# Data Stream Management Systems for Computational Finance

Badrish Chandramouli, Mohamed Ali, Jonathan Goldstein, Beysim Sezgin, Balan S. Raman
Microsoft Corporation
One Microsoft Way, Redmond, WA 98052
e-mail: {badrishc, mali, jongold, beysims, sethur}@microsoft.com

*Abstract*—**Computational finance leverages computer technologies to build models from large amounts of data to extract insight. In today's networked world, the amount of data available to build and refine models has been increasing exponentially. Often times the difference between seizing an opportunity and missing one is the latency involved in processing this data. The need to do complex processing with minimal latency over large volumes of data has led to the evolution of different data processing paradigms. Increasingly there is a need to develop an event-oriented application development paradigm to decrease the latency in processing large volumes of data. Such an event-oriented system incorporates the lessons that we have learnt from earlier data processing platforms, i.e., declarative programming, etc. and adapts them for incremental processing. Some of the systems have adopted techniques from artificial intelligence while others have adopted techniques from database management systems.**

**This technical report motivates the need for using a specialized platform for such applications, and delves into a deeper discussion about systems that are architected to operate over temporal data (streams of events), called *Data Stream Management Systems* (*DSMSs*). With the aid of a running example from the financial domain, it describes the salient characteristics of these platforms and the advantages they offer.**

## I. Introduction

### A. Need for a New Data Processing Paradigm

The computational finance application domain leverages the ability to process and analyze large amounts of data and extract useful insight. With the recent proliferation of computer networks along with the computerization of almost every business process, increasing quantities of data are being produced by machines, sensors, and applications, and are becoming available as input for analysis and mining by financial applications. There are several interesting aspects of this data:

- The data is available *instantaneously* and *continuously*, being delivered to applications as an unending "stream" of data.
- The data is inherently time-oriented (*temporal*) in nature. Thus, to derive maximum value out of the data, there is a need to process queries over this data incrementally and on-the-fly, and incorporate the results into ongoing business processes such as algorithmic stock trading.
- The temporal data is also stored offline for subsequent analysis, mining, and back-testing, often using the same queries that are used in the real-time scenario.

In this fast-paced environment, the difference between a leveraged opportunity and a missed one is often the latency involved in processing the stream of temporal data. Given the importance of processing temporal data, financial applications may choose to build customized vertical solutions to solve specific problems. While such solutions could be efficient for a particular application, it is hard to generalize the effort across diverse applications, and further, it can be expensive to build, maintain, manage, and re-use such customized systems.

*Software platforms* solve the re-usability problem by providing a generic substrate upon which many user applications can be easily built. One such platform that has been extensively used in many application domains in the past is *databases*, which exposes a simple yet powerful data model called *relational algebra*, to enable applications to issue and execute complex declarative *queries* over set-oriented data. While databases are well-suited for set-oriented offline analysis of infrequently changing datasets, the new environment of long and possibly unending sequential data streams (in real time as well as over historical temporal data) poses new challenges beyond the scope of database systems. Given the high data arrival rates, we are no longer in a situation where we can afford to collect all the data, load it into a database system, build database indexes over the data for query efficiency, and finally allow users to execute one-time queries over the data. Rather, we need to allow applications to issue queries a priori; in this case, the results of the query need to be continuously and incrementally computed and updated as new relevant data arrives from the data sources. Here, incremental computation denotes the capability to update the result of a query when new data arrives, without having to re-process earlier data. Incremental computation is necessary for a low-latency response to incoming data, but databases, due to the one-time nature of database queries, do not directly support quick incremental result computation as new data arrives as the database.

### B. An Event-Based Temporal Data Processing Paradigm

Figure 1 shows a spectrum of data processing applications and solutions in terms of the data rates and latencies that they target. Given the unsuitability of custom solutions and database systems for managing high-volume temporal data at low latency, there have been many recent efforts to build *Data Stream Management Systems* (*DSMSs*).

A DSMS enables applications to issue *long-running queries* over streaming data sources. For example, we may have a stream (or data feed) of stock quotes, where each event (or notification) in the stream contains a stock symbol and price. An application may wish to continuously monitor this stream and produce an alert when the price crosses some pre-defined threshold. Such a requirement would be expressed as a long-running query, called a *continuous query* (*CQ*).

CQs are registered with the DSMS by applications. The DSMS monitors and efficiently processes streams of data in real time. CQs are processed by the DSMS in an event-driven manner – for every new event, the DSMS efficiently and incrementally checks if the query result needs to be updated due to the event – if yes, an output event is produced. In our example, whenever we receive a stock quote, the DSMS checks to see if the price exceeds the threshold, and if yes it would produce the corresponding quote as output of the CQ. A CQ receives as well as produces event streams, and thus CQs can be *composed* to achieve more complex application requirements.

DSMSs are used for real-time in-memory data processing in a broad range of applications including fraud detection, monitoring RFID readings from sensors (e.g., for manufacturing and inventory control), and algorithmic trading of stocks. In a typical application for computational finance, there are several points during the workflow where there could be friction between the application and its data processing needs: getting data into the system, decoupling application logic from the actual management of data, and the ability to plug-in the results into business logic or workflow. These points of frictions can be alleviated by using a DSMS, and at the same time it can provide low-latency data processing which is crucial for many applications in this domain.

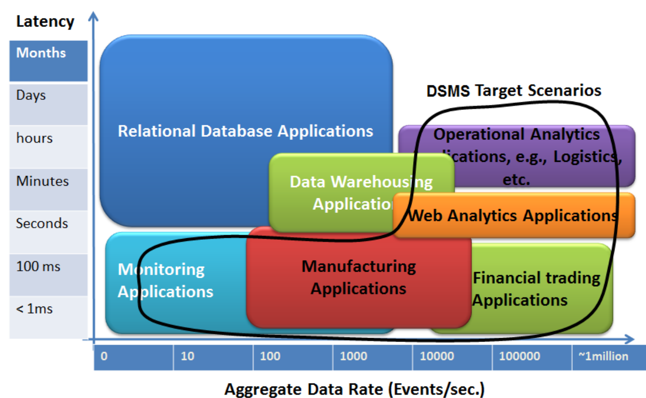



Figure 1. Scenarios for Event-Driven Applications.

## C. Lifecycle for Business Applications

The value proposition of a DSMS to a business is captured by the "monitor-manage-mine ($M^3$) cycle" shown in Figure 2. We monitor data in real-time to answer continuous queries and detect interesting patterns. The results of these operations are used to manage and perform daily business actions such as algorithmic stock trading, fraud detection, etc. Finally, the raw temporal data is stored offline, so that the business can mine the historical data to determine new CQ requirements or patterns that are fed back to the monitor phase.

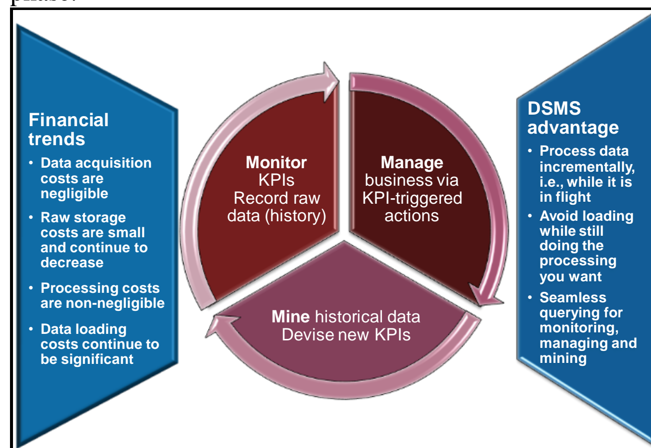



Figure 2. The monitor-manage-mine ($M^3$) cycle.

In deploying the $M^3$ cycle, applications may often need to execute the same queries over offline temporal data, for example, to perform back-testing over historical data or mining to discover interesting trends and patterns. We will discuss this aspect further in the context of the $M^3$ cycle in Section II, where we will also cover the other concepts depicted in Figure 2 in greater detail.

## D. Running Example

We next formalize a concrete end-to-end stock trading application, to serve as a running example for the rest of our discussion.

**Example 1 (Stocks Trading Application)** Assume that we have stock trade messages reaching the application as events, from various stock exchanges. Each event consist of the timestamp of the trade and a stock symbol, along with the last traded price and volume. For each stock that the trader is interested in tracking, the trader may wish to first "smooth out" the signal with a moving average, and then perform some customized computation such as detecting a *chart pattern*. The set of stocks that the trader is interested in tracking may itself change over time based on the current stock portfolio, or there may instead be a blacklist of stocks such that they wish to track all stocks other than those in the blacklist. Note that in this example, one can model both the stock trades and the updates to the whitelist/blacklist as event streams, since both activities consist of notifications that report either a stock trade or a change to the whitelist/blacklist of stock symbols.

## E. Challenges for DSMSs

There are several classes of challenges unique to this new environment for processing queries over large-scale unending stream data, which a DSMS needs to effectively address; these challenges are briefly summarized below.

#### 1) ***Ease and completeness of specification***

*a) Ability to express fundamental query logic:*

It is important for a DSMS user to easily specify their query logic declaratively as a composable sequence of queries. Declarative programming allows the query writer to focus on query logic expression without having to worry about how it would be implemented or executed. Further, since a DSMS is intimately related to temporal (time-oriented) data, it is necessary to incorporate concepts such as *windowed processing* and *time progress* as a core component of the query logic.

*b) Ability to handle late-arriving data or corrections*

Given that a DSMS needs to operate over temporal data, issues such as late arrival of events need to have a deterministic resolution. Another issue is the case where a source generates an event, but later detects an error and either changes or completely removes the earlier event. One option is to delay processing until we are guaranteed to see no late-arriving events, but this could increase the latency in producing results. Another alternative is to aggressively process events and produce tuples, issuing compensations when we receive disordered events.

*c) Extensibility*

While the toolbox of operations provided natively by a DSMS may cover many scenarios, given the complexity and diversity of applications, there is a need for a general extensibility framework using which applications can deploy custom streaming logic as part of the DSMS. Challenges here include the ability to support legacy libraries, provide the ability to easily write streaming building blocks that leverage windowing and incremental computations, and yet offer flexibility to power-users. Extensibility is important in the computational finance domain, particularly given the fact that custom libraries and technology is where financial institutions tend to mainly develop and maintain competitive advantage.

*d) Genericity of specification*

We would like the specification and semantics of query logic to be independent of when and how we physically read the events. For example, the output should be the same for a set of events and a specified query, regardless of whether we read it from a file on from a sensor in real time. Recall the $M^3$ cycle from Section 1 – it is clear that genericity of specification allows us to use a DSMS to complete the loop without duplicating effort, by preventing an impedance mismatch between the mental model (and queries) over historical data for back-testing, and real time data for a live deployment.

Genericity of specification is achieved by employing application time instead of system time, to define the semantics of stream operations. The basic idea of application time is that time forms a part of the actual event itself (in the form of a control parameter), and output of the DSMS is independent of when tuples physically get processed – depending only on the timestamps present in individual events. The semantics of stream operators are usually specified as a temporal algebra over event payloads and application timestamps – an extension of the relational algebra with time-oriented semantics.

*e) Ease of debugging*

An important challenge that DSMSs need to resolve is the capability to perform debugging of stream queries. This is particularly useful in a streaming system since data is constantly being processed in an online fashion, which makes correctness verification an apparently infeasible task. One important concept that helps debugging is the use of application time instead of system time. The use of application time in association with a temporal algebra and semantics underlying the query specification, makes query results deterministic regardless of when or how individual events in a stream are processed. This form of repeatability is useful to trace the root cause of any anomalous results observed in an output stream.

*f) Deployment obliviousness*

We would like the user of the streaming system to be isolated from system aspects such as the number of machines that the query is being executed upon, and the fact that redundancy may be employed for high-availability. Ideally, one would like the user to submit their queries and operational requirements in terms of *quality of service* (*QoS*), to a single DSMS endpoint, and the system to transparently scales out or run in highly available mode based such on user specifications.

#### 2) ***Performance***

Performance is of primary concern to applications of a DSMS because of its common use in real-time applications. Unlike databases, DSMSs usually do not have the flexibility of writing data to disk and processing it later. Thus, in order to achieve low latency, DSMSs usually maintain internal state in main memory and discard events/state when they are no longer needed for processing CQs in the future. When the event arrival rate becomes higher than processing capability of the system, events are buffered in in-memory *event queues* that exist between operators. Techniques such as flow control and load shedding are sometimes necessary if the DSMS is unable to keep up with incoming data rates and event queues get filled up as a consequence.

There are various metrics that the DSMS needs to optimize performance for, such as (1) *latency*, the time from when an event enters the DSMS to when its effect is observed in the output, i.e., the time spent by events waiting in queues and getting processed by operators; (2) *throughput*, the number of result events that the DSMS produces per second of runtime; (3) *correctness*, which reflects how much the query result deviates from the true value due to actions such as dropping of events due to load shedding, delayed arrival of

events, etc.; (4) *memory usage*, the amount of in-memory state accumulated during runtime due to query processing.

Note that performance metrics may sometimes be conflicting, and the system may need to choose to optimize performance for a user-specified QoS specfication.

#### 3) ***Ease of management***

The task of the DSMS does not end with issuing and answering queries over streaming data. A DSMS needs to provide a rich end-to-end management experience. Issues that need to be addressed include query composability, easy deployment over a variety of environments (single servers, clusters, cellphones, etc.), and security features to avoid data being accessed by users without the correct credentials. Query composability requires the ability to "publish" query results, as well as the ability for CQs to consume results of existing CQs and streams. Composability is a crucial building block for complex event-driven applications.

Another crucial feature is the ability to perform system health monitoring, where the user is able to easily query the current behavior of the system in terms of performance metrics such as throughput, memory usage, etc.

#### 4) ***Inter-operability***

A DSMS needs to operate closely with data from a variety of sources such as sensors, message buses, network ports, databases, distributed caches, etc. This facility is important because the DSMS is often used in combination with other middleware technologies in order to implement a rich end-to-end user experience.

### F. *Summary and Overview*

We find that DSMSs perform very good job of addressing these challenges, surpassing custom oriented techniques as well as database systems, due to the following characteristics:

- Ability to natively handle time (temporal semantics) and support for temporal concepts such as windows.
- High-performance main-memory-based execution
- Performing incremental computation of CQ results

Thus, DSMSs are a particularly good fit for computational finance applications. While DSMSs employ a fundamentally different push-based or event-driven processing model, they leverage many useful ideas from databases (declarative querying paradigm, strongly typed data with schema, leveraging indexes for performance, etc.). In this paper, we provide a detailed overview of stream processing systems from the perspective of our example financial application, walking the user through the process of creating and executing a complex continuous query over temporal data (real-time or offline).

The rest of this report is organized as follows. We first discuss the use of a DSMS for online as well as offline data in the context of the *monitor-manage-mine* loop in Section II. Then, we illustrate various aspects of using a DSMS for computational finance, along with the capabilities and advantages of using a DSMS, using the running example query introduced earlier. Specifically, Section III covers the problem of getting data into and out of the DSMS. Section IV covers basic operators, while Section V discusses the handling of user-provided computations. Section VI discusses other operations such as partitioned computations and correlations. Section VII covers several tools that are useful supplements in a DSMS. Section VIII presents design choices made by various DSMSs. Section IX presents a closer look at Microsoft StreamInsight as an example DSMS. We finally conclude the report in Section X.

## II. THE MONITOR-MANAGE-MINE CYCLE

The notion of time has been very crucial in many application scenarios across various business domains. To better understand how time is involved in these business domains, we conducted a survey over a wide range of customers, both in the financial and the non-financial domains. The survey investigates the range of time that is of interest to a specific domain.

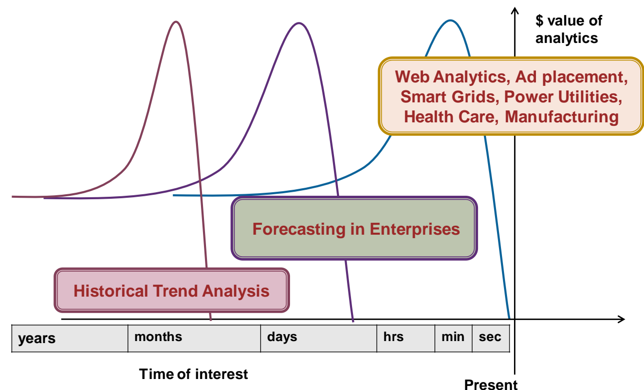



Figure 3. The value of time in analytics.

Going outside the scope of financial applications for a while, (Figure 3) illustrates the outcome of the survey across these business domains. For example, historical trend analysis applications build models, profiles, sketches or trends of users' behavior, stock prices or performance measures using years or months of historical data. Forecasting in enterprises consider months to days worth of data. Interestingly, some business domains are by far more time critical where minutes and seconds make a difference. Manufacturing is one of these domains where the failure of a sensor-monitored production line is not tolerable. Similarly, web analytics and online ad placement take advantage of the current user's behavior (up to the latest web click) to deliver targeted ads as the user navigates the web [1]. Most applications in health monitoring, power utilities and smart grids require, by nature, real time or near real-time decision making.

Now, moving back to the financial domain which is the focus of this paper, financial applications span the entire spectrum over the timeline. For example, building a trading model considers years of historical data while risk management applications detect various patterns of changes to stock prices according to latest stock tick. It is even more interesting, that financial recommendation systems combine the past and the future and blend them together according to a *magic* formula to deliver one piece of recommendation or to suggest a trading action. More specifically, algorithmic trading recommends a trading action by considering the pattern(s) exhibited by the latest stock tick(s) as well as the history of the monitored stock symbols. It looks up the past history for conditions that are similar to the current ones and predicts the future in the guidelines of the past experience.

To formalize one of the core concepts in decision making systems or recommendation based systems, the term *Key Performance Indicator* (KPI) is usually used to refer to the values, measures, or functions that are be monitored before a decision or a recommendation can be made. In particular, two questions need to be answered:

- What KPIs are of interest and are descriptive with respect to the monitored environment?
- What actions are to be taken given specific values of the KPIs?

The answers to the above questions are domain specific and very cumbersome. It is definitely the net outcome of years of domain expertise that shapes the answers to these questions. However, in the remainder of this section, we provide an approach where DSMSs help alleviate some of the pain involved in the design and deployment of a decision making or recommendation based system. We first provide some financial trends and facts learned over the past few decades before we describe the DSMS approach in more detail.

### A. *Trends and facts*

On one hand, the cost of data acquisition has become negligible thanks to the advances in hardware and software technologies, and more specifically thanks to the advances in communication technologies. Similarly, thanks to the advances in massive data storage devices, the cost of raw data storage is small and continues to decrease. On the other hand, data processing costs are non-negligible despite the advances in CPU technologies, cloud computing and parallel and distributed processing. Moreover, enormous amount of data is being acquired on a regular basis and is possibly stored to massive storage devices. The cost of loading such data (from storage devices) for processing and analysis purposes is significant and continues to increase with the increase in the data arrival rate. Therefore, although the advances in data acquisition and in data storage devices have been tangible, these advances resulted in large archives of data. These archives of data keep waiting for processing resources and are eventually held back from acquiring these resources due to the significant cost of data loading.

Fortunately, with the ability to process data incrementally while the data is in flight, DSMSs avoid data loading costs and provide the ability to process the data as it is being "*streamed on the wire*". Figure 2 introduced in Section I summarizes some of the previously-mentioned key trends and highlights some of the DSMS advantages. More interestingly, Figure 2 illustrates the $M^3$ cycle, which stands for "monitor-mine-mange". This concept is widely adopted by DSMSs to continuously monitor KPIs and to adapt to changes in the business needs as well as to changes in the behavior of the monitored environment. In the monitor-manage-mine cycle, a DSMS monitors a predefined set of KPIs, manages the application domain through a set of KPI-triggered actions and mines for interesting patterns and better KPIs as the system gets the opportunity to be trained over time using the incoming streams of data. In the following section, we elaborate on the role of a DSMS in the monitor-manage-mine cycle in more detail.

### B. *A dual role and a DSMS approach*

Figure 4 is a data flow diagram that illustrates the role of a DSMS as the monitor-manage-mine cycle takes place. The figure highlights five major steps (shown as numbered arrows in the figure). We summarize these steps as follows:

1. **Monitor and record:** Real time data streams are archived in a data store. Usually, many companies and institutions maintain years or months worth of data for operational and legal purposes. This step takes place outside a DSMS to collect an archive of historical data.
2. **Mine and design:** The historical data is replayed and pushed through a DSMS to mine for interesting patterns and to design good, meaningful, and effective KPIs as defined by business standards. In this "training" phase, several KPIs are devised and tested thanks to the ease and completeness of specification offered by DSMSs (as described in Section 1).
3. **Deploy:** Out of the devised KPIs (as described in step 2), a selected subset of KPIs are deployed over a DSMS instance that reads real time data streams.
4. **Manage and benefit:** At real time, a DSMS monitors the KPIs and manages the application domain according to a predefined set of KPI triggered actions. This step is termed as "*manage and benefit*" thanks to low-latency high-throughput nature of DSMS that gives the ability to invoke beneficial actions at real time.
5. **Feedback:** The resultant output of the real time processing (on the form of KPI values along the timeline) is sent back and archived in the data store. These KPIs values are analyzed to figure out how descriptive this selected subset of KPIs has been over the last deployment cycle.

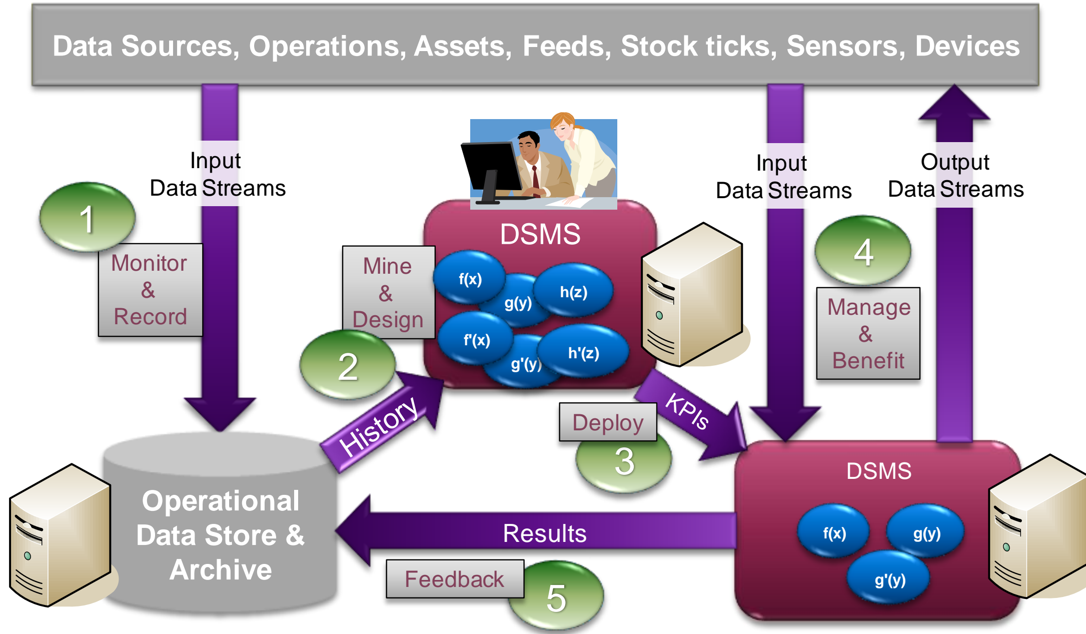



Figure 4. A data flow diagram for historical and real-time data through a DSMS.

Another shot of the "mine and design" phase (step 2) is conducted to further optimize the KPIs and come up with even more effective KPIs; and hence, the monitor-manage-mine cycle is reinitiated.

Note that a DSMS plays a dual role in the context of the monitor-manage-mine cycle. The DSMS is deployed over both historical and real time data. For historical data, a DSMS goes under a training phase to mine for interesting patterns and to help design effective KPIs. For real time data, a DSMS is deployed to monitor the previously designed KPIs and to manage the environment using KPI-triggered actions. The continuous monitor-manage-cycle ensures adaptivity to changes in the behavior of the underlying stream sources as well as flexibility to emerging business requirements.

## III. Getting Data In and Out of the Streaming System

From a ten thousand foot architectural view of traditional database management systems (DBMSs), we notice that DBMSs encompass two major components: a storage engine and a query processing engine. Regardless of where the data is coming from, the data end up being stored and indexed in secondary storage devices, at the discretion of the storage engine, in the "*best*" possible format. The query processing engine executes subsequent queries by first loading the data from the storage engine. The data moves from one operator to another through the query pipeline and the result is sent back to the query consumer. Therefore, data loading (in traditional databases) is carried over through, and only through, a storage engine. Also, note that both the input and the output of a query are potentially large, yet, *finite* bags of events.

With the paradigm shift in streaming-oriented workloads and the movement towards real-time low-latency requirements, the notion of a storage engine is getting out of the picture. The cost to materialize the data in secondary storage devices and to subsequently load it for query processing is a severe barrier in front of the previously mentioned real time low latency requirements. Moreover, the nature of the continuous queries that run over *infinite* (usually high-rate) input streams necessitate that query processing takes place and copes with the events as they are being streamed into the system's input buffers. Hence, a stream query processing engine needs to interact with the stream sources in real time and without a storage manager. Typically, the notion of an input adapter is used to refer to the mechanism that is used to push data from stream sources (e.g., sensor readings, stock quotes, etc.) to the DSMS. Similarly, and output adapter streams out the query result from the DSMS to the query consumer.

Adapters sit on the border between stream sources and the DSMS to moderate the communication between the two sides. Therefore, adapters are expected to provide several functionalities and to maintain several properties. In the remainder of this section we summarize some of these key functionalities and properties.

### A. *The Functionalities of an Adapter*

An *input* adapter writer needs to go through several steps to understand the properties of the stream source and to properly stream events out of the source. The adapter conveys the generated events to the DSMS as well the underlying properties of the stream source for optimization purposes. Given the fact that the output of a DSMS is a data

stream that is of similar nature to the input data stream, similar steps are needed to write an *output* adapter. The output adapter conveys the resultant stream events from the DSMS to the sink node or to the target consumer of the query output as well as the output stream properties.

We summarize the steps needed by an adapter writer as follows:

**Understand the format of the event.** Adapters need to understand the format of the events generated by the source and provide a schematized format that describes the event. For example, a stock ticker adapter might provide a stream with a schema that contains two of fields (symbol String, price decimal)

**Augment each event with temporal attributes.** Because each event occurs at a specific point in time and may last for a known (or an unknown) period of time, the adapter needs to augment the schema with one or more timestamps to denote the temporal attributes of the event. More precisely, we can think of three formats through which the adapter conveys the temporal information to the DSMS:

a- An event that occurs at a point in time, that is on the form of (*timestamp DateTime,* symbol String, price decimal), where *timestamp* denotes the point in time at which the stock price is reported. Events represented by this format are usually referred to as *point events*.

b- An event that spans an a priori known period of time, that is on the form of (*Vs DateTime, Ve Datetime,* symbol String, price decimal), where *Vs* and *Ve* are the start and end times of the *validity interval* over which the reported stock price has been valid, respectively. Events represented by this format are known as *interval events*.

c- An event that spans an a priori unknown period of time, where the event is represented by two signals: (*Vs DateTime,* symbol String, price decimal) denotes the time the event starts. Later on, another signal comes on the form of (*Ve DateTime,* symbol String, price decimal) to denote the end of the interval once information about the end time becomes available. Events represented by start/end signals are referred to as "*edge events*".

A stock tick is an example of point events as it ticks at a specific point in time. Alternatively, the stock price may be viewed as interval event that spans the duration in between two ticks. Finally, the intent of a customer to "sell" stocks for a certain price is an edge event. The event starts as soon as the transaction is placed. The end time is not known in advance. The event remains open till the point in time the sell transaction takes place. This is when the end of the event can be signaled.

**Handle out-of-order and retraction events.** Adapters communicate to the DSMS whether or not the events in the stream are coming in the order of their timestamps. Also, adapters interrogate the source for stability guarantees. Stability guarantees tell whether a generated event is final or it can be *retracted* and its value is altered later on. If late arrivals or retractions are expected from the input source, the adapter should be able to generate punctuations to denote stability guarantees and insert these punctuations in the input stream. Consequently, the DSMS uses these punctuations to provide correctness and stability guarantees on the generated output.

**Understand the delivery model of the streaming source (push versus pull based).** Depending on how source data arrives and interfaces provided by the DSMS system, an adapter might use a pull-based or a push-based model. Usually, real time events are pushed asynchronously while historical events are pulled synchronously. When pushing data to DSMS, adapter might have to negotiate the acceptable event rate with the DSMS (through control flow APIs) if the event rate exceeds DSMS's processing capabilities.

**Push computations as close to the source as possible.** DSMS's might optimize computations by filtering and reducing the data, as early as possible in the computation pipeline. The earliest possible stage would be the input adapter, thus a good adapter should have a way to negotiate filter and project pushdowns as close to the sources as possible. Note that this step is specific to input adapters and does not apply to output adapters.

### *B. Adapter-level High Availability and Scalability*

Besides the previously mentioned functionalities of adapters, there are two desirable properties adapters are expected to hold for robust deployment. First, an adapter is expected to be highly available and has the ability to interact with a highly-available DSMS. Note that having an adapter that is highly available on its own is a separate issue from the ability to interact with highly-available system. On one hand, a highly available adapter is an adapter that is resilient to failures (e.g., network failures) and has the ability to resume once the failure is over. On the other hand, highly-available DSMS have the ability to recover from failures. However, such recovery is not feasible without the help of input/output adapters. To recover from a failure, the DSMS may request the input adapter to replay portions of the input streams that have been previously submitted to the system and that got dropped due to an unexpected failure. Moreover, the DSMS may request the output adapter to rip out portions of the output streams that have been partially generated but abruptly interrupted due to the failure.

Second, an adapter is expected to be performant and to support the interaction with scalable DSMSs. A performant adapter is an adapter that does not stand as a bottleneck in the course of the continuous query execution. Multi-threading, buffering and stream multiplexing are example techniques adapters utilize to achieve high performance. In addition to being performant on their own, adapters should promote scalability to achieve high performance in large scale systems. Hence, adapters are expected to communicate with scalable DSMSs. In this situation, adapters are expected to partition the input streams and feed them to multiple nodes (each node is running an instance of the DSMS) and, later on, collect the output streams and merge them back to reconstruct the output stream.

### *C. Built-in versus Custom Adapters*

To ensure that adapters meet a minimum level of quality and to help developers write their own adapters, commercial DSMSs deliver a set of built-in adapters for widely used data sources and provide a mechanism for third-party vendors to build domain specific adapters. Example input adapters include modules to replay data from storage media (e.g., text files, event logs, or databases) and feed them to the DSMS in the form of streams. Also, an input adapter that reads live data from a web service is another interesting example. Output adapters compliment the process by providing a mechanism to stream the query output to text files, event logs, databases, or web services. More interestingly, some output adapters visualize the data graphically in a way that captures the trend and the intrinsic properties of the stream. Section VII provides example adapters for visualization purposes provided as part of commercial DSMS.

## IV. Basic Stream Operations

Now that the stock data is available in our system, we are ready to write some queries. Whenever we design a query, we begin with a clear statement of the query in English:

**Q1 : Select all stock quotes for the ticker symbol "MSFT", removing the unnecessary ticker symbol information in the output.**

In order to clearly understand the semantics of this query in precise, streaming terms, we provide example finite input and output tables:

StockTickerInfo:

| *Timestamp* | *Symbol* | *Price* |
|---|---|---|
| **0** | MSFT | 25.00 |
| **0** | IBM | 110.00 |
| **1** | MSFT | 25.03 |

SelectQuery:

| *Timestamp* | *Price* |
|---|---|
| **0** | 25.00 |
| **1** | 25.03 |

While the tables above are finite, none of the streaming operations we perform on the data assume finite data. Rather, we use this finite example as a way to drive understanding of the operations we wish to perform on our stream. Note that we are assuming that all stock readings are point events. Queries like these may be implemented using filter and project operations, where each event in the table is subjected to a test in the form of a boolean predicate. If the test passes, the event is retained, and a subsequent project removes the unnecessary ticker symbol information in the output.

Our next query is stated as follows:

**Q2 : Take the output signal produced by FilteredStream, and smooth it by reporting a 3 second trailing average every second.**

For exposition purposes, assume that the lifetime units are seconds. The following are example input and output streams:

FilteredStream:

| *Timestamp* | *Price* |
|---|---|
| **3** | 18.00 |
| **4** | 24.00 |
| **5** | 18.00 |
| **6** | 30.00 |

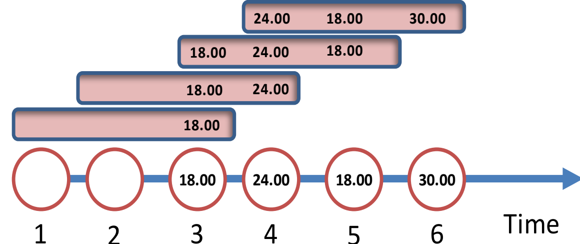
SmoothedStream:

| *Timestamp* | *Price* |
|---|---|
| **3** | 18.00 |
| **4** | 21.00 |
| **5** | 20.00 |
| **6** | 24.00 |
| **7** | 24.00 |
| **8** | 30.00 |

There are some interesting properties of the above table. First, there are no implicit zero values in the stream. Therefore, as we consider trailing averages prior to time 5 and after time 6, there are fewer than 3 readings used to calculate the average.

Computations like these may be modeled using the powerful windowing capability. One way to model this computation is using hopping windows, which group events together into buckets using two parameters. The first parameter, window size, determines the duration of time which each bucket corresponds to. In this case, since we have a trailing average of 3 seconds, we want to aggregate 3 seconds worth of data with each result, so the window size is 3 seconds. The second parameter is the hop size, which specifies how often these buckets are created along the timeline. In this case, since we are reporting every second, the hop size is 1 second.

A 3 second hopping window with a 1 second hop will create windows along the timeline in the following manner:




A hopping average computed with the above hopping window will produce, for each window above, an event whose lifetime is the lifetime of the associated window, and which averages the data whose lifetime intersects the window definition. Note that since all time intervals are inclusive of the start time, and exclusive of the end time, the first window which contains actual data is the one which starts at time 1, which includes only the first data point (at time 3).

## V. PERFORMING CUSTOM COMPUTATIONS

A DSMS provides a toolbox of "off-the-shelf" operations that are common across business applications. Very often, this toolbox is sufficient for the most common application requirements. However, it is natural for each business sector to have unique demands that are specific to its business logic. Business logic is the outcome of domain expertise in a specific sector over years. It is hard to expect that a single platform can *out-of-the-box* cover domain expertise in different domains. Thus, for broad applicability, a DSMS is expected to have an extensibility mechanism that can seamlessly integrate domain-specific business logic into the incremental in-memory streaming query processing engine.

### A. Design of an Extensibility Mechanism

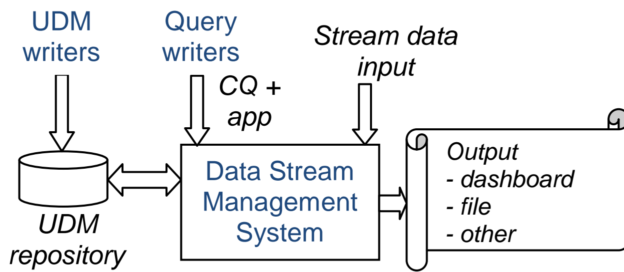



Figure 5: Entities in an extensible streaming solution for a domain.

As shown in Figure 5, there are three distinct entities that collaborate to extend a streaming system. The *user defined module* (*UDM*) *writer* is the domain expert who writes and packages the code that implements domain-specific operations as libraries. The *query writer* invokes a UDM as part of the query logic. A query is expected to have one or more UDMs wired together with standard streaming operators (e.g., filter, project, joins). Note that multiple query writers may leverage the same existing repository of UDMs for accomplishing specific business objectives. The *extensibility framework* is the component that connects the UDM writer and the query writer. The framework executes the UDM logic on demand based on the query to be executed. Thus, the framework provides convenience, flexibility, and efficiency for both the UDM writer and the query writer.

### B. An example UDM for pattern matching

As a concrete example, we discuss how one can implement a UDM that allows complex pattern matching over event streams. We assume that the user specifies the pattern to be matched in the form of a *non-deterministic finite automaton* (*NFA*). In order to support complex chart patterns, we add the concept of a *register* that consists of added runtime information associated with the automaton during transitions. We call such an augmented pattern matching specification as *augmented finite automaton (AFA).* The automaton consists of directed acyclic graph of states, including a start state and a set of final states. The states are connected by arcs or transitions, each of which is associated with a *fence function* (to determine if the arc can be triggered by an incoming event) and a *transfer function* (to compute the new value of the register in case the arc can be triggered). Both these functions have two input parameters: the new event and the current register value.

Assume we are currently in state X with register value R. If a new event arrives, we evaluate the fence function for each outgoing arc (from state X) to test whether that arc can be traversed due to the event. If yes, we would evaluate the transfer function to compute the new register R' associated with the destination state X'. If the X' is a final state, we produce an output event indicating a successful pattern match. For more details on AFA, the reader is encouraged to look at our paper [2]. In the next subsection, we will show how a complex chart pattern can be specified using AFA.

The AFA pattern matcher can be implemented as a UDM that can be reused by multiple query writers for matching different patterns. The query writer defines the actual automaton, and provides this AFA as input to the UDM. The UDM receives events from the system. For every event, the UDM performs some computation, updates its internal state, and optionally produces output events. Note that the query writer can use the UDM in a similar fashion as the native operators, combining it with windowing, selections, and other stream operations.

### C. Example: Chart Patterns over Smoothed Stock Stream

**Q3: Detect the head & shoulders chart pattern over the stock stream for MSFT, over a ten-minute window.**

Consider the smoothed stock stream in our running example. The query writer uses the AFA UDM to detect the *head & shoulders chart pattern*, where we look for a trading pattern that starts at price $p_1$, moves up to local maximum $p_2$, declines to local minimum $p_3 > p_1$, climbs to local maximum $p_4 > p_2$, declines to local minimum $p_5 > p_1$, climbs again to local maximum $p_6 < p_4$, and finally declines to below the starting price $p_1$. An example of such a pattern is shown in Figure 6.

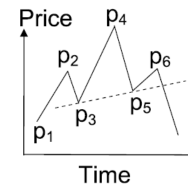



Figure 6: The head & shoulders chart pattern

We can use the AFA of Figure 7 to detect this pattern, where we use three registers $<r_1, r_2, r_3>$ to track the prices $p_1$, $p_2$, and $p_4$ respectively. In the figure, each arc in the automaton is annotated with the fence function and the transfer function (separated by a semicolon). State $q_0$ is the start state, while $q_7$ is the final state in the automaton. Stock upticks and downticks events are represented as up(e) and down(e) respectively. For example, the arc $a_2$ checks if we see a stock uptick; if yes, we store the current stock price ($p_2$) in register $r_2$, for later comparison within arc $a_6$.

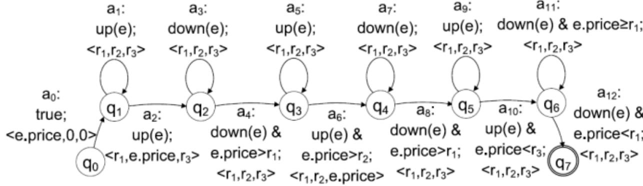
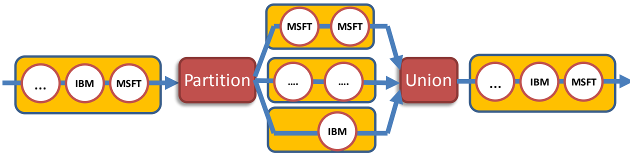



Figure 7: The head & shoulders chart pattern

## VI. PARTITIONED COMPUTATIONS, CORRELATIONS, AND ANTI-CORRELATIONS

In our examples so far, we are performing operations on an individual stock. In many cases, we wish to perform operations like these for each stock in the stock universe. For these types of operations, we use group and apply, which applies some stream valued function to partitions of a stream according to some partitioning function. We now write a query which applies the head and shoulders pattern detector to each stock in the stock universe:

**Q4: For each unique ticker symbol, detect the occurrence of the head & shoulders chart pattern of query Q3.**




The first part of this query partitioned the incoming stock ticks by the ticker symbol. The second part applies, to each of these partitions, our head and shoulders pattern matcher. The output of this query is all the outputs produced for each partition combined together into one unioned output stream.

This seemingly simple operation is surprisingly powerful in practice, and is leveraged to achieve high utilization of multiple cores.

So far, all of our queries have been queries over single streams of input. We now introduce a query that brings together two clearly different sources of data:

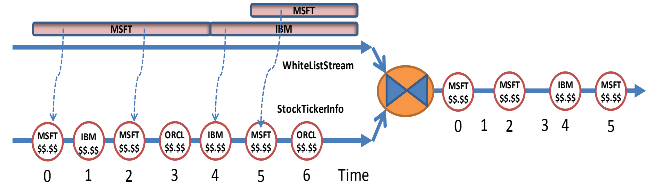
**Q5: For each stock on a dynamic "white list", determine all occurrences of the head & shoulders chart pattern.**

In addition to our stock ticker data, we now have another stream which describes the set of stocks which are being monitored at a particular time. Each event in this stream will have a ticker symbol and associated lifetime. For instance, the following is a valid white list stream (WhiteListStream):

| *Vs* | *Ve* | *Symbol* |
|---|---|---|
| **0** | 4 | MSFT |
| **4** | 7 | IBM |
| **5** | 7 | MSFT |

The above white list tells us that we will search for our head and shoulders patterns for Microsoft using data from time 0 to time 10, IBM from time 2 to time 20, and Microsoft again, but at the later time interval of 20 to 100.

To achieve the desired effect, we will join the above white list with the raw ticker data. Join takes two input streams, and correlates the events between them in the following way: Events on each side are matched to events on the other side whose lifetimes intersect, and for which a join condition involving the events from both sides holds. For each matching pair of events, an output is produced whose lifetime is the intersection of the input lifetimes, and whose payload combines the information from the two matching events.




For instance, if we join WhiteListStream and StockTickerInfo, and specify a join condition which matches the Symbol fields, the output is the filtered set of point events from StockTickerInfo which occurred during a matching lifetime specified in the WhiteListStream.

While the query above returns all stock tick info which conforms to our white list, we can also easily remove stock tick info using a black list. For instance, consider the following query:

**Q6: For each stock not present in a dynamic "black list", determine all occurrences of the head & shoulders chart pattern.**

Given a BlackListStream with a schema which is identical to the WhiteListStream, we could apply our black list by replacing join with an *anti-semi-join* operator. Anti-semi-join is used to detect the absence of information, and can also be used to detect the non-occurrence of an event.

## VII. SOFTWARE TOOLS

DSMSs are expected to be equipped with powerful engines that provide the necessary processing capabilities for real time data streams. The low-latency and high throughput aspects of these stream processing engines have been discussed throughout the sections of this paper. In this section, we describe some software tools that complement the processing capabilities of DSMS engine and provide smooth development and deployment experience of streaming applications. In this section, we focus mainly on programmability, manageability and visualization tools.

### A. *Programmability Tools*

Programmability tools enable developers to write, test and debug queries. With the continuous nature of streaming oriented workloads, debugging a continuous query becomes a challenge compared to ad hoc or snapshot queries in traditional database systems. Moreover, the temporal attributes of incoming events pose a major challenge in

determining the validity of the results under diverse and dynamic stream behavior. Hence, a tool is needed to facilitate the debugging and the tracing of events as they flow in the pipeline of the continuous query.

Note that there are fundamental differences between a control flow debugger of a programming language (for example, a C# or C++ debugger) and an event flow debugger in DSMSs. In control flow debugging, the developer "builds" a program written in a specific language in debug mode, enables breakpoints at specific statements or junctions in the control flow of the program, "runs" the program until these specific breakpoints are hit, reasons about the code and the state of the system, steps into or over functions and procedures, watches variables and so on until the completion of execution. Temporal reasoning of data variables (that is, analyzing the transformation of these variables through the passage of time) is limited or non-existent.

In contrast, event flow debugging traces an event through the passage of time, as it proceeds from one operator in the query to the next. Here, debugging involves understanding the effect an event has on the stream and how new events are generated as a result of the computations. The emphasis in event flow debugging is on how the operator's semantics (Filter, Project, Join, Aggregate, and so on) affect the event, rather than on the (control flow) execution of the operators themselves.

### B. *Manageability tools*

Managing the state of a DSMS involves tracking the overall health of the system and measuring the performance of running queries. Maintaining statistics that show the number of events produced/consumed by the system as well as CPU and memory usages help diagnose problems and troubleshoot the system in a timely manner. It also helps for resource planning and load balancing.

It is worth mentioning that maintaining statistics over fine grained granularities can be very costly in the resource intensive environment of streaming applications. Statistics are usually aggregated by time (e.g., hour, day, week, etc) and/or by query components. For example, component-level statistics are maintained at the operator level, query level, or even aggregated to the entire system level.

### C. *Visualization Tools*

Visualization tools are utilized for different purposes in the context of DSMSs. Yet, we summarize three major uses of visualization tools in this section. First, some DSMSs offer UI-based tools to compose queries graphically in a drag and drop fashion. Operators are represented as boxes while the flow of events among these operators is represented by arrows. Second, some DSMSs offer tools to visualize the query plans of queries that are being executed along with running statistics over each query component. Last but not least, there are visualization tools that enable users to monitor the data streams or aggregations/summaries over the data streams at real and compare them against each other and against historical data.

## VIII. CHALLENGES AND CHOICES IN EXISTING DSMSS

In this section, we dive into some of the more important and fundamental issues that a DSMS needs to solve, and discuss how various systems and research proposals have chosen to architect their solutions to address these issues. Our subjective analysis is not intended to comprehensively cover all choices in the commercial market, and is primarily intended to demonstrate some variety of choices, along with the strategies adopted by different systems. In Section IX, we will take a closer look at solutions adopted by Microsoft StreamInsight.

### A. *Range of Stream Processing Systems*

We are seeing a wide range of DSMS offerings in both research and commercial domains. The Stanford STREAM project [6] was one of the earlier stream processing systems proposed in the academic community; the Oracle CEP [10] engine [10] mostly follows STREAM's execution model. Another academic prototype named Aurora/Borealis [5] was later commercialized as the StreamBase [4] product. Coral8 (now acquired by Sybase CEP) is a popular DSMS, whose model is different from both Oracle CEP and StreamBase. Esper is yet another offering that focuses mostly on complex pattern detection over streaming data. The CEDR project [3] from Microsoft Research proposed a temporal algebra inherited from relational algebra and motivated by early research on temporal databases. Microsoft StreamInsight [1] uses an underlying execution model that is derived from this research. IBM InfoSphere Streams [8] is yet another streams offering whose roots are from the System S research project at IBM Watson Research.

### B. *Application time vs. System time*

Most stream processing systems consider time as a "first-class citizen", and offer native constructs such as temporal correlations (joins) and windows that allow operations over well-defined subsets of time (systems such as IBM InfoSphere Streams do not mandate time-oriented computations but can support them if needed by the application). StreamBase associates windows with operators, whereas systems such as StreamInsight associate window semantics with events, in the form of event lifetime which denotes the period of time during which an event contributes to output. An advantage of associating lifetimes with events is that it naturally fits into the intuitive definition of when an event is active, and also allows windowed operations to fit well into a temporal algebra (we discuss this in detail later).

The traditional notion of time is implicitly the *system time* at which the DSMS receives or processes events – indeed the first generation of streaming systems such as Stanford STREAM and Aurora used system time for

defining and computing windows. This choice has carried over to some commercial streaming systems such as Esper. Some streaming systems offer the ability to use and operate over timestamps specified by the application, called *application time*. StreamBase uses application time, but re-assigns application time to events based on the current system time. Such use of system time during stream processing has the potential of producing varying results across runs, for the same input dataset and query.

There are many advantages to separating application time (timestamps provided by the application and contained within the event) from system time (the time at which the DSMS receives or processes events), and further ensuring that all computations are deterministic in terms of application time. For example, it allows repeatable results regardless of when the DSMS processes the input, and how long individual operators take to process individual events. This helps query writers in understanding and reasoning about the operational semantics of the system, and can also enable easy debugging.

### C. Execution Model, Semantics, and Algebra

The semantics of query specification and how/when output is produced varies across systems. For example, in Oracle CEP and Coral8, an operator (such as moving average) result is produced whenever the window content changes (e.g., when an event gets added or removed from the window). On the other hand, StreamBase produces result events every pre-determined period of time (e.g., every second) regardless of when actual events enter or exit windows. The systems also differ in whether they report only when windows end or otherwise. Further, Coral8 supports an atomic bundling mechanism, using which we can simulate either tuple-driven output or time-driven output by simply controlling the content of each bundle/batch. Reference [12] gives a good overview of the range of different semantics and execution models across systems.

Microsoft StreamInsight recognizes and gives primary importance to semantics-related issues. One design goal for StreamInsight has been to ensure deterministic well-defined semantics for streaming queries. StreamInsight incorporates an underlying temporal algebra that treats events as having lifetimes, and query output as a time-varying database over the set of events. Application times, and not system time, is used to define operational semantics. The output at any given application time T is the result of applying the query semantics to the set of events that are "alive" at time T, i.e. all events whose lifetimes contain time T.

### D. Event Types

Recall the basic event types of point, edge, and interval (see Section II). Most commercial systems support only point events. In systems like STREAM, there is indirect support for edge events using the concepts of *Istreams* (insert streams) and *Dstreams* (delete streams) which correspond to start and end edge events. Each of these event types can be represented in Microsoft StreamInsight using events with lifetimes. In case of edge events, a start edge corresponds to an interval with infinity as endpoint, while an end edge sets the right endpoint to the correct value. Point events in StreamInsight are simply intervals with the right endpoint set to the left endpoint plus a *chronon* – the smallest possible time unit.

Interestingly, StreamInsight leverages its temporal algebra for testing the DSMS internally – a SQL Server database system is used to compute the expected output, which is compared with actual output to detect bugs in the DSMS engine (see [13] for details).

### E. Query Specification

We see a remarkable diversity in languages and specification techniques in the streams community, that we overview and summarize in this subsection.

The first streaming systems derived from databases, and thus inherited the use of SQL as the specification language. For example, the Stanford STREAM project proposes CQL (Continuous Query Language), which is a variant of SQL with windowing primitives. Commercial offering such as Oracle [10] and StreamBase [4] have followed this trend, offering variants of a language called StreamSQL (although convergence between the semantics is still an issue [9]).

While SQL-like queries may appeal to users coming to the streams world from a database background, one problem with this approach is the need for the application to construct query statement strings within the application and "hike them over the wall" to the DSMS for processing. This disconnect can affect the ease of entry into smoothly integrating stream processing into an end-to-end application.

IBM InfoSphere Streams uses an Eclipse-based integrated development environment, with a language called SPADE (Stream Processing Application Declarative Engine). Systems like Esper propose their own event processing language with constructs for pattern matching.

Microsoft StreamInsight uses the .NET and Visual Studio development environment for writing streaming applications, with the use of LINQ [11] for queries, with added temporal concepts for temporal specifications (e.g., windows). In this environment, the application writer gets the full power and flexibility of an integrated environment with facilities such as code correction and auto-complete while composing streaming queries. We discuss StreamInsight in detail in Section IX.

### F. Disorder and Revisions in Streams

Events typically arrive at the DSMS in non-decreasing timestamp order. However, there may be several reasons for events to arrive out-of-order – due to network delays and varying packet routes, merging streams from different sources, etc. Most commercial systems handle disorder by assuming that events are buffered and re-ordered before entering the DSMS. However, this approach has limitations:

it is difficult to predict the amount of disorder and hence buffering, and we may need to unnecessarily wait for a long time for late arrivals, even if there is no disorder in the stream.

Microsoft StreamInsight can internally operate directly over disordered data. The basic idea is that at any given time, it produces the best-effort result given the current set of events that have been delivered to the DSMS. If an event arrives late, it is incorporated into computations and the DSMS produces "revision" events to indicate a change in the actual result. This fits well with the temporal algebra, where at any given point in time, the DSMS result matches the expected output assuming we had seen everything in the input stream.

However, such a solution needs some notion of progress of time – for guaranteeing correctness of delivered output as well as for cleanup of internal state. This is achieved by using CTI (Current Time Increment) events. CTI events (also called *punctuations)* are associated with an application time T, that guarantee that no future event will have a timestamp earlier than T.

A similar concept exists in other systems, for the purpose of synchronizing system time with application time, such as heartbeats in STREAM, TIMEOUT in StreamBase, and MAXDELAY in Coral8.

### G. *Native Operations and Extensibility*

Given the diverse application domains where DSMSs are deployed, there is a strong need for a rich yet efficient set of extensibility mechanisms in the DSMS. For example, IBM InfoSphere Streams allows the ability to extend the set of built-in operators with user-defined ones, programmable in either C++ or Java. Microsoft StreamInsight incorporates a rich extensibility framework, giving application writers the flexibility to either re-use existing libraries or write new operators specifically for streaming environments. A range of windowing constructs and support is provided to users of this framework. User-defined operators can be written in any .NET language such as C#, and fits nicely into the single Visual Studio-based programming environment.

### H. *Query Composition*

It is necessary for a DSMS to allow the composition of queries by allowing new queries to re-use the results of existing queries. Most DSMSs allow some form of composition. Aurora uses a boxes-and-arrows style query construction technique that can be used for query composition. IBM InfoSphere Streams uses SPADE to support modular component-based programming. Microsoft StreamInsight supports *dynamic query composition*, where queries can re-use prior published stream endpoints.

## IX. MICROSOFT STREAMINSIGHT – A CLOSER LOOK

Micrrosoft StreamInsight is Microsoft's commercial DSMS that currently ships as a part of SQL Server 2008 R2. The StreamInsight data and execution model and programming surface is carefully designed to solve the challenges discussed in this report. In this section, we walk through the process of writing and executing queries in StreamInsight.

### A. *Getting data into the DSMS*

StreamInsight applications are written using Visual Studio. Thus, events types are simply classes or structs. For example, we may define a stocks event type as:

```
struct StockEvent {
   string Symbol;
   float Price;
}
```

The adapter developer writes input and output adapters. The input adapter reads data from the source and converts them into a stream of events of type `StockEvent`. Once the data has been defined and pulled in from the external source, we can write the streaming query logic (discussed in the next subsection) over this typed input stream. Finally, the output adapter reads the result stream and delivers it back to the application, for example using a graphical dashboard, on a console, or to a file on disk. StreamInsight supports a variety of adapter types, and a rich flow control model to control the buffering of events in the DSMS.

### B. *Writing a Continuous Query in LINQ*

StreamInsight queries are written in LINQ [11]. Our LINQ query for selecting events pertaining to a particular stock (M SFT) is written as follows (query Q1):

```
var filteredStream =
  from e in StockTickerInfo
  where e.Symbol == "MSFT"
  select new (Price = e.Price);
```

This query has 3 sections. The first section, the from clause, specifies that this query will process events from the StockTickerInfo stream. For each of these events e, we apply the filter specified in the where clause, which retains events where the ticker symbol equals MSFT. The select line says that the output of the query is a stream of events with one payload column, Price, and that Price is passed along unmodified from e.Price. We associate with the output stream, the variable FilteredStream.

Next, we wish to smooth the filtered stock stream, in order to report a 3 second trailing average every second (query Q2). This operation is performed using the following query.

```
var hoppingAvg =
from w in filteredStream.HoppingWindow
 (TimeSpan.FromSeconds(3),
  TimeSpan.FromSeconds(1),
  HoppingWindowOutputPolicy.ClipToWindowEnd)
select new { SmoothedPrice =
             w.Avg(e => e.Price) };
```

Therefore, the output of this hopping average is very close to our desired output. The only difference is that the output lifetimes aren't quite right. More specifically, we need to shift the output start times forward 2 secs, and reduce the lifetime to a chronon (instead of the 3 sec window description). This may be accomplished with an AlterEventLifetime, which modifies the lifetime of each event in the stream with two functions. The first function modifies the event start time, while the second modifies the event duration. The following query computes the final smoothed output for our query.

```
var correctedHoppingAvg =
hoppingAvg.AlterEventLifetime
  (e => e.StartTime+TimeSpan.FromSeconds(2),
  e => TimeSpan.FromTicks(1));
```

Note that the above query does not have a from clause. Some operations, like AlterEventLifetime, operate on streams, not events. As a result, these operations may be applied anywhere where a stream is required, including in the from clause of a query, which would apply the operation to the input before being processed by the rest of the query.

Another noteworthy feature of the above query is the use of *lambda expressions* ( e => etc…), which is a notation for defining functions within an expression. Each of these functions takes as input an event e. The first function computes, from this e, the new valid start time. The second function computes, from this e, the new event duration.

Next, assume that we wish to detect the head & shoulders chart pattern over a tumbling window of 10 minutes (query Q3). A non-incremental version of an AFA-based user-defined module (UDM) for StreamInsight is available at the StreamInsight blog [7]. We can write the AFA specification for the head & shoulders chart pattern depicted in Figure 7, and invoke our AFA UDM (called AfaOperator) using the LINQ query shown below:

```
var HSPattern =
from w in filteredStream.HoppingWindow
 (TimeSpan.FromMinutes(10),
  HoppingWindowOutputPolicy.ClipToWindowEnd)
select w.AfaOperator(HeadShoulderPattern);
```

Next, in order to detect the pattern for every unique stock symbol in a stream with quotes for multiple stocks (query Q4), we perform a grouping operation called Group&Apply (by stock symbol) using the following LINQ queries:

```
var EachHeadAndShoulders =
  from e in StockTickerInfo
  group e by e.Symbol;

var AllHeadAndShoulders =
  EachHeadAndShoulders.ApplyWithUnion(
    applyIn => HeadAndShoulders(applyIn),
    e => new {
      e.Payload,
      e.Key });
```

Finally, if we wished to limit pattern detection to only stock trades pertaining to a "whitelist" of stocks that we are interested in tracking (query Q5), we need to perform a join operation with the whitelist stream, before feeding the stock quotes to the grouping queries above. This is written in LINQ as follows:

```
var InterestingStream =
  from e1 in StockTickerInfo
  join e2 in WhiteListStream
  on e1.Symbol equals e2.Symbol
  select new { Symbol = e1.Symbol,
               Price = e1.Price }
```

Now that we have selected all the desired stock information using our white list, we may apply our other query logic to the reduced ticker stream to achieve the desired results.

While the query above returns all stock tick info which conforms to our white list, StreamInsight can also easily remove stock tick info using a black list (query Q6). For instance, given a BlackListStream with a schema which is identical to the WhiteListStream, we could apply our black list using the following LINQ query:

```
 var InterestingStream =
  from e1 in StockTickerInfo
  where (from e2 in BlackListStream
         where e1.Symbol == e2.Symbol
         select e2).IsEmpty()
  select e1;
```

The above query, for each event in StockTickerInfo, tries to join it to the BlackListStream. If the event from StockTickInfo doesn't join to any event in BlackListStream (the join result is empty), the event from StockTickerInfo is output. This query construct is called anti-semi-join, and was discussed briefly in Section VI.

Note from this example that StreamInsight queries are fully composable. Further, using a facility called dynamic query composition, one can publish/feed the result of one query, so that it can be re-used as the input to several another separately issued queries.

### C. *Software Tools in StreamInisght*

Figure 8 gives a snapshot of an example event flow debugger, the Microsoft StreamInsight Event Flow Debugger, which is an event trace-based debugging tool to trace events and to keep track of their effect on the resultant output stream. The debugger illustrates the query plan graphically and visualizes the movement of events across operators as the query is executed. The debugger gives users

the ability to pause and resume execution as well trace the effect of a single event in the output.

For query plan visualization purposes, Microsoft StreamInsight Event Flow Debugger (Figure 8) connects to the DSMS and visualizes all running query plans graphically. Meanwhile, for manageability purposes, Microsoft StreamInsight Event Flow Debugger (Figure 8) serves as a vehicle for exposing operator-level, query-level and server-level statistics to the user. These statistics are referred to as "diagnostic views" given the fact that these statistics are used to diagnose the state of the system and its running queries.

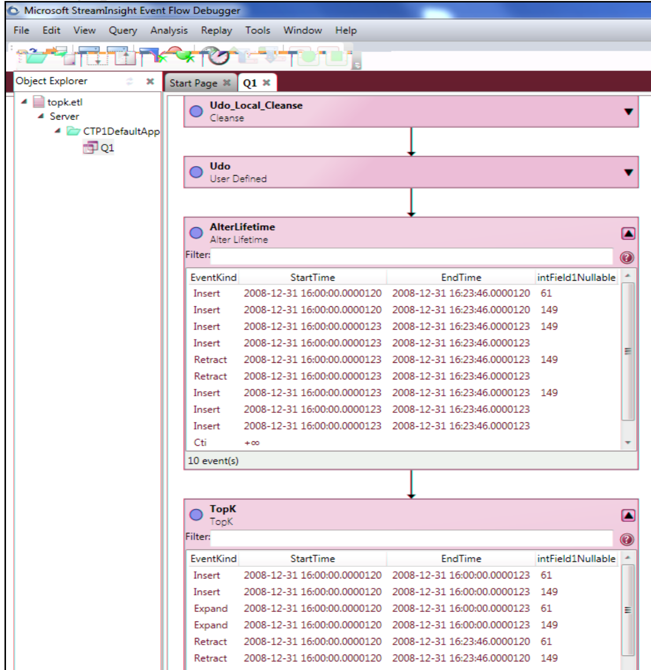



Figure 8: Microsoft StreamInsight Event Flow Debugger.

As an example, Microsoft StreamInsight provides an output adapter that continuously sends the output stream events to a Microsoft Excel sheet. Excel-based output adapters leverage all the visualization tools offered by Microsoft Excel. Moreover, Microsoft StreamInsight provides a Silverlight-based output adapter that enables output streams to be visualized from a web browser as shown in Figure 9. In the figure, streams of stock prices of various stock symbols are being visualized in real time. The combined value of programmability, manageability and visualization tools enable developers and administrators to take full advantage of the capabilities of DSMSs.

## X. CONCLUSIONS

In this report, we motivated the need for considering a new form of continuous data processing for computational finance. We introduced data stream management systems (DSMSs) as a general middleware for processing and

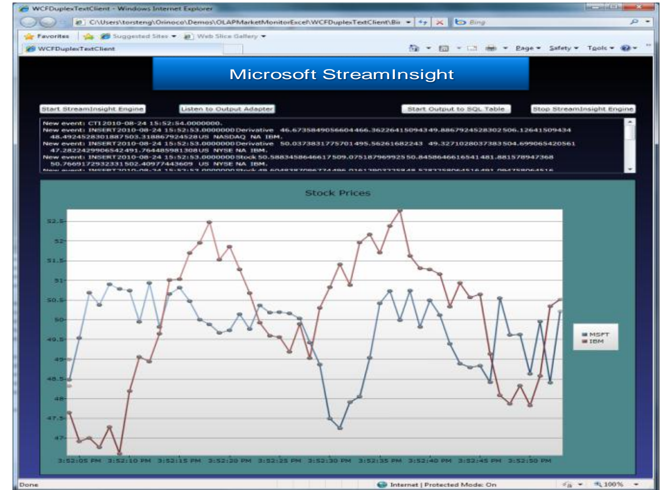



Figure 9: Microsoft StreamInsight Sliverlight based output adapter.

issuing long-running queries over temporal data streams. We detailed the unique challenges that DSMSs need to tackle, and showed how these challenges are addressed in the context of a running end-to-end example application in the financial domain. Given the inherent temporal nature of data in computational finance, our belief is that DSMSs are a perfect fit for many applications over both streaming data as well as offline data archived for historical analysis.

## XI. ACKNOWLEDGMENTS

We would like to thank the Microsoft StreamInsight team for nominating us to represent the team and supporting us in preparing this report.

## REFERENCES

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

[2] B. Chandramouli, J. Goldstein, and D. Maier. High-Performance Dynamic Pattern Matching over Disordered Streams. In VLDB 2010.

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

[4] StreamBase. http://www.streambase.com/

[5] D. Abadi et al. The design of the Borealis stream processing engine.

[6] In CIDR, 2005.Arvind Arasu, Shivnath Babu, Jennifer Widom: CQL: A Language for Continuous Queries over Streams and Relations. DBPL 2003: 1-19

[7] Pattern Detection with StreamInsight. http://tinyurl.com/2afzbhd.

[8] IBM InfoSphere Streams. http://www.ibm.com/software/data/infosphere/streams/

[9] N. Jain et al. Towards a streaming SQL standard. In VLDB, 2008.

[10] Oracle. http://www.oracle.com/

[11] Language Integrated Query (LINQ). http://msdn.microsoft.com/en-us/netframework/aa904594.aspx

[12] I. Botan et al. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. In VLDB 2010.

[13] A. Raizman et al. An Extensible Test Framework for the Microsoft StreamInsight Query Processor. In DBTest 2010.