DOI:10.1145/1629175.1629195

Article development led by acmqueue
queue.acm.org

How streaming SQL technology can help solve the Web 2.0 data crunch.

BY JULIAN HYDE

Data in Flight

WEB APPLICATIONS PRODUCE data at colossal rates, and those rates compound every year as the Web becomes more central to our lives. Other data sources such as environmental monitoring and location-based services are a rapidly expanding part of our day-to-day experience. Even as throughput is increasing, users and business owners expect to see their data with ever-decreasing latency. Advances in computer hardware (cheaper memory, cheaper disk, and more processing cores) are helping somewhat, but not enough to keep pace with the twin demands of rising throughput and decreasing latency.

The technologies for powering Web applications must be fairly straightforward for two reasons: first, because it must be possible to evolve a Web application rapidly and then to deploy it at scale with a minimum of hassle; second, because the people writing Web applications are generalists and are not prepared to learn the kind of complex, hard-to-tune technologies used by systems programmers.

The streaming query engine is a new technology that excels in processing rapidly flowing data and producing results with low latency. It arose out of the database research community and therefore shares some of the characteristics that make relational databases popular, but it is most definitely not a database. In a database, the data arrives first and is stored on disk; then users apply queries to the stored data. In a streaming query engine, the queries arrive before the data. The data flows through a number of continuously executing queries, and the transformed data flows out to applications. One might say that a relational database processes data at rest, whereas a streaming query engine processes data in flight.

Tables are the key primitive in a relational database. A table is populated with records, each of which has the same record type, defined by a number of named, strongly typed columns. Records have no inherent ordering. Queries, generally expressed in SQL, retrieve records from one or more tables, transforming them using a small set of powerful relational operators.

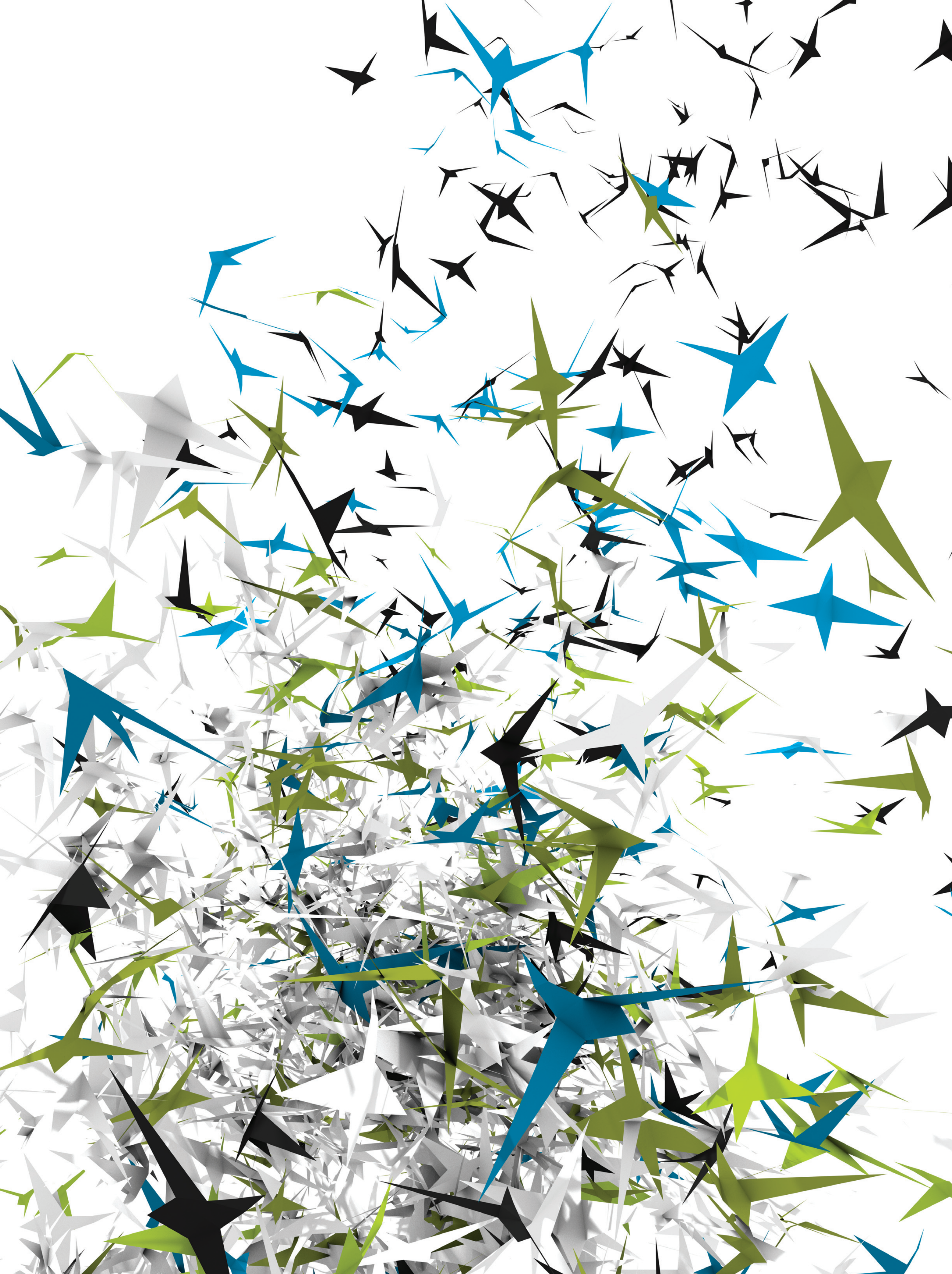
Streams are the corresponding primitive in a streaming query engine. A stream has a record type, just like a table, but records flow through a stream rather than being stored. Records in a streaming system are inherently ordered; in fact, each record has a time stamp that indicates when it was created. The relational operations supported by a relational database have analogues in a streaming system and are sufficiently similar that SQL can be used to write streaming queries.

To illustrate how a streaming query engine can solve problems involving data in flight, consider the following example.

Streaming Queries for Click-Stream Processing

Suppose we want to monitor the most popular pages on a Web site. Each Web server request generates a line to the

ILLUSTRATION BY RYAN ALEXANDER



Web server's log file describing the time, the URI of the page, and the IP address of the requester; and an adapter can continuously parse the log file and populate a stream with records. This query computes the number of requests for each page each minute, as shown in the accompanying table.

The example here is expressed in SQLstream's query language, as are others in this article. The language is standard SQL plus streaming extensions.[4] Other streaming query engines have similar capabilities.

```
SELECT STREAM ROWTIME,
  uri,
  COUNT(*)
FROM PageRequests
GROUP BY
  FLOOR(ROWTIME TO MINUTE),
  uri;
```

The only SQL extensions used in this particular query are the `STREAM` keyword and the `ROWTIME` system column. If you removed the `STREAM` keyword and converted `PageRequests` to a table with a column called `ROWTIME`, you could execute the query in a conventional database such as Oracle or MySQL. That query would analyze all requests that have ever occurred up until the current moment. If `PageRequests` is a stream, however, the `STREAM` keyword tells SQLstream to attach to the `PageRequests` stream and to apply the operation to all future records. Streaming queries run forever.

Every minute this query emits a set of rows, summarizing the traffic for each page during that minute. The output rows time-stamped 10:00:00 summarize all requests between 10:00 and 10:01 (including the 10:00 end point but not including 10:01). Rows in the `PageRequests` stream are sorted by their `ROWTIME` system column, so the 10:00:00 output rows are literally pushed out by the arrival of the first row time-stamped 10:01:00 or later. A streaming query engine tends to process data and deliver results only when new data arrives, so it is said to use push-based processing. This is in contrast to a relational database's pull-based approach where the application must poll repeatedly for new results.

The example in Figure 1 computes URIs for which the number of requests is much higher than normal. First, the `PageRequestsWithCount` view computes the number of requests per hour for each URI over the past hour and averaged over the past 24 hours. Then a query selects URIs for which the rate over the past hour was more than three times the hourly rate over the past 24 hours.

Unlike the previous query that used a `GROUP BY` clause to aggregate many records into one total per time period, this query uses windowed aggregate expressions (*aggregate-function* `OVER` *window*) to add analytic values to each row. Because each row is annotated with its trailing hour's and day's statistics, you need not wait for a batch of rows to be complete. You can use such a query to continuously populate a "Most popular pages" list on your Web site, or an e-commerce site could use it to detect products selling in higher than normal volumes.

Figure 1. Streaming query to find Web pages with higher than normal volume.

```
CREATE VIEW PageRequestsWithCount AS
SELECT STREAM ROWTIME,
    uri,
    COUNT(*) OVER lastHour AS hourlyRate,
    COUNT(*) OVER lastDay / 24 AS hourlyRateL-
astDay
FROM PageRequests
WINDOW lastHour AS (
        PARTITION BY uri
        RANGE INTERVAL '1' HOUR PRECEDING)
    lastDay AS (
        PARTITION BY uri
        RANGE INTERVAL '1' DAY PRECEDING);

SELECT STREAM *
FROM PageRequestsWithCount
WHERE rate > hourlyRateLastDay * 3;
```

Comparing Databases and Streaming Query Engines

A database and a streaming query engine have similar SQL semantics, but if you use the two systems for problems involving data in flight, they behave very differently. Why is a streaming query engine more efficient for such problems? To answer that question, it helps to look at its pedigree.

Some use the term *streaming database*, which misleadingly implies that the system is storing data. That said, streaming query engines have very strong family connections with databases. Streaming query engines have roots in database research, in particular the Stanford STREAMS project,[1] the Aurora project at MIT/Brown/Brandeis,[2] and the Telegraph project at Berkeley.[3] Streaming query engines are based on the relational model that underlies relational databases and, as we shall see, those underpinnings give them power, flexibility, and industry acceptance.

The relational model, first described by E.F. Codd in 1970, is a simple and uniform way of describing the structure of databases. It consists of relations (named collections of records) and a set of simple operators for combining those relations: *select*, *project*, *join*, *aggregate*, and *union*. A relational database naturally enforces data independence, the separation between the logical structure of data and the physical representation. Because the query writer does not know how data is physically organized, a query optimizer is an essential component of a relational database, to choose among the many possible algorithms for a query.

SQL was first brought to market in the late 1970s. Some say it is not theoretically pure (and it has since been extended to encompass nonrelational concepts such as objects and nested tables), but SQL nevertheless embodies the key principles of the relational model. It is declarative, which enables the query to be optimized, so you (or the system) can tune an application without rewriting it. You can therefore defer tuning a new database schema until the application is mostly written, and you can safely refactor an existing database schema. SQL is simple, reliable, and forgiving, and many developers understand it.

Streams introduce a time dimension into the relational model. You can still apply the basic operators (*select, project, join*, and so forth), but you can also ask, "If I executed that *join* query a second ago, and I execute it again now, what would be the difference in the results?"

This allows us to approach problems in a very different way. As an analogy, consider how you would measure the speed of a car traveling along the freeway. You might look out the window for a mile marker, write down the time, and when you reach the next mile marker, divide the distance between the mile markers by the elapsed time. Alternatively, you might use a speedometer, a device where a needle is moved based on a generated current that is proportional to the angular velocity of the car's wheels, which in turn is proportional to the speed of the car. The mile-marker method converts position and time into speed, whereas the speedometer measures speed directly using a set of quantities proportional to speed.

Position and speed are connected quantities; in the language of calculus, speed is the differential of position with respect to time. Similarly, a stream is the time differential of a table. Just as the speedometer is the more appropriate solution to the problem of measuring a car's speed, a streaming query engine is often much more efficient than a relational database for data-processing applications involving rapidly arriving time-dependent data.

Output from query.

ROWTIME	uri	COUNT(*)
10:00:00	/index.html	15
10:00:00	/images/logo.png	19
10:00:00	/orders.html	6
10:01:00	/index.html	20
10:01:00	/images/logo.png	18
10:01:00	/sitemap.html	2
...		

Streaming Advantage

Why is a streaming query engine more efficient than a relational database for data-in-flight problems?

First, the systems express the problems in very different ways. A database stores data and applications fire queries (and transactions) at the data. A streaming query engine stores queries, and the outside world fires data at the queries. There are no transactions as such, just data flowing through the system.

The database needs to load and index the data, run the query on the whole dataset, and subtract previous results. A streaming query system processes only new data. It holds only the data that it needs (for example, the latest minute), and since that usually fits into memory easily, no disk I/O is necessary.

A relational database operates under the assumption that all data is equally important, but in a business application, what happened a minute ago is often more important than what happened yesterday, and much more important than what happened a year ago. As the database grows, it needs to spread the large dataset across disk and create indexes so that all of the data can be accessed in constant time. A streaming query engine's working sets are smaller and can be held in memory; and because the queries contain window specifications and are created before the data arrives, the streaming query engine does not have to guess which data to store.

A streaming query engine has other inherent advantages for data in flight: reduced concurrency control overhead and efficiencies from processing data asynchronously. Since a database is writing to data structures that other applications can read and write, it needs mechanisms for concurrency control; in a streaming query engine there is no contention for locks, because incoming data from all applications is placed on a queue and processed when the streaming query engine is ready for it.

In other words, the streaming query engine processes data asynchronously. Asynchronous processing is a feature of many high-performance server applications, from transaction processing to email processing, as well as Web crawling and indexing. It allows a system to vary its unit of work—from a record at a time when the system is lightly loaded to batches of many rows when the load is heavier—to achieve efficiency benefits such as locality-of-reference. One might think an asynchronous system has a slower response time, because it processes the data "when it feels like it," but an asynchronous system can achieve a given throughput at much lower system load, and therefore have a better response time than a synchronous system. Not only is a relational database synchronous, but it also tends to force the rest of the application into a record-at-a-time mode.

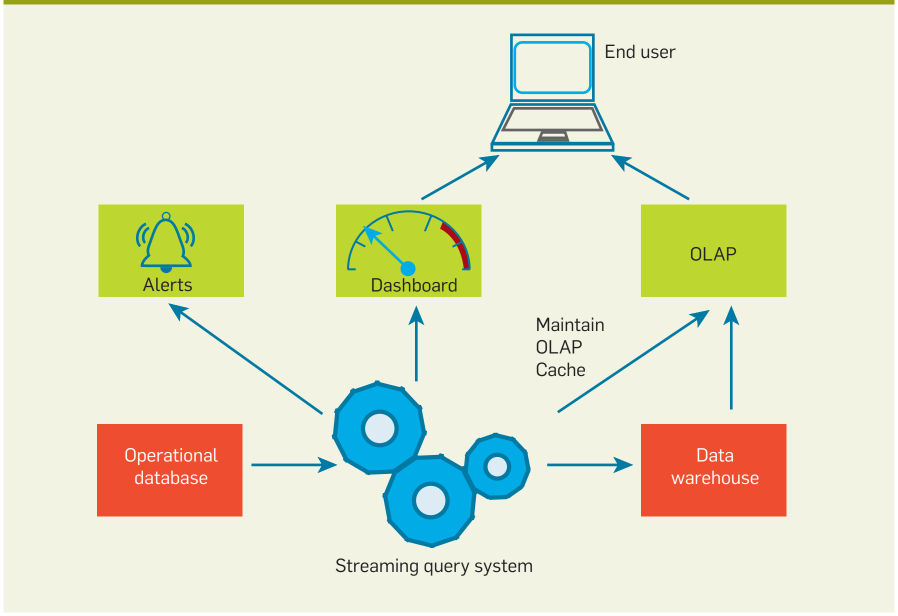
It should be clear by now that push-based processing is more efficient for data in flight; however, a streaming query engine is not the only way to achieve it. Streaming SQL does not make anything possible that was previously impossible. For example, you could implement many problems using a message bus, messages encoded in XML, and a procedural language to take messages off the bus, transform them, and put them back onto the bus. You would, however, encounter problems of performance (parsing XML is expensive), scalability (how to split a problem into sub-problems that can be handled by separate threads or machines), algorithms (how to combine two streams efficiently, correlate two streams on a common key, or aggregate a stream), and configuration (how to inform all of the components of the system if one of the rules has changed). Most modern applications choose to use a relational database management system to avoid dealing with data files directly, and the reasons to use a streaming query system are very similar.

Other Applications of Streaming Query Systems

Just as relational databases are a horizontal technology, used for everything from serving Web pages to transaction processing and data warehousing, streaming SQL systems are being applied to a variety of problems.

Application areas include complex event processing (CEP), monitoring, population data warehouses, and middleware. A CEP query looks for sequences of events on a single stream or on multiple streams that, together, match a pattern and create a "complex event" of interest to the business. Applications of CEP include fraud detection and electronic trading.

Figure 2. Continuous ETL using a streaming query system.



CEP has been used within the industry as a blanket term to describe the entire field of streaming query systems. This is regrettable because it has resulted in a religious war between SQL-based and non-SQL-based vendors and, in overly focusing on financial services applications, has caused other application areas to be neglected.

The click-stream queries here are a simple example of a monitoring application. Such an application looks for trends in the transactions that represent the running business and alerts the operations staff if things are not running smoothly. A monitoring query finds insights by aggregating large numbers of records and looking for trends, in contrast to a CEP query that looks for patterns among individual events. Monitoring applications may also populate real-time dashboards, a business's equivalent of your car's speedometer, thermometer, and oil pressure gauge.

Because of their common SQL language, streaming queries have a natural synergy with data warehouses. The data warehouse holds the large amount of historical data necessary for a "rear-view mirror" analysis of the business, while the streaming query system continuously populates the data warehouse and provides forward-looking insight to "steer the company."

The streaming query system performs the same function as an ETL (extract, transform, load) tool but operates continuously. A conventional ETL process is a sequence of steps invoked as a batch job. The cycle time of the ETL process limits how current the data warehouse is, and it is difficult to get that cycle time below a few minutes. For example, the most data-intensive steps are performed by issuing queries on the data warehouse: looking up existing values in a dimension table, such as customers who have made a previous purchase, and populating summary tables. A streaming query system can cache the information required to perform these steps, offloading the data warehouse, whereas the ETL process is too short-lived to benefit from caching.

Figure 2 shows the architecture of a real-time business intelligence system. In addition to performing continuous ETL, the streaming query system populates a dashboard of business metrics, generates alerts if metrics fall outside acceptable bounds, and proactively maintains the cache of an OLAP (online analytical processing) server that is based upon the data warehouse.

Today, much "data in flight" is transmitted by message-oriented middleware. Like middleware, streaming query systems can deliver messages reliably, and with high throughput and low latency; further, they can apply SQL operations to route, combine, and transform messages in flight. As streaming query systems mature, we may see them stepping into the role of middleware and blurring the boundaries between messaging, continuous ETL, and database technologies by applying SQL throughout.

Conclusion

Streaming query engines are based on the same technology as relational databases but are designed to process data in flight. Streaming query engines can solve some common problems much more efficiently than databases because they match the time-based nature of the problems, they retain only the working set of data needed to solve the problem, and they process data asynchronously and continuously.

Because of their shared SQL language, streaming query engines and relational databases can collaborate to solve problems in monitoring and real-time business intelligence. SQL makes them accessible to a large pool of people with SQL expertise.

Just as databases can be applied to a wide range of problems, from transaction processing to data warehousing, streaming query systems can support patterns such as enterprise messaging, complex event processing, continuous data integration, and new application areas that are still being discovered.

Related articles on queue.acm.org

A Call to Arms
Jim Gray and Mark Compton
http://queue.acm.org/detail.cfm?id=1059805

Beyond Relational Databases
Margo Seltzer
http://queue.acm.org/detail.cfm?id=1059807

A Conversation with Michael Stonebraker and Margo Seltzer
http://queue.acm.org/detail.cfm?id=1255430

References
1. Arasu, A., Babu, S., Widom, J. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical Report. Stanford University, Stanford, CA, 2003.
2. Aurora project; http://www.cs.brown.edu/research/aurora.
3. Chandrasekaran, S., et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of Conference on Innovative Data Systems Research* (2003).
4. SQLstream Inc.; http://www.sqlstream.com.

Julian Hyde is chief architect of SQLstream, a streaming query engine. He is also the lead developer of Mondrian, the most popular open source relational OLAP engine and a part of the Pentaho open source BI suite. An expert on relational technology, including query optimization and streaming execution, Hyde introduced bitmap indexes into Oracle and led development of the Broadbase analytic DBMS.