An Extensible Test Framework for the Microsoft StreamInsight Query Processor

Alex Raizman[1], Asvin Ananthanarayan[1], Anton Kirilov[1], Badrish Chandramouli[2], Mohamed Ali[1]

[1]Microsoft SQL Server, {alexr, asvina, antonk, mali }@microsoft.com

[2]Microsoft Research, {badrishc}@ microsoft.com

ABSTRACT

Microsoft StreamInsight (*StreamInsight*, for brevity) is a platform for developing and deploying streaming applications. StreamInsight adopts a deterministic stream model that leverages a temporal algebra as the underlying basis for processing long-running continuous queries. In most streaming applications, continuous query processing demands the ability to cope with high input rates that are characterized by imperfections in event delivery (i.e., incomplete or out-of-order data). StreamInsight is architected to handle imperfections in event delivery, to generate real-time low-latency output, and to provide correctness guarantees on the resultant output.

On one hand, streaming operators are similar to their well-understood relational counterparts - with a precise algebra as the basis of their behavior. On the other hand, streaming operators are unique in their non-blocking nature, which guarantees low-latency and incremental result delivery. While our deterministic temporal algebra paves the way towards easier testing of the streaming system, one unique challenge is that as the field evolves with more customers adopting streaming solutions, the semantics, behavior, and variety of operators is constantly under churn. This paper overviews the test framework for the StreamInsight query processor and highlights the challenges in verifying the functional correctness of its operators. The paper discusses the extensibility and the reusability of the proposed streaming test infrastructure, as the research and industrial communities address new and constantly evolving challenges in stream query processing.

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems – Query Processing, Relational databases.

General Terms

Algorithms, Design, Verification.

Keywords

Data Streaming, SQL Server, StreamInsight, Testing, Verification.

DBTest'10, June 7, 2010, Indianapolis, Indiana, USA.

1. INTRODUCTION

Recent advances in sensor network technologies, GPS devices, RFIDs, and wireless communications have resulted in wide-spread real-time stream data acquisition. Example data streaming applications include network monitoring, web-click analytics, telecommunications data management, intrusion detections, manufacturing, geosensing, traffic management, and online stock trading. While the amount of streamed data acquired from sensors has increased substantially, the inability to process, mine, and analyze this data in a timely manner prevented researchers from making full use of the incoming stream data. Consequently, an ongoing effort in both research and industry has been established to develop data stream systems that are capable of processing hundreds of thousands of events per second.

Microsoft StreamInsight [4] (*StreamInsight*, for brevity) is a platform for developing and deploying streaming applications that run continuous queries over high-rate streaming events. StreamInsight is an event stream processing system featured by its declarative query language and its multiple consistency levels of stream processing. StreamInsight adopts a temporal stream model, where a data stream is modeled as a time-varying relation. This paper introduces the ongoing effort at Microsoft SQL Server to build a test framework to verify the functional correctness of StreamInsight. The paper presents the challenges in testing data stream systems, discusses the design principles originated by the nature of streaming applications, and delves into the architecture of the proposed test infrastructure.

1.1 Challenges

There are three main aspects in which data stream systems differ in terms of surface area from their traditional relational (DBMS) counterparts. We next summarize these differences and explain how each surface area difference poses a challenge against the data stream test framework.

First, a data stream system is characterized by its expected and frequent imperfections in the delivery of stream events. Due to the nature of sensing devices, stream events are continuously transmitted over the network to the data stream system. Taking network delays and the unreliable nature of the transmission channels into consideration, some stream events are expected to arrive late, to be out of order, or to be duplicated on their way to the system. Hence, the test space needs to be augmented by an additional dimension for the event arrival pattern.

Second, data stream systems generate output in real time. An incoming event gets processed immediately by the query pipeline its effect gets reflected in the output immediately. As a result, streaming queries may produce output that may require *compensation* (deletion or modification) in the future, e.g., due to

out-of-order events in the stream, errors that are subsequently corrected at the source, or upstream operators that modify their output. We refer to the generation of output that may require subsequent correction as *speculation*. As a consequence, the test framework needs mechanisms to verify the functional correctness of operators in the presence of speculative input and output.

Third, the data streaming domain is a new area compared to traditional databases that have matured over decades. While some efforts have been made recently [8, 9], there are as yet no standards or semantics that have gained consensus across the community. As a result, the streaming test infrastructure has to be extensible and well-prepared for the new challenges in this area. As a concrete example, test scenarios need to be declaratively described by their intent, with functional descriptions of scenarios and dependencies on query languages kept to a minimum.

1.2 Design Principles

The proposed test framework has been architected with several design principles in mind. These design principles are crucial for traditional database testing. However, these principles are stretched along different dimensions as the focus is shifted from traditional database systems to data stream systems. We summarize the design principles as follows:

- **Declarative scenario description and language independence.** Given the lack of standardization in the relatively new streaming domain, test scenarios are expressed in a declarative language with the possibility of mapping the declared intent across multiple languages.
- **Composability.** Streaming operators are characterized by the stream-in/stream-out feature, i.e., the output of an operator is a stream that is of similar nature to the input stream. Hence, single-operator tests are composed to form composed test scenarios. More interestingly, composed test scenarios are grouped iteratively to form complex testing scenarios.
- **Leveraging existing test infrastructures.** Data stream systems share the relational basis with traditional database systems. A test framework for data stream systems is expected to reuse the enormous effort that has been conducted in database testing and augment it with the proper components for stream-oriented workloads.
- **Extensibility and reusability.** This principle addresses how the test infrastructure scales as new challenges are being addressed by the research community and as more customers adopt the data streaming technologies.

The remainder of this paper is organized as follows. Section 2 introduces some basic concepts in the StreamInsight query processor. Section 3 presents the overall architecture of the test framework. Declarative intent-based generation of test scenarios is described in Section 4 while the stream event generator is described in Section 5. Result verification is described in Section 6. Section 7 discusses the extensibility and reusability of the test framework. Section 8 concludes the paper.

2. Background

A streaming system [4, 6, 7, 10] allows applications to execute long-running *continuous queries* (*CQs*) that monitor and process data streams. While the core concepts are generalizable to any streaming system, this paper focuses on *Microsoft StreamInsight*, which is based on the CEDR [2, 3] research project.

2.1 Logical and Physical Streams and Events

A physical stream is a sequence $\{e_1, e_2, \ldots, \}$ of *events*. An event e_i = <p, c> is a notification from the outside world that contains: (1) a payload $p = \langle p_1, \ldots, p_k \rangle$, and (2) a control parameter *c* that provides event metadata. While the exact set of control parameters associated with events varies across systems [2, 11, 12], two common notions are: (1) an event generation time, and (2) a duration, which indicates the period of time over which an event can influence output. We capture these by defining *c* = <LE, RE>, where the interval [LE, RE) specifies the period (or lifetime) over which the event contributes to output. The *left endpoint* (LE) of this interval, also called *start time*, is the application time of event generation, also called the event timestamp. Assuming the event lasts for *x* time units, the *right endpoint* of an event, also called *end time*, is simply RE = LE + *x*.

Compensations StreamInsight allows users to issue compensations (or corrections) for earlier reported events, by the notion of *retractions* [2, 13, 14], which indicates a modification of the lifetime of an earlier event. This is supported by an optional third control parameter RE_{new}, that indicates the new right endpoint of the corresponding event. Event deletion (called a full retraction) is expressed by setting RE_{new}=LE (i.e., zero lifetime).

Canonical History Table (CHT) This is the logical representation of a stream. Each entry in a CHT consists of a lifetime (LE and RE) and the payload. All times are application times, as opposed to system times. Thus, StreamInsight models a data stream as a time-varying relation, motivated by early work on temporal databases by Snodgrass et al. [17]. Table 1 shows an example CHT. This CHT can be derived from the actual physical events (either new inserts or retractions) with control parameter *c* = <LE, RE, RE_{new}>. For example, Table 2 shows one possible physical stream with an associated logical CHT shown in Table 1. Note that a retraction event includes the new right endpoint of the modified event. The CHT (Table 1) is derived by matching each retraction in the physical stream (Table 2) with its corresponding insertion (i.e., matching by event ID) and adjusting the RE point of the event accordingly.

Table 1 – Example of a CHT

ID	LE	RE	Payload
E0	1	5	P1
E1	4	9	P2

Table 2 – Example of a physical stream

ID	Type	LE	RE	RE_{new}	Payload
E0	Insertion	1	∞	-	P1
E0	Retraction	1	∞	10	P1
E0	Retraction	1	10	5	P1
E1	Insertion	4	9	-	P2

2.2 Event Classes

Users can use lifetimes to model different application scenarios. For instantaneous events with no lifetime, RE is set to LE+*h* where *h* is a chronon, the smallest possible time-unit. We refer to such events as *point events*. On the other hand, there may be

events that model an underlying continuous signal being sampled at intervals. In this case, each event samples a particular value, and has a lifetime until the beginning of the next event sample. We refer to such events as *edge events*. The most general form of events have arbitrary endpoints depending on when the modeled event came into and went out of existence – these events are referred to as *interval events*.

2.3 Detecting Progress of Time

We need a way to ensure that an event is not arbitrarily out-of-order, which is realized using time-based punctuations [2, 15, 16]. A time-based punctuation is a special event that is used to indicate time progress. These punctuations are called *Current Time Increments* (or *CTIs*) in StreamInsight. A CTI is associated with a timestamp *t* and indicates that there will be no future event in the stream that modifies any part of the time axis that is earlier than *t*. Note that we could still see retractions for events with LE less than t, as long as both RE and RE_{new} are greater than or equal to *t*.

2.4 Streaming Queries and Operators

A streaming *continuous query* (*CQ*) consists of a tree of operators, each of which performs some transformation on its input streams and produces an output stream. Queries are expressed in a high-level language such as StreamSQL or, in case of StreamInsight, using LINQ. LINQ queries are converted into an equivalent XML representation of the stream query plan, and users can also directly submit queries in XML format. StreamInsight operators are well-behaved and have clear semantics in terms of their effect on the CHT. This makes the underlying temporal operator algebra deterministic, even when data arrives out-of-order.

Data enters the streaming system via *input adapters*, which convert external sources into events that can be processed by the streaming system. Output events exit the system via *output adapters*. There are two main classes of operators: *span-based* and *window-based*.

Span-based operators A span-based operator accepts events from an input, performs some computation for each event, and produces output for that event with the same or possibly altered output event lifetime. Examples of single-input span-based operators include *filter* (to select events that match a specified condition) and *project* (to select certain input columns or expressions over columns from the input stream). A two-input span-based operator is *temporal join*, which correlates events across the two streams. The lifetime of the output event is equivalent to the entire "span" of the input event's lifetime for single-input operators, or the intersection of contributing event lifetimes in case of multi-input operators.

Window-based operators *Aggregation operators* such as Count, Top-K, Sum, etc. work by reporting a result (or set of results) for every unique *window*. The result is computed using all events that belong to that window. StreamInsight supports several types of windows: *snapshot* (equivalent to sliding), *hopping*, *tumbling*, and *count-based* windows.

Other Operators One stream can be output to multiple operators using an operator called *multicast*, while multiple streams are merged using a *union* operator. StreamInsight allows per-group computation using an operation called *Group&Apply*, where the same subplan (called the *apply branch)* to be applied in parallel for every *group* (defined by a grouping key) in a stream. The results of all the groups are merged (using the union operator) as the final operator output. In addition, StreamInsight supports *user-defined operators* that users can use to express custom computations (span-based and window-based) on streaming data.

3. Architecture of the Test Framework

Figure 1 depicts the architecture of the proposed test framework. Test scenarios are represented by a declarative testing approach that describes the scenario intent for both queries and stream event data. As discussed in [1], declarative testing approaches focus on "*what*" to accomplish rather than on the imperative details of "*how*" to manipulate the state of the system. Testers describe their queries and the nature of the generated input data through an *intent tree*. An example intent tree would read "Execute an equi-join query over out-of-order stream events". (Formal examples are presented in Section 4.)

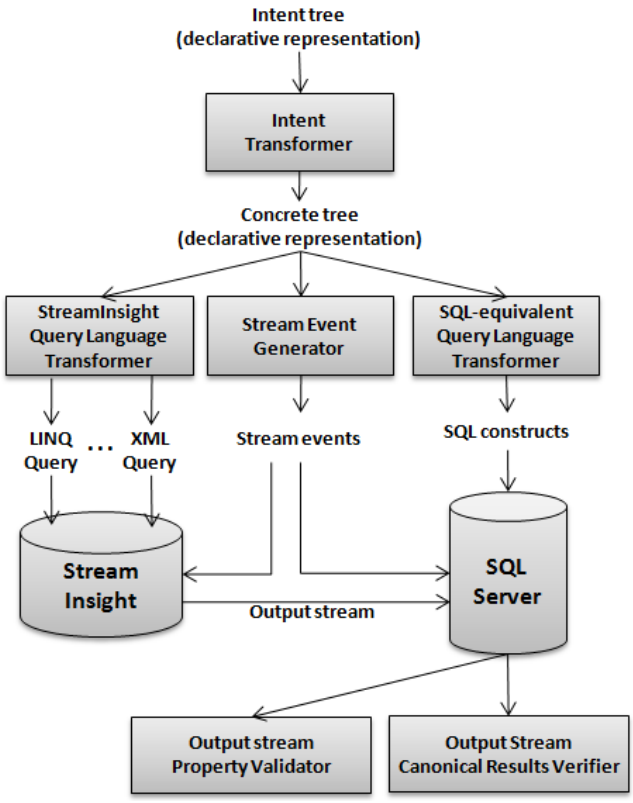

Figure 1: System architecture.

The intent tree goes through an *Intent Transformer* to fill in the omitted elements that are required to execute the test. The outcome of the *Intent Transformer* is called a *concrete tree*. Concrete trees are still expressed in declarative form but, unlike intent tree, has no omitted elements left. An example concrete tree would read "Execute an equi-join query **between $Stream_1$ and $Stream_2$** using an equality join expression **$Stream_1.Field_a$ == $Stream_2.Field_b$** over **highly** out-of-order stream events". Note that the concrete tree has hints that control certain aspects or properties of the event data generator. These properties are used by the *Stream Event Generator* to generate stream events. The *Stream Event Generator* is described in section 5.

The concrete tree then is translated into one of the supported StreamInsight interface languages. Note that StreamInsight is

designed to support multiple language interfaces. Figure 1 shows XML and LINQ as two example languages that are supported by the StreamInsight interface. The resultant query (either XML or LINQ) is shipped and executed within the StreamInsight engine.

Meanwhile, the stream event generator output is transformed into a time varying relation and is inserted into relational tables in an SQL Server database. Also, the concrete tree is transformed to an SQL-equivalent query using techniques described in recent work [18], and is shipped to SQL Server to be executed over the time-varying relation. The output stream from StreamInsight is finally shipped to SQL Server for verification purposes. The *Canonical Results Verifier* checks that the StreamInsight output is equivalent to the SQL Server output from a canonical perspective, and the *Property Validator* performs additional checks. Result verification is covered in the technical report [18], while its application in the test framework is briefly summarized in Section 6.

4. Intent-based Testing

In this section, we present an example test scenario that is expressed in a declarative format and how it gets transformed from an intent tree to a concrete tree. To construct the intent tree, we write a test scenario using a C# code snippet. An equijoin scenario is expressed as follows:

```
var q = left.Join(right).On(
            leftExpression.Equals(
                rightExpression));
```

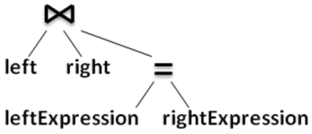

Figure 2: An example intent tree.

The test framework transforms the test scenario (expressed in C#) into an intent tree (see **Figure 2**). An intent tree represents a wide test space constrained by the intent definition. Concrete trees are specific test instances or points that fall in the test space that is defined by the intent tree. Figure 3 depicts two examples of possible concrete trees for the intent tree depicted in **Figure 2**. These concrete trees are obtained by filling in the omitted elements in the intent tree. For example, the "left", "right", "leftExpression", and "rightExpression" in **Figure 2** are examples of such omitted elements. These elements are replaced by the actual input streams and expressions to yield a concrete tree. Numerous concrete trees are constructed from the same intent tree by intelligently sampling the test space constrained by the intent tree. The test scenario now consists of all the concrete trees.

The intent tree also describes the expected nature of the input stream events declaratively, e.g., out-of-orderness, event durations, event overlap, etc. The *Intent Transformer* formalizes the declarative description of the intent input data into concrete parameters, decorates the concrete trees with these parameters and passes these values to the *Stream Event Generator*.

5. Stream Event Generation

Traditional data generators for testing relational database systems focus primarily on data distributions to generate payload values. As described in Section 2, data generators for data stream systems need to be augmented (1) by temporal attributes that represent the event's endpoints (i.e., LE and RE) and (2) by an event type (i.e., insertions and retractions) that is crucial to represent the imperfections in event delivery. In the remainder of this section, we describe the generation of the event's temporal attributes and the generation of the event type. For payload generation, we simply leverage the available payload generators in the current SQL Server test infrastructure.

5.1 Temporal Aspect Generation

Event generation in the proposed test framework follows the declarative concepts outlined in the Section 3. Testers specify their intent to the generator by declaratively expressing various properties for the temporal aspects. The event generator examines these properties and transforms them into a sequence of timestamp pairs. Each timestamp pair represents the endpoints (LE and RE) of the event. We support the following aspects in our generator.

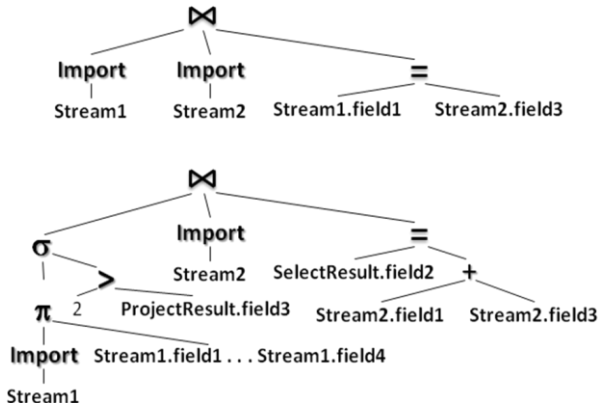

Figure 3: Concrete tree examples.

Event duration – StreamInsight events are point, interval or edge events. We need to test the entire space that ranges from very short lived events (e.g., point events) to events that span the entire temporal space (LE = -∞ and RE = ∞). Testers express their intent by specifying the maximum life time of an event which is described in terms of Point, Short, Long, Infinite. Event generator then translates this property into a timestamp that is used to generate the end time of the event.

Event overlap – Overlap between events is another important test dimension. This controls behavior of operators such as Join as well as window based operators. Testers describe their intent by specifying a subset of overlap space. The overlap space currently include: High, Medium and Low.

Controlling liveliness – Liveliness of a query in StreamInsight is governed by CTI events. As described in Section 2, a CTI event guarantees that no events will arrive earlier than the CTI's timestamp. Hence, CTIs advance the time forward and places stability guarantees on the input streams. Testers describe their intent by providing a property that specifies how eager the event generator should be in generating CTIs.

User defined temporal patterns – For the sake of extensibility, the event generator exposes an interface that enables testers to explicitly control the shapes and patterns of the interval events that are generated. The details are omitted due to lack of space.

Simulation of typical and boundary cases - The event generator produces events to test a wide range of temporal cases that include:

- *Non overlapping events*, where none of the events overlap with each other. This is a typical case for low-rate sensors that intermittently produce events.
- *Highly-overlapping events*, where the event endpoints can be as close to each other as one tick, e.g. $e_i.LE = e_j.RE \pm 1$. This is a typical case for high-rate sensors or large sensor networks that generate a dense sample set of the surrounding environment.
- *Back-to-back events*, where the event endpoints (LE/RE) are aligned to the same point in time, e.g., $e_i.LE = e_j.RE$. This is a typical case for fixed-rate sensors that reads a sample every *t* time unites (e.g., one sample per minute).
- *Duplicate events*, where multiple sensors or RFID readers acquire readings for the same event or RFID tag. Duplicate events are also common in unreliable networks due to repeated transmission in response to lost acknowledgements.

5.2 Imperfections in Event Delivery

The tester's intent that describes imperfections in event delivery is represented by two facets:

Out-of-orderness – The proposed test framework supports the following values for the out-of-orderness facet: In-order, Slightly-out-of-order, Out-of-order and Higly-out-of-order. The stream event generator uses specified property value to control the degree of out of orderness. Although a finer and a more granular specification of the property value is possible, the proposed four values seems good enough in practice to cover the test space.

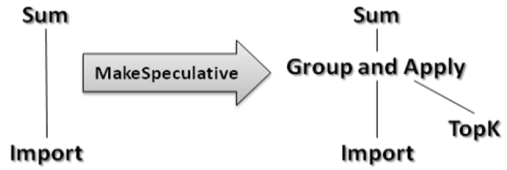

Figure 4. MakeSpeculative query transformation.

Speculation - As described in section 2, speculation causes a previous event to be retracted. In order to test that individual operators behave correctly with retraction events, we use an innovative approach to generate retractions. We create a query fragment, called MakeSpeculative, which uses existing streaming operators. MakeSpeculative uses TopK and Group and Apply operators (refer to Section 2 for background on Group and Apply) in a combination with out of order events to generate speculative output. The group and apply operator gives us events that overlap with each other. The Top K operator generates "speculated output" based on the set of events it received and, then, retracts the speculated output and generates compensation output as late and out-of-order events are received. The result of this fragment is a rich event stream that contains insertion and retraction events with high degree of overlap. This approach has the advantage that the test framework makes use of the stream-in stream-out feature of streaming operator and avoids the need of writing specialized modules. Testers leverage this construct in their tests by specifying their intent as "testing with speculative input", which is data property to begin with. The event generator applies a transform on the concrete tree and introduces the MakeSpeculative query construct above the import operator. Figure 4 is an example query where the tester's intent is to validate the SUM aggregate. The test framework transforms the concrete tree as shown.

This demonstrates an interesting paradigm of "influencing data by manipulating the query shape" using a declarative approach. We also use this paradigm to generate duplicate events. By applying a multicast-union query fragment above the MakeSpeculative fragment we can generate duplicate retraction events.

6. Canonical Result Verification

6.1 Query Translation

We test the functional correctness of the StreamInsight engine by first executing the generated streaming query over the generated event stream. In order to verify the result of the streaming query, technical report [18] leverages the CEDR temporal algebra and operator semantics that StreamInsight is based on. Briefly, the well-defined operator semantics and our use of application time instead of system time enable us to translate any StreamInsight query into a sequence of SQL queries with the following property:

Given a streaming query operating on a physical stream S1 and producing physical stream S2 as result, the corresponding sequence of SQL queries applied to CHT(S1) produces table CHT(S2).

Here, *CHT(S)* refers to the canonical history table (see Section 2) corresponding to the event stream *S*. Note that any physical stream can be converted into its canonical form by removing all CTIs and applying retractions to the corresponding insert events. It has been shown [18] that any streaming query can be converted into an equivalent sequence of SQL queries.

6.2 Expression Translation

This step translates all *expressions* (if any) that are part of the streaming query. For example, in case of the join operator, the join predicate expression and the project expressions are translated. The test framework could have attempted to translate every streaming expression into a corresponding SQL expression using the SQL expression services and type system. However, this would create a disparity because StreamInsight is based on the *CLR* (Common Language Runtime) type system whereas SQL defines its own type system. Additionally, certain types like System.TimeSpan are not available in SQL Server.

Instead, we leverage SQL CLR for this purpose. Every primitive StreamInsight type is wrapped into a SQL CLR User Defined Type (UDT) and provisioned in SQL Server. A Data property on the UDT returns the actual C# primitive type. Any expression in StreamInsight can be represented as a C# expression because it uses CLR as the basis of its expression services. The abstract expression tree from the declarative component is transformed into a C# expression. This C# expression is wrapped into a SQL CLR User Defined function and provisioned in SQL Server. The payload fields in the expression become input parameters to the SQL CLR user defined function and the return type of the UDF is the C# return type of the expression. The SQL CLR UDF is then used in its parent operator in the appropriate context. For example, if it is a join operator, the UDF is the join predicate and if it is a select operator, the UDF is the filter predicate.

This technique of using SQL CLR UDFs allows the test framework to avoid issues involved with translating arbitrary C# expressions into SQL expressions using the SQL type system.

Although SQL CLR UDFs are used for payload fields, the *datetime2* SQL type is used for representing the application time of StreamInsight events. This is because *datetime2* is equivalent to the CLR System.DateTime type.

6.3 Output Stream Verification

The event stream generated by StreamInsight is persisted in the SQL Server table by the output adapter. The output stream verifier is a collection of SQL stored procedures which are executed over the persisted table. The following properties are verified:

- Absence of any *CTI violations* (i.e., an event that modifies a time t that is earlier than the most recent CTI).
- *Stream shape*, i.e., whether the shape (point, edge, interval) is identical the one specified by the tester
- *Stream ordering*, i.e., whether the events are fully ordered (by LE) or ordering is assured only across an insert and its subsequent lifetime modifications.

We verify the correctness of the output stream using the following steps:

1. Transform the input stream into a CHT C1
2. Translate the streaming query and the expressions into a sequence of SQL queries D1
3. Run the streaming query S1 on StreamInsight and generate the corresponding output stream CHT C2
4. Execute query sequence D1 (on SQL Server) over a database table T1 corresponding to C1, to generate an output table T2
5. Finally, test the table T2 for equivalence with CHT C2. A mismatch in the results indicates a bug in StreamInsight, assuming no errors in either the SQL Server query processor or our SQL query generator.

7. Extensibility and Reusability

In this section, we describe three major themes in the proposed test framework that promote extensibility and reusability of the framework for future scenarios. First, the adoption of the declarative testing approach was one of the main aspects which enabled us to design robust workflow that is able to keep up with growing demands and complexity of the StreamInsight engine during its development. As the streaming language constructs mature over years and as the operator semantics are agreed upon, the declarative approach secures the test scenarios from the need to be edited or changed. Moreover, additional languages (other than XML and LINQ) can be supported by adding additional language transformers to the test architecture.

Second, thanks to the canonical result verifier, more operators can be added to the StreamInsight engine and integrated in the test framework without the need to worry about the verification of the operator's functional correctness. As long as the new operator is expressed in an equivalent SQL sequence of statements, the *SQL Language Transformer* is augmented to include these statements and other components in the test framework remain intact.

Third, the proposed framework is capable of composing declarative test scenarios to formulate complex test scenarios. Thanks to the stream-in/stream-out feature of streaming operators. The output of one test scenario is fed as input to another test scenario. Alternatively, one test scenario can be embedded as a node in another test scenario. This approach tests the cross interaction among streaming operators as more operators are added to the system. It also facilitates implementation of rich test models which can be used both to create intent trees and enrich *Intent Transformer* to generate multiple concrete trees within constraints of initial intent tree.

8. Conclusions

In this paper we presented a declarative testing approach to transform the tester's intent into a streaming query. We elaborated the challenges that face the testing of a data streaming system that include the real time nature of stream events with a likelihood of late and out-of-order events. The proposed test framework has been designed to cope with differences between traditional database systems and data stream systems. The notions of extensibility and reusability have been given a priority to foresee the expected growth in the requirements and semantics of continuous query processing as it gets more popular amongst database vendors.

9. REFERENCES

[1] Ed Triou, Zafar Abbas, Sravani Kothapalle, "Declarative Testing: A Paradigm for Testing Software Applications," itng, pp.769-773, 2009 Sixth International Conference on Information Technology: New Generations, 2009.

[2] Roger S. Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. Consistent Streaming Through Time: A Vision for Event Stream Processing. In CIDR, 2007.

[3] Jonathan Goldstein, Mingsheng Hong, Mohamed Ali, and Roger Barga. Consistency Sensitive Streaming Operators in CEDR. Technical Report, MSR-TR-2007-158, Microsoft Research, Dec 2007.

[4] Mohamed Ali et al.: Microsoft CEP Server and Online Behavioral Targeting. In VLDB 2009.

[5] B. Chandramouli, J. Goldstein, and D. Maier. On-the-fly Progress Detection in Iterative Stream Queries. In VLDB, 2009.

[6] Arvind Arasu, Shivnath Babu, Jennifer Widom: CQL: A Language for Continuous Queries over Streams and Relations. DBPL 2003: 1-19.

[7] Theodore Johnson, S. Muthukrishnan, Vladislav Shkapenyuk, Oliver Spatscheck: A Heartbeat Mechanism and Its Application in Gigascope. In VLDB, 2005.

[8] Irina Botan et al, "A Demonstration of the MaxStream Federated Stream Processing System", Demonstration, In ICDE, 2010.

[9] N. Jain et al. Towards a Streaming SQL Standard. In VLDB, 2008.

[10] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo and C. Venkatramani. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In SIGMOD 2006.

[11] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, David Maier: *Out-of-order Processing: A New Architecture for High-Performance Stream Systems*. PVLDB 1(1):274-288 (2008).

[12] Moustafa A. Hammad et al.: Nile: A Query Processing Engine for Data Streams. In ICDE, 2004.

[13] E. Ryvkina et al. Revision processing in a stream processing engine: a high-level design. In ICDE, 2006.

[14] R. Motwani et al. Query processing, approximation, and resource management in a DSMS. In CIDR, 2003.

[15] Peter Tucker, David Maier, Tim Sheard, Leonidas Fegaras: *Exploiting Punctuation Semantics in Continuous Data Streams*. IEEE TKDE 15(3): 555-568 (2003).

[16] Utkarsh Srivastava, Jennifer Widom. *Flexible Time Management in Data Stream Systems.* In PODS, 263-274, 2004

[17] C. Jensen and R. Snodgrass. *Temporal Specialization*. In ICDE, 1992.

[18] J. Goldstein et al. Historic Stream Query Processing in SQL. Technical Report, Microsoft Research.

[19] Rathakrishnan, B. et al. Using CLR Integration in SQL Server 2005. SQL Server Books Online. http://msdn2.microsoft.com/en-us/library/ms345136.aspx.