WHEN NOW MEANS RIGHT NOW.

The Eight Rules of Real-Time Stream Processing

Mike Stonebraker, CTO, StreamBase Systems
Ugur Cetintemel, Senior Architect, StreamBase Systems
Stanley Zdonik, Chief Architect, StreamBase Systems

Applications that process real-time streams of data are pushing the limits of traditional data-processing infrastructures. This calls for a new class of systems software—a stream processing engine—whose attributes can be characterized by a core set of eight general rules.

Overview

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 quotes and trades from options exchanges—estimates peak rates of 125,000 messages per second in 2005, with rates doubling every year. This dramatic escalation in feed volumes is stressing or even 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 millisecond-level processing of feed data—or faster—with very low latency.

Until now, off-the-shelf system software to manage streaming data has been largely unavailable for applications to process tens of thousands of messages per second. Previously available approaches for developing stream processing applications include the following:

- **Custom code** has traditionally been the only solution to support high-volume, low-latency streaming applications. However, resorting to a custom-coded solution is far from ideal in most environments because the result is inflexible, costly to develop and maintain, and often difficult to modify in response to new feature requests.

- **Traditional database systems** offer general data management functionality and are designed to handle applications on static data, ranging from Online Transaction Processing (OLTP) to data warehousing. In order to reliably store large, finite data-sets and efficiently process human-initiated queries, the data is stored on disk and indexed before any query processing can take place and outputs produced. (See Figure 1).

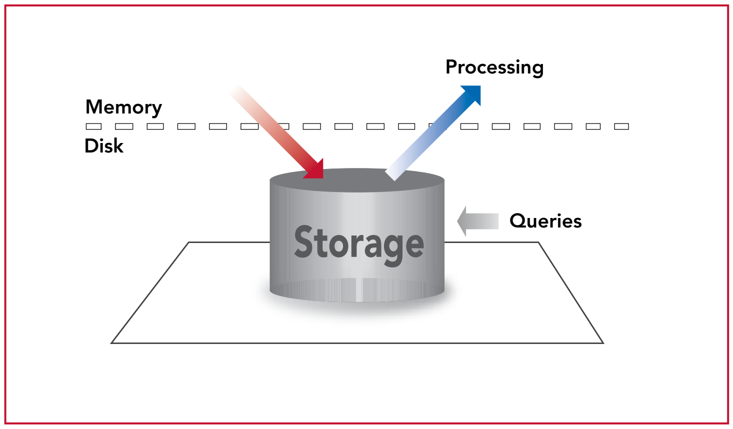

Figure 1: Traditional relational database system stores before querying.

  The storage, indexing, and ad hoc query execution in this approach all introduce latencies that are unacceptable for fast data streams.

- **Main-memory/specialized databases** are higher throughput versions of traditional database systems, as they can avoid going to disk for most operations, given sufficient memory. Despite being faster than traditional Database Management Systems (DBMSs), these systems cannot scale to very high stream rates as they still fundamentally "shoehorn" stream processing into the prevailing processing model inherent in all relational databases.

- **Rules engines** were first proposed by the artificial intelligence (AI) community in the 1970s, with various types marketed in recent years to the financial services community. A rule is typically expressed as a condition/action pair, where the action is enabled whenever its condition is met. When rules are simple, the paradigm works well; however, as the size of a rule set grows (such as tick-by-tick monitoring of multiple securities for specific movement patterns across price, volume, and time), it quickly becomes unmanageable. Since multiple rules can be triggered whenever the data changes, it becomes very difficult to manage the flow of control and execution ordering of a large intertwined set of rules. Moreover, rules engines are not architecturally optimized for low latency and comprehensive state management.

- **Point solutions** typically implement a specific application suitable for solving one specific problem (e.g. algorithmic trading), provided that the application's underlying data model, programming/user interfaces, and analytic capabilities exactly meet the needs of the user. Unfortunately, this often is not the case. Additionally, point solutions rarely meet the demanding real-time processing needs of multiple organizations across the firm beyond one problem area.

Recently, several technologies have emerged—including off-the-shelf stream processing engines—specifically to overcome these limitations and address the challenges of processing high-volume, real-time data without requiring the use of custom code. Within several years, it is expected that these engines will be as ubiquitous for processing real-time data as relational databases are for processing stored data. In order to effectively address the **performance and agility requirements** of stream processing applications, these systems need not only be extremely efficient in real time stream processing, but also shield the users from the inherent complexities of dealing with imperfect data streams and underlying physical resources.

This paper outlines eight characteristics that such stream processing engines should possess to excel at a variety of streaming applications.

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 2 for an architectural example of this processing paradigm.

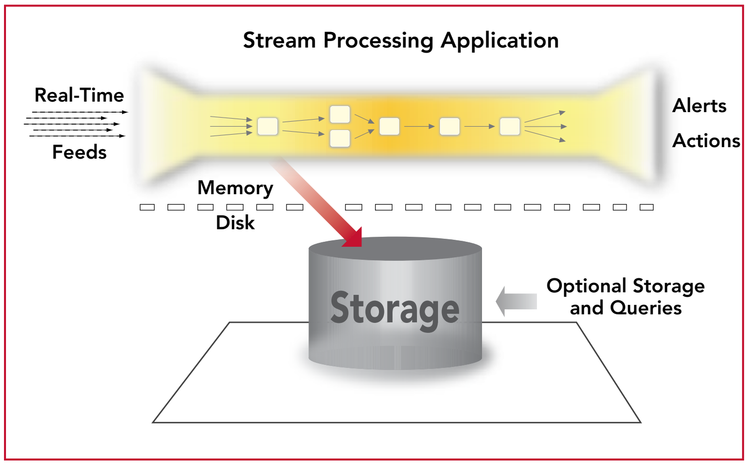

Figure 2: Stream processing engine processes data continuously on-the-fly, with optional storage.

An additional latency problem exists with systems that are **passive**, because 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.

Rule 2

Query Using SQL on Streams (StreamSQL)

In streaming applications, some querying mechanism must be used to issue arbitrary queries on moving data and compute real-time analytics.

Just as the value of SQL has been and remains its ability to issue arbitrary queries against a data store, the same type of querying capability must exist against streams. Historically, for streaming applications, general purpose languages such as C++ or Java have been the workhorse development and programming tools; however, relying on low-level tools such as these languages results in long development cycles and high maintenance costs. Two alternative higher-level approaches exist: a rules language (limitations described above) or SQL for data streams (StreamSQL).

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 so that its meaning can be easily understood independently from run-time 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.

Thus, SQL should be the logical starting point for a stream processing engine. Since standard SQL runs queries on records in a finite stored data set, to deal with continuous event streams and time-based records (tuples), SQL must be extended to become **StreamSQL**. StreamSQL should retain the core capabilities of standard SQL, while adding new ones such as a rich windowing system, the ability to mix stored data with streaming data, and the ability to extend the primitives to include powerful custom logic (such as analytic functions) that are capable of **transforming** the data in arbitrary ways.

As with SQL, StreamSQL must provide **a high level of abstraction**, making the complicated concepts associated with stream processing accessible to the user without the need for low-level programming or deep knowledge about the underlying physical resources or infrastructure. A StreamSQL implementation must also include a well-optimized execution strategy that can provide very low response times even in the presence of high volume data.

Rule 3

Handle Stream Imperfections—Delayed, Missing, and Out-of-Order Data

In a conventional database, data is always present before it is queried, 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 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 Securities & Exchange Commission (SEC) orders a stop to trading in one of the 25 securities, then the calculation will block indefinitely.

In a real-time processing 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.

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 is 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. ACID transactions used by traditional DBMSs are also insufficient, as they cannot guarantee repeatability but can only enforce serializability.

The ability to produce predictable, repeatable 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.

Rule 5

Integrate Stored and Streaming Data

Another requirement for streaming applications is that they are capable of storing and accessing current or historical state information, preferably using a familiar standard such as SQL commands.

Storage of state data is desired almost universally, whether it is yesterday's business analytics or control strategies to apply in a specific trading situation. In addition, for many situations, events of interest depend partly on real-time data and partly on history, as in the following example:

> *"Issue an alert when the volume-weighted average price of IBM shares over the last 10 ticks exceeds the same statistic over the last 50 ticks, as long as this has not happened more than 5 times in the last seven hours of trading."*

A very popular extension of this requirement comes from firms with electronic trading applications, who want to test a trading algorithm on historical data to see how it would have performed, and also test alternative scenarios. When the algorithm works well on historical data, it can be seamlessly switched over to a live feed without application modification.

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 manual intervention.

For low-latency streaming data applications, interfacing with a client-server DBMS connection will add excessive latency and overhead to the application. Therefore, state must be stored in the same operating system address space as the application. As such, a DBMS solution will only satisfy this requirement if the stream processing infrastructure is extended with an embedded DBMS. Hence, the scope of a StreamSQL command must be either a real-time stream, for example TICKS, or a stored table in the embedded DBMS. See Figure 3 for an example of such an architecture. For low latency processing, the application state should be stored and processed **entirely in main memory**.

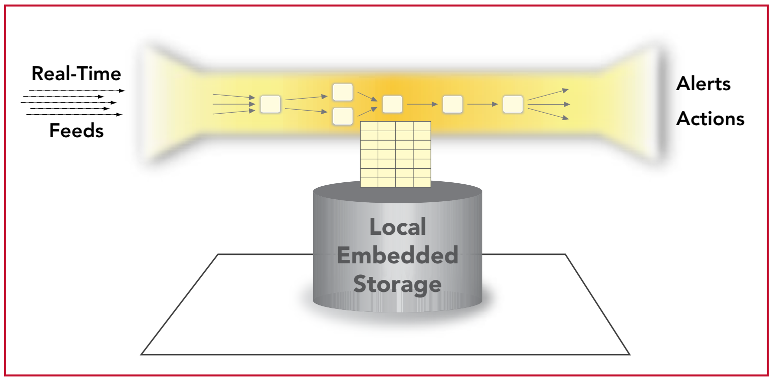

Figure 3: Stream processing engine with in-process, embedded database.

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 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, 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 depicted in Figure 4.

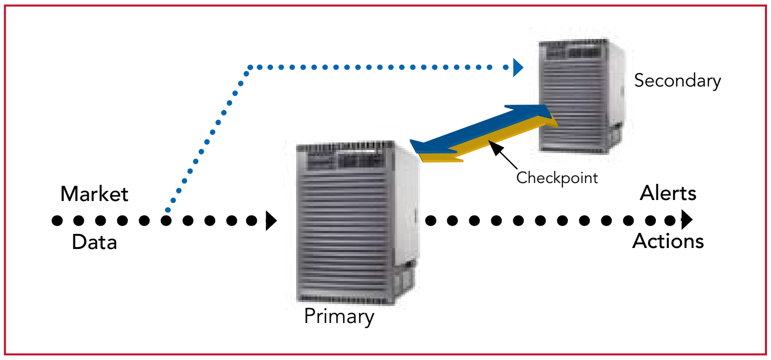

Figure 4: "Tandem-style" hot backup and failover ensures high availability for real-time data processing.

Rule 7

Partition and Scale Applications Automatically

It must be possible to split an application over multiple processors or machines for scalability, without the developer having to write low-level code.

Distributed operation is becoming increasingly important given the favorable price-performance characteristics of low-cost commodity clusters. Stream processing engines must also support multi-threaded operation to take advantage of modern multi-processor (or multi-core) computer architectures. Even on a single-processor machine, multi-threading is crucial 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.

Rule 8

Process and Respond Instantaneously

None of the preceding rules will make any difference alone unless an application using these combined capabilities on one infrastructure 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 commercial off-the-shelf hardware—while also handling stream imperfections, integrating real-time and stored data, scaling over distributed systems, and maintaining fault-tolerant operation.

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.

Summary

This paper has discussed eight rules which characterize the requirements for real-time stream processing. As more and more organizations realize significant benefits of using a stream processing engine vs. custom-coding—such as low latency, business agility, greater developer productivity, and flexibility to quickly capture new opportunities—this technology will become even more widespread as a fundamental systems software infrastructure. Over time, stream processing engines will become as broadly deployed for querying and processing real-time data as relational databases are today for processing stored data. Since a firm's ability to gain competitive advantage is greatly enhanced via adoption of such technologies, IT and application development organizations should consider and evaluate stream processing engines today. These Eight Rules serve to illustrate the necessary features required for any product in this category that will be used for high-volume low-latency applications.

About the Authors

Mike Stonebraker, Ph.D., CTO of StreamBase Systems, has been a pioneer of database research and technology for more than 30 years and is the 2005 recipient of the IEEE's von Neumann Award. He was the main architect of the Ingres DBMS, the object-relational Postgres DBMS, and the federated data system, Mariposa. All three were developed at the University of California at Berkeley where Stonebraker was a professor of computer science for 25 years. Stonebraker also served as founder and CTO of Ingres Corporation, Illustra, and Cohera Corporation as well as the CTO of Informix. In addition to driving the technology vision for StreamBase, Mike is presently adjunct professor of computer science at the Massachusetts Institute of Technology and he serves on the board of directors of a number of emerging technology companies.

Ugur Cetintemel, Ph.D. Senior Architect of StreamBase Systems, received his doctorate in computer science from the University of Maryland, College Park in 2001. He is currently an assistant professor at the department of Computer Science at Brown University. His work focuses on the architecture and performance of advanced information systems and databases. Çetintemel has published numerous papers in leading databases and systems conferences, primarily in the areas of data stream processing, distributed data storage, and replication. He won the prestigious CAREER award from the National Science Foundation in 2004.

Stanley Zdonik, Ph.D., Chief Architect of StreamBase Systems, is also a professor of computer science at Brown University, where he has led the advanced data management research group since 1983. His team has researched object-oriented database systems, semantic query optimization, transaction management, network information systems, data management for mobile systems, data dissemination and stream processing. Zdonik has written more than 90 research papers and, is a member of the board of the Very Large Database (VLDB) Endowment and an editor of several academic journals. He also has been program chair for both the VLDB and the International Conference on Data Engineering (ICDE). Previously, Zdonik worked on an advanced data management system for pharmacologists (called Prophet) at Bolt, Beranek and Newman. He has been a consultant to major corporations including Verizon, Xerox, Digital Equipment Corporation, Object Design, and Computer Corporation of America.

About StreamBase

Backed by top-tier venture capital firms and founded by Dr. Mike Stonebraker, one of the world's foremost database experts, StreamBase Systems is at the vanguard of a sea change in the use, value, processing, and analysis of real-time event streams.

The company has developed a new class of systems software, called a stream processing engine, designed to help organizations meet the performance, agility, and return-on-investment challenges posed by high-volume, high-velocity streaming data applications. The company's StreamBase platform is the first in this new breed of systems software. It offers opportunities for competitive advantage for a number of world-class organizations today, including financial services firms, telecommunications providers, government and military agencies, and other leaders of industry.

More information about StreamBase and its capabilities and areas of expertise can be found at www.streambase.com.

Corporate Headquarters
181 Spring Street
Lexington, MA 02421
1.866.STRMBAS
1.866.787.6227
1.781.761.0800

New York City Office
220 West 42nd Street
20th Floor
New York, NY 10036
1.866.STRMBAS
1.866.787.6227

Reston, VA Office
11921 Freedom Drive
Suite 550
Reston, VA 20190
1.703.608.6958

London Office
107-111 Fleet Street
London EC4A 2AB
United Kingdom
+44 (0) 20 7936 9050