

QueryTemplate qt = app.CreateQueryTemplate("Query", "", res);

QueryBinder qb = new QueryBinder(qt);

# Discussion on the Different Design Options

Embedding StreamInsight inside SQL Server Integration Services can help you make use of event processing during data integration. Using these technologies in combination, your organization can benefit from improvements in the areas of temporal data processing, large data reduction, and continuous data processing in the ETL pipeline.

In the POC, we developed the StreamInsight SSIS transformation in order to understand the design space for using StreamInsight and Integration Services together. In addition, the POC provided an opportunity for deep performance investigations on the performance of the integration, and also identify areas in both products that need to be improved in order to ship the component:

* **Dealing with out-of-order data** – The Telco CDR data is usually out-of-order, the data often comes from multiple switches, and they usually need to be unioned together in order to process the data.
* **Efficient mechanism for passing data to/from Integration Services to StreamInsight** – Integration Services and StreamInsight have different mechanisms of working with streams of data and efficiently passing data between two components would impact how well the pipeline component performs.
* **Providing the right user interface for defining the LINQ query** – in order to have a general purpose pipeline component that enables temporal data processing, there needs to be a way to let the SSIS package designer input the temporal query in a familiar or accessible form.

## Integration Services and StreamInsight Working Together

StreamInsight is designed to be embeddable and Integration Services is designed to be extensible, so embedding StreamInsight inside Integration Services as an extended pipeline component is a natural fit. The only challenge is how to make them “talk” to each other in the most efficient way.

StreamInsight uses an input adapter to bring data in and an output adapter to send data out. The input adapter pushes the data into StreamInsight whenever it has data ready. StreamInsight usually takes these events, but it can also push back when it’s busy. The output adapter uses the pull model. StreamInsight tells the output adapter to pull the data whenever the data is ready.

Integration Services pushes data into the pipeline component in batches. When the data buffer is ready, Integration Services calls into the pipeline component’s code and ask it to process the buffer with a batch of events. The pipeline component then must process all events in the buffer before the component returns.

In order to avoid the data copying between Integration Services and StreamInsight, SQL Server coordinates the data pushing of Integration Services and StreamInsight by doing handshaking between the two components – whenever a data buffer is ready, Integration Services calls into StreamInsight component to push the data into StreamInsight. The Input adapter of StreamInsight attempts to push all data into StreamInsight. If it encounters pushback from StreamInsight, the adapter waits for StreamInsight to be ready again, and then it keeps pushing until the buffer is ready.

**Pushing the Limits - Getting Higher Performance from Embedding the StreamInsight Engine Within an Integration Services Component**

During performance tuning of both StreamInsight and Integration Services, we realized that in order for the embedded StreamInsight and Integration Services to achieve higher performance, Integration Services needs to be able to push multiple buffers to the embedded StreamInsight input adaptors in parallel. An existing limitation of the SSIS engine is that it is capable of processing the inputs in a sequential manner only. Consequently, if the StreamInsight pipeline component takes multiple input streams, the events have to be sent to StreamInsight one stream at a time.

## Processing Out-of-Order Data

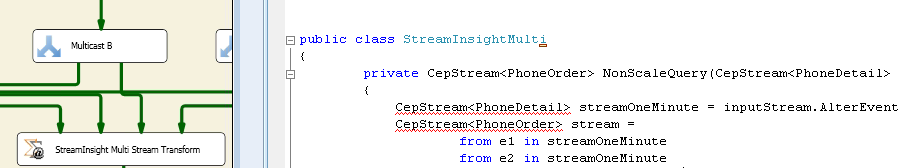
Because the incoming CDR data can arrive late, it is important to be able to handle it correctly. If the system waits too long for the late-arriving data, the output latency increases. If the system waits for a shorter time, late-arriving data has to be dropped. This impacts the accuracy of the overall results.

StreamInsight has built-in support for dealing with out-of-order data. The concept of current time increment (CTI) is critical for communicating the progress of application time. The AdvanceTimeSettings can be used during query build time to help issue CTIs for the application. The input adapter can also issue CTI appropriately to communicate with the StreamInsight engine to advance application time.

## Development Experience

In order to provide customers with a rich development experience when designing using StreamInsight as an Integration Services component, you should plan to provide a UI for designing and editing the StreamInsight LINQ query for the StreamInsight Integration Services component. This provides the ability to allow users to modify the StreamInsight query in the component.

One possible development experience is to make use of the Visual Studio environment when developing the LINQ query used by the Integration Services component. Today, Integration Services uses Visual Studio Tools for Applications (VSTA) in the Script Component and Script Task. VSTA provides Integration Services users with the ability to author code that can be used in a control flow or data flow task. Similarly, it enables users to make use of the VSTA environment for authoring the LINQ query used by the StreamInsight transform in the Integration Services development environment. The following diagram shows the steps that are performed when defining the LINQ query and ensuring that it can be used by the StreamInsight transform.



Create the StreamInsight Transform Component

Create Input Metadata

Edit the LINQ Query

Auto Compile and Link

In addition, the user can debug the LINQ query if an error occurs during execution.

Another possible development experience is to provide a graphical way of composing the LINQ query. This is similar to the Query-By-Example (QBE) concept. Because the StreamInsight transform maintains information about the number of inputs (from various upstream Integration Services components), you can provide a graphical designer to show the available input streams, and users can use the designer to describe the exact steps in an environment that is similar to Integration Services. The graphical representation can then be converted to a LINQ query that can be used during execution.

## Scale-Out and Scale-Up

In order to process events with the maximum throughput and minimum latency, you usually need to consider how to make the best use of the available resources. In order to maximize CPU usage, you often need to parallelize the work so that most or all of the CPU can be busy most of the time. Both Integration Services and StreamInsight have a data flow architecture that enables pipeline parallelism. It is very easy to use multiple cores in a computer to achieve parallel processing. In addition, StreamInsight has a built-in construct – Group&Apply operator—that you can use to allow parallel processing of events that belong to different logical groups.

# Summary

In this white paper, we presented how SQL Server Integration Services and StreamInsight can be used together in data integration scenarios. Using scenarios from the telecommunications sector, we discussed several solution architectures that use Integration Services and StreamInsight in a single data integration solution. The design options that are covered in each of these solution architectures can be implemented using Integration Services and StreamInsight capabilities that are part of SQL Server 2008 R2. As a proof of concept, we shared the key learnings from building a custom Integration Services-StreamInsight transform, which embeds StreamInsight capabilities. We evaluated the performance of the custom Integration Services-StreamInsight transform in the SQLCAT customer lab, and the data we gathered there provides a reference baseline for future work in building similar custom SSIS transforms that embed StreamInsight capabilities. From the performance study, we documented the type of hardware that is needed to support given data throughputs.