2002

A Stream Database Server for Sensor Applications

Moustafa A. Hammad

Walid G. Aref
Purdue University, aref@cs.purdue.edu

Ann C. Catlin

Mohamed G. Elfeky

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

Report Number:
02-009

A STREAM DATABASE SERVER FOR SENSOR APPLICATIONS

Moustafa A. Hammad
Walid G. Aref
Ann C. Catlin
Mohamed G. Elfeky
Ahmed K. Elmagarmid

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #02-009
May 2002

A Stream Database Server for Sensor Applications

Moustafa A. Hammad, Walid G. Aref, Ann C. Catlin,
Mohamed G. Elfeky and Ahmed K. Elmagarmid

Abstract

We present a framework for stream data processing that incorporates a stream database server as a fundamental component. The server operates as the stream control interface between arrays of distributed data stream sources and end-user clients that access and analyze the streams. The underlying framework provides novel stream management and query processing mechanisms to support the online acquisition, management, storage, non-blocking query, and integration of data streams for distributed multi-sensor networks. In this paper, we define our stream model and stream representation for the stream database, and we describe the functionality and implementation of key components of the stream processing framework, including the query processing interface for source streams, the stream manager, the stream buffer manager, non-blocking query execution, and a new class of join algorithms for joining multiple data streams constrained by a sliding time window. We conduct experiments using real data streams to evaluate the performance of the new algorithms against traditional stream join algorithms. The experiments show significant performance improvements and also demonstrate the flexibility of our system in handling data streams. A multi-sensor network application for the intelligent detection of hazardous materials is presented to illustrate the capabilities of our framework.

Index terms: Multi-sensor processing framework, Stream database, Stream manager, Stream query processing, Stream scan, Window-join.

1. Introduction

The widespread use of sensing devices that generate digital data streams and the enormous value of the information that can be extracted from them have led to an explosion of research in the development and application of sensor data stream processing systems. Applications that process streams have provided great insights into many physical systems, however sensor application development is complicated by the continued reexamination of basic components, such as stream management and stream processing, during the design and implementation of each application. An advanced sensor-processing framework simplifies application development by providing powerful components for the acquisition, query, analysis, and integration of streams of data. Our framework for stream data processing incorporates a stream database management server as an integral and fundamental component. The server operates as the stream control interface between arrays of distributed data stream sources and the end-user clients that access and analyze the streams. It provides the underlying database technologies for online data stream management with real-time constraints, online and long-running stream query processing, and stream query

operators for stream analysis and data mining. The stream manager has well-defined interfaces for integrating stream pre- and post-processing units, and the query processor supports the integration of application-specific modules to aid in the delivery of decision support based on stream queries. A high-level view of the *STEAM* framework is shown in Figure 1.

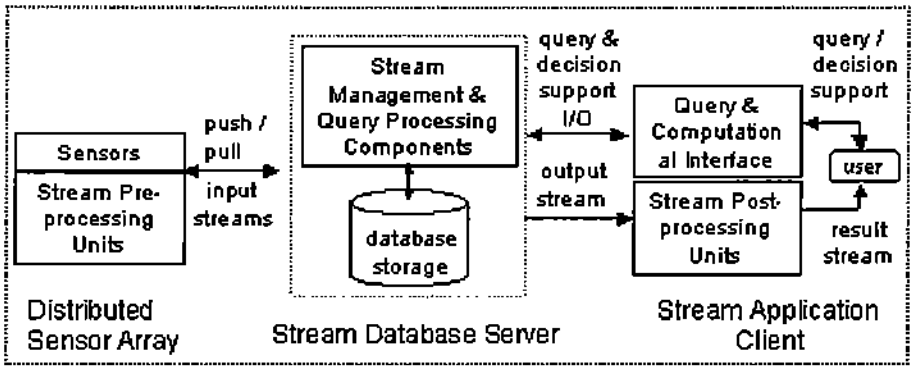

Figure 1. High-level view of the three elements of the STEAM framework. The stream database server provides stream management and query processing for a network of sensors, and interacts with clients to admit user queries and return results for decision support.

The Purdue University *Boilermaker STEAM* framework introduces the advanced database technology required for managing and processing online stream data. In this paper, we describe the functionality, design, and implementation of key components of the *STEAM* system. In Section 3, we establish our stream model, steam representation, stream data type, and the *STEAM* system architecture. In Section 4, we give a detailed description of the query processing interface for source streams. The stream manager is described in Section 5, and we include a description of our mechanism for calling individual streams. We also discuss stream buffer management in Section 5, and describe the *STEAM* mechanisms for handling multiple streams and for sharing streams between multiple queries. Query execution is discussed in Section 6. We examine the scheduling of query operators in the query plan and give a detailed description of a class of join algorithms, the window-join, for joining multiple data streams. To motivate and illustrate the capabilities of the *STEAM* database server, we describe a multi-sensor network application implemented within the *STEAM* framework in Section 7. In the application, data streams are generated from a sensor network of chemical and biological detectors that progressively collect and stream multi-dimensional data to the *STEAM* database server. Online stream data mining algorithms aid in determining whether hazardous materials are present.

The *STEAM* project research and development is based on experience and insight gained through our ongoing research initiatives for advancing video database technology, which has produced a video stream database management system [6,7] offering comprehensive and efficient database management for the real-time query, analysis, retrieval and streaming of video data. Our fundamental concept was to provide a full range of functionality for the *video stream* as a fundamental database object. Research and development efforts for this system have produced some of the most advanced techniques and models [17,18,19,29] currently available in streaming video database management, and have provided the foundation for *STEAM* research.

The key contributions of the *STEAM* project are the following:

- A stream processing framework with a powerful stream database server that provides advanced stream management and query processing capabilities to support the acquisition, management, storage, online query, online analysis, and integration of data streams for distributed multi-sensor networks.

- A class of algorithms for joining multiple data streams which addresses the infinite nature of streams by joining stream data items that lie within a sliding window over time. The algorithms are non-blocking and can easily be implemented in the pipeline query plan.

- Application development based on the stream database framework. Our system for the intelligent detection of hazardous materials includes extensions to the underlying query processing component to support online data mining and analysis techniques.

2. Related Work

Many ongoing research projects address sensor and stream query processing, and their methods for handling the processing of multiple continuous streams share many characteristics. This section presents several research projects which have developed or are currently developing systems for data stream processing.

The COUGAR [11] system focuses on executing queries over both sensor and stored data. Sensors are represented as a new data type, with special functions to extract sensor data when requested. The system addresses scalability (increasing numbers of sensors) by introducing a virtual table where each row represents a specific sensor. The table handles the asynchronous behavior of the sensor as well as the return of multiple values for a single sensor request. The COUGAR system inspired many of the implementation issues addressed by *STEAM*. The *STEAM* database server is built on top of PREDATOR [35] and Shore [37], which is similar to the COUGAR system implementation.

The STREAM [9] project at Stanford addresses new demands imposed on data management and processing techniques by data streams. They suggest a query execution mechanism based on a separate scheduler to independently schedule the operators, which are connected by queues. The work on STREAM also addresses the processing of query operators using a limited memory space [2], suggesting that some queries over data streams (e.g., projection with duplicate elimination operators) may be answered using limited memory by considering the relationship between the terms in the where clause. The Fjord project [32] proposes a framework for query execution plans involving both sensor and stored data. Operators are represented as modules that are connected to each other through push or pull queues. If the operator receives data from a push queue, a specialized scheduler repeatedly schedules the operator; otherwise the pull queue invokes the source operator.

The Telegraph project and the work on eddy [1] introduce a data flow system where the order of executing the query operators can be changed during query execution. Their recent work [33] addresses the adaptation of eddies to run queries over data streams and share the status of the query operators between concurrent continuous queries. They suggest a multi-way join for

joining multiple data streams over a window interval, SteM. The SteMs are unary operators that are probed with new tuples for a match. Their methods for updating join buffers and verifying window constraints are not discussed. Continuous queries are also addressed in the context of the Niagara project [13], which addresses group optimization over continuous and long running queries. Recent work [42] suggests a rate-based optimization strategy to select the best plan to output tuples more quickly. Tribeca [38] is a specialized query processing system designed to support network traffic analysis. The system mainly focuses on query processing over streams of network traffic, either online or off-line.

Various research efforts have studied the algorithmic complexity of computations over streams [27] and the computation of correlated averages over streams using fixed or sliding windows [22]. In [23] the authors propose the use of wavelet transformation methods to provide small space representations of streams for answering aggregate queries. Recent work of Datar et al. [14] introduces an approximate algorithm to compute count and sum within a sliding window defined over the count of arriving data items. Praveen et al. [36] provide the SEQ model and implementation for a sequence database. The sequence is defined as a set with a mapping function defined to an ordered domain. The work in [28] provides a data model for chronicles (sequences) of data items and discusses the complexity of executing a view described by the relational algebra operators. The band join [15] technique addresses the problem of joining two relations of fixed size for values within a "band" of each other. The band-join addresses the same problem as joining streams within a window of time, however the suggested solution is based on stored relations (partitioning), which is not applicable for streams. Index and partition-based algorithms are presented in [31,44] for temporal-joins over finite relations.

In the *STEAM* system, we address the sharing of input data streams by multiple concurrent queries at a level below query execution. We suggest an efficient and simple scheduling mechanism that allows non-blocking query execution, and we introduce a stream manager to interface query requests with the retrieval of data from source streams. None of the systems described above address the handling and representation of source data streams in this manner. We also introduce a novel multi-way stream window join that provides an efficient online approach for verifying window constraints and updating the join buffers during execution. We plan to address group query optimization and sharing of execution states between concurrent queries.

3. An Advanced Stream Database Server

The nature of stream data, whether processed by the database server to answer queries or delivered to the client from the database server, requires the extension of underlying database technology to support real-time processing for data streams [8,9,11,13,27]. We address the research issues involved in the development of *STEAM* functionality by first establishing the definition and model of the data stream on a suitable level of database abstraction and then defining the representation of stream characteristics within *STEAM*.

3.1 The Stream Model

We consider a stream to be an infinite sequence of data items, where items are appended to the sequence over time and items in the sequence are ordered by a time-stamp. Accordingly, we model each stream data item as a tuple $< v, t >$ where v is a value (or set of values) representing

the data item content, and t is the time at which this item joined the stream. A sensor identifier is used to retrieve sensor-specific information from the *STEAM* database storage. The data content v can be a single value, a vector of values or NULL, and each value can be a simple or composite data type. Time t is our ordering mechanism, and the time stamp is the sequence number implicitly attached to each new data item. The time stamp may identify either the valid time or the transaction time, where valid time refers to the time assigned to the item at its source, and transaction time refers to the time assigned to the data item at the query processor. A sensor is any data stream source that is capable of producing infinite streams of data, either continuously or asynchronously.

3.2 The STEAM System Architecture

The *STEAM* stream-processing framework is shown in Figure 2. The source of stream data is a distributed array of sensors, each of which provides infinite streams of raw data. The pre-processing units receive raw streams from the sensors and prepare them for database processing operations. The functionality of a preprocessing unit is application dependent; it may prepare raw video streams by extracting image-based feature information, network traffic streams by extracting packet headers, etc. These units may also perform other functions such as filtering stream content and projecting portions of the stream.

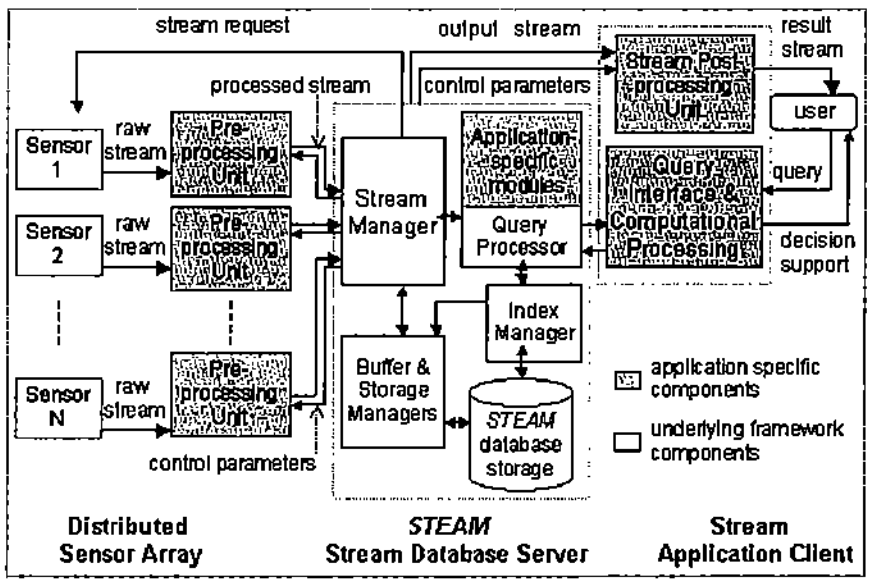

Figure 2. Architecture of the STEAM stream processing framework.

The stream database server keeps information about the streaming sources in database storage. The information may include error probabilities associated with incoming data, statistical distributions of stream values, headers associated with groups of streaming items, and the average or maximum rate of streaming. The *STEAM stream manager* handles multiple incoming streams and acts as a buffer between the stream source and stream query processing. The main function of the stream manager is to register new stream-access requests, retrieve data from the registered streams into local stream buffers, and supply data to the *query processor.*

The post-processing units export a processed output stream to the requesting client as a result stream, and additional computations may be required to generate data for user decision support. Information related to post-processing, such as additional headers or a requirement to execute specific client software, can also be stored in the database. Some of the processing units may not be needed or may be integrated with other units. Preprocessing may be integrated into the sensor source and post-processing may not be needed at all. This functionality is entirely dependent on the stream processing application.

3.3 Stream Representation and Data Type

Our two principal objectives for *STEAM* sensor representation are the following: 1) Each sensor must be identified according to both static and dynamic information. I_{static} includes the sensor identification number, location, physical features, dimensions, etc. $I_{dynamic}$ represents the real-time value information of the sensor, and the sequence of dynamic values constitutes the stream data. Both static and dynamic information are eligible for queries in the stream database system. For example, a user may request the maximum value reported by one sensor during some time interval or the maximum value reported at this moment by a subset of sensors located within a specific area. In the first example, $I_{dynamic}$ for a single sensor is accessed. In the second example, both I_{static} and $I_{dynamic}$ are accessed for the query predicates. 2) The sensor representation must be scalable, i.e., it must be capable of handling simultaneous values from multiple (possibly thousands) of sensors with low overhead for their storage, access, and query in the database system.

We considered two representations for the sensor. In the first, the sensor is defined as a relation that grows over time as data values from the sensor arrive. This representation is typical of the table functions approach [34] that has been implemented in IBM DB2 to support external sources. The disadvantage of this approach is that each sensor is considered a separate table, and a query that spans thousands of sensors must enumerate the sensors in the query syntax. Another disadvantage is the difficulty in querying static information associated with the sensors. The second alternative considers sensors as tuples, whose attributes describe both static and dynamic information. This representation scales very well for increasing numbers of sensors, and both dynamic and static information can be queried in a straightforward way using SQL syntax. We have adopted the second representation, and we view collections of sensors as a single relation (e.g., all sensors in a given application that have common static information.) For the dynamic attribute type, we introduced the *user-defined stream data type* (SDT.)

The stream type is an abstract user-defined data type that represents source data types of streaming capability. The value assigned to an attribute of type *stream* represents static information, such as the communication port number of the sensor, the URL of the web page, etc. Dynamic information is retrieved only at run time, when the stream is referenced by a query. As part of the stream type definition, the user must provide implementations for the following interface functions: InitStream, ReadStream, and CloseStream. These functions represent the basic protocol routines that are called by other stream processing components of *STEAM*; any sensor specific code can be encapsulated there. InitStream performs the necessary initializations, allocates resources, and starts up communication with the physical sensor. ReadStream retrieves the current value of the sensor, and each invocation of ReadStream produces a new value in the system. CloseStream is invoked when no further data is needed from the sensor. The system

maintains the state between function invocations and between repeated calls to ReadStream using allocated storage space for each stream.

Each value from a stream data type is associated with a time stamp. The default time stamp is the time at which the value was received by *STEAM*, but the user may override the default with a different value within ReadStream. In both cases, the system verifies that time is assigned in increasing order for each data value received. Since incoming data items are associated with a time stamp, we associate an implicit attribute to any table that has an attribute of type SDT. The attribute maintains the time associated with the tuples in that table.

4.0 The Stream Query Interface

Traditional databases provide access methods that are used to retrieve data from stored relations. Techniques such as the sequential scan and index scan are the most widely used mechanisms for accessing stored data. The type of access method is determined during query optimization. The same technique can be used to access the contents of a table with attributes of type stream; however, a new mechanism must be defined to report the dynamic values in each stream attribute. In this section, we describe StreamScan as a mechanism for accessing the content of a sensor table.

In *STEAM*, we rely on the traditional cost-based query optimizer to choose either the FileScan or IndexScan method for accessing the stored content of the sensors table; we refer to this method as the InitialScan. The method is passed to the physical query operator StreamScan, which provides the typical interfaces functions Open, GetNext, and Close [21] to use in the iterative execution of the query. During the call to Open, the sensors table is accessed using InitialScan, and the retrieved tuples are registered for subsequent streaming. GetNext reports the ready tuples to other operators in the query execution plan. When the query is terminated, the Close routine is called to perform any de-allocation and to confirm query termination to other query operators. In Section 5, we elaborate on the mechanism used by StreamScan to retrieve new streaming values.

StreamScan is more sophisticated than traditional file and index scans because it must first access the sensor table to retrieve tuples with static information about the individual sensors, and then contact those sensors continuously to get data. As described in the previous section, the sensor table includes the static information for the sensors and place-holders to receive dynamic values at run-time. Consider a sensor table with the following schema:

Sensors = (SensorID: int, SensorLocation: String, SensorValue: SDT)

and assume the table *Sensors* has values:

(1, http://alkhwarizmi.cs.purdue.edu:5001,"")
(2, http://cybil.cs.purdue.edu:5002, "")
(3, http://lisa.cs.purdue.edu:5003, "")
(4,http://ector.cs.purdue.edu:5004,"")
(5,http://icds32.cs.purdue.edu:5005,"")

The URL in SensorLocation represents the sensor location, which will be contacted to retrieve sensor values. The user sends this query:

Select S.SensorValue.temperature FROM Sensors S WHERE S.SensorID=2 AND S.SensorID=4

We adopt the strategy of pushing down any selection that references static sensor information in the query plan. We also push down any projection that does not include the SDT attributes in the query plan. The target of projection pushdown is to accommodate the possibility that the user query may only be interested in non-stream attributes of the sensor table. We determine this very early during query optimization to avoid using the StreamScan operator to scan the sensor table. In the example, the predicate *S.SensorID=2 AND S.SensorID=4* will be pushed down to the StreamScan, restricting sensors table access to tuples with SensorIDs equal to either 2 or 4. StreamScan selects only tuples one and four for streaming. This step is performed within Open, where an InitialScan is opened over the sensor table and the tuples are retrieved accordingly. The selection predicates are applied to the returned tuples, which are projected as in the projection list.

Resulting tuples are registered in the stream manager so that the corresponding data items can be retrieved. All registered streams from a single sensors table are attached to a single input queue StreamsQueue, which is maintained by the stream manger for holding StreamScan data items as the stream manager retrieves them from the corresponding sensors. Finally, the StreamScan closes the Initial Scan. The Open interface for StreamScan is illustrated in Figure 3.

```
Stream Scan: Open(Sensors Table) {
  StreamsQueue = NULL;
  Tuple t = NULL;
  InitialScan.Open(Sensors Table)    // Open scan over the sensors table
  If( table not empty){
     StreamsQueue = StreamManager.CreateStreamsQueue(); // create StreamsQueue for current scan
     While( (t = InitialScan.GetNext(Sensors Table))!=NULL)
          If (t satisfies selection predicates){
             t = projection(t)
             StreamManager.Register(t, StreamsQueue)      // register stream and attach to StreamsQueue
          };
  }
  InitialScan.Close(Sensors Table)  // Close scan over the sensors table
```

Figure 3. The Open interface for the StreamScan Operator.

The GetNext interface of the StreamScan dequeues the StreamsQueue for new tuples. If StreamsQueue is empty, Stream Scan reports *NULL tuple* to the next query plan operator. The propagation of NULL in the query plan is used to self-schedule query operators; this mechanism is described in Section 6. Our query interface handles input streams with an infinite supply of data items, but we also consider cases where input streams terminate. Termination of a single stream detaches it from the corresponding StreamsQueue, and when all streams of a single StreamsQueue are terminated, StreamScan reports an end-of-streaming message to higher query operators. Close is called when all attached streams are terminated or when the user halts the

query. It unregisters any streams attached to the StreamsQueue and flushes the StreamsQueue contents.

5.0 The Stream Manager

We have developed a real-time *stream manager* component for *STEAM* that manages incoming and outgoing data streams. The stream manager handles multiple incoming streams and acts as a buffer between the stream source and stream query processing. It services new data item requests from the query processor using StreamScan, retrieves data from the corresponding sensors into local buffers, and supplies data to the query execution engine. Stream manager operation is illustrated in Figure 4.

The StreamScan registers new streams in the stream manager. The registration request for a new stream contains the tuple value that represents the sensor's static and dynamic information. The stream is enqueued in CurrStreamsQueue and any necessary initialization is performed, such as calling InitStream for the corresponding stream data type. The stream manager attaches the stream request to the assigned StreamsQueue, which stores the streamed data items and serves as the input to the StreamScan. Many streams can be attached to a single StreamsQueue, for example all streams that belong to the same sensor table. A scheduler has been implemented to serve the current requests in the CurrStreamsQueue in a round robin fashion. The current implementation is a continuously running thread that polls the streams for new data items through calls to ReadStream, and stores them in the corresponding StreamsQueue. We are investigating interrupt-driven scheduling, where the scheduler is notified when a stream has a new tuple to retrieve.

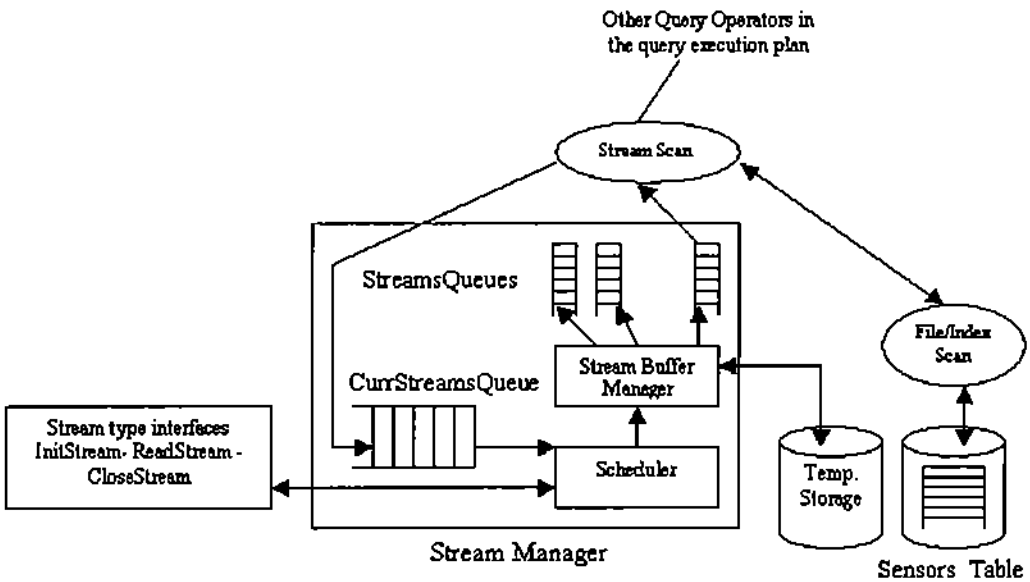

Figure 4. The stream manager retrieves data from the sensor sources and supplies them to the query physical plan. The StreamScan is the interface between the stream manager and the query execution plan.

The ReadStream interface function returns three types of values: 1) the return status (STREAM_NEW_VALUE, STREAM_NO_VALUE or STREAM_END), 2) a new tuple with the original

static information and new dynamic values (if any), and 3) the time-stamp (optional, with default equal to the current time.) Stream manager response is determined by the value of the return status. For STREAM_NEW_VALUE, the new tuple is added to the corresponding StreamsQueue. STREAM_NO_VALUE needs no action and STREAM_END tells the stream manager to decrement the number of streams attached to the corresponding StreamsQueue and remove this stream request from the CurrStreamsQueue. The stream manager also calls the CloseStream interface of the SDT. To preserve the sequence order of data streams over time, the stream manager performs an internal verification that data in the StreamsQueue are added in increasing time-stamp order. Since the rates of input streams attached to a specific StreamsQueue may be greater than the capacity of the query execution to consume the tuples, the StreamsQueue may grow beyond the available memory storage. We now discuss buffer management for the StreamsQueue.

5.1 Stream Buffer Management

The stream manager separates the process of accessing the source streams and processing the stream queries. Thus, stream query operators focus only on optimized execution of the query operations and can ignore buffering for incoming data streams. The stream manager handles the buffer requirements by storing the incoming data streams in StreamsQueue for each SteamScan operator. StreamsQueue management is complicated for query execution rates that are either faster or slower than the stream arrival rate. In the first case, the StreamsQueue memory buffer should not overflow beyond the assigned memory space since the query execution engine consumes each arrival tuple immediately. However, stream processing systems must be capable of processing multiple queries at the same time, and furthermore, a single query may reference a large number of data streams. The effect of increased numbers of input streams and/or increased numbers of concurrent queries is that the actual rate of query processing is lower than the total arrival rate of the input streams, and the StreamsQueue will eventually overflow. To handle this, we have introduced a double buffering design for StreamsQueue in *STEAM*.

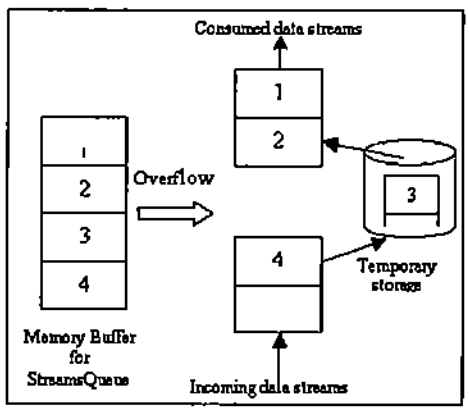

Figure 5. The partitioning of the memory buffer as the StreamsQueue overflows.

Each StreamsQueue is assigned a limited memory buffer space and, as the buffer overflows, the tail of the StreamsQueue is swapped to temporary disk storage as shown in Figure 5. When overflow occurs, the buffer is divided into four partitions. The first two *out partitions* are used as an input double buffer for query processing. The remaining two *in partitions* are used to receive incoming data items from source streams. The stream manager schedules swapping for the *in* partitions when they are full, and prefetching for the *out* partitions when they become empty.

5.2 Handling Concurrent Queries over a Single Stream

We now consider the case for multiple queries referencing the same input stream(s). Recent research adddresses query execution techniques for handling multiple concurrent queries that reference the same data [13,33], focusing on the interaction between concurrent queries and sharing the run-time state between operators from different queries. Our focus in this section is different; we address the concurrency between streams *before* processing by the query engine. The interaction between query operators from multiple queries is beyond the scope of this paper.

A single StreamsQueue can hold data from multiple streams. In addition, a single data stream can be part of multiple StreamsQueues, e.g., when multiple concurrent queries are referencing the same stream. These cases can be handled by assigning to a single stream in the CurrStreamsQueue a list of StreamsQueue that should receive the incoming data items. Any new stream request S_{req_i} is first compared against current streams requests in CurrStreamsQueue. If a similar request S_{req_j} is found, the StreamsQueue of the new request is added to the list of StreamsQueue associated with S_{req_j}. When serving S_{req_j}, the new data items are added to all StreamsQueues included in its list. The approach is simple and optimal (in terms of memory requirement) when the level of interaction between concurrent queries is small or none. As the number of common streams increases between queries, buffer space is wasted due to increased numbers of duplicate tuples in the StreamsQueue. In the worst case, this scheme will require buffer size Q*S, where Q is the number of concurrent queries and S is the number of shared streams. We propose a second scheme that scales with large numbers of data streams and increased degrees of interaction between the queries.

In the second scheme, each stream is assigned a separate buffer space (StreamBuffer) and each StreamsQueue is assigned a different buffer space. The StreamsQueue does not include the actual streamed tuple. Instead it contains a reference to the stream where the tuple actually resides. StreamBuffer contains the actual tuple, a unique identifier for the tuple, a count of number of StreamsQueues that references the tuple, and the time-stamp. StreamsQueue contains the identifier of the stream and the identifier of the tuple. The insertion algorithm for a new tuple is more complicated as it needs to add the tuple to StreamBuffer, assign its count using the number of StreamsQueue that reference the tuple, and add a tuple reference to all StreamsQueues that reference the tuple. The StreamsQueue is scanned in the same way as the first scheme. However, for each consumed tuple from StreamsQueue, the corresponding count field of that tuple in StreamBuffer must be decremented. A tuple is removed from the StreamBuffer only when its count reaches zero, that is, when it has been complete consumed by queries. The total size of the buffers is additive in terms of the number of queries and the number of streams, but the processing time is greater than that of the first scheme, since two accesses are needed to reference a tuple.

6.0 Query Execution

In this section, we describe the mechanism for query execution used by *STEAM* to achieve non-blocking execution of query operators. We restrict the discussion to select-project-join type queries, and illustrate how this technique is used to handle joins between multiple data streams that are constrained by a sliding window over time.

6. 1. Self-Scheduling for Stream Query Operators

Traditional query execution mechanisms generally utilize iterator-based query operators connected together in a pipelined fashion. The interface between operators is through calls to GetNext between parent operators and their children, and the parent operator finally reports an output to a higher operator in the query plan. The operators schedule each other in a simple routine call fashion. This mechanism proves efficient for pull-based data sources (usually disk-based), where the system has full control over data sources and the delays associated with the retrieving the data can be predicted. Even with traditional and non-streaming data sources, new execution mechanisms are introduced to speed up the execution of either a single operator [41] or the set of operators in the query plan [1]. Their goal is to avoid blocking the pipeline if one operator is incapable of producing new data items. For query operators that handle data streams, the need for a scheduler was introduced in [9,33], where a separate scheduler module schedules the different operators and the operators are connected through queues. The management of queues between the operators introduces additional overhead, especially when queues need to be swapped to disk. The proposed solution in eddy [1], which is mainly designed to handle dynamic ordering between operators of the query plan, maintains additional routing information per tuple.

We have developed a simple schedule mechanism that avoids using a separate scheduler module and also avoids the large size queues between the different query operators. Our mechanism fits easily in the traditional pipelined execution without introducing any tuple overhead. The proposed scheduler is non-blocking; if one of the operators has no new data from its underlying sources, the operator simply relinquishes control to next operator in the pipeline to proceed. This step is repeated until arriving at the top level operator in the pipeline, where it repeats execution of previous operators (through calls to GetNext.) It is important to note that the release of control is not the result of the empty *output* of the operator execution, which may be caused by other factors such as high selectivity of the operator. Rather, it is associated with the blocking of all the input sources. Since the blocking of all input sources indicates that the operator is incapable of producing more output tuples, releasing control to other higher operators is a reasonable alternative. Note that with traditional pipeline execution, control is passed to the higher operators only when new tuples are produced. Note also that operator optimizations to speed up the production of output tuples [41] are still applicable using our mechanism. In *STEAM*, we represent the release of control from one operator to the next operator in the query plan as a production of a NULL tuple. The NULL tuple is not actually processed in the query execution and simply signals the blocking of the current operator to the next operator.

Our technique provides fair scheduling between operators and, at the same time, guarantees execution of any operator that has tuples to consume. The self-scheduling technique reduces to the traditional pipeline scheduling when input sources always have input tuples waiting to be processed.

6.2 Joining Multiple Streams Using Window Constraints

We have developed a general class of algorithms for joining multiple infinite data streams, and have integrated these algorithms into the *STEAM* system. The algorithms address the infinite nature of streams by joining stream data items that lie within a sliding window over time and match the specified join condition. Our *stream window-join* (W-join) can be applied to many practical queries that occur within the context of processing simultaneous online data streams. In

this section, we introduce the class of algorithms for performing the W-join. The algorithms are non-blocking and can easily be implemented in the pipeline query plan. For comparison, we have also implemented stream join algorithms from the literature in *STEAM*, and we present experimental results comparing W-join to existing algorithms using real data streams. Our results demonstrate that the new W-join algorithms outperform existing algorithms by an order of magnitude under a variety of stream data rates and stream delays.

The data streams processed by W-join represent continuously incoming data for multi-sensor network applications. An essential property for stream query operators is that they be non-blocking and able to process infinite amounts of data. Some ongoing research projects have focused on window aggregate operations for data streams [22], where the window is defined to be the entire contents of the stream seen thus far (the stream *prefix*.) Other studies [32, 38] consider a join operation between a data stream and a typical database relation, or a self-join over a single stream. However, none of these studies address the join operation for multiple streams. Some systems that execute join queries on both streamed and stored data apply only to special cases of the join. For example, the Fjord system [32] restricts the join operation to the equi-join, with no joining between two streams. More recent stream join techniques, such as ripple-join [24] and the non-blocking hash-joins [20,40] emphasize producing early results during join execution. However, these operators all require processing the entire stream prefix, as well as all incoming tuples. Stream join queries can either execute separately or be combined with aggregate functions to produce summary information. While these queries are somewhat similar to temporal queries, efficient algorithms for temporal joins [31,44] depend on the underlying access structure, which is not available for online stream data sources. In addition, temporal join algorithms do not consider long-running queries over infinite data sources.

In contrast to existing stream join algorithms, our class of algorithms joins multiple infinite data streams over a sliding window of time and can be applied to any join condition. W-join can take several forms depending on the variations and/or existence of window constraints between each pair of joined streams. A *uniform window constraint* is used to join all input streams using a single, fixed window size. For example, to monitor the simultaneous occurrence of a hazardous material in multiple distributed sensors with a sliding one hour interval w, we issue the query:

SELECT A.GasName FROM SensorTables A, B, C
WINDOW =w
WHERE A.GasId=B.GasId AND B.GasId= C.GasId

The W-join for *different window constraints* is applied to cases where window sizes vary between pairs of input sensors. To track the spread of a hazardous gas that is detected by multiple sensors, where the maximum time for the gas to travel between the sensors defines the time windows, w_1, w_2, w_3, ... for the join, we issue the query:

SELECT A.GasId FROM SensorTables A, B, C
WINDOW(A,B) =w_1 AND WINDOW(B,C) =w_2 AND WINDOW(A,C) =w_3
WHERE A.GasId=B.GasId AND B.GasId= C.GasId

The W-join can also handle joins *where some window constraints do not exist* between pairs of input streams, e.g., if there is a barrier preventing the propagation of the gas. Assume paths exist between sensors A and B and between B and C, but not between A and C. We issue the query:

SELECT A.GasId FROM SensorTables A, B, C
WINDOW(A,B) =w_1 AND WINDOW(B,C) =w_2
WHERE A.GasId=B.GasId AND B.GasId= C.GasId

Our class of W-join algorithms can be used to answer any of the above forms of stream join. It avoids repeated iterations over non-window related tuples by verifying window constraints between the input streams and then updating join buffers to contain only eligible tuples. All algorithms are non-blocking to adapt for variations in stream data rates. For the uniform window constraint, we present detailed descriptions of the nested-loop (NLW-join) and hash (HW-join). The two algorithms use a similar approach for tuning intermediate join buffers to maintain window-related tuples from different streams, and both algorithms can be easily integrated into query pipeline execution. We have also developed a merge (MW-join) algorithm that can be generalized to address the variations and local/global optimizations related to the nature of the window constraints. Due to space limitations, we omit the description of MW-join and the handling of different or missing windows constraints. These algorithms are described in [25].

6.2.1 The W-join Operation

We define the operation of W-join as follows:

> *Given N data streams and a join condition which is represented as a Boolean expression of the values of the tuples, find the tuples that satisfy the join condition and that are within a sliding time window of length w time units from each other.*

To illustrate the W-join, Figure 6 shows the operation of W-join among five data streams, A through E, where the position of the data stream tuples on the x-axis represents their arrival order over time. The heavy black dots correspond to tuples that satisfy the join condition, and the window restriction implies that only tuples within a window of each other can be joined. Thus, the tuple $<a_2,b_2,c_1,d_2,e_2>$ is a candidate for W-join, but $<a_2,b_2,c_1,d_2,e_1>$ is not since d_2 and e_1 are separated by more than a window.

We clearly need an efficient approach for verifying window constraints between the input streams and updating join buffers to contain only eligible tuples. The brute-force approach requires verifying constraints between each pair of *N* streams. A more efficient approach for verifying that *N* objects, each from a different class, are within a fixed distance of each other has been suggested by Aref et al. [3] and we adopt a similar approach for verifying window constraints among the individual tuples. While the algorithms in [3] cannot deal with infinite streams and may block if data is delayed, our approach is non-blocking and does not require complete scans over the streams. For updating the join buffers we provide an online approach that updates the join buffers either periodically or as storage overflows.

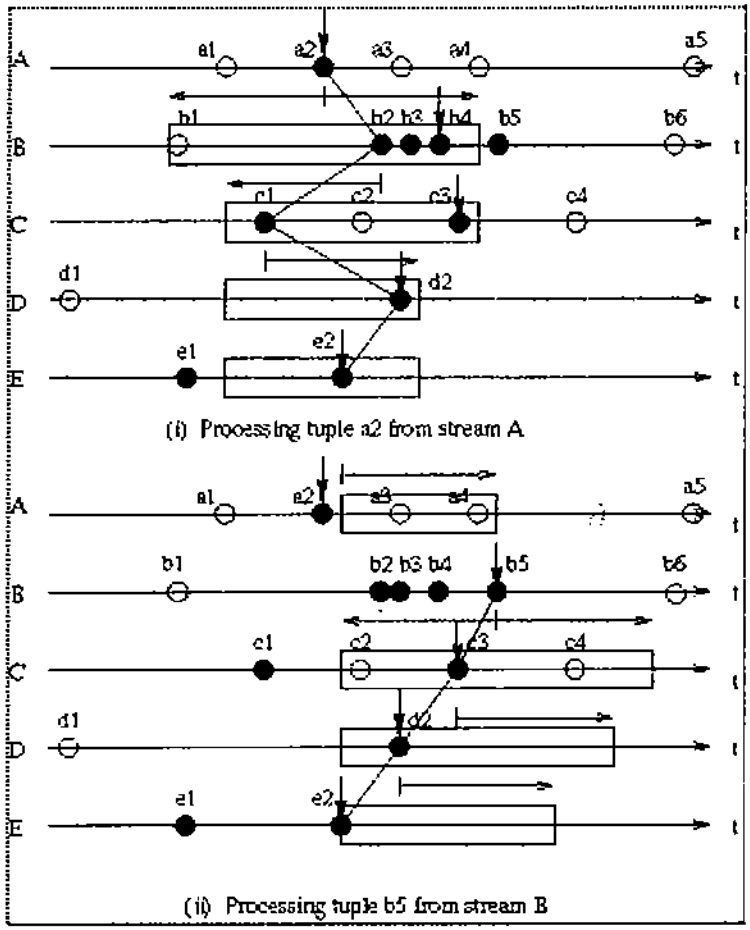

Figure 6. Two iterations of the W-join for five streams.

In Figure 6, five streams are joined together using a single window constraint of length w that applies to each pair of streams. W-join maintains a buffer for each stream, and the vertical bold arrows point to the last tuple processed from each stream. The algorithm processes each stream in order: A, B, C, D, E, A, B, etc. At each iteration, a new tuple from a different stream is processed. If the current stream has no tuples, the next stream is processed so that the algorithm does not block waiting for tuples from a single stream.

Figure 6(i) depicts the status of the algorithm as it processes a_2 from stream A, forming a window of length $2w$ centered at a_2. The algorithm then iterates over all tuples of stream B that are within the window of a_2. These tuples from stream B are shown inside the rectangle over B. Since b_1 is white, it does not satisfy the query predicate and cannot join with a_2. Tuple b_2, however, satisfies the join and is located within the required window of a_2. Thus the period is modified (reduced) to include a_2, b_2, and all tuples within w of either of them. This new period, shown as a rectangle over stream C, is used to test the tuples in C.

Tuple c_1 satisfies the join predicate and lies within the rectangle, so a new period is calculated that includes a_2, b_2, c_1 and all tuples within w of any of them. This period is shown as a rectangle over D. In stream D, d_2 satisfies the join and is located within the required window –

creating a new period that include these tuples and any others within length w of any of them. This period is shown as a rectangle over stream E. We repeat this test for E, and the 5-tuple $<a_2,b_2,c_1,d_2,e_2>$ is reported as output. The algorithm recursively backtracks to consider other tuples in D, then C and finally B. The final collection of 5-tuples in the iteration beginning with tuple a_2 is: $<a_2,b_2,c_1,d_2,e_2>$, $<a_2,b_2,c_3,d_2,e_2>$, $<a_2,b_3,c_1,d_2,e_2>$, $<a_2,b_3,c_3,d_2,e_2>$, $<a_2,b_4,c_1,d_2,e_2>$, $<a_2,b_4,c_3,d_2,e_2>$. When the iteration for a_2 is complete, the algorithm begins again using a different stream. In Figure 6 we advance the pointer of stream B to process tuple b_5. This iteration is shown in Figure 6(ii), where periods over streams C, D, E, and A are constructed. This iteration produces no output since no tuples join in these rectangles.

W-join must address the removal of old tuples from the buffer associated with each stream, where old tuples are those which will never W-join with any incoming tuples. We remove a tuple from a stream if it is located a distance more than w from the last tuple in *all* other streams. In Figure 6(i), we remove d_1 as we process stream D, since it is located more than w away from the last tuple of all streams, namely a_2, b_4, c_3, and e_2. Note that a tuple cannot be removed merely because it is more than length w away from a *single* stream.

W-join is non-blocking in the sense that it does not stop if one of the streams has no tuples; instead it continues to process tuples from other streams. In addition, W-join can easily be implemented in the pipeline query plan, since it is clear that new tuples from each stream are only compared with tuples in the period constructed thus far, and those tuples can be produced from lower levels in the pipeline tree.

6.2.2 The NLW-join Algorithm.

We consider a left deep, pipelined execution plan of the joins, where at each level in the pipeline an additional stream is introduced to the join and the source streams correspond to leaves in the execution tree. The left stream may be a source stream or an intermediate stream from a lower level in the tree, but the right stream is always a source stream. We let "m-tuple" denote the W-joined tuples between m streams which are constructed at intermediate stages of W-join execution. Except at the lowest node in the tree, tuples in the tree do not have a single time stamp per period. As the left m-tuple climbs the tree, additional time stamps are added. We store the times stamps of the joined tuples with the tuples themselves in order to handle tuple removal. The period of each m-tuple can easily be computed as $[T_{max} - w, T_{min} + w]$ where T_{max} is the maximum of all times stamps in the m-tuple and T_{min} is the minimum.

All tuples received from one stream and not yet dropped are stored in a stream buffer, which is either small enough to fit in memory or is swapped in part (the tail) to disk. The pipelined algorithm alternates between retrieving tuples from the left and right streams, and does not block if one of the streams has no tuples. In this case, it continues to retrieve tuples from the other stream, and if the other stream is also empty, the join produces an empty tuple to higher levels. An empty tuple signals higher levels to either process other non-blocking streams or to call the lower level again. As an m-tuple arrives from the left stream, it is used to iterate over all qualifying tuples in the right stream. Only tuples that are located within the period of the m-tuple are tested for the join predicate. If it satisfies the join from the right stream, a new (m+1)-tuple is reported to higher levels and the time stamp of the right tuple is added to the set of time stamps already in the m-tuple. The process is repeated as each tuple arrives from the right stream.

To remove old tuples from the buffer of source and intermediate streams, we need to verify that the tuples are not included within the window of any arriving tuples from *all* the streams. For this, we keep a vector of time stamps for each buffer which maintains the maximum time stamps appearing in each source stream. For right streams, the vector V_{right} has a single value that represents the maximum time stamp for that stream. For left streams, V_{left} is a vector of maximum time stamps for each source stream in its subtree. During the iteration of one stream, we remove the tuple that is located a distance more than w from the tuples in the time stamp vector of the other stream. This tuple is guaranteed not to W-join with later tuples appearing in the other stream.

6.2.3 The HW-join Algorithm.

For the hash-based implementation, we assume the join predicate is an equality predicate over the join attribute. The approach for updating the window is similar to the NLW-join, but the hash algorithm builds hash tables based on the join equality attribute for both streams. The iteration of HW-join is also similar to NLW-join, except that a new tuple is used to probe the hash table of the other stream instead of scanning it. As with NLW-join, we need to update the buffer (hash table) of one stream as tuples arrive from the other streams. For hashing, the probing tuple only visits part of the hash table – the bucket that has the same hash key. This may result in old tuples occupying buckets that are probed infrequently. To handle this situation, we call an update routine to probe buckets that are rarely probed. It is worth noting that with large numbers of arriving tuples and non-skewed values for the hashing attribute, the buckets are regularly updated without the need to call an update routine.

6.2.4 Performance Studies for W-join

We now evaluate the performance of NLW-join and HW-join using real data streams. We also compare the performance of the W-join algorithms with versions of the ripple-join and Xjoin which have been adapted to consider window constraints and periodic updates of their join buffers. We first describe our workload data, and then discuss the experimental results.

Logs of Wal*Mart transactions were used for experiments based on real data. Our query joined multiple stores, where the join predicate was equality of the item number sold in different stores. Sold items appear as transactions of customer purchases. A single transaction for each store includes the item number, description, purchase time, etc. A single store represents a stream of items that have been sold, and the time stamp of the tuple is the time of transaction. The data was extracted form the NCR Teradata machine that hold 70 Gbytes of Wal*Mart data.[1] The performance measure is the service time for each arriving tuple, which represents the time needed to handle an arriving item in the W-joins. The service time depends on the number of comparisons in each iteration, and reflects the efficiency of the algorithm. We collected the service times averaged by 1000 input tuples during the experiment, and repeated the experiment multiple times to validate the trends in the performance curves.

Figure 7 shows the results of {2,3,4}-way joins using real data streams with the sliding window set to one hour. In Figure 7(a), we compare the NLW-join with a version of ripple-join that is periodically updating its join buffers every 10,000 tuples to remove non-window related

[1] Computer hardware and software were donated to Purdue University by Wal*Mart and NCR Corporation.

tuples. In Figure 7(b), the same experiment is performed to compare the HW-join and a version of Xjoin, where the hash tables are updated periodically in Xjoin. The steep increase and decrease in the curves for 2-way joins are due to the nature of the real data, since it represents transactions during the hours of store operation. Note that the plots do not build up as more tuples are processed from the streams due to the effect of updating the join buffers. This is demonstrated even more clearly when comparing the plots of the W-joins with the adapted ripple-join and Xjoin. The experimental results show that W-join outperforms the ripple-join and Xjoin in all cases.

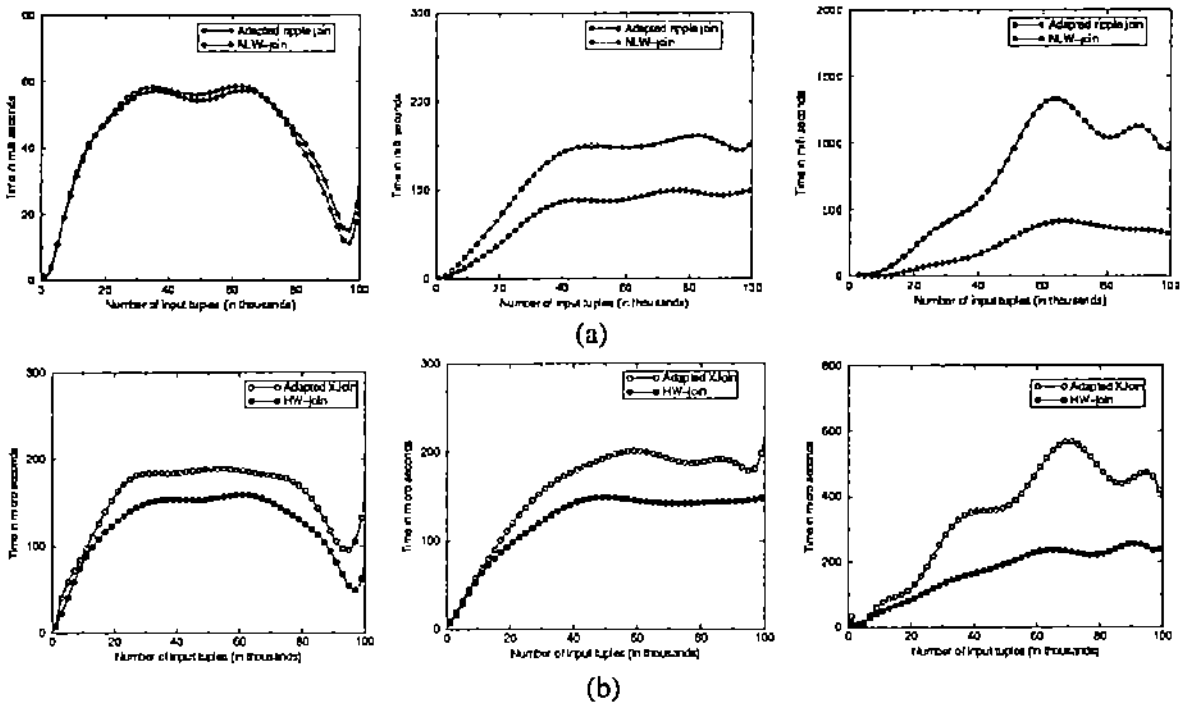

Figure 7. Performance comparisons using real streams for (a) the adapted ripple-join and NLW-join for 2, 3 and 4 streams (left to right) and (b) the adapted Xjoin and HW-join for 2, 3 and 4 streams (left to right)

7. The Intelligent Detection of Hazardous Materials (IDHM)

The *STEAM* server processes continuous, incoming streams generated by sensor networks that feature external data sources with dynamic characteristics. Our framework provides support for interactions with stream pre- and post-processing units, scalable integration of sensors, and scalable support for internal stream processing requests. In this section, we present an application which was built using *STEAM* technology.

7.1 Rationale for IDHM

Airline security, hazardous material detection, and counter-terrorist efforts are leading national priorities. Present technology is not capable of providing fast, cost-effective and reliable detection, and no single technology can attain high detection rates and low false signals for all types of hazardous materials. There is a critical need for the development of detection systems

that integrate multiple detector technologies, utilizing detectors of different types, data fusion and sensor response integration. The Center for Sensing Science and Technology (CSST) has been established to research and develop deployable detection systems [12] and *STEAM* provides the database server framework to build such applications. For our initial effort, we have integrated the *STEAM* engine with the gamma detectors developed by CSST. The application addresses the use of detection technology to identify concealed hazardous materials in airline passenger baggage, and was developed in collaboration with CSST and the Physics Department at Purdue University.

This system consists of 1) a distributed array of gamma detectors, each of which captures the energy spectrum of a black box (e.g., suitcase) during exposure to neutrons produced by a neutron generator, 2) the *STEAM* stream database server, and 3) a stream client application which provides an interface to the user for entering queries and receiving output streams and action recommendations. The system attempts to identify the existence and type of hazardous materials in a black box by applying data mining query operators to the data streams produced by the gamma detector, supported by an underlying knowledge base of energy spectra information. Figure 8 shows a functional view of our framework for this application. The preprocessing unit forms energy spectra from the data values generated by the gamma detectors. The stream manager handles the incoming streams and provides an interface to the query processor. Data-mining operators in the query execution plan analyze the spectra, and results are reported to the client.

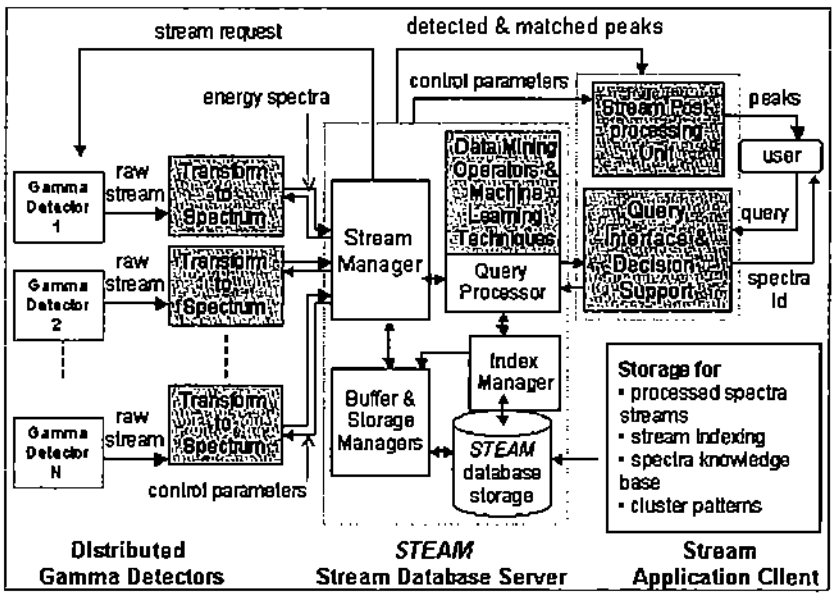

Figure 8. Functional view of IDHM.

7.2 Data Mining Operators

The discovery of knowledge from incoming data streams is an essential element of the IDHM application. Online and incremental stream processing is needed to avoid complete passes over entire streams and to handle the continuous and increasing numbers of arrivals, while consulting

both past and current streams. Data mining tools in STEAM are operators, both logical and physical, which are integrated in the query processor.

We have developed and implemented several new data mining operators to aid in online analysis and decision-making. Our operators support incremental and online mining of partial periodic patterns [4,26], periodicity and cycle detection [16], high-dimensional clustering and the discovery of frequent subsequences appearing across all incoming data streams.

7.3 IDHM Stream Processing

Data arrives from the gamma detectors in bursts, and a new energy spectrum curve for the current box arrives every n seconds. When new data arrives, the system concludes either that the box is safe or that more data is required to make a decision. More data may be needed due to background material in the box that has affected the spectrum or because prolonged exposure is required to produce better peak data. One metric to measure the effectiveness of the detection methodology is the average time for reaching a decision. In some preliminary results, the system was able to detect certain peaks from the first incoming spectra streams, while other peaks required significantly more time. Each peak is modeled and evaluated separately, but we plan to apply temporal stream data mining to capture the evolution of the energy spectrum over time. This area of research may allow the system to predict upcoming energy spectrums to speed the decision process.

Analysis of an energy spectrum consists of two phases: location of the peaks and correspondence of the peaks to known hazardous materials. Our data mining methods apply machine-learning techniques to model the hazardous materials and match them against known curves using decision tree induction, nearest neighbor methods, and clustering. The machine learning approach views this as a pattern classification problem and attempts to assign an input pattern to one of a finite number of known classes. Each incoming energy spectrum defines a pattern, and the classes in the knowledge base include known hazardous materials and the "none" class corresponding to a safe box. Background materials can significantly affect the energy spectrum and must be filtered out.

We store a significant portion of the preprocessed data streams from the sensors in the *STEAM* storage layer, which serves as a repository and index for data collected from different types of detectors. Our data mining operators provide analysis and integration of sensor readings, increasing the level of confidence in reliability and accuracy of results. We have applied the subsequence sensor analysis operators for temporal stream mining, subsequence similarity matching, periodicity detection, and the mining of partial periodic patterns. The *STEAM* database server is responsible for requesting and collecting data from the gamma detectors. The protocol interface between the *STEAM* stream manager and the gamma detectors is based on the pull paradigm and includes the interface functions Open, Get_Next and Close. The query interface is used to initiate the detection process and receive results in the form of actual and matched energy spectra, identification of the matched spectra pattern, and data to support action recommendations.

8. Conclusion

We have introduced the *STEAM* system as a framework for building distributed multi-sensor applications. The framework incorporates a stream database management server as an integral and fundamental component to provide the underlying database technologies needed for online data stream management with real-time constraints, stream buffer management, online and long-running stream query processing, and advanced stream query and data mining operators for stream analysis. We established the stream model and database representation developed for *STEAM* and described the functionality, design and implementation for key stream management and query processing components. We described two new algorithms, the NLW-join and HW-join, for performing join operations between multiple data streams using a sliding window over time. The algorithms use an efficient approach for verifying the window requirement and provide online updating of join buffers. The new algorithms outperform the ripple join and Xjoin for queries with time window constraints.

We also presented a system for the intelligent detection of hazardous materials which was created using the *STEAM* framework. The system attempts to identify the existence and type of hazardous materials by applying *STEAM's* data mining query operators to the data streams. We plan to continue the design and implementation of new multi-sensor processing applications based on the *STEAM* framework. The applications are used to motivate and advance the underlying stream processing functionality of *STEAM*

References

[1] Avnur, R. and Hellerstein, J. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data.* Dallas, Texas. May 16-18, 2000. Vol. 29. pp. 261-272. May, 2000.

[2] Arasu, A., Babcock, B., Babu, S., McAlister, J. and Widom, J. *Characterizing Memory Requirements for Queries over Continuous Data Streams. In Proc. Of PODS 2002.*

[3] Aref, W., Barbará, D., Johnson, S. and Mehrotra, S. Efficient Processing of Proximity Queries for Large Databases. *In Proc. of the 11th ICDE, March, 1995.*

[4] Aref, W., Elfeky, M. and. Elmagarmid, A. Incremental, Online and Merge Mining of Partial Periodic Patterns in Time-Series Databases. Submittted.

[5] Aref, W., Kamel, I. and Ghandeharizadeh, S. Disk scheduling in video editing systems. *IEEE Trans. on Knowledge and Data Engineering.* 13(6). pp. 933-950. November/December 2001.

[6] Aref, W., Catlin, A., Elmagarmid, A., Fan, J., Hammad, M., Ilyas, I., Marzouk, M., Zhu, X. Search and discovery in digital video libraries. *CDS TR #02-005, Computer Sciences Department, Purdue University.* February 2002.

[7] Aref, W., Catlin, A., Elmagarmid, A., Fan, J., Guo, J., Hammad, M., Ilyas, I., Marzouk, M., Prabhakar, S., Rezgui, A., Teoh, S., Terzi, E., Tu, Y., Vakali, A. and Zhu, X. A distributed server for continuous media. In *ICDE'02 Proc. of the 18th International Conference on Data Engineering.* February 26-March 1. San Jose, California. February 2002.

[8] Babcock, B., Babu, S., Datar, M., Motwant, R. and Widom, J. Models and issues in data stream sytems. *Invited talk PODS 2002.*

[9] Babu, S. and Widom, J. Continuous queries over data streams. In *SIGMOD Record.* 30(3). September 2001.

[10] Berchtold, S., Böhm, C., Jagadish, H., Kriegel, H-P. and Sander, J. Independent quantization: An index compression technique for high-dimensional data spaces. In *ICDE'00 Proc. of the 16th International Conference on Data Engineering.* San Diego, CA. pp. 577-588. February 2000.

[11] Bonnet, P., Gehrke, J. and Seshadri, P. Towards sensor database systems. In *Proceedings of the Second International Conference On Mobile Data Management.* Hong Kong. January 2001.

[12] Center for Sensing Science and Technology: IDHM. http://www.physics.purdue.edu/csst/

[13] Chen, J., DeWitt, D., Tian, F. and Wang Y. Niagracq: A scalable continuous query system for Internet databases. In *Proc. of the 2000 ACM SIGMOD International Conf. on Management of Data.* May 16-18. Dallas, TX. Vol. 29. pp. 379-390. 2000.

[14] Datar, M., Gionis, A., Indyk, P. and Motwani, R. Maintaining Stream Statistics over Sliding Windows. In *Proc. of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2002.

[15] DeWitt, D., Naughton, J. and Schneider, D. An evaluation of non-equijoin algorithms. *17th VLDB Conf. on Very Large Data Bases*. September 1991.

[16] Elfeky, M., Aref, W., Atallah, M. and Elmagarmid, A. Periodicity detection in time-series databases. *CDS TR #02-120, Computer Sciences Department, Purdue University.* May 2002.

[17] Fan, J., Aref, W., Elmagarmid, A., Hacid, M-S., Marzouk, M. and Zhu, X. Multiview: Multi-level video content representation and retrieval. *Journal of Electrical Imaging*, 10(4). pp. 895-908. October 2001.

[18] Fan, J. and Elmagarmid, A. Semi-automatic semantic algorithm for video object extraction and temporal tracking. *Signal Processing: Image Communication.* Vol. 17. 2002. To appear.

[19] Fan, J., Hacid, M-S. and Elmagarmid, A. Model-based video classification for hierarchical video access. *Multimedia Tools and Applications.* Vol. 15. October 2001.

[20] Florescu, D., Levy, A., Manolescu, I. and Suciu, D. Query optimization in the presence of limited access patterns. In *Proceedings of ACM SIGMOD Conference.* June 1999.

[21] Garcia-Molina, H., Ullman, J., and Widom, J. Database System Implementation. Prentice Hall, 2000.

[22] Gehrke, J., Korn, F. and Srivastava. On computing correlated aggregates over continual data streams. In *Proceedings of ACM SIGMOD Conference.* May 2001.

[23] Gilbert, A., Kotidis, Y., Muthukrishnanm, S. and Strauss, M. Surfing wavelets on streams: one-pass summaries for approximate aggregate queries. *In Proc. of 27th VLDB Conference, September, 2001.*

[24] Haas, J. and Hellerstein, J. Ripple joins for online aggregation. In *Proc. ACM SIGMOD Conference.* 1999.

[25] Hammad, M., Aref, W. and Elmagarmid, K. Joining Multiple Data Streams with Window Constraints. *CDS TR #02-115, Computer Sciences Department, Purdue University.* May 2002.

[26] Han, J., Dong, G.and Yin, Y. Efficient Mining of Partial Periodic Patterns in Time Series Databases. In *Proc. of 1999 Int. Conf. on Data Engineering*, Sydney, Australia, March 1999.

[27] Henzinger, M., Raghavan, P. and Rajagopalan, S. Computing on data streams. In *Technical Note 1998-011.* Digital Systems Research. 1998.

[28] Jagadish, H., Mumick, I. and Silberschatz, A. Vew Maintenance Issues for the Chronicle Data Model. *In Proc. of PODS, May, 1995.*

[29] Jiang, H., Helal, A., Elmagarmid, A., and Joshi, A. Scene change detection for video database systems. *Journal on Multimedia Systems*, 6(2), pp.186–195. May 1998.

[30] Katayama, N. and Satoh, S. The SR-tree: An index structure for high dimensional nearest neighbor queries. *SIGMOD Record, ACM Special Interest Group on Management of Data*, 26(2). 1997.

[31] Lu, H., Ooi , B. and Tan, K. On the spatially partitioned temporal join. In *Proc. of 20th VLDB Conference*, Sept. 1994.

[32] Madden, S. and Franklin, M. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE'02 Proc. of the 18th International Conference on Data Engineering*. February 26-March 1. San Jose, California. February 2002.

[33] Madden, S., Shah, M., Hellerstein,J., and Raman, V. Continuously Adaptive Continuous Queries over Streams. In Proc. of SIGMOD 2002.

[34] Reinwald, B., Pirahesh, H., Krishnamoorthy, G., Lapis, G., Tran, T. and Vora, S. Heterogeneous query processing through SQL table functions. In *ICDS'99 Proceedings of the 15th International Conference on Data Engineering*. March 23-26. Sydney, Australia. pp. 366-373. 1999.

[35] Seshadri, P. Predator: A resource for database research. *SIGMOD Record*. 27(1). pp. 16-20. 1998.

[36] Seshadri, P., Livny, M. and Ramakrishnan, R. The design and implementation of a sequence database system. In *Proc. of 22th VLDB Conference, Sept., 1996.*

[37] Storage Manager Architecture. *Shore Documentation, Computer Sciences Department*. UW-Madison, June 1999.

[38] Sullivan, M. and Heybey, A. Tribeca: A system for managing large databases of network traffic. *USENIX, New Orleans. LA*. June 1998.

[39] Thomas, M., Carson, C. and Hellerstein, H. Creating a customized access method for blobworld. March 2000.

[40] Urhan, T. and Franklin, M. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*. 23(2). 2000.

[41] Urhan, T. and Franklin, M. Dynamic Pipeline Scheduling for Improving Interactive Query Perfromance. *In Proc. of VLDB Conference. 2001.*

[42] Viglas, S., and Naughton, J. Rate-Based Query Optimization for Streaming Information Sources. *In Proc. ACM SIGMOD Conference. 2002.*

[43] Wilschut, A. and Apers, P. Dataflow query execution in a parallel main-memory environment. In *Proc. of the 1st PDIS Conf.* December 1991.

[44] Zhang, D. Tsotras, V., and Seeger, B. Efficient temporal join processing using indices. In *ICDE'02 Proc. of the 18th International Conference on Data Engineering*. February 26-March 1. San Jose, California. February 2002.