



DATA STREAM MANAGEMENT SYSTEMS FOR COMPUTATIONAL FINANCE

Badrish Chandramouli, Mohamed Ali, Jonathan Goldstein, Beysim Sezgin, and Balan Sethu Raman
Microsoft

Because financial applications rely on a continual stream of time-sensitive data, any data management system must be able to process complex queries on the fly. Although many organizations turn to custom solutions, data stream management systems can offer the same low-latency processing with the flexibility to handle a range of applications.

Computational finance applications have unique needs born of ubiquitous networking and increasingly automated business processes. To be effective, such applications require the rapid processing of an unending data stream. Any system to manage this stream must process queries over continually changing data and be able to incorporate the results into ongoing business processes incrementally—all with extremely low latency. The system must also support the processing of data archived for historical analysis, which involves mining and back-testing real-time queries.

In this fast-paced environment, most organizations realize that reduced processing latency is a key requirement. For this reason, financial application developers often prefer customized solutions to solve specific problems. But customization is expensive, and, as a rule, customized

solutions do not generalize well, making this option a poor choice for an enterprise with diverse applications.

Organizations might also use databases to build such financial applications. In addition to forming a data-processing substrate that provides a simple yet powerful data model, databases enable applications to issue and execute declarative queries over set-oriented data. However, current databases cannot support the unique requirements of financial applications, which demand high-performance, complex, and time-oriented query processing over temporal data.

Recognizing the need for a more responsive general platform, many organizations are turning instead to data stream management systems (DSMSs), middleware that processes and issues long-running queries over temporal data streams. With a DSMS, applications can register queries before continuously and incrementally computing the result as new data arrives.

DSMSs use a cycle of data monitoring, managing, and mining to accommodate complex queries in streaming applications like computational finance. The notion of time is crucial in such applications, which include activities with both immediate and historic real-time data streams. Stock trading is an example. Risk management applications must detect various price-change patterns according to the latest tick, yet a trading model is based on years of historical data. Other applications make trading decisions on the basis of algorithms, delivering financial recommendations through some magic formula that blends past and future.

DSMSs are a perfect fit for these applications, which must query over streaming data as well as data archived for historical analysis.¹ Indeed, several DSMSs have already become popular, each with its own execution model and approach to stream management.

MEETING REAL-TIME NEEDS

To address the challenges of real-time applications, a DSMS must support or have certain features. One is specification ease and completeness. DSMS users, whose programming expertise varies widely, want to worry less (if at all) about application execution, focusing instead on query logic—how to compose the most effective query. Users might wish to encapsulate query logic into higher-level meaningful components and compose them into complex queries. A DSMS must also be able to handle late-arriving data or corrections. Computational finance activities, such as stock trading, are dynamic, which means that the DSMS must deterministically resolve late-arriving events. Also, a source can generate an event, later detect an error, and then either change or remove the previously generated event.

Ease of query debugging is useful because it is difficult to verify the correctness of data that the system is constantly processing.

Another desirable feature is extensibility. The DSMS toolbox covers many scenarios, but computational finance applications are complex and can evolve quickly, requiring some framework for extending existing tools. Specific challenges include supporting legacy libraries and writing streaming building blocks.

Ease of query debugging is also useful because it is difficult to verify the correctness data that the system is constantly processing. Using application time instead of system time, along with concrete temporal algebra semantics, makes query results deterministic regardless of the order in which the system processes individual events.

Finally, high performance is of paramount importance in real-time applications. To achieve high performance, DSMSs usually maintain internal state in main memory and discard events or state when they are no longer needed. When event arrival rate exceeds the system's processing capability, the system buffers events in the in-memory event queues between operators. Performance measurement is usually based on three definitions:

- *measurable latency period*—the point at which an event enters the DSMS to the point at which the user observes its effect in the output;

- *throughput*—the number of result events that the DSMS produces per second of runtime; and
- *memory use*—the in-memory state accumulated during runtime as a result of query processing.

Beyond answering queries over streaming data, a DSMS must support the ability to publish query results and enable queries to consume the results of other published queries and streams at runtime. A DSMS should be easy to deploy in a variety of environments, including single servers, clusters, and smartphones. Like other business systems, it should implement security features to avoid unauthorized data access and monitor system health against defined performance metrics.

HOW THE SYSTEM WORKS

Figure 1 shows how a DSMS might support a stock trading application by managing the flow of both historical and real-time data. As the figure implies, stock trading can have multiple data streams, but the most obvious one is stock quotes, in which each event (or notification) in the stream represents a single quote.

For each stock, the trading application smoothes out the signal using a moving average to rid the data stream of spikes or outliers and then performs some customized computation, such as detecting a chart pattern. The stocks that the application is interested in tracking can change according to the stock portfolio, and the application typically has a stock white list (stocks to trade) or blacklist (stocks to avoid). The DSMS continually monitors the event stream and alerts the trading application when it detects a chart pattern for a relevant stock.

Support of the trading application begins when the application registers a continuous query (CQ) with the DSMS. The DSMS then processes incoming stock quotes against the registered CQ, considering quotes that satisfy the black- or white list and performing the requested CQ operations, such as grouping and pattern matching. Stock trade messages reach the DSMS as events, each of which consists of a trade timestamp, stock symbol, and last traded price. The trading model views both stock trades and list updates as event streams, since both activities consist of notifications that report either a stock trade or a change to a set of stock symbols.

The DSMS produces a new event when the CQ results require updates or other changes. For example, if the DSMS detects a chart pattern for a particular stock, it produces output conveying this discovery. The trading application can then take appropriate action such as buying or selling. Because a CQ receives as well as produces event streams, a DSMS user can compose CQs to satisfy a variety of complex requirements.

Monitor-manage-mine cycle

Computational finance applications like stock trading rely on key performance indicators (KPIs), which refer to the values, measures, or functions the system must monitor before making a decision or a recommendation. Historically, application designers have attempted to answer two questions about KPIs:

- What KPIs are of interest in the monitored environment?
- Given specific KPI values, what actions should the system take?

The answers to these questions depend, of course, on the domain and the questioner's expertise, but a DSMS can help alleviate some of the angst of seeking them through its monitor-manage-mine (M3) cycle. In this cycle, the DSMS continuously monitors a predefined set of KPIs, helps manage the application domain through a set of KPI-triggered actions, and then mines data for interesting patterns and better KPIs.

As Figure 1 shows, the M3 cycle has five major steps, which together give the DSMS the flexibility to accommodate changing needs in both business processes and the monitored environment's behavior.

1. *Acquire and archive.* The DSMS acquires data from real-time streams, makes it available for real-time data processing, and archives it for subsequent operations or to satisfy legal requirements. As part of this process, the system can clean or compress the acquired data.
2. *Mine and design.* The DSMS replays historical data to execute queries that mine for interesting patterns. The overall goal is to design meaningful and effective KPIs.
3. *Deploy.* After finding the KPIs, the DSMS deploys a selected subset as queries that read real-time data streams.
4. *Manage.* The DSMS computes the selected KPIs over the real-time data from Step 1 and manages the application domain according to a predefined set of actions governed by the KPIs. The business benefits from low latency and high throughput.
5. *Feed back.* The DSMS archives the real-time processing output and analyzes KPI values to determine KPIs' effectiveness. After this step, another iteration of Steps 2 through 4 occurs, completing the M3 cycle.

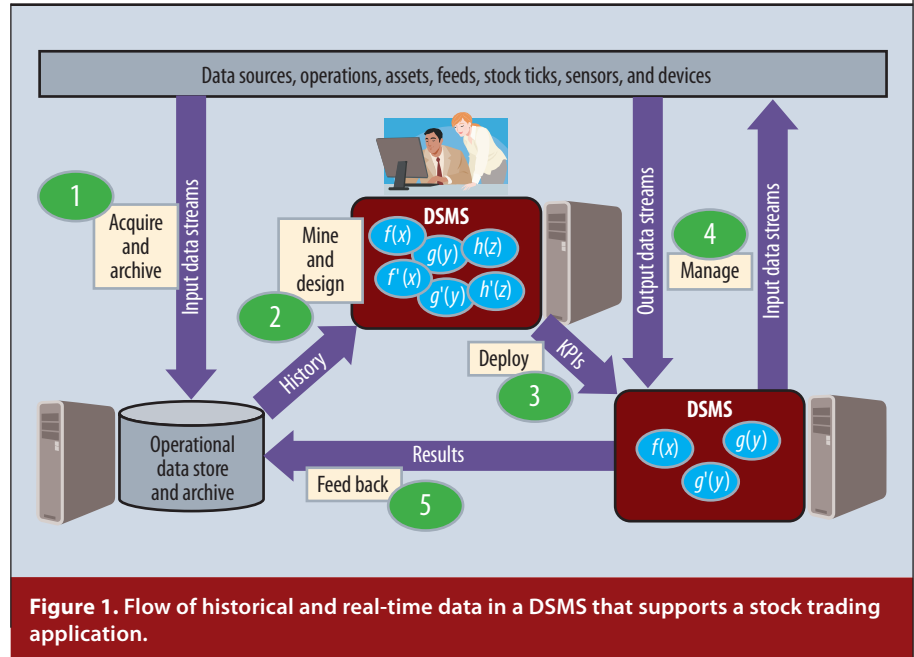


Figure 1. Flow of historical and real-time data in a DSMS that supports a stock trading application.

Data input and output

Any data exchange requires a communication moderating mechanism. In a DSMS, adapters fill this role. An input adapter pushes data from stream sources to the DSMS; an output adapter streams query results from the DSMS to the consumer.

The adapter writer must first understand the source event's format and provide a schema that describes the event. A stock ticker adapter, for example, might provide a stream with a schema that contains two fields (symbol: string, price: decimal).

The writer must also augment each event with temporal attributes. Because each event occurs at a specific time and generally lasts for some period (known or unknown), the adapter must augment the schema with one or more timestamps to denote the event's temporal attributes. The writer chooses one of three formats for conveying the temporal information:

- *Point event:* An event occurs at a point in time, taking the form (timestamp: DateTime, symbol: String, price: decimal), where timestamp denotes the time at which the stock price is reported.
- *Interval event:* An event that spans an a priori known period, taking the form (Vs: DateTime, Ve: Datetime, symbol: string, price: decimal), where Vs and Ve are the start and end times of the interval during which the reported stock price was valid.
- *Edge event:* An event that spans an a priori unknown period and is represented by two signals: (Vs: DateTime, symbol: string, price: decimal) to denote the interval's start and (Ve: DateTime, symbol: string, price: decimal) to denote the interval's end.

```

struct StockEvent // Stock event type
{
    string Symbol;
    float Price;
}

// Filter and project (Q1)
var filteredStream =
    from e in StockTickerInfo
    where e.Symbol == "MSFT"
    select new { Price = e.Price };

// Windowing operation (Q2)
var hoppingAvg =
    from w in filteredStream.HoppingWindow
        (TimeSpan.FromSeconds(3),
         TimeSpan.FromSeconds(1),
         HoppingWindowOutputPolicy.
         ClipToWindowEnd)
    select new { SmoothedPrice =
        w.Avg(e => e.Price) };

// User-defined operator (Q3)
var HSPattern =
    from w in filteredStream.HoppingWindow
        (TimeSpan.FromMinutes(10),
         HoppingWindowOutputPolicy.
         ClipToWindowEnd)
    select w.AfaOperator(HeadShoulderPattern);

// Group and apply (Q4)
var EachHeadAndShoulders =
    from e in StockTickerInfo
    group e by e.Symbol;

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

// Join or correlation (Q5)
var InterestingStream =
    from e1 in StockTickerInfo
    join e2 in WhiteListStream
    on e1.Symbol equals e2.Symbol
    select new { Symbol = e1.Symbol,
        Price = e1.Price };

// Anti-join (Q6)
var InterestingStream =
    from e1 in StockTickerInfo
    where (from e2 in BlackListStream
        where e1.Symbol == e2.Symbol
        select e2).IsEmpty()
    select e1;

```

Figure 2. Query construction for a stock trading application using Microsoft Streamlight. The DSMS uses LINQ queries, each of which has three sections: “from” identifies the source stream; “where” applies filters; and “select” performs projection to ensure that the query output has events with the expected schema and content.

A stock tick could be a point event, since it ticks at a specific time, but it could also be an interval event that spans the duration between consecutive ticks. The customer’s intent to sell stocks for a certain price is an edge event, starting as soon as the customer places the transaction. The end time is not known ahead of time; rather, the event remains open until the sell transaction takes place, signaling the end of the event.

Table 1. Sample finite input (StockTickerInfo) for “Select stock quote events for symbol ‘MSFT,’ discarding all other events.”

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

Table 2. Sample finite output (SelectQuery) for “Select stock quote events for symbol ‘MSFT,’ discarding all other events.”

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

Query construction

To illustrate query construction, we describe the process in terms of Microsoft’s StreamInsight, a commercial DSMS that currently ships as a part of SQL Server 2008 R2. Figure 2 shows the StreamInsight queries for the stock trading application, which are written in Language Integrated Query (LINQ; <http://msdn.microsoft.com/en-us/netframework/aa904594.aspx>). The six queries we describe are applicable to many scenarios in stream-oriented workloads.

Query construction begins with a statement in English, which has from, where, and select clauses. The results of LINQ queries are associated with variables that subsequent queries reference.

The informal English version of the first query in Figure 2 is

Q1: Select stock quote events for symbol “MSFT,” discarding all the other events.

Tables 1 and 2 provide insights into the precise semantics of this query.

We have made Tables 1 and 2 finite to drive the understanding of what operations the system must perform on

this data stream. In reality, typical streaming operations do not assume finite data. We also assume that all stock readings are point events, so it is possible to implement Q1 using filter and project operations. A filter operation subjects the event to a Boolean test to compare the symbol to MSFT. If the event passes the test, the operation retains it and removes the unnecessary ticker symbol column in the output.

The second query is

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

This query introduces windowing extensions that use a hopping window approach to create windows using two parameters to group events into buckets. The first parameter, window size, determines the duration of each bucket. In the query, the trailing average is 3 seconds, so the goal should be to aggregate 3 seconds worth of data with each result, making the window 3 seconds long.

The second parameter in the hopping window approach is hop size, which specifies how often the system creates buckets along the timeline. Because the query specifies that reports occur every second, the hop size is 1 second. It is also feasible to use a sliding window, which reports updates to the result whenever the DSMS receives a new event.

For each window, a hopping average computed with these parameters will produce an event with a price equal to the average price across all the events in that window. The first window that contains data starts at time 1 and contains only one event (with timestamp 3).

Tables 3 and 4 show sample (again finite) input and output streams. We assume that time is in seconds.

The third query highlights the need for an extensibility mechanism to cover more specific business logic:

Q3: Detect the head-and-shoulders pattern over the stock stream for MSFT, over a 10-minute window.

A DSMS toolbox has many operations that are common across business applications, making it sufficient for most requirements, but each business sector tends to have its own demands. The extensibility mechanism seamlessly integrates specific logic into the stream processing.

Extending the DSMS is a collaboration between the user-defined module (UDM) writer and the query writer. The two collaboratively use an extensibility framework to execute the UDM. A domain expert writes the UDM, which essentially comprises libraries of domain-specific operations. The query writer invokes a UDM as part of the query logic. The extensibility framework executes the UDM logic on demand, in essence connecting the UDM and query writers and making the writing process convenient and efficient for both.

Table 3. Sample finite input (FilteredStream) for “Take the output signal produced by FilteredStream, and smooth it by reporting a 3-second trailing average every second.”

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

Table 4. Sample finite output (SmoothedStream) for “Take the output signal produced by FilteredStream, and smooth it by reporting a 3-second trailing average every second.”

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

As Figure 3 shows, the head-and-shoulders trading pattern 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.

Pattern matching is a typical use case for the extensibility framework. DSMSs generally use native operators to support basic pattern matching, but complex pattern matching, which targets financial applications, is beyond the capabilities of simple pattern languages. Researchers have proposed several alternative techniques that are more suitable for matching complex patterns over streams.

Microsoft is exploring a technique² to augment a finite state automaton with a register to track added

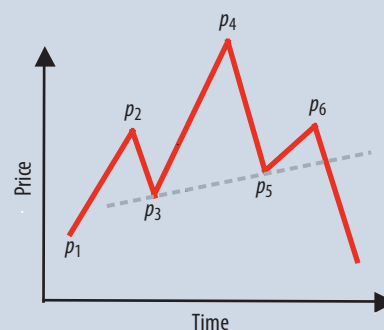


Figure 3. The head-and-shoulders trading pattern. This complex pattern is beyond the capability of simple pattern languages, such as regular expressions.

Table 5. White list stream (WhiteListStream) in response to the query “For each stock on a dynamic ‘white list,’ determine all occurrences of the head-and-shoulders chart pattern.”

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

runtime information, which is associated with the automation during event-triggered transitions between states. (A UDM for pattern matching with this technique is available at the StreamInsight blog; <http://blogs.msdn.com/b/streaminsight/archive/2010/09/02/pattern-detection-with-streaminsight.aspx>).

The queries so far concern operations on a single stock, but a trader might want to perform similar operations for multiple stocks. The next query performs grouped computation using the “group by” LINQ clause, with the repeated computation (“apply”) specified as a separate query.

These operations rely on the notion of group-and-apply, which distributes some stream computation to every partition of the stream according to a specific grouping or partitioning function.

The fourth query applies the head-and-shoulders pattern detector to each symbol in the stream:

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

The first part of Q4 partitions the incoming stock ticks by the ticker symbol. The second part applies per-symbol pattern matching² to each partition. The query output combines all the outputs produced for each partition. This group-and-apply operation (grouped computation) is surprisingly powerful in practice and is common in DSMS queries. The operation partitions a stream into substreams, each of which has the same nature as the parent stream, making it easier to compose and scale streams. In that sense, it is similar both to the standard relational group-by operation and—for large amounts of stream data—to the map-reduce operation.

In contrast to stream partitioning, the fifth query brings together two data sources—stock ticker data and the set of stocks monitored over a particular interval:

Q5: For each stock on a dynamic “white list,” determine all occurrences of the head-and-shoulders chart pattern.

Each event in this stream will have a ticker symbol and associated lifetime. Table 5 shows a white-list stream (WhiteListStream).

According to this stream, particular symbols are of interest only during the corresponding event lifetimes.

For events with matching symbols, a join of StockTickerInfo and WhiteListStream correlates the two streams and produces output only if lifetimes intersect and the join condition is satisfied. Hence, the join operation exploits the temporal aspects of the incoming stream events.

There is often a need to check for the absence of certain information. For example, although Q5 returns all stock tick events that satisfy the white-list condition, it is also easy to remove stock tick information using a blacklist.

Q6: For each stock that is NOT on a dynamic “blacklist,” determine all occurrences of the head-and-shoulders chart pattern.

This query yields BlackListStream, which has the same schema as WhiteListStream. Replacing the join operator with an anti-join operator detects the nonoccurrence of an event (applies the blacklist).

EXISTING SYSTEMS AND DESIGN CHOICES

Several organizations, both academic and commercial, have developed prototype and fully functional DSMSs, including but not limited to

- Oracle Complex Event Processing (CEP),³ based on the Stanford Stream research model;
- StreamBase,⁴ based on the Aurora and Borealis research models;
- StreamInsight,⁵ based on Microsoft’s Complex Event Detection and Response (CEDR) project;
- IBM InfoSphere Streams,⁶ based on the System S research project; and
- Coral8 (acquired by Sybase).

Developers of each system have chosen various execution models, semantics, query processing approaches, and extensibility mechanisms.

Execution model and semantics

Systems vary in their use of query specification semantics and how and when they produce output.⁷ For example, Oracle CEP and Coral8 produce an operator result whenever the window content changes. When the user sends events to the system, the system assigns the event to the appropriate window according to the window definition the user provides as part of the CQ specification. StreamBase produces result events every predetermined period, (for example, every second) regardless of when actual events enter or exit windows.

Microsoft 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 time helps define operational semantics. The output at any given application time T is the

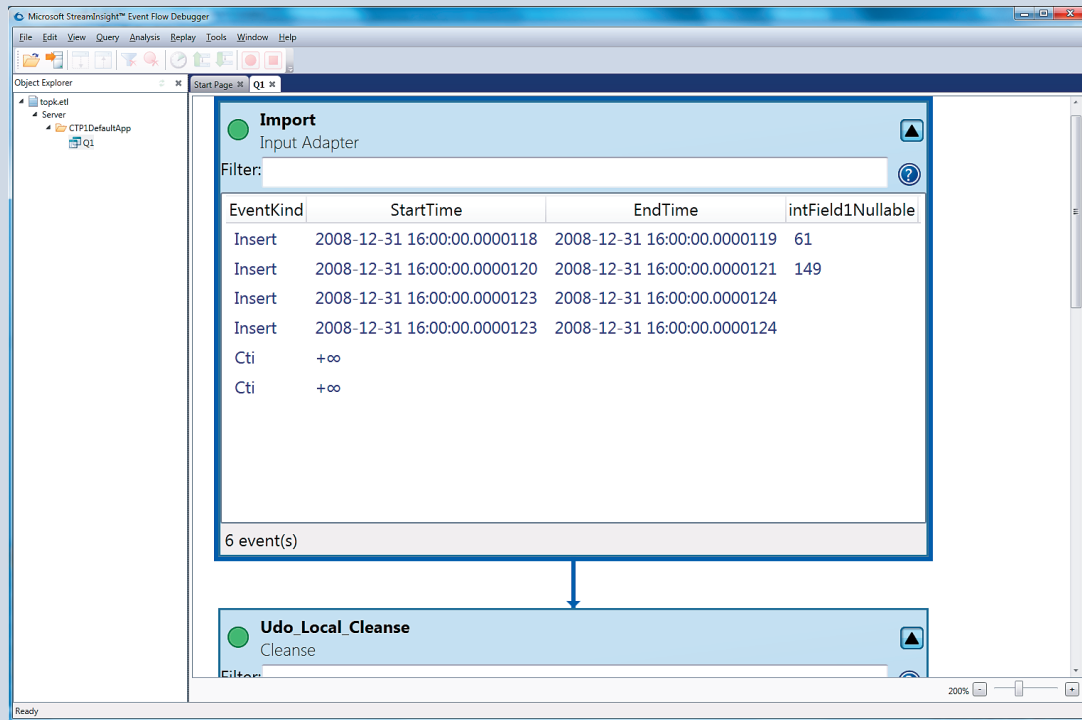


Figure 4. Microsoft StreamInsight’s event flow debugger. The tools trace events and keep track of how they affect the output stream.

result of applying the query semantics to the set of events that are alive at time T —all events whose lifetimes contain time T .

Toolsets

Software tools combine features such as programmability, ease of management, and powerful visualization capability to enhance the DSMS user’s system experience and productivity. Several tools are integral to DSMS operation, and are usually offered as part of the DSMS. Event flow debuggers are event trace-based tools that track the effect of events on the resultant output stream. The debuggers illustrate the query plan graphically and visualize event movement across operators as the query executes. Event flow debuggers also let users pause and resume execution as well as trace the effect of a single event in the output. Figure 4 gives a snapshot of the event flow debugger in the StreamInsight DSMS.

Output adapters are also useful tools because they enable desktop connectivity to streams through information tools such as Web browsers, spreadsheets, word processors, and report generators.

Query specification

The languages and specification techniques used in the streams community demonstrate remarkable diversity. The first DSMSs derived from databases and thus

inherited the use of SQL (with temporal extensions). Oracle and StreamBase have followed this trend with the StreamSQL language, although convergence between the semantics is still an issue.⁸ IBM InfoSphere Streams uses an Eclipse-based integrated development environment, with a language called SPADE.⁶ Microsoft StreamInsight uses the .NET and Visual Studio IDE for writing streaming applications, with LINQ for queries.

Stream disorder

Events can arrive late or out of order for many reasons, including network delays and varying packet routes and merging streams from different sources. Most commercial systems handle disorder by assuming that events are buffered and reordered before entering the DSMS. The Stanford Stream project uses heartbeats to bound disorder; StreamBase uses Timeout, and Coral8 uses Max-Delay. However, bounding disorder has disadvantages. It is difficult to predict the amount of disorder, and the wait for late arrivals might be long, even without disorder in the stream.


StreamInsight takes a different approach, operating directly over disordered data. If an event arrives late, the DSMS incorporates it into computations and produces “revision” events to indicate a changed result. Such a solution needs some notion of time progress to bound disorder. StreamInsight uses punctuations associated with an appli-

cation time T , which guarantees that no future event will have a timestamp earlier than T .

Extensibility and composition

IBM InfoSphere Streams allows the extension of operator sets with user-defined operators, programmable in either C++ or Java. StreamInsight incorporates an extensibility framework that permits the addition of user-defined operators in any .NET language such as C# in Visual Studio. Most DSMSs allow some form of composition. IBM InfoSphere Streams uses SPADE to support modular component-based programming. StreamInsight supports dynamic query composition, in which queries can reuse published stream endpoints.

Computational finance has unique requirements that argue strongly for a new form of continuous data processing. DSMSs show promise in satisfying the special needs of real-time applications with their ability to process and issue long-running queries over temporal data streams.

As with any maturing technology, work remains to refine the concept. Research must be devoted to identifying a common language and semantics standard and providing better real-time analysis, mining, and learning techniques and tools. Such tools will complete the M3 cycle, making it fully real time. Despite these open issues, DSMSs appear to be a near-perfect fit for many applications over both streaming data as well as offline data archived for historical analysis. 

Acknowledgments

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

References

1. B. Chandramouli et al., "Data Stream Management Systems for Computational Finance," tech. report, MSR-TR-2010-130, Microsoft Research, Sept. 2010; <http://research.microsoft.com/pubs/138844/streams-finance.pdf>.
2. B. Chandramouli, J. Goldstein, and D. Maier, "High-Performance Dynamic Pattern Matching over Disordered Streams," *Proc. Conf. Very Large Databases (VLDB 10)*, ACM Press, 2010, pp. 220-231.
3. A. Arasu, S. Babu, and J. Widom, "CQL: A Language for Continuous Queries over Streams and Relations," *Proc. 9th Int'l Workshop Database Programming Languages*, ACM Press, 2003, pp. 1-11.
4. D.J. Abadi et al., "The Design of the Borealis Stream Processing Engine," *Proc. 2nd Biennial Conf. Innovative Data Systems Research (CIDR 05)*, ACM Press, 2005, pp. 277-289.
5. R.S. Barga et al., "Consistent Streaming Through Time: A Vision for Event Stream Processing," *Proc. 3rd Biennial Conf. Innovative Data Systems Research (CIDR 07)*, ACM Press, 2007, pp. 363-374.
6. B. Gedik et al., "SPADE: The System S Declarative Stream Processing Engine," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD 08)*, ACM Press, 2008, pp. 1123-1134.
7. I. Botan et al., "SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems," *Proc. Conf. Very Large Databases (VLDB 10)*, ACM Press, 2010, pp. 232-243.
8. N. Jain et al., "Towards a Streaming SQL Standard," *Proc. Conf. Very Large Databases (VLDB 08)*, ACM Press, 2008, pp. 1379-1390.

Badrish Chandramouli is a researcher in the database group at Microsoft Research. His research interests include stream processing, publish-subscribe systems, networks, and database query processing. Chandramouli received a PhD in computer science from Duke University. He is a member of IEEE and the ACM. Contact him at badrishc@microsoft.com.

Mohamed Ali is a senior software design engineer in the StreamInsight group at Microsoft. His research interests include advancing the state of the art in streaming systems to cope with the requirements of emerging applications. Ali received a PhD in computer science from Purdue University. He is a member of the ACM. Contact him at mali@microsoft.com.

Jonathan Goldstein is part of Microsoft's StreamInsight development team. His research interests include query optimization and processing, multimedia retrieval, and streaming. Goldstein received a PhD in computer science from the University of Wisconsin. Contact him at jongold@microsoft.com.

Beysim Sezgin is a principal architect in Microsoft's StreamInsight group. His interests include the architecture and design of low-latency event data processing and continuous query processing systems. Sezgin received a BS in computer science from Ege University, Izmir, Turkey. Contact him at beysims@microsoft.com.

Balan Sethu Raman is a Distinguished Engineer in Microsoft's Business Platform Division and head of the StreamInsight group. His interests include distributed systems and algorithms, query processing, and information retrieval. Raman received an MTech in computer science from the Indian Institute of Technology, Chennai. He is a member of the ACM. Contact him at sethur@microsoft.com.



Selected CS articles and columns are available for free at <http://ComputingNow.computer.org>