STREAM: The Stanford Stream Data Manager
(Demonstration Proposal)

Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar,
Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom*

Stanford University
http://www-db.stanford.edu/stream

1 Introduction

We propose to demonstrate a *Data Stream Management System* (*DSMS*) called *STREAM*, for *STanford stREam datA Manager*. The challenges in building a DSMS instead of a traditional DBMS arise from two fundamental differences:

- In addition to managing traditional stored data such as relations, a DSMS must handle multiple continuous, unbounded, possibly rapid and time-varying *data streams*.
- Due to the continuous nature of the data, a DSMS typically supports long-running *continuous queries*, which are expected to produce answers in a continuous and timely fashion.

STREAM is a general-purpose DSMS that supports a declarative query language and is designed to cope with high data rates and large numbers of continuous queries. A description of our current (Fall 2002) language design, algorithms, system design, and system implementation efforts can be found in [MW+03]. Needless to say, we expect additional functionality to be in place by Spring 2003. Our proposed demonstration will focus on two aspects of our system:

- Registering and observing multiple continuous queries over multiple continuous data streams. Queries can be registered using our declarative query language *CQL* (Section 3.1), or by entering query plans directly (Section 3.2).
- An interactive interface for online visualization and management of query execution and resource utilization, and for easy experimentation with DSMS query processing techniques (Section 3.3).

2 System Architecture

A simplified view of a Data Stream Management System is shown in Figure 1(a). On the left are the incoming *Data Streams*, which produce data indefinitely and drive query processing. Although we are concerned primarily with the online processing of continuous queries, in many applications stream data also may be copied to an *Archive*, for preservation and possible offline processing of expensive analysis or mining queries. Finally, processing of continuous queries typically requires intermediate state, which we denote as *Scratch Store* in the figure. This state could be stored and accessed in memory or on disk, although in STREAM we currently use memory only.

Across the top of the figure we see that users or applications register *Continuous Queries*, which remain active in the system until they are explicitly deregistered. Results of continuous queries are generally transmitted as output data streams, but they could also be relational results that are updated over time (similar to materialized views).

Currently STREAM offers a Web system interface through direct HTTP, and we are planning to expose the system as a Web service through SOAP. Thus, remote applications can be written in any language and on any platform. They can register queries and receive the results of a query as a streaming HTTP response in XML. For human users, we have developed a Web-based GUI exposing the same functionality, as well as providing an interactive interface for visualizing and modifying system behavior (see Section 3.3).

2.1 Query Plans

When a continuous query is registered using our declarative query language CQL it is compiled into a *query plan*. Currently a separate query plan is generated for each query, although we expect to develop plan merging algorithms,

*Contact author: widom@stanford.edu

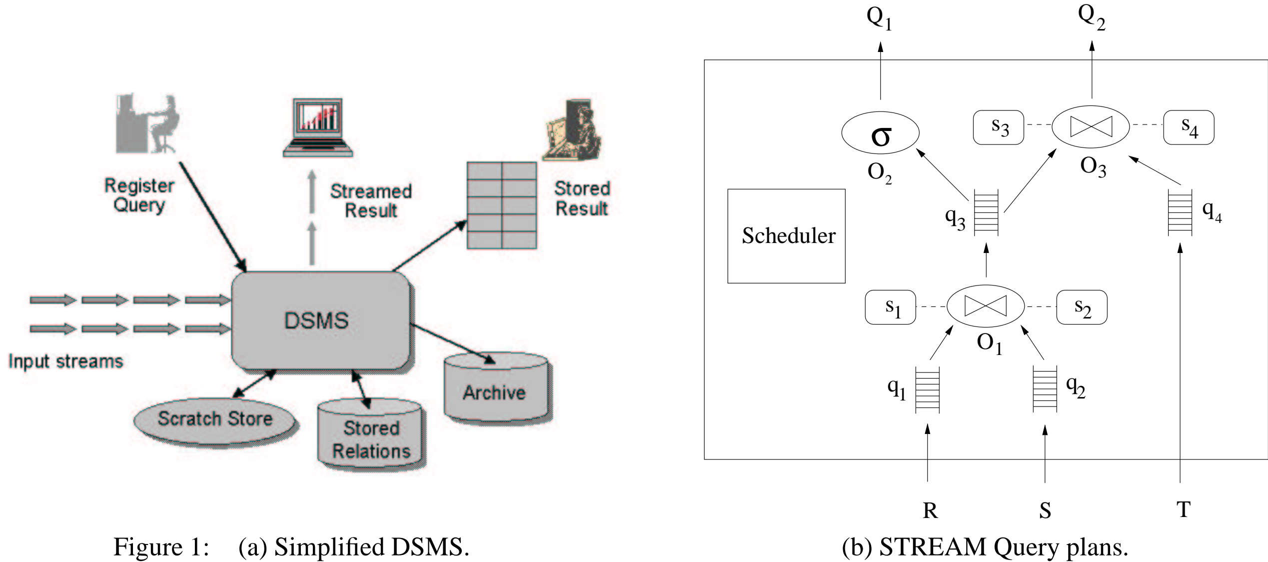

Figure 1: (a) Simplified DSMS. (b) STREAM Query plans.

especially in cases when approximate query results are tolerated. Alternatively, query plans can be entered directly, and plans can be merged "by hand." A query plan in our system runs continuously and is composed of three different types of components:

- Query *operators*, similar to those in a traditional DBMS. Each operator reads a stream of tuples from a set of input queues, processes the tuples based on its semantics, and writes its output tuples into a single output queue.
- Inter-operator *queues*, also similar to the approach taken by some traditional DBMS's. Queues connect different operators and define the paths along which tuples flow as they are being processed.
- *Synopses*, used to maintain state associated with operators and discussed in more detail next.

Operators implemented so far include the standard relational operators (including aggregation), *windowed* versions of some operators (e.g., join, duplicate elimination), and *sampling* operators. Note that the queues and synopses in the query plans active in the system comprise the *Scratch Store* depicted in Figure 1(a).

A synopsis summarizes the tuples seen so far at some intermediate operator in a running query plan, as needed for future evaluation of that operator. For example, for full precision a join operator must remember all the tuples it has seen so far on each of its input streams, so it maintains one synopsis for each (similar to a symmetric hash join). On the other hand, simple filter operators, such as selection and duplicate-preserving projection, do not require a synopsis since they need not maintain state. So far in our system we have focused primarily on *sliding-window* synopses, although we also have implemented *reservoir sample* synopses and will soon add *Bloom filters* [MW+03].

Figure 1(b) illustrates plans for two queries, Q_1 and Q_2. Together the plans contain three operators O_1–O_3, four synopses s_1–s_4 (two per join operator), and four queues q_1–q_4. Query Q_1 is a selection over a join of two streams R and S. Query Q_2 is a join of three streams, R, S, and T. The two plans share a subplan joining streams R and S by sharing its output queue q_3. Execution of query operators is controlled by a global *scheduler*. The scheduler currently uses a simple round-robin scheme for scheduling operators that are ready to execute. We plan to implement more sophisticated scheduling schemes shortly.

2.2 Plan Implementation

In the implementation of our system, operators, queues, and synopses all are subclasses of a generic `Entity` class. Each entity has a table of attribute-value pairs called its *Control Table* (CT for short), and each entity exports an interface to query and update its CT. (CT contents can also be updated by the entity itself.) The CT serves two purposes in our system so far. First, some CT attributes are used to dynamically control the behavior of an entity. For example, the amount of memory used by a synopsis S can be controlled by updating the value of attribute `Memory` in S's control table. Second, some CT attributes are used to collect statistics about entity behavior. For example, the

number of tuples that have passed through a queue q is stored in attribute `Count` of q's control table. The CT approach has been instrumental in implementing the management interface described in Section 3.3.

3 Demonstration Content

Our demonstration will consist of three parts:

(1) Basic stream registration, specification of continuous queries in CQL, and observation of streamed query results.

(2) Direct entry of query plans through the graphical interface and using XML.

(3) Online visualization and management of query execution and resource allocation through the graphical interface.

3.1 The CQL Language

The declarative query interface to STREAM uses a language we developed called *CQL*, for *Continuous Query Language*. Syntactically, CQL is a superset of SQL (although many esoteric or complex SQL constructs are not yet implemented), with constructs added for *sliding windows* and for *sampling*.[1] However, the semantics of continuous queries over streams (and relations) turns out to be fairly subtle, and some subtleties of CQL semantics will be illustrated in the demonstration. Several example CQL queries are provided in [MW+03].

3.2 Direct Entry of Query Plans

STREAM provides a graphical interface for users to create query plans and enter them directly into the system, bypassing the declarative CQL interface. The query plan construction interface was created initially so that system developers could experiment with the query processing engine. The same interface is useful for a "power user" who wants to override the plan generator. The interface also can be used to create merged plans or to merge components of separately generated plans.

Since we want continuous queries in a DSMS to persist until explicitly deregistered, our main-memory plan structures (as in Figure 1(b)) are mirrored in XML files. We expose this feature to users so they can create, edit, and transfer XML plans offline and enter them directly into the system. The XML plan interface offers our third (and most primitive) method of registering continuous queries.

3.3 Plan Execution and Management Interface

We are developing a comprehensive interactive interface for STREAM users, system administrators, and system developers to visualize and modify query plans as well as query-specific and system-wide resource allocation while the system is in operation. Note that the rapid implementation of this interface is enabled by our generic CT structure described in Section 2.2.

3.3.1 Query Plan Execution

STREAM provides a graphical interface to visualize the execution of any registered continuous query. Recall that query plans are implemented as networks of *entities* (Section 2.2), each of which is an operator, a queue, or a synopsis. The query plan execution visualizer provides the following features.

(1) View the structure of the plan and its component entities.

(2) View specific attributes of an entity, e.g., the amount of memory being used by a synopsis in the plan.

(3) View data moving through the plan, e.g., tuples entering and leaving inter-operator queues, and synopsis contents growing and shrinking as operators execute. Depending on the scope of activity individual tuples or tuple counts can be visualized.

[1] Actually our windowing construct is derived from SQL-99, and our sampling construct is similar to a recently proposed standard, so syntactically there is little new in our language.

3.3.2 Global System Behavior

The query execution visualizer described in the previous section is useful for visualizing the execution and resource utilization of a single query or a small number of queries that may share plan components. However, a system administrator or developer might want to obtain a more global picture of DSMS behavior. STREAM provides an interface to visualize system-wide query execution and resource utilization information. The supported features include:

(1) View the entire set of query plans in the system, with the level of detail dependent on the number and size of plans.

(2) View the fraction of memory used by each query in the system, or in more detail by each queue and each synopsis.

(3) View the fraction of processor time consumed by each query in the system.

3.3.3 Controlling System Behavior

Visualizing query-specific and system-wide execution and resource allocation information is important for system administrators and developers to understand and tune the performance of a DSMS running long-lived continuous queries. A sophisticated DSMS should adapt automatically to changing stream characteristics and changing query load, but it is still useful for "power users" and certainly useful for system developers to have the capability to control certain aspects of system behavior. We support the following features:

(1) Run-time modification of memory allocation, e.g., increasing the memory allocated to one synopsis while decreasing memory for another.

(2) Run-time modification of plan structure, e.g., changing the order of synopsis joins in a query over multiple streams, or changing the type of synopsis used by a join operator.

(3) Run-time modification of the scheduling policy, choosing several alternative policies.

4 Demonstration Application Domain

Much of our work has been driven by the *network monitoring* domain, for which a DSMS is particularly well-suited [BSW01]. We are likely to use this domain for at least a portion of our demonstration, although we plan to explore a number of alternate application domains in the coming months. One unusual application we are investigating is *system reflection*. Since our CT interface (Section 2.2) can generate many continuous streams of many different internal STREAM system measurements, we are exploring whether we can use the STREAM query processor itself to support components of the monitoring interface described in Section 3.3, or even to automatically adapt STREAM system behavior. If successful, system reflection will form a component of our demonstration.

5 Conclusion and Acknowledgments

Please visit `http://www-db.stanford.edu/stream` for continuously updated information. We have benefited greatly from discussions with our non-coding participants in the STREAM project and from interactions with the wider data streams research community.

References

[BSW01] S. Babu, L. Subramanian, and J. Widom. A data stream management system for network traffic management. In *Proceedings of the Workshop on Network-Related Data Management*, Santa Barbara, California, May 2001.

[MW+03] R. Motwani, J. Widom, et al. Query processing, approximation, and resource management in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, Monterey, California, January 2003. To appear.