PIPES: A Multi-Threaded Publish-Subscribe Architecture for Continuous Queries over Streaming Data Sources

Michael Cammert Christoph Heinz Jürgen Krämer Alexander Markowetz
Bernhard Seeger
Department of Mathematics and Computer Science
University of Marburg
35032 Marburg, Germany
{cammert, heinzch, kraemerj, markow, seeger}@informatik.uni-marburg.de

Abstract

In contrast to traditional query processing based on persistent data, new application scenarios arise that heavily rely on the continuous evaluation of data streams. These streams often emerge from autonomous data sources. In this paper, we present the novel publish-subscribe architecture PIPES (Public Infrastructure for Processing and Exploring Streams), which easily allows the composition of complex query graphs with an inherently dynamic resource sharing, even during runtime. We sketch essential design and implementation considerations of PIPES, which contains a generic operator framework for streaming data. PIPES is completely implemented and integrated in XXL, a flexible and extensible Java library for data processing. In contrast to related system prototypes, whose communication between operators solely depends on queues, we permit direct interoperability. In our new hybrid scheduling approach for PIPES, we intend to dynamically vary the number of concurrently running lightweight processes, called threads. First experimental studies show its advantages of neither being restricted to a single thread nor assigning a separate thread per operator.

1. Introduction

Nowadays several scientific research projects deal with data stream processing. Inspired by different applications based on autonomous data sources, like sensors, click streams, financial tickers or network traffic, first prototypical systems have been developed [4, 11, 26, 13]. There are still basic challenges associated with the differences in processing data streams and conventional persistent relations.

Due to characteristics of autonomous data sources, several issues of database management systems (DBMS) have to be reconsidered, e.g. memory management, operator scheduling and query processing in general. One of the open aspects, which still has to be addressed, is the design of a standardized operator-interface similar to the well-known ONC interface for relational operators [19]. Such an interface has to be powerful enough to model arbitrary query graphs. In contrast to the original ONC query processors, an explicit operator scheduling is of utmost importance for an efficient processing of continuous queries. The overall architecture should support a flexible operator scheduling that grants high output rates, induces low overhead and allows to meet Quality of Service (QoS) constraints.

So far, there have been two approaches that each meet one of the criteria, but fail to meet the other.
In *operator-threaded scheduling (OTS)*, each of the operators runs in a separate lightweight process (thread) [2, 11, 27]. This yields a high output rate for moderate query sizes. However, the thread management by its own creates a significant overhead. Additionally, this noticeably decreases the output rate for large query graphs.
In *graph-threaded scheduling (GTS)*, the entire query graph runs in a single thread [26, 12, 3]. Within this thread, a selector determines the next operator to be executed. The selector then hands control over to this operator and receives it back, once exactly one element is entirely processed. Given a lean selector, this approach induces very low overhead. Problems however arise, when the atomic processing of an element requires a substantial amount of time. This may lead to a buffer overflow when incoming data cannot be kept in memory anymore, or cause the output rate to temporarily drop to zero. An interruption of the operator could avoid both cases. Furthermore, GTS does not make use of today's multi-processor hardware.
Due to the drawbacks of both strategies, we have developed a *hybrid multi-threaded scheduling* (HMTS) that contains the best of both worlds as well as an orthogonal approach

of *direct interoperability* (DI). The latter merges a sequence of operators to a virtual operator. It is particularly suitable for handling low-cost operators since it eliminates communication overhead and intermediate buffering.

We have developed a library PIPES (Public Infrastructure for Processing and Exploring Streams) for processing queries on streams that fully implements HMTS. Based on a flexible publish-subscribe architecture, PIPES allows the construction of complex query graphs over arbitrary data sources. It contains interfaces that model the data flow and control flow between different operators and provides easily extensible abstract classes as well as a rich set of ready-to-use operators.

PIPES is completely integrated in XXL [6, 10], a Java library developed by our team. XXL provides a modular software infrastructure for efficient query processing under the terms of the GNU Lesser General Public License. The goal of XXL was to separate the different components of a DBMS such as an object-relational algebra and a framework for index-structures. For sake of increased flexibility and extensibility, we decided not to build just another system from scratch. The streaming architecture of PIPES seamlessly extends XXL's scope towards data-driven query processing.

The rest of the paper is organized as follows. Section 2 provides a survey of related projects with regard to their architecture and interoperability. In Section 3, we introduce the design and convenient construction of query graphs in PIPES. We present the details of HMTS in Section 4, dedicating a subsection to a novel strategy for the inevitable synchronization of operators in a multi-threaded environment. In Section 5, we report the results of a preliminary experimental study indicating that PIPES (and its HMTS) achieves high output rates while consuming little memory. Finally, we present our conclusion and future work in Section 6.

2. Related Work

In this section, we provide a brief overview of current work with regard to data stream management:

The *Aurora* project [11] builds a system for applications monitoring data streams and attempts to satisfy realtime requirements. Support of QoS is a central goal, for which techniques like load shedding and memory-aware operator scheduling are adopted. Aurora supports resource sharing by communication via queues that have to be synchronized. While in [11] OTS is proposed, later work [12] uses GTS. Aurora introduces *connection points*, a particular kind of temporal storage. Several operators can be connected to such a point that additionally supports historical adhoc queries.

NiagaraCQ [15], a subsystem of *Niagara* [28], offers scalable continuous query processing over multiple, distributed XML files in a wide area network. Typical applications consist of a change- or timer-based monitoring of XML data sources. Based on the assumption that many web queries share similar structures, NiagaraCQ establishes scalability with a general strategy of incremental group optimization. Intermediate files store the results of overlapping queries and are treated like original data sources.

STREAM [27] is a centralized data stream management system that relies on the relational data model. Queries, which can also be continuous, are formulated in a declarative query language derived from SQL. In order to reduce resource requirements, STREAM summarizes data with synopses and provides a global scheduler that minimizes the queue's memory usage. Similar to Aurora, a queue connects two successive operators and guarantees uni-directional communication. The architecture described in [27] relies on an OTS model, while later work on scheduling [3] bases on GTS. In contrast to Niagara, STREAM takes approximate query answers into account. Another important idea mentioned in [4] considers joins between streams and relational data.

The *TelegraphCQ* [13] system combines prior work in *Fjords*, *Eddies* and *PSoup*. Thereby, the Fjords API [26] defines a set of operators, which communicate via either "push" (asynchronous) or "pull" (synchronous) modalities. Fjords allows OTS as well as GTS. Eddies [2] are specific operators that decide how to route elements adaptively among other operators. These operators run in separate threads. PSoup [14] extends the work on Eddies by treating data and queries symmetrically. It allows multiple continuous and ad-hoc historical queries. The FluX operator introduces load balancing and fault-tolerance into TelegraphCQ.

The evaluation of continuous queries (CQ) in the *Alert* [33] system depends on the usage of event-based triggers. The *Tribeca* [35] system provides efficient filter- and aggregation operations on network data streams. The *Tukwila* [21] system is mainly concerned with the integration of distributed and autonomous data sources. [22] describes an extension of Tukwila, in order to achieve adaptivity and approximative query processing. The *X-Filter* [1] system realizes a publish-subscribe architecture for continuous queries that extracts information from XML documents stored in a central data warehouse based on user profiles. *Xyleme* [29] monitors changes in HTML or XML documents collected from the internet by a crawler.

3. Query Graph

This section presents the publish-subscribe architecture of PIPES that supports the dynamic construction of directed query graphs during runtime.

Continuous query execution starts with a set of heteroge-

nous data sources. From here, the data flows through intermediate operators composed in the query graph. The operators perform the actual processing and eventually forward the data to consuming applications. We will imagine the *initial sources* being at the bottom of a physical query graph, *terminal sinks* at the top and data flowing strictly upward.

In the context of streaming data, there are several requirements the query processor should met. First, since currently running queries often overlap, system resources should be saved if multiple queries exploit the same physical operators. Hence, the set of all queries is combined in a single directed acyclic query graph where a node can have multiple inputs as well as multiple outputs.

Second, the query graph should support dynamic addition and removal of queries at runtime. In order to avoid expensive restructuring, it might be advisable to keep queries in the graph though they are not currently in use. If a user issues a similar query at a later point in time, only minimal effort is required.

Third, wrappers are required for a seamless integration with persistent data sources such as provided by XXL's cursors.

In this section, we present the basic interfaces of logical graph nodes and introduce our concept of direct interoperability. We demonstrate how to integrate new functionality and provide a brief overview of ready-to-use components such as join operators and wrappers for the integration of streaming data and persistent data.

3.1. Logical Nodes

In conformity to the ONC interface, this subsection focuses solely on the data flow and control flow, while leaving operator functionality and actual implementations open and abstract. A logical query graph consists of three types of nodes: A *source* has multiple outputs, a *sink* has multiple inputs and an intermediate node, a so-called *pipe*, has multiple inputs as well as multiple outputs. This abstraction closely corresponds to the one used in data flow languages [34]. The most important methods of the three interfaces are discussed in the following:

- The interface `Source` (see Fig. 1) can be explained by drawing its analogy to ONC. The method `open` opens a source and allocates resources. The method `close` does the exact opposite. After a call to `open`, elements continuously flow upwards. A method *next*, as known from ONC, does not exist for data streams. The interface `Source` additionally contains a pair of methods `pause` and `resume` for temporarily suspending data reception.
  While implementing XXL, we soon realized that several semantic checks need to be performed in an *init* phase, before `open` can be called. Thus, it is advisable to enhance the original ONC approach. Similar checks

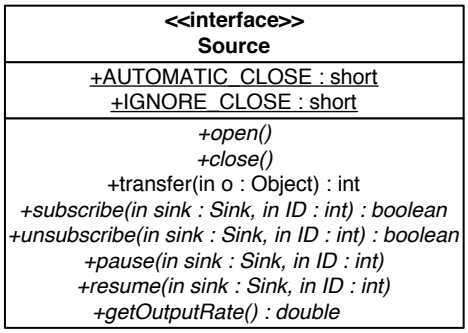

Figure 1. Interface for sources

for data streams require the corresponding subgraph to be built in advance. For this reason, we provide a method `subscribe`, which basically establishes a connection between a sink and this source. Since we also allow the query graph to be re-arranged at runtime, methods for adding and removing queries, without closing the source, are required. These methods are `subscribe`, as described above, and its counterpart `unsubscribe`. The former subscribes new sinks to this source at runtime. The latter cancels the subscriptions without necessarily harming the source.

The reaction of a source to an unsubscription depends on its current closing-mode. If `AUTOMATIC_CLOSE` is set, the source gets closed, if the last sink cancels its subscription. In contrast, `IGNORE_CLOSE` preserves the state of this source for future subscriptions.

In a *strict* ONC architecture it is clear at any point in time, which source is accessed next, while in data stream processing such an implicit scheduling would reduce performance. Because explicit scheduling plays an important role in stream processing, we define a method `transfer` that can be invoked by our scheduler. The method `transfer` delivers the next qualifying element to all subscribed sinks and generally triggers the processing of an element for each sink. This method abstracts from the underlying transportation technique such as queues and sockets.

A source's current output rate is returned by a call to `getOutputRate`. In stream processing, the output rate is an essential criterion to ensure adaptivity. For

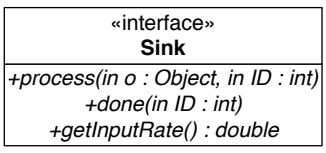

Figure 2. Interface for sinks

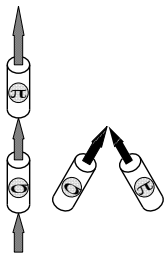

Figure 3. Composition of query graphs

instance, a query plan may be optimized according to output rates [38].

- The interface `Sink` (see fig. 2) consumes the elements of its subscribed sources, which are distinguished by an identifier `ID`. The method `process` handles incoming elements associated with the ID of its source. The ID is required for sinks with multiple sources, e. g. multi-way joins. A `done` call signals that the source has run out of elements. In conformity with the interface `Source`, this interface also allows the measuring of a sink's current input rate.
- Operators represent intermediate nodes in a query graph. We define such an intermediate operator as a `Pipe`. Data flows in at its bottom and streams out at its top. On the one hand, a pipe acts as a source and on the other hand as a sink. Obviously, the interface `Pipe` extends both interfaces `Source` and `Sink`.

Figure 3 depicts a PIPES query graph. Initial sources are located at the bottom; terminal sinks at the top. The nodes in-between represent pipes. Their design enables recursive method invocations up and down the query graph. The methods `open`, `close`, `pause` and `resume` are recursively passed downwards the query graph, while `done` and `transfer` run upwards.

3.2. Direct Interoperability

All research prototypes related to PIPES make use of buffers in-between operators. No matter on which level the operators are scheduled, a certain overhead is created with respect to both, CPU (scheduling) and memory (buffers). This overhead is significant in scenarios with many low-cost operators. PIPES provides a new approach to directly plug operators into each other without any queues in-between. For a contiguous subgraph above a certain source, the processing of the elements is performed in a depth-first manner. When an operator finished processing an element, it immediately hands it over to the next operator, which is instantly executed. This leads to a chain reaction that forwards results to the top sinks of the subgraph. We denote this as *direct interoperability* (DI). DI realizes an implicit depth-first strategy, with no overhead. It basically wraps an entire subgraph into a single virtual operator. The following example illustrates the implementation of DI:

```
public class Filter implements Pipe {
  ...
  public void process(Object o, int ID){
    if (predicate.invoke(o))
      transfer(o);
  }
}
```

In this example, the generic filter operation implements the interface `Pipe` with the desired functionality of `process`. A user-defined predicate is applied to each arriving element. If this predicate is satisfied, the element is immediately transferred. Within the method `transfer` the specified element is forwarded to all subscribed sinks by individually invoking the sinks' method `process`.

3.3. Integration of new Functionality

User-defined functionality can seamlessly be integrated in PIPES. We provide an abstract implementation for each interface, including constructors. This reduces the lines of code to be written enormously while granting extensibility and allows the user to concentrate on the core functionality. The relationships among the interfaces and their corresponding abstract classes are shown in figure 4. The interface `Pipe` extends the interfaces `Source` and `Sink`. The abstract class `AbstractPipe` implements the interface `Pipe` and contains two unilateral associations to the set of sources and the set of sinks to which this operator is connected to. In the following, we discuss the integration of new graph nodes in more detail:

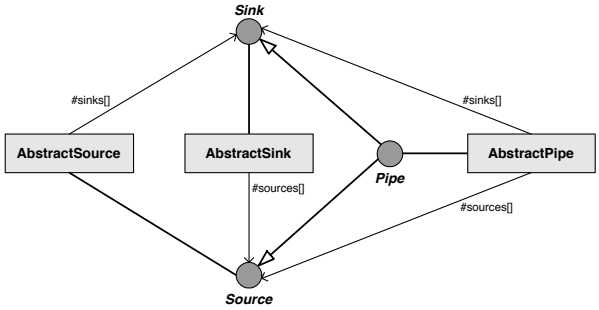

Figure 4. Interface dependencies and abstract classes for physical nodes in a query graph

Initial Source. The class `AbstractSource` contains a pre-implementation of the interface `Source`. Every time a new element is ready, a simple call to `transfer` suffices to send it upwards in the graph. In addition to invoking `transfer`, a developer solely needs to implement `open` and `close`. All further methods such as `subscribe`, `transfer` and `pause` etc. are readily implemented.

Terminal Sink. Various kinds of sinks may be considered, e.g. incoming data can be sent over a network or is presented to a user via a graphical user interface. The class `AbstractSink` represents a default implementation of the interface `Sink` in analogy to the class `AbstractSource`. In order to specify the user-defined processing steps and the release of resources, the methods `process` and `done` have to be overwritten. `Process` contains the functionality of the sink, while `done` defines the behavior when an underlying source has run out of elements. In the latter case, the receiving application might write into a log or inform an administrator.

Operator. Developers can construct their own operators from the class `AbstractPipe` or use existing operators as templates. The abstract class includes the subscription and temporary inactivation mechanisms as well as the computation of input and output rates. The method `done` received a default implementation that simply forwards the call to all subscribed sinks. The method `transfer` is also readily implemented and behaves analogously.
When describing DI above, we have shown a snippet of code from an individually implemented filter. In the following code section, using `AbstractPipe`, we'll show, that this snippet is about all that is needed:

```
public class Filter extends AbstractPipe
{
  protected  Predicate predicate;

  public Filter(Source s, int ID,
                Predicate p) {
    super(s, ID);
    this.predicate = p;
  }

  public void process(Object o, int ID){
    if (predicate.invoke(o))
      transfer(o);
  }
}
```

3.4. Ready-To-Use Functionality

In addition to interfaces and abstract classes, PIPES provides a rich collection of fully implemented and ready-to-use components.

Initial Sources. Real world users of PIPES will implement an extension to `AbstractSource` to feed data streams into a PIPES query graph. Initial sources typically run in their own thread.
For testing and experiments, PIPES provides synthetic initial sources. These run a timer that allows to model variable delays between two successive elements. In addition to constant output rates, we derived a timer that determines the delay between two successive elements in its output stream according to a Poisson distribution. This proved to be powerful enough to reproduce the experiments on synthetic network traffic by [38].
Moreover, we implemented another timer that generates massive fluctuations in a source's output rate and particularly bursts. The delays between two elements are based on a uniformly distributed random number in $(0, T]$, where T denotes an upper bound in milliseconds. By this delay, we model varying output rates. After a fixed time interval expired, we change the output rate by generating a new random delay. This feature enabled us to validate our framework with regard to scalability and availability.

Furthermore, we have implemented wrappers that feed data from persistent data sets, such as relations, into a data stream. This topic will be addressed in subsection 3.5.

Terminal Sinks We have implemented a preliminary interactive graphical interface, which lets users start, stop, pause and resume queries. For direct observation, results are written to a text window. Additionally, PIPES contains a terminal sink that appends incoming elements to a `java.io.PrintStream`. For this class, Java provides a rich functionality for further processing, storing and displaying.

Operators The most powerful part of PIPES is its collection of readily implemented operators. These are strictly generic, because they are parameterized by higher order functions and predicates [5]. For instance, we provide *filter*, *map*, *join*, *difference*, *online aggregation*, *reservoir sampling*, *group* and *sort* functionality. Blocking operators do not make sense in the context of continuous data streams. In order to migrate their functionality, we introduce an abstraction of sweep-line status structures, called `SweepArea` [10]. The latter also comes in handy, whenever memory can be traded for accuracy.

In the following, we exemplarily discuss the join operator with the intention to illustrate the generic operator de-

sign in PIPES.
The construction of a join generally uses the following parameters: two input streams, two `SweepArea`s and a binary function that creates the results.
Thereby, the `SweepArea`s determine the type of a join. Each input stream is associated with exactly one `SweepArea`. A `SweepArea` represents a status-based data structure with efficient support for insertion, retrieval and continuous reorganization. The interface described in [10] represents an abstraction of the sweep-line status structures introduced in [16, 17]. Related approaches have been published in [25, 32].

Our generic join framework is based on the ripple join [20] and extends it towards data-driven processing. Each incoming element is inserted into its stream's `SweepArea` and queried against the other. Thereafter, the results are appended to the output stream. Finally, the `SweepArea` corresponding to the input stream is reorganized.

We support different implementations of `SweepArea`s, for example hash- or list-based. Moreover, a `SweepArea` can be realized as a memory-adaptive synopsis, commonly restricted to a sliding window. Regarding to `SweepArea`s, PIPES supports symmetric joins as well as asymmetric joins. If both `SweepArea`s are hash-based, we receive a symmetric hash-join [39]. Asymmetric joins [24] are created by specifying one hash-based `SweepArea` and one list-based `SweepArea`, which may additionally realize sliding windows. Hash-based, memory-adaptive `SweepArea`s using external memory enable us to emulate the double-pipelined hash-join [21] as well as its extension, the X-Join [36].

As an example of our generic approach, we present a symmetric hash-join:

```
new Join(
  new RandomNumber(
      RandomNumber.DISCRETE, 15
  ),
  new RandomNumber(
      RandomNumber.DISCRETE, 25
  ),
  HashBagSweepArea.FACTORY_METHOD,
  Tuplify.DEFAULT_INSTANCE
)
```

The input streams deliver uniformly distributed random numbers at equidistant time delays of 15 and 25 milliseconds. By using a factory method [18], two hash-based `SweepArea`s, called `HashBagSweepArea`, are constructed. For the creation of resulting tuples, a default instance of the binary function `Tuplify` is passed.

This generic design principle of joins allows a natural extension towards new join functionality. Moreover, we use `SweepArea`s in various implementations of other operators, such as *difference* and *distinct*.

3.5. Data Flow Translation

XXL includes a package `xxl.cursors` with a rich collection of demand-driven operators that extend the interface `java.util.Iterator` [5]. The package `xxl.pipes` extends XXL's functionality towards data-driven processing. Both packages can be used separately or combined. Their combination is based on special wrappers that put data *flow translation* control operators into practice [19].
In order to switch from a demand-driven flow to a data-driven flow, a thread periodically consumes the next element of the underlying passive source and forwards it to all subscribed sinks.
Transforming a data-driven flow into demand-driven flow requires an operator that buffers the incoming elements until these are explicitly requested.
PIPES contains both wrappers, called `CursorSource` and `SinkCursor`. These adapters allow to wrap a cursor as an initial source and a terminal sink as a cursor, respectively.
Because of these powerful adapters, XXL supports arbitrary joins of persistent sources, such as relations, and data streams. This feature has been considered as an import aspect for query processing [23, 9, 4].
Not only can we now access both kinds of data, but we can also make use of the whole range of different functionality that has been implemented in XXL over the past years. This enables us to rebuild, validate and significantly enhance components proposed by other developers.
For example, we can rebuild the concept of connection points, as introduced in [11], and substitute the original B-trees with multi-dimensional indices at virtually no extra cost. The same applies to bulk-loading functionality as proposed in [7].

4. Hybrid Multi-Threaded Scheduling

In this section, we present our *hybrid multi-threaded scheduling* (HMTS). It encompasses the two concepts lined out in the introduction (OTS, GTS) as well as the direct interoperability presented in section 3.2. Thus, our architecture consists of three layers, of which two allow an explicit scheduling, while DI runs the already discussed implicit depth-first scheduling.
We present our queueing operator that performs the necessary decoupling of ordinary operators for the former two concepts. Eventually, we identify arising synchronization problems and outline our solutions.

Our scheduling concept offers several advantages. On the one hand, we allow parts of the query graph to run in extra threads, and thus make the processing of elements non-atomar. Therefore, the execution of a slow operator cannot stall the rest of the query graph, like in GTS. This is a fundamental prerequisite for the production of stable output rates. On the other hand, we do not necessarily assign a new thread to each operator (OTS), because our experimental studies showed in compliance with [3, 12, 26] that an extensive thread usage causes loss in performance.
HMTS allows to dynamically adapt the number of threads and to keep the balance between concurrency and overhead. Obviously, we can easily rebuild the two extreme cases.

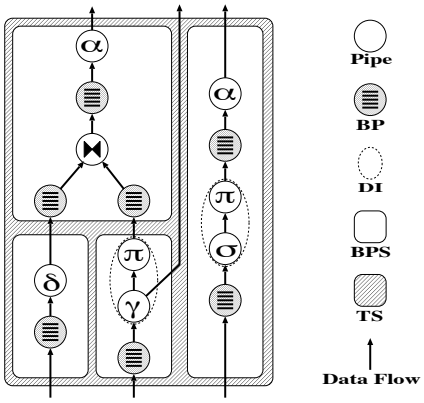

Figure 5. Hybrid Multi-Threaded Scheduling

4.1. Decoupling

For any explicit scheduling, the decoupling of operators by using queues becomes a must. Our queuing-operator `BufferPipe` (BP) realizes a buffering between two successive nodes. In the context of several threads, this operator behaves similar to the *Exchange* operator [19]. Interestingly enough, this single class serves for decoupling on both levels: between operators running in different threads as well as between operators running in the same thread.
BPs can be dynamically inserted and removed at runtime. Only after an initial source, decoupling is always necessary, because the independent execution of the source's thread has to be guaranteed.

4.2. Three-Level Architecture

In this subsection, we present the details of the three-level architecture of HMTS. On the first level, our architecture consists of physical operators and virtual operators. Virtual operators group physical operators forming a subgraph through direct interoperability, using the implicit scheduling strategy.

On the second level, subgraphs of first-level operators are constructed where a BP is inserted in front of each node. At this level, explicit scheduling becomes possible. This is achieved by associating a `BufferPipeScheduler` (BPS) to the BPs of a subgraph. The BPS schedules the operators in the subgraph by calling their BP's `transfer` method. It entirely hands control to the operator and waits for its return. The strategy, by which BPS selects the operator being executed next, is left open to a specific implementation of a strategy pattern. This unit, which consists of a BPS and its associated BPs, run in a single thread.

The third level is concerned with running multiple second-level units concurrently. Concurrency is managed by a high-priority thread, called `ThreadScheduler` (TS). This concept breaks up atomic execution inherent to second-level units. Again, specific strategies by which TS assigns CPU cycles are current work and beyond the scope of this paper.

Figure 5 illustrates our scheduling architecture by showing possible relations between the different components. It depicts the partitions of a query graph corresponding to second-level units that are scheduled by a TS. Within these units, a BPS schedules first-level operators. In case of virtual operators, the underlying physical operators are shown as well.

The default `ThreadScheduler`, implemented in PIPES, accomplishes a preemptive priority-based scheduling strategy. It determines the next thread to be executed ensuring that starvation is prevented. The distribution of CPU resources results from priorities (weights) that can be adapted during runtime. The implementation of the TS is basically orientated on the scheduling concept for Java introduced in [30]. Thereby, the basic idea is to associate a high Java priority to the scheduler, which guarantees its preferential execution. The BPs are primarily equipped with a lower thread priority. In order to execute the next thread, its Java priority is temporarily increased.
Depending on the Java Virtual Machine (JVM), the scheduling may conflict with the mapping of Java threads to operating system processes. The *Green-Thread* model ensures that all threads run in the same system process and obey our TS. In the *Native Thread Model*, threads are mapped to separate system processes and therefore the scheduling can interact with that of the underlying operating system. There are also commercial JVMs like BEA's JRockit, that allow m:n mappings between threads and system processes.

4.3. Synchronization

Each approach realizing concurrency has to deal with synchronization aspects. Related projects implementing OTS synchronize on their communication queues. The latter however do not necessarily exist in a PIPES query graph,

namely when direct interoperability is used. Therefore, PIPES is equipped with a different synchronization mechanism ensuring consistency.
We differ two levels of synchronization: At the *Thread-level*, the execution of several threads has to be managed. At the *Variables-level*, the access to shared variables needs to be controlled.
We generally obey the concept of object-related synchronization provided with Java's Thread API. This includes synchronized access to code sections (*mutual exclusion*) by monitoring object locks. The communication between threads is realized with Java's `wait-notify` mechanism integrated in the superclass `Object`, which avoids polling and saves CPU resources.

In the following, we discuss the relevant synchronization aspects for PIPES.

THREAD-LEVEL. On the thread-level, we distinguish between two kinds of synchronization:

- *Inter-node synchronization*
  The following concurrency issues can occur between successive nodes in an acyclic directed query graph:

  1. Multiple sources access the same sink at the same time. Due to our inter-node communication process, this can happen for example within a union or join operator. This problem is most commonly caused by data flow mechanisms. In addition, methods like `done` can trigger such an event.
  2. Different sinks access the same source at the same time. Therefore, methods related to the control flow of a source in a query graph require additional synchronization.
  3. In the hybrid case, a source and a sink access a pipe at the same time. Thereby, conflicts arise that rely on the converse directions in data flow and control flow.

  Initial object-related synchronization via locking an entire node would solve the concurrency problems only in the first two cases. In the hybrid case this locking cannot avoid the following deadlock constellation: While the methods `done` and `transfer` run upwards, all others related to the control flow descend the query graph recursively. If locks on two connected nodes are acquired, a mutual exclusion may appear, if both nodes try to fetch locks on each other at the same time. This constellation can occur for example, if the two different kinds of control flow integrated in our architecture are aligned conversely, i.e., if a `done` call moves bottom up while a `close` call is invoked top down in a query graph. The methods `transfer` and `close` may cause the same effect, because data flow and control flow also point in reverse directions.

  We solved these problems using a fine-grained synchronization mechanism. We do not lock an entire node but monitor the access to its referenced sources and sinks. Therefore, we introduce two explicit lock objects. A source contains a lock object that has to be acquired, if its set of subscribed sinks is accessed. Analogously a sink owns a lock object for its sources.
  In the first of the three cases listed above, the methods `process` and `done` have to be synchronized. This can be achieved by requesting the lock associated with the sources of a sink. In the second case concurrent access is controlled by acquiring the lock associated with the sinks of a source. For example, the methods `subscribe`, `unsubscribe` and `open` are synchronized this way.
  The hybrid case needs both locks, but must not use them in a nested way. To avoid locking an entire node and thus producing a deadlock, a fixed order for acquiring both locks is enforced. This sequential locking implies that operators are able to delegate method calls running in both directions in a query graph. Mutual dependencies are resolved, because different nodes initially acquire different locks and release them before acquiring the next.

- *Intra-node synchronization*
  Another important aspect of synchronization is related to threads used inside a node. These internal threads may conflict with outer threads accessing inner-node components.
  Inside a BP two threads may interact on its buffer. While the BPS consumes the buffer and transfers its elements, another thread may insert elements. Consistency is achieved by object-related synchronization within our implementation. In order to avoid busy waiting, we use Java's built-in `wait-notify` mechanism.
  The X-Join [37] serves as another example, where a separate thread is needed. This thread swaps elements between main and external memory with regard to the current system load. Thereby, the threads transferring elements to the join operator and the thread used for internal maintenance have to be synchronized.

VARIABLES-LEVEL. In order to avoid inconsistent states, synchronization needs to be taken into account at the variables-level.
A source stores references to all subscribed sinks in an internal array. Analogously, a sink manages its sources in an array, whereas an operator contains both arrays (see fig. 4). In addition, we store separate arrays with IDs that correspond to the connected nodes. These are used for internal

communication purposes (see interface definitions in fig. 1 and fig. 2). The separate usage of IDs becomes necessary and profitable in the context of distributed query processing and self-joins, because passing references in the `process` method is not satisfying. Both types of internal arrays, reference and ID array, require an explicit synchronization with the intention to control concurrent read and write access. This is implicitly realized with our presented object lock mechanism.

Measuring the input rate and output rate requires additional attributes in each node, e.g. counters, which also need to be synchronized. In particular, updates of internal status attributes may conflict with concurrent read access.

In addition to the locking mechanisms discussed above, we had to deal with cache effects of the JVM, when implementing PIPES. We consider thread termination as example. A thread is terminated with the help of a boolean attribute that triggers the termination. This attribute is typically set by other threads, e.g. the inner thread of a BPS is terminated after a `close` call. The access to the boolean termination attribute either has to be synchronized including read and write access or it has to be signed with Java's modifier `volatile`. Both solutions update a thread's cache and enforce an explicit reload of the attribute every time this is accessed. In PIPES, we prefer the latter.

The issues discussed above outline that thread-safe programming plays an important role in an adequate and performant processing of multiple data streams. The architecture of PIPES maintains consistency with an explicit synchronization. Our experiments show that the overhead is almost negligible.

5. Experimental Evaluation

In this section, we present a preliminary experimental evaluation. From our set of experiments, we report four showing that DI, OTS as well as GTS do not perform optimal by their own. Our presented hybrid scheduling architecture (HMTS) outperforms these single solutions, if the query graph is partitioned appropriately.

Our experiments rely on synthetic data sets, because testing our architecture requires only streams of arbitrary objects. In order to allow total runtime comparisons, we used data streams with a finite number of elements. The hardware setup involved a Pentium IV with 2.4 GHz and 1 GB main memory. All experiments were performed on Windows XP running Sun's Server JVM 1.4.1.

In our first experiment, we show that direct interoperability (DI) may induce a loss in performance. For a join setting, we demonstrate the necessity of decoupling. Each input of the binary symmetric hash join (SHJ) and the symmetric nested loops join (SNJ) consists of 40,000 uniformly distributed random numbers within range $[0, 10^5]$. In order to simulate real bursty traffic, the inter arrival rate between two successive elements follows a Poisson distribution according to the experimental setup in [3]. Figure 6 depicts the input rates and output rates of the two equi-join algorithms. Each join operator directly runs in the thread of its autonomous data sources. As illustrated, SHJ does not slow down any of its inputs. Due to the linear increasing size of the `SweepAreas`, SNJ breaks down after circa 40 seconds. Therefore, the data flow between the source and the SNJ has to be decoupled, while SHJ still can run directly without any data loss.

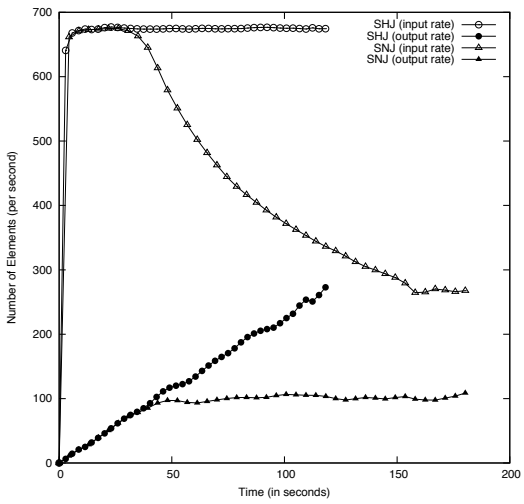

Figure 6. Dependencies between input rate and output rate of different join algorithms

In the next experiment, we point out that the synchronization overhead in PIPES is almost negligible. Additionally, we compare four different data processing techniques regarding to their runtime. We compare the direct data-driven interoperability of PIPES with the concept of running each operator in a separate thread (OTS). Furthermore, we consider the demand-driven counterparts using `xxl.cursors`. Because a cursor processes passive data sources in an ONC style, we differ between two settings: a pure demand-driven execution based on passive sources and a demand-driven execution applied to wrapped autonomous data sources, i.e., a `SourceCursor` is prepended.

We measured the performance according to the following query: $\alpha(\pi(\bowtie(\sigma(A), B)))$. The 10^6 elements of each data source, A and B, are uniformly distributed over $[0, 10^6]$. A filter σ is firstly evaluated over A with selectivity 0.5. The results are joined with input B in a symmetric hash equi-join $\bowtie$. Finally, the standard deviation of the first attribute of the join results is computed by an online aggregation α. Figure 7 shows the number of results over time, while each query is executed as fast as possible, i.e., with-

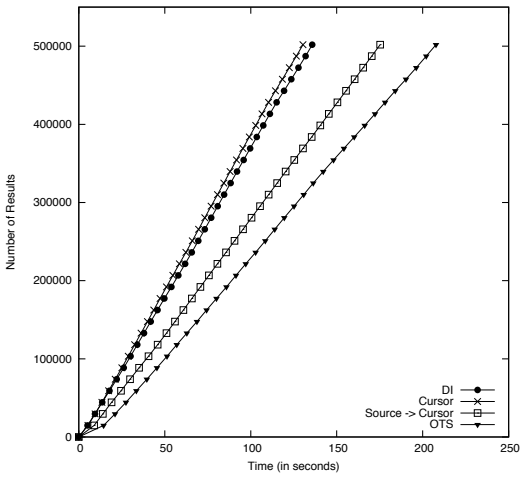

Figure 7. Performance of different data processing techniques

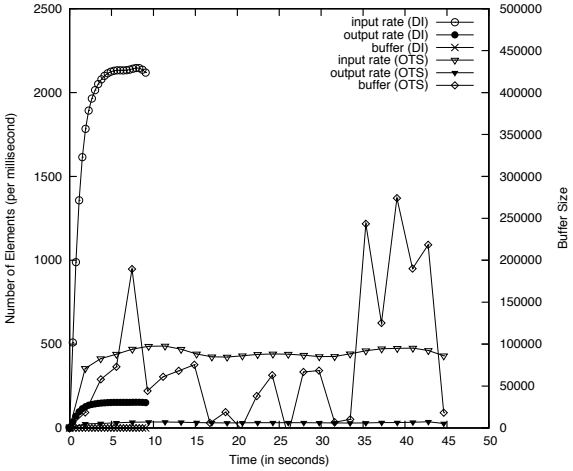

Figure 8. Performance of DI vs. OTS

out any synthetic delays. Benefiting from its passive data source, purely demand-driven processing (`Cursor`) outperforms all other techniques. In this case, no synchronization is needed because the query runs in a single thread. The marginal overhead for concurrency and synchronization compared to PIPES is only about 5%, generally almost negligible. In OTS, the performance decreases significantly due to buffer-related overhead. Even using a wrapper transforming a `Source` into `Cursor` (`Source` $\rightarrow$ `Cursor`) runs rather slowly.

Our third experiment (see fig. 8) emphasizes the inherently reduced memory requirements and significantly shorter runtime of DI in contrast to OTS. While OTS heavily relies on communication via queues, a high output rate of a source causes performance losses and tremendous increases in memory usage. The query is a sequence of operators $\alpha(\sigma(\gamma(A)))$ consisting of a hashed group operation γ, a selection σ and an online aggregated average α. The autonomous data source A delivers $2 \cdot 10^7$ elements uniformly distributed over $[0, 2 \cdot 10^7]$. If the query is processed with an average input rate of almost 2,000 elements per millisecond, DI still needs no buffering. Contrarily, OTS buffers up to $3 \cdot 10^5$ elements for a moderate input rate of 500 elements per millisecond. Moreover, the total output rate is by a factor of 4 lower in comparison to DI.

The fourth experiment demonstrates the necessity of multi-threading and shows the advantages of HMTS in contrast to GTS. We posed a query with two successive filter operations having the selectivity $9 \cdot 10^{-4}$ and 0.3, respectively. We simulated a complex predicate by introducing a time delay of 5 milliseconds every time the second filter's predicate is evaluated. Thus, an atomar processing of an element implies a significant stall of the other operators.
The tests rely on an autonomous data source, which transfers a total of $5 \cdot 10^6$ elements uniformly distributed in the range $[0, 5 \cdot 10^6]$. In our first setting (HMTS), we decoupled the data flow twice, between the source and the first filter as well as between the filters. In the second setting (GTS), we inserted buffers in-between two successive nodes and processed the buffers via a round-robin scheduler running in a separate thread as proposed in [3, 12, 26]. As depicted in fig. 9, HMTS shows a marginal shorter runtime and a significantly smaller memory usage. In the case of GTS, the atomic processing causes an enormous increase of the buffer size.
Priority based strategies like Chain [3] reduce memory requirements, but suffer from increased runtime. While our experiments rely on round-robin scheduling, we also support priority based scheduling strategies, but these are beyond the scope of this paper.

The results of our experiments demonstrate that the separate usage of DI, OTS and GTS is not optimal. The tradeoff between the advantages of concurrency and the resulting overhead concerning synchronization significantly outlines that our hybrid approach (HMTS) outperforms the three other solutions, if the query graph is partitioned appropriately. The experiments show the benefits of HMTS with regard to CPU and memory resources. Moreover, HMTS allows to scale concurrency and DI at runtime.
For sake of clearness, we presented experiments comprising of a small number of operators. More complex query graphs will result in even greater performance benefits.

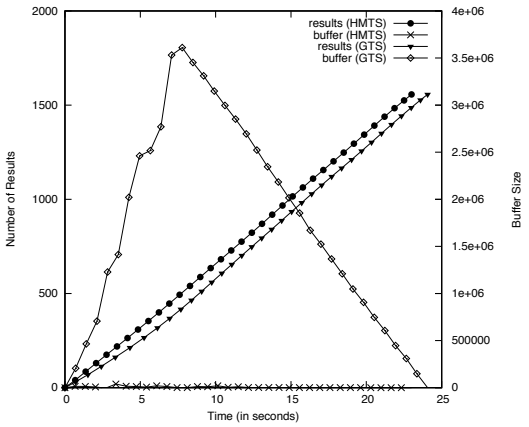

Figure 9. HMTS in comparison to GTS

6. Conclusion and Future Work

The processing of autonomous data sources requires adequate models for handling data streams. The characteristics of data streams imply a reconsideration and modification of traditional query processing techniques.
We presented a novel framework, PIPES, with an inherent publish-subscribe mechanism that allows resource sharing. Starting from query plans, an uni-directional query graph is constructed.

We developed a three level architecture for our hybrid multi-threaded scheduling (HMTS). On the first level, operators are directly connected forming virtual operators. This enables direct interoperability. On the second level, virtual operators are combined inserting buffers in-between. Separate threads control the so formed subgraphs. Finally, on the top level, a scheduler enables concurrent execution of these subgraphs.
This architecture includes both extremes of thread assignment: Using a separate thread for each operator (OTS) or a single thread for the complete query graph (GTS). By placing HMTS in-between these two extremes, we intend to reduce the number of concurrently running threads, while still benefiting from concurrency. Moreover, this multi-threaded dynamic scheduling allows to adaptively scale concurrency and consequently saves system resources as validated by our experiments.
Because in HMTS several threads may run concurrently, explicit synchronization is inevitable. We have identified the arising synchronization problems and discussed our solutions.

PIPES is seamlessly integrated into our freely available Java library XXL. Therefore, it serves as an easily adaptable testbed for various stream processing algorithms as well as their interaction with demand-driven ones. The components of this library act as fundamental and generic building blocks for implementations of data stream management systems. Experimental evaluations validated the usability and excellent performance of our architecture. PIPES has already been applied in the context of online clustering and data mining on continuous streams [8].

The current focus of our work revolves around three questions:

- How to assign the operators to the three layers of HTMS?
- How to schedule on the top two layers (in the presence of QoS)?
- How to go about reassignment at runtime?

In addition, we develop a central memory manager that continuously distributes memory at runtime. Moreover, our join framework is extended to include multi-way joins as well as sort-based implementations. In general, we strive for building a dynamic query optimizer.

The fully documented implementation of our architecture PIPES is available under [31].

Acknowledgements
This work has been supported by the German Research Society (DFG) under grant no. SE 553/4-1.

References

[1] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 53–64. Morgan Kaufmann, 2000.
[2] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. of the ACM SIGMOD*, pages 261–272. ACM Press, 2000.
[3] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In *Proc. of the ACM SIGMOD*, pages 253–264, 2003.
[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Symp. on Principles of Database Systems*, pages 1–16, 2002.
[5] J. Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger. XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 39–48, 2001.
[6] J. Bercken, J.-P. Dittrich, and B. Seeger. javax.XXL: A prototype for a Library of Query processing Algorithms. In *Proc. of the ACM SIGMOD*, page 588, 2000.
[7] J. Bercken and B. Seeger. An Evaluation of Generic Bulk Loading Techniques. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 461–470, 2001.
[8] J. Beringer and E. Hüllermeyer. Online Clustering of Data Streams. Technical report, University of Marburg, 2003. CS-31 (submitted for publication).

[9] P. Bonnet, J. Gehrke, and P. Seshadri. Towards Sensor Database Systems. In *Proc. of 2nd Int. Conf. on Mobile Data Management*, pages 3–14, 2001.

[10] M. Cammert, C. Heinz, J. Krämer, and B. Seeger. A Status Report on XXL - A Software Infrastructure for Efficient Query Processing. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2003.

[11] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring Streams: A New Class of Data Management Applications. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 215–226, 2002.

[12] D. Carney, U. Cetintemel, S. Zdonik, A. Rasin, M. Cerniak, and M. Stonebraker. Operator Scheduling in a Data Stream Manager. In *Proc. of the Conf. on Very Large Databases (VLDB)*, 2003. (to appear).

[13] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, 2003.

[14] S. Chandrasekaran and M. J. Franklin. Streaming Queries over Streaming Data. In *Proc. of the Conf. on Very Large Databases (VLDB)*, 2002.

[15] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of the ACM SIGMOD*, pages 379–390, 2000.

[16] J.-P. Dittrich and B. Seeger. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *Proc. of the IEEE Conference on Data Engineering*, pages 535–546, 2000.

[17] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer. Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 299–310, 2002.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[19] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[20] P. J. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation. In *Proc. of the ACM SIGMOD*, pages 287–298, 1999.

[21] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. In *Proc. of the ACM SIGMOD*, pages 299–310, 1999.

[22] Z. G. Ives, A. Y. Levy, D. S. Weld, D. Florescu, and M. Friedman. Adaptive Query Processing for Internet Applications. Overview paper, University of Washington, INRIA Rocquencourt, Viathan Corp., 2002.

[23] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View Maintenance Issues for the Chronicle Data Model. In *Symp. on Principles of Database Systems*, pages 113–124, 1995.

[24] J. Kang, J. Naughton, and S. Viglas. Evaluating Window Joins over Unbounded Streams. In *Proc. of the IEEE Conference on Data Engineering*, 2003.

[25] W. Li, D. Gao, and R. T. Snodgrass. Skew Handling Techniques in Sort-Merge Join. In *Proc. of the ACM SIGMOD*, pages 169–180, 2002.

[26] S. Madden and M. J. Franklin. Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data. In *Proc. of the IEEE Conference on Data Engineering*, 2002.

[27] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olsten, J. Rosenstein, and R. Varma. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, 2003.

[28] J. Naughton, D. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24(2):27–33, 2001.

[29] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the Web. In *Proc. of the ACM SIGMOD*, pages 437–448, 2001.

[30] S. Oaks and H. Wong. *Java Threads*. O' Reilly, 1999.

[31] Philipps University of Marburg: The Database Research Group. XXL – eXtensible and fleXible Library. URL: http://www.mathematik.uni-marburg.de/DBS/xxl/, 2003.

[32] V. Raman, A. Deshpande, and J. Hellerstein. Using State Modules for Adaptive Query Processing. In *Proc. of the IEEE Conference on Data Engineering*, 2003. (to appear).

[33] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 469–478, 1991.

[34] R. Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7):491–541, 1997.

[35] M. Sullivan and A. Heybey. Tribeca: A System for Managing Large Databases of Network Traffic. In *In Proc. of the USENIX Annual Technical Conference*, pages 13–24, 1998.

[36] T. Urhan and M. J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[37] T. Urhan and M. J. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 501–510, 2001.

[38] S. D. Viglas and J. F. Naughton. Rate-Based Query Optimization for Streaming Information Sources. In *Proc. of the ACM SIGMOD*, pages 37–48, 2002.

[39] A. N. Wilschut and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proc. of the First Int. Conf. on Parallel and Distributed Information Systems (PDIS 1991)*, pages 68–77, 1991.