The 8 Requirements of Real-Time Stream Processing

Michael Stonebraker
Computer Science and Artificial Intelligence Laboratory, M.I.T., and StreamBase Systems, Inc.
stonebraker@csail.mit.edu

Uğur Çetintemel
Department of Computer Science, Brown University, and StreamBase Systems, Inc.
ugur@cs.brown.edu

Stan Zdonik
Department of Computer Science, Brown University, and StreamBase Systems, Inc.
sbz@cs.brown.edu

ABSTRACT

Applications that require real-time processing of high-volume data steams are pushing the limits of traditional data processing infrastructures. These stream-based applications include market feed processing and electronic trading on Wall Street, network and infrastructure monitoring, fraud detection, and command and control in military environments. Furthermore, as the "sea change" caused by cheap micro-sensor technology takes hold, we expect to see everything of material significance on the planet get "sensor-tagged" and report its state or location in real time. This sensorization of the real world will lead to a "green field" of novel monitoring and control applications with high-volume and low-latency processing requirements.

Recently, several technologies have emerged—including off-the-shelf stream processing engines—specifically to address the challenges of processing high-volume, real-time data without requiring the use of custom code. At the same time, some existing software technologies, such as main memory DBMSs and rule engines, are also being "repurposed" by marketing departments to address these applications.

In this paper, we outline eight requirements that a system software should meet to excel at a variety of real-time stream processing applications. Our goal is to provide high-level guidance to information technologists so that they will know what to look for when evaluation alternative stream processing solutions. As such, this paper serves a purpose comparable to the requirements papers in relational DBMSs and on-line analytical processing. We also briefly review alternative system software technologies in the context of our requirements.

The paper attempts to be vendor neutral, so no specific commercial products are mentioned.

1. INTRODUCTION

On Wall Street and other global exchanges, electronic trading volumes are growing exponentially. Market data feeds can generate tens of thousands of messages per second. The Options Price Reporting Authority (OPRA), which aggregates all the quotes and trades from the options exchanges, estimates peak rates of 122,000 messages per second in 2005, with rates doubling every year [13]. This dramatic escalation in feed volumes is stressing or breaking traditional feed processing systems. Furthermore, in electronic trading, a latency of even one second is unacceptable, and the trading operation whose engine produces the most current results will maximize arbitrage profits. This fact is causing financial services companies to require very high-volume processing of feed data with very low latency.

Similar requirements are present in monitoring computer networks for denial of service and other kinds of security attacks. Real-time fraud detection in diverse areas from financial services networks to cell phone networks exhibits similar characteristics. In time, process control and automation of industrial facilities, ranging from oil refineries to corn flakes factories, will also move to such "firehose" data volumes and sub-second latency requirements.

There is a "sea change" arising from the advances in micro-sensor technologies. Although RFID has gotten the most press recently, there are a variety of other technologies with various price points, capabilities, and footprints (e.g., mote [1] and Lojack [2]). Over time, this sea change will cause everything of material significance to be sensor-tagged to report its location and/or state in real time.

Military has been an early driver and adopter of wireless sensor network technologies. For example, the US Army has been investigating putting vital-signs monitors on all soldiers. In addition, there is already a GPS system in many military vehicles, but it is not connected yet into a closed-loop system. Using this technology, the army would like to monitor the position of all vehicles and determine, in real time, if they are off course.

Other sensor-based monitoring applications will come over time in non-military domains. Tagging will be applied to customers at amusement parks for ride management and prevention of lost children. More sophisticated "easy-pass" systems will allow congestion-based tolling of automobiles on freeways (which was the inspiration behind the Linear Road Benchmark [5]) as well as optimized routing of cars in a metropolitan area. The processing of "firehoses" of real-time data from existing and newly-emerging monitoring applications presents a major stream processing challenge and opportunity.

Traditionally, custom coding has been used to solve high-volume, low-latency streaming processing problems. Even though the "roll your own" approach is universally despised because of its inflexibility, high cost of development and maintenance, and slow response time to new feature requests, application developers had to resort to it as they have not had good luck with traditional off-the-shelf system software.

Recently, several traditional system software technologies, such as main memory DBMSs and rule engines, have been repurposed and remarketed to address this application space. In addition, Stream Processing Engines (e.g., Aurora [8], STREAM [4], TelegraphCQ [9]), a new class of system software, have emerged to specifically support high-volume, low-latency stream processing applications.

In this paper, we describe eight characteristics that a system software must exhibit to excel at a variety of real-time stream processing applications. Our goal is to provide information technologists high-level guidance so that they know what to look for when evaluating their options. Thus, this paper shares a similar goal with earlier papers that present requirements for relational DBSMs [10, 11] and on-line analytical processing [12].

We present these features as a collection of eight rules in the next section. We then review the alternative technologies and summarize how they measure up for real-time stream processing in Section 3. We conclude with final remarks in Section 4.

2. EIGHT RULES FOR STREAM PROCESSING

Rule 1: Keep the Data Moving

To achieve low latency, a system must be able to perform message processing without having a costly storage operation in the critical processing path. A storage operation adds a great deal of unnecessary latency to the process (e.g., committing a database record requires a disk write of a log record). For many stream processing applications, it is neither acceptable nor necessary to require such a time-intensive operation before message processing can occur. Instead, messages should be processed "in-stream" as they fly by. See Figure 1 for an architectural illustration of this *straight-through processing* paradigm.

An additional latency problem exists with systems that are *passive*, as such systems wait to be told what to do by an application before initiating processing. Passive systems require applications to continuously *poll* for conditions of interest. Unfortunately, polling results in additional overhead on the system as well as the application, and additional latency, because (on average) half the polling interval is added to the processing delay. Active systems avoid this overhead by incorporating built-in event/data-driven processing capabilities.

> *The first requirement for a real-time stream processing system is to process messages "in-stream", without any requirement to store them to perform any operation or sequence of operations. Ideally the system should also use an active (i.e., non-polling) processing model.*

Rule 2: Query using SQL on Streams (StreamSQL)

In streaming applications, some querying mechanism must be used to find output events of interest or compute real-time analytics. Historically, for streaming applications, general purpose languages such as C++ or Java have been used as the workhorse development and programming tools. Unfortunately, relying on low-level programming schemes results in long development cycles and high maintenance costs.

In contrast, it is very much desirable to process moving real-time data using a high-level language such as SQL. SQL has remained the most enduring standard database language over three decades. SQL's success at expressing complex data transformations derives from the fact that it is based on a set of very powerful data processing primitives that do filtering, merging, correlation, and aggregation. SQL is explicit about how these primitives interact

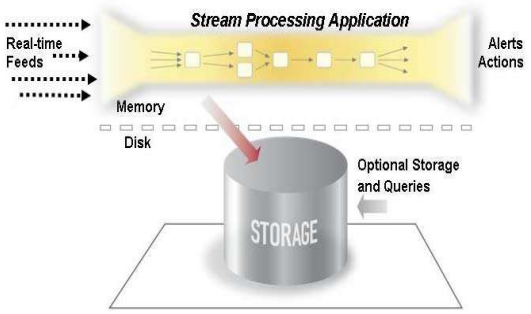

Figure 1: "Straight-through" processing of messages with optional storage.

so that its meaning can be easily understood independently from runtime conditions. Furthermore, SQL is a widely promulgated standard that is understood by hundreds of thousands of database programmers and is implemented by every serious DBMS in commercial use today, due to its combination of functionality, power, and relative ease-of-use. Given that millions of relational database servers running SQL are already installed and operating globally today, it makes good sense to leverage the familiar SQL querying model and operators, and simply extend them to perform processing on continuous data streams.

In order to address the unique requirements of stream processing, StreamSQL, a variant of the SQL language specifically designed to express processing on continuous streams of data, is needed. StreamSQL should extend the semantics of standard SQL (that assumes records in a finite stored dataset) by adding to it *rich* windowing constructs and stream-specific operators.

Although a traditional SQL system knows it is finished computing when it gets to the end of a table, because streaming data never ends, a stream processing engine must be instructed when to finish such an operation and output an answer. The *window* construct serves this purpose by defining the "scope" of a multi-message operator such as an aggregate or a join.

Windows should be definable over time (probably the most common usage case), number of messages, or breakpoints in other attributes in a message. Such windows should be able to *slide* a variable amount from the current window (e.g., a window could be five ticks wide and the next window could slide by one tick from the current one). As a result, depending on the choice of window size and slide parameters, windows can be made disjoint or overlapping. A sliding window example is shown in Figure 2.

Furthermore, new stream-oriented operators that are not present in the standard SQL are needed. An example is a "Merge" operator that multiplexes messages from multiple streams in a manner that

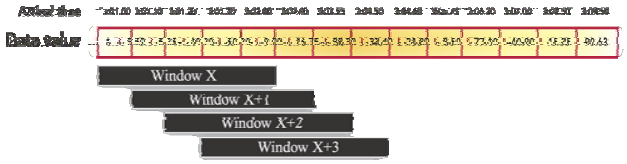

Figure 2: Windows define the scope of operations. The window has a size of 4 messages and slides by 1 each time the associated operator is executed. Consecutive windows overlap.

is sensitive to arrival times and ordering of data messages.
Finally, the operator set must be extensible, so that developers can easily achieve new processing functionality within the system (e.g., to implement a proprietary analysis algorithm on the streaming data).

The second requirement is to support a high-level "StreamSQL" language with built-in extensible stream-oriented primitives and operators.

Rule 3: Handle Stream Imperfections (Delayed, Missing and Out-of-Order Data)

In a conventional database, data is always present before it is queried against, but in a real-time system, since the data is never stored, the infrastructure must make provision for handling data that is late or delayed, missing, or out-of-sequence.

One requirement here is the ability to time out individual calculations or computations. For example, consider a simple real-time business analytic that computes the average price of the last tick for a collection of 25 securities. One need only wait for a tick from each security and then output the average price. However, suppose one of the 25 stocks is thinly traded, and no tick for that symbol will be received for the next 10 minutes. This is an example of a computation that must *block*, waiting for input to complete its calculation. Such input may or may not arrive in a timely fashion. In fact, if the SEC orders a stop to trading in one of the 25 securities, then the calculation will block indefinitely.

In a real-time system, it is *never* a good idea to allow a program to wait indefinitely. Hence, every calculation that can block must be allowed to *time out*, so that the application can continue with partial data. Any real-time processing system must have such time-outs for any potentially blocking operation.

Dealing with out-of-order data introduces similar challenges. Ordinarily, a time window (e.g., [9:00 – 9:01]) would be closed once a message with a timestamp greater than the window's closing time is received. However, such an action assumes that the data arrives in timestamp order, which may not be the case. To deal with out-of-order data, a mechanism must be provided to allow windows to stay open for an additional period of time. One solution specified in Aurora was the notion of *slack* [3].

The third requirement is to have built-in mechanisms to provide resiliency against stream "imperfections", including missing and out-of-order data, which are commonly present in real-world data streams.

Rule 4: Generate Predictable Outcomes

A stream processing system must process time-series messages in a predictable manner to ensure that the results of processing are deterministic and repeatable.

For example, consider two feeds, one containing TICKS data with fields:

TICKS (stock_symbol, volume, price, time),

and the other a SPLITS feed, which indicates when a stock splits, with the format:

SPLITS (symbol, time, split_factor).

A typical stream processing application would be to produce the real-time split-adjusted price for a collection of stocks. The price must be adjusted for the cumulative split_factor that has been seen. The correct answer to this computation can be produced when messages are processed by the system in ascending time order, regardless of when the messages arrive to the system. If a split message is processed out-of-order, then the split-adjusted price for the stock in question will be wrong for one or more ticks. Notice that it is insufficient to simply sort-order messages before they are input to the system—correctness can be guaranteed only if time-ordered, deterministic processing is maintained throughout the entire processing pipeline.

The ability to produce predictable results is also important from the perspective of fault tolerance and recovery, as replaying and reprocessing the same input stream should yield the same outcome regardless of the time of execution.

The fourth requirement is that a stream processing engine must guarantee predictable and repeatable outcomes.

Rule 5: Integrate Stored and Streaming Data

For many stream processing applications, comparing "present" with "past" is a common task. Thus, a stream processing system must also provide for careful management of stored state. For example, in on-line data mining applications (such as detecting credit card or other transactional fraud), identifying whether an activity is "unusual" requires, by definition, gathering the usual activity patterns over time, summarizing them as a "signature", and comparing them to the present activity in real time. To realize this task, both historical and live data need to be integrated within the same application for comparison.

A very popular extension of this requirement comes from firms with electronic trading applications, who want to write a trading algorithm and then test it on historical data to see how it would have performed and to test alternative scenarios. When the algorithm works well on historical data, the customer wants to switch it over to a live feed *seamlessly*; i.e., without modifying the application code. Seamless switching ensures that new errors are not introduced by changes to the program.

Another reason for seamless switching is the desire to compute some sort of business analytic starting from a past point in time (such as starting two hours ago), "catch up" to real time, and then seamlessly continue with the calculation on live data. This capability requires switching automatically from historical to live data, without the manual intervention of a human.

For low-latency streaming data applications, interfacing with a client-server database connection to efficiently store and access persistent state will add excessive latency and overhead to the application. Therefore, state must be stored in the same operating system address space as the application using an embedded database system. Therefore, the scope of a StreamSQL command should be either a live stream or a stored table in the embedded database system.

The fifth requirement is that a stream processing system should have the capability to efficiently store, access, and modify state information, and combine it with live streaming data. For seamless integration, the system should use a uniform language when dealing with either type of data.

Rule 6: Guarantee Data Safety and Availability

To preserve the integrity of mission-critical information and avoid disruptions in real-time processing, a stream processing system must use a high-availability (HA) solution.

High availability is a critical concern for most stream processing applications. For example, virtually all financial services firms expect their applications to stay up all the time, no matter what happens. If a failure occurs, the application needs to failover to

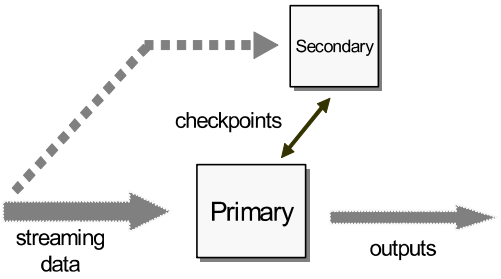

Figure 3: "Tandem-style" hot backup and failover can ensure high availability for real-time stream processing.

backup hardware and keep going. Restarting the operating system and recovering the application from a log incur too much overhead and is thus not acceptable for real-time processing. Hence, a "Tandem-style" hot backup and real-time failover scheme [6], whereby a secondary system frequently synchronizes its processing state with a primary and takes over when primary fails, is the best reasonable alternative for these types of applications. This HA model is shown in Figure 3.

The sixth requirement is to ensure that the applications are up and available, and the integrity of the data maintained at all times, despite failures.

Rule 7: Partition and Scale Applications Automatically

Distributed operation is becoming increasingly important given the favorable price-performance characteristics of low-cost commodity clusters. As such, it should be possible to split an application over multiple machines for scalability (as the volume of input streams or the complexity of processing increases), without the developer having to write low-level code.

Stream processing systems should also support multi-threaded operation to take advantage of modern multi-processor (or multi-core) computer architectures. Even on a single-processor machine, multi-threaded operation should be supported to avoid blocking for external events, thereby facilitating low latency.

Not only must scalability be provided easily over any number of machines, but the resulting application should automatically and transparently load-balance over the available machines, so that the application does not get bogged down by a single overloaded machine.

The seventh requirement is that a stream processing system must be able to distribute its processing across multiple processors and machines to achieve incremental scalability. Ideally, the distribution should be automatic and transparent.

Rule 8: Process and Respond Instantaneously

None of the preceding rules will make any difference alone unless an application can "keep up", i.e., process high-volumes of streaming data with very low latency. In numbers, this means capability to process tens to hundreds of thousands of messages per second with latency in the microsecond to millisecond range on top of COTS hardware.

To achieve such high performance, the system should have a highly-optimized execution path that minimizes the ratio of overhead to useful work. As exemplified by the previous rules, a critical issue here is to minimize the number of "boundary crossings" by integrating all critical functionality (e.g., processing and storage) into a single system process. However, this is not sufficient by itself; all system components need to be designed with high performance in mind.

To make sure that a system can meet this requirement, it is imperative that any user with a high-volume streaming application carefully test any product he might consider for throughput and latency on his target workload.

The eighth requirement is that a stream processing system must have a highly-optimized, minimal-overhead execution engine to deliver real-time response for high-volume applications.

3. SYSTEM SOFTWARE TECHNOLOGIES for STREAM PROCESSING

3.1 Basic Architectures

In addition to custom coding, there are at least three different software system technologies that can potentially be applied to solve high-volume low-latency streaming problems. These are DBMSs, rule engines, and stream processing engines, which we discuss below:

- **Database Management Systems (DBMSs)** are widely used due to their ability to reliably store large data sets and efficiently process human-initiated queries. Main-memory DBMSs can provide higher performance than traditional DBMSs by avoiding going to disk for most operations, given sufficient main memory.

  Figure 4(i) illustrates the basic DBMS architecture. Streaming data is entered into the DBMS directly or through a loading application. A collection of applications can then manipulate DBMS data. A client can use these pre-built applications, often with run-time arguments, and can also

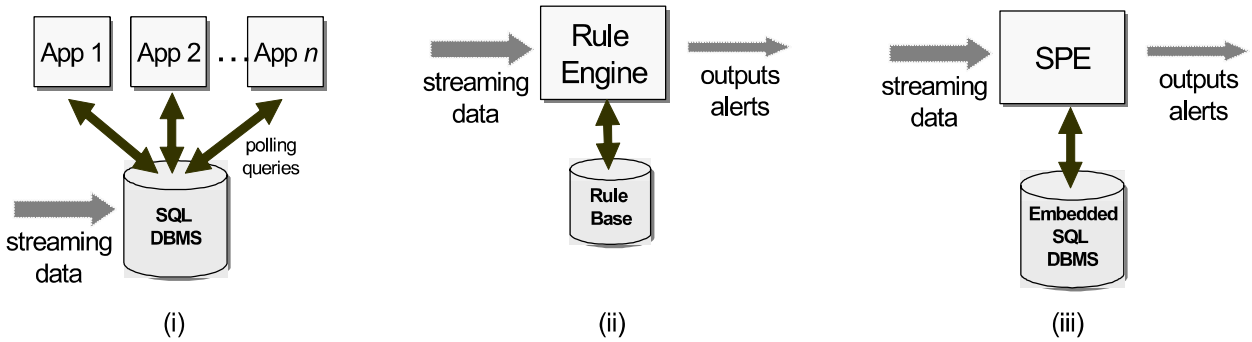

Figure 4: Basic architectures of (i) a database system, (ii) a rule engine, and (iii) a stream processing engine.

code additional ones in a general purpose language such as C++ or Java, using embedded SQL calls to the DBMS.

- **Rule engines** date from the early 1970's when systems such as PLANNER and Conniver were initially proposed by the artificial intelligence community. A later more widespread rule engine was Prolog (1980s), and there have been several more recent examples (e.g., OPS5 [7]). A rule engine typically accepts condition/action pairs, usually expressed using "if-then" notation, watches an input stream for any conditions of interest, and then takes appropriate action. In other words, a rule engine enforces a collection of rules that are stored in a rule base.

  Figure 4(ii) illustrates the basic rule engine model for stream processing. The rule base provides persistent storage for rules. As streaming data enters the system, they are immediately matched against the existing rules. When the condition of a rule is matched, the rule is said to "fire". The corresponding action(s) taken may then produce alerts/outputs to external applications or may simply modify the state of internal variables, which may lead to further rule firings.

- **Stream Processing Engines (SPEs)** are specifically designed to deal with streaming data and have recently gotten attention as a third alternative. Their basic architecture is shown in Figure 4(iii).

  SPEs perform SQL-style processing on the incoming messages as they fly by, without necessarily storing them. Clearly, to store state when necessary, one can use a conventional SQL database embedded in the system for efficiency. SPEs use specialized primitives and constructs (e.g., time-windows) to express stream-oriented processing logic.

Next, we briefly evaluate these systems on the basis of the requirements we presented in Section 2.

3.2 How do they measure up?

DBMSs use a "process-after-store" model, where input data are first stored, potentially indexed, and then get processed. Main-memory DBMSs are faster because they can avoid going to disk for most updates, but otherwise use the same basic model. DBMSs are passive; i.e., they wait to be told what to do by an application. Some have built-in triggering mechanisms, but it is well-known that triggers have poor scalability. On the other hand, rule engines and SPEs are both active and do not require any storage prior to processing. Thus, DBMSs do not **keep the data moving**, whereas rule engines and SPEs do.

SQL was designed to operate on finite-size stored data and thus needs to be extended in order to deal with potentially unbounded streams of time-series data. SQL/Temporal is still in its infancy and, SQL, as implemented by the DBMS vendors, supports only a rudimentary notion of windowed operations (i.e., sort and aggregate). Rule languages need to be extended in a similar manner so that they can express conditions of interest over time. Moreover, rule languages also need the notion of aggregation, a common operation in many streaming applications. Therefore, SPEs support **SQL-style processing on streams**, whereas DBMSs and rule engines do not.

In rule engines and SPEs, it is possible to code operations that might block. Hence, any implementation of these systems should support time-outs. In a DBMS solution, applications have to explicitly specify their polling behavior to simulate the effect of time-outs and receive partial data. On the other hand, a DBMS triggering system has no obvious way to time out. Dealing with out-of-order data exhibits similar challenges for a DBMS. Overall, **handling stream imperfections** is much easier with rules engines and SPEs than with DBMSs.

To **generate predictable outcomes**, an SPE or a rule engine must have a deterministic execution mode that utilizes timestamp order of input messages. DBMSs have particular difficulty with this requirement simply because they are passive—some external system would have to control the order in which messages were stored and processed. In addition, application programs are fundamentally independent and their execution is controlled by an operating system scheduler. Enforcing some order on the execution of application programs is another task that would have to be done by some external software.

Seamlessly **integrating stored and streaming data** is problematic for both DBMSs and rules engines. Storing state is what DBMSs do naturally. As argued earlier, however, DBMSs cannot cope well with streaming data. Even if only used for state storage in a streaming application, a client-server DBMS will be ineffective as it will incur high latency and overhead. As such, a DBMS solution will only be acceptable if the DBMS can be embedded in the application.

In contrast, a rule engine can effectively keep data moving, but has problems when dealing with state storage. The reason is that a rule engine relies on local variables for storage, and there is no easy way to query local variables. To cope with a large amount of

state, one must then somehow graft an embedded DBMS onto a rule engine. In this case, it is necessary to switch from local variables to a DBMS paradigm, an unnatural thing to do. Hence, a rule engine is ill-suited for storing significant amounts of state information. An SPE, on the other hand, should be able to support and seamlessly integrate streaming and stored data by simply switching the scope of a StreamSQL command from a live feed to a stored table.

All three systems can incorporate appropriate mechanisms to **guarantee data safety and availability**. Similarly, there are no fundamental architectural impediments to prevent these systems from **partitioning and scaling applications**.

Finally, all architectures can potentially **process and respond instantaneously**; however, DBMSs are at a big disadvantage here as they do not employ a straight-through processing model.

3.3 Tabular results

In Table 1, we summarize the results of the discussion in this section. Each entry in the table contains one of four values:

- **Yes**: The architecture naturally supports the feature.
- **No**: The architecture does not support the feature.
- **Possible**: The architecture *can* support the feature. One should check with a vendor for compliance.
- **Difficult**: The architecture *can* support the feature, but it is *difficult* due to the non-trivial modifications needed. One should check with the vendor for compliance.

SPEs offer the best capabilities since they are designed and optimized from scratch to address the requirements of stream processing. Both DBMSs and rule engines were originally architected for a different class of applications with different underlying assumptions and requirements. As a result, both systems fundamentally "shoehorn" stream processing into their own model. It is, thus, not surprising to see that they have fundamental limitations for this domain. In particular, neither system has the capability to efficiently and uniformly deal with *both* streaming and stored data.

	DBMS	Rule engine	SPE
Keep the data moving	No	Yes	Yes
SQL on streams	No	No	Yes
Handle stream imperfections	Difficult	Possible	Possible
Predictable outcome	Difficult	Possible	Possible
High availability	Possible	Possible	Possible
Stored and streamed data	No	No	Yes
Distribution and scalability	Possible	Possible	Possible
Instantaneous response	Possible	Possible	Possible

Table 1: The capabilities of various systems software.

4. CONCLUDING REMARKS

There is a large class of existing and newly emerging applications that require sophisticated, real-time processing of high-volume data streams. Although these applications have traditionally been served by "point" solutions through custom coding, system software that specifically target them have also recently started to emerge in the research labs and marketplace.

Based on our experience with a variety of streaming applications, we presented eight rules to characterize the requirements for real-time stream processing. The rules serve to illustrate the necessary features required for any system software that will be used for high-volume low-latency stream processing applications. We also observed that traditional system software fails to meet some of these requirements, justifying the need for and the relative benefits of SPEs.

REFERENCES

[1] Crossbow Technology Inc., 2005. http://www.xbow.com/.

[2] Lojack.com, 2005. http://www.lojack.com/.

[3] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. VLDB Journal, 2003.

[4] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nizhizawa, J. Rosenstein, and J. Widom. STREAM: The Stanford Stream Data Manager. In ACM SIGMOD Conference, June 2003.

[5] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Benchmark for Stream Data Management Systems. In Very Large Data Bases (VLDB) Conference, Toronto, CA, 2004.

[6] J. Barlett, J. Gray, and B. Horst. Fault tolerance in Tandem computer systems. Tandem Computers TR 86.2., 1986.

[7] L. Brownston, R. Farrell, E. Kant, and N. Martin, Programming Expert Systems in OPS5: Addison-Wesley, 1985.

[8] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams: A New Class of Data Management Applications. In proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02), Hong Kong, China, 2002.

[9] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In Proc. of the 1st CIDR Conference, Asilomar, CA, 2003.

[10] E. F. Codd. Does your DBMS run by the rules? ComputerWorld, October 21, 1985.

[11] E. F. Codd. Is your DBMS really relational? Computerworld, October 14, 1985.

[12] E. F. Codd. Providing OLAP to User-Analysts: An IT Mandate. Codd and Associates, Technical Report 1993.

[13] J. P. Corrigan. OPRA Traffic Projections for 2005 and 2006. Technical Report, Options Price Reporting Authority, Aug, 2005. http://www.opradata.com/specs/projections_2005_2006.pdf.