# Adaptive Ordering of Pipelined Stream Filters*

Shivnath Babu
Stanford University

Rajeev Motwani
Stanford University

Kamesh Munagala
Stanford University

Itaru Nishizawa
Hitachi, Ltd.

Jennifer Widom
Stanford University

{shivnath,rajeev,kamesh,widom}@cs.stanford.edu, itaru@crl.hitachi.co.jp

## ABSTRACT

We consider the problem of *pipelined filters*, where a continuous stream of tuples is processed by a set of commutative filters. Pipelined filters are common in stream applications and capture a large class of multiway stream joins. We focus on the problem of ordering the filters adaptively to minimize processing cost in an environment where stream and filter characteristics vary unpredictably over time. Our core algorithm, *A-Greedy* (for *Adaptive Greedy*), has strong theoretical guarantees: If stream and filter characteristics were to stabilize, A-Greedy would converge to an ordering within a small constant factor of optimal. (In experiments A-Greedy usually converges to the optimal ordering.) One very important feature of A-Greedy is that it monitors and responds to selectivities that are correlated across filters (i.e., that are nonindependent), which provides the strong quality guarantee but incurs run-time overhead. We identify a three-way tradeoff among provable convergence to good orderings, run-time overhead, and speed of adaptivity. We develop a suite of variants of A-Greedy that lie at different points on this tradeoff spectrum. We have implemented all our algorithms in the STREAM prototype Data Stream Management System and a thorough performance evaluation is presented.

## 1. INTRODUCTION

Many modern applications deal with data that is updated continuously and needs to be processed in real-time. Examples include network monitoring, financial monitoring over stock tickers, sensor processing for environmental monitoring or inventory tracking, telecommunications fraud detection, and others. These applications have spurred interest in general-purpose stream processing systems, e.g., [5, 6, 11, 26]. A fundamental challenge faced by these systems is that data and arrival characteristics of streams may vary significantly over time [6]. Since queries in stream systems frequently are long-running, or *continuous* [8, 25], it is important to consider *adaptive* approaches to query processing [1]. Without adaptivity, performance may drop drastically over time as stream and other characteristics change.

*This work was supported by the National Science Foundation under grants IIS-0118173 and IIS-9817799. Author Munagala is supported by grant NIH 1HFZ465.


*SIGMOD 2004* June 13-18, 2004, Paris, France.


This paper addresses the processing of data streams whose characteristics vary unpredictably over time. We focus in particular on streams processed by a set of commutative *filters*, where overall processing costs depend on how the filters are ordered, and the best orderings are dependent on current stream and filter characteristics. Commutative filters capture many stream applications, including a wide class of stream joins. Let us consider an example drawn from the network monitoring domain.

*Example 1.* Stream processing systems can be used to support traffic monitoring for large networks such as the backbone of an Internet Service Provider (ISP) [11, 13]. A sample continuous query in this domain is [13]:

> *Monitor the amount of common traffic flowing through four routers, $R_1$, $R_2$, $R_3$, and $R_4$, in the ISP's network over the last 10 minutes.*

A network analyst might pose this query to detect service-level agreement violations, to find opportunities for load balancing, or to monitor network health [13].

Data collection in the routers [13] feeds four streams which for convenience we also denote $R_1$, $R_2$, $R_3$, and $R_4$. Each stream tuple contains a packet identifier *pid*, the packet's *size*, and its destination *dest*. Figure 1 depicts an execution strategy for this query. Each stream feeds a 10-minute sliding window, $W_i$ for $R_i$, with a hash index on *pid* [22]. Tuples are inserted and deleted from these windows as data arrives and time advances.

When a tuple $t$ is inserted into $W_1$, the other three windows are probed with $t.pid$ in some order, e.g., the order $W_3$, $W_4$, $W_2$ is used in Figure 1. If all windows contain a tuple matching $t.pid$, then $t$ is sent as an insertion to the aggregation operator *sum(size)*. Otherwise, processing on $t$ stops at the first window that does not contain a matching tuple. Similar processing occurs for deletions from $W_1$, and for insertions and deletions to the other three windows (not shown in the figure). Note that each $W_i$ has its own order for probing the other windows. □

The stream of insertions and deletions from each window in Figure 1 is processed by a conjunction of filters in a pipelined fashion. In this example, each filter is an index-lookup on another window, which returns true if a matching tuple is found. Pipelined filters are very common in stream applications [14, 29]. In general, filters can vary considerably in cost and complexity, e.g., simple predicates like *dest* = "12.34.56.78", index-lookups on windowed streams as in our example, *longest-prefix-match* joins with stored tables [11], regular-expression matching [11], expensive user-defined functions like *domain*(*dest*) = "yahoo.com", and others. Pipelined filters also capture a large class of multiway joins over streams as we will show in this paper.

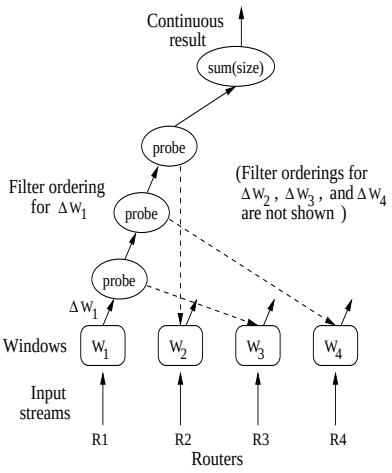


**Figure 1: Pipelined filters for updates to $W_1$**

Overall processing costs can vary widely across different filter orderings. In our example, suppose most of the traffic flowing through router $R_1$ comes from $R_3$ or $R_4$, and rarely comes from $R_2$. Then, an ordering that begins with $W_2$ is preferred for processing tuples from $W_1$ since $W_2$ will "drop" as quickly as possible $W_1$ tuples that will be dropped eventually.[1] Further suppose that most traffic that flows through $R_2$ and $R_1$ comes from $R_3$, and rarely comes from $R_4$. Then, the ordering $W_2, W_4, W_3$ for processing tuples from $W_1$ will have lower processing cost than the ordering $W_2, W_3, W_4$.

## 1.1 Filter Ordering

Our example motivates the problem of selecting good orderings for pipelined filters. This problem poses several challenges:

1. Ordering decisions are based on filter selectivities, and as we have seen, selectivities across filters may be *correlated*, i.e., nonindependent. In traditional database systems, correlations are known to be one of the biggest pitfalls of query optimization [9, 30]. We are able to make strong theoretical claims about our ordering algorithms, and achieve good performance in practice, by monitoring and taking into account correlated filter selectivities. To our knowledge this work is the first to give polynomial-time algorithms with provable quality guarantees for ordering correlated filters and joins.

2. For $n$ filters, an exhaustive algorithm must consider $n!$ possible orderings. This approach may be feasible for one-time optimization and small $n$, but it quickly becomes infeasible as $n$ increases, and also is impractical for an online adaptive approach. We present a *greedy ordering algorithm* that is provably within a small constant factor of optimal for our cost model, and that nearly always finds the optimal orderings in practice.

3. Stream data and arrival characteristics may change over time [6]. In addition, since we are considering arbitrary filters which could involve, for example, joins with other data or network communication, filter selectivities and processing times also may change considerably over time, independent of the incoming stream characteristics. Overall, an ordering that is optimal at one point in time may be extremely inefficient later on. Our adaptive approach for handling this problem is discussed next.

[1]Note that if a packet $p$ flows from router $R_1$ to $R_2$, then the tuple corresponding to $p$ in stream $R_1$ will be dropped during filter processing, but it remains in $W_1$ and will later match with the tuple corresponding to $p$ in stream $R_2$.

## 1.2 Adaptive Ordering of Filters

Even with an effective greedy algorithm for filter ordering, we face additional challenges in the continuous query environment when stream and filter characteristics may change considerably during the lifetime of a query.

- **Run-time overhead:** Suppose we have established a filter ordering based on current data and processing-time statistics. To adapt to changes in stream and filter characteristics, we must monitor continuously to determine when statistics change. In particular, we must detect when statistics change enough that our current ordering is no longer consistent with the ordering that would be selected by the greedy algorithm. Since the greedy algorithm is based on correlated filter selectivities, in theory we must continuously monitor selectivities for all possible orderings of two or more filters to determine if a better overall ordering exists. We provide a variety of techniques for efficiently approximating these statistics.
- **Convergence properties:** An adaptive algorithm continuously changes its solution to a problem as the input characteristics change. Thus, it is difficult to prove anything concrete about the behavior of such an algorithm. One approach for making claims is to prove *convergence properties* of an adaptive algorithm: Imagine that data and filter characteristics stabilize; then the adaptive algorithm would converge to a solution with desirable properties. Our core adaptive algorithm, *A-Greedy* (for *Adaptive Greedy*), has good convergence properties: If statistics were to stabilize, A-Greedy would converge to the same solution found by the static greedy algorithm using those statistics, which is provably within a small constant factor of optimal.
- **Speed of adaptivity:** Even if an adaptive algorithm has good convergence properties, it may adapt slowly. In reality, statistics may not stabilize. Thus, an algorithm that adapts slowly may constantly lag behind the current optimal solution, particularly in a rapidly-changing environment. On the other hand, an algorithm that adapts too rapidly may react inappropriately to transient situations. We show how A-Greedy balances adaptivity and robustness to transient situations.

It turns out we face a three-way tradeoff: Algorithms that adapt quickly and have good convergence properties require significantly more run-time overhead. If we have strict limitations on run-time overhead then we must sacrifice either speed of adaptivity or convergence. We develop a suite of variants on our statistics collection scheme and our A-Greedy algorithm that lie at different points along this tradeoff spectrum. We have implemented all of our algorithms in a Data Stream Management System and a thorough performance evaluation is presented.

## 1.3 Outline of Paper

Section 2 discusses related work. Section 3 formalizes the problem and establishes notation. Section 4 presents the A-Greedy algorithm and analyzes its convergence properties, run-time overhead, and speed of adaptivity. Sections 5–7 present variants of A-Greedy. Section 8 shows how pipelined filters and our algorithms directly handle a large class of multiway stream joins. Section 9 presents experimental evaluation of our algorithms and we conclude in Section 10.

# 2. RELATED WORK

We classify related work into three categories: pipelined filters, adaptive query processing, and multiway stream joins.

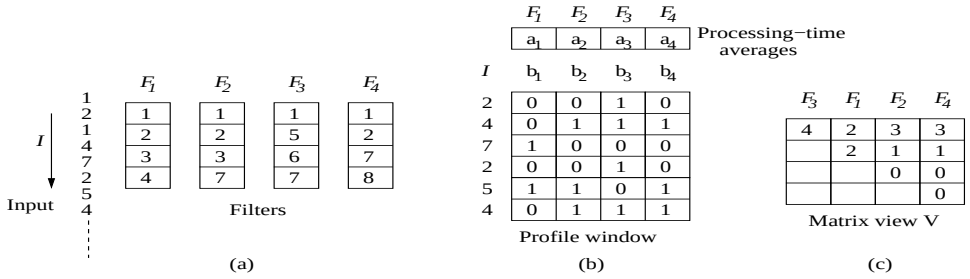


**Figure 2: Running example**

When filters are independent, the optimal ordering of a set of pipelined filters can be computed in polynomial time. Most previous work on pipelined filters and related problems makes the independence assumption and uses this ordering technique, e.g., [7, 18, 24]. Without the independence assumption, the problem is NP-hard. Previous work on ordering of correlated filters, e.g., [23, 29], either uses exhaustive search or proposes simple heuristics with no provable guarantees on the solution obtained. Approximation algorithms for the NP-hard *pipelined set cover* (or *min-sum set cover*) problem are proposed in [10, 15, 27]. Our analysis of the convergence properties of our adaptive algorithms is based on a mapping from pipelined filters to pipelined set cover. In [27] we develop a linear-programming framework to analyze several approximation algorithms for pipelined set cover.

Previous work on adaptive query processing considers primarily traditional relational query processing. One technique is to collect statistics about query subexpressions during execution and use the accurate statistics to generate better plans for future queries [4, 30]. Two other approaches [20, 21] reoptimize parts of a query plan following a *materialization point*, based on accurate statistics on the materialized subexpression. *Convergent query processing* is proposed in [19]: a query is processed in stages, each stage leveraging its increased knowledge of input statistics from the previous stage to improve the query plan. Unlike our algorithms, the algorithms proposed in [19] do not extend to continuous queries and provide no guarantees on convergence. Reference [31] explores the idea of moving to different parts of a query plan adaptively when input relations brought from remote nodes incur high latency.

The novel *Eddies* architecture [1, 6, 12, 25, 28] enables very fine-grained adaptivity by eliminating query plans entirely, instead *routing* each tuple adaptively across the operators that need to process it. Our approach is more coarse-grained than Eddies, since at a given point in time one ordering is followed by all tuples. Nevertheless, one of our algorithms (the *Independent* algorithm) is comparable to the original Eddies algorithm proposed in [1]. A more recent Eddies paper [12] reduces run-time overhead by routing tuples in batches. However, the effects on adaptivity are not studied, and the statistics that need to be monitored is exponential in the number of inputs and operators. Note that one application of our algorithms could be to provide efficient routing schemes for Eddies.

Multiway windowed stream joins have received a great deal of attention recently, although no work has considered adaptive ordering to the best of our knowledge. A multiway stream join operator called *MJoin* is specified in [32] which studies the advantages of MJoins over trees of binary joins in the stream environment. Algorithms to maintain windows lazily in MJoins to improve performance are presented in [16]. Reference [17] proposes different types of sliding window specifications and algorithms to reduce run-time memory requirements for multiway joins based on these specifications. Multiway joins in the Eddies context are considered in [25, 28]: [28] extends the original Eddies architecture so that the router can simulate MJoins, while [25] considers sharing of computation across queries, which we do not address in this paper.

# 3. PRELIMINARIES

Let *query* $Q$ process input stream $I$, applying the conjunction of $n$ commutative filters $F_1, F_2, \ldots, F_n$. Each filter $F_i$ takes a stream tuple $e$ as input and returns either *true* or *false*. If $F_i$ returns *false* for tuple $e$ we say that $F_i$ *drops* $e$. A tuple is emitted in the continuous query result if and only if all $n$ filters return *true*.

A *plan* for executing $Q$ consists of an *ordering* $P = F_{f(1)}, F_{f(2)}, \ldots, F_{f(n)}$.[2] When a tuple $e$ is processed by $P$, first $F_{f(1)}$ is evaluated. If it returns *false* ($e$ is dropped by $F_{f(1)}$), then $e$ is not processed further. Otherwise, $F_{f(2)}$ is evaluated on $e$, and so on.

At any time, the *cost* of an ordering $O$ is the expected time to process an incoming tuple in $I$ to completion (either emitted or dropped), using $O$. Consider $O = F_{f(1)}, F_{f(2)}, \ldots, F_{f(n)}$. We use $d(i|j)$ to denote the conditional probability that $F_{f(i)}$ will drop a tuple $e$ from input stream $I$, given that $e$ was not dropped by any of $F_{f(1)}, F_{f(2)}, \ldots, F_{f(j)}$. The unconditional probability that $F_{f(i)}$ will drop an $I$ tuple is $d(i|0)$. We use $t_i$ to denote the expected time for $F_i$ to process one tuple. Note that $d(i|j)$ and $t_i$ are expected to vary over time as input characteristics change; we always refer to their current values. With this notation we can now formalize the cost of $O = F_{f(1)}, F_{f(2)}, \ldots, F_{f(n)}$ as:

$$\sum_{i=1}^{n} t_i D_i, \text{ where } D_i = \begin{cases} 1 & i = 1 \\ \prod_{j=1}^{i-1}(1 - d(j|j-1)) & i > 1 \end{cases} \quad (1)$$

Our goal is to maintain filter orderings that minimize this cost at any point in time.

*Example 2.* Figure 2(a) shows a sequence of tuples arriving on stream $I$: $1, 2, 1, 4, \ldots$. We have four filters $F_1$–$F_4$, each a hash-indexed set of values such that $F_i$ drops a tuple $e$ if and only if $F_i$ does not contain $e$. Let us consider the number of hash probes required to process the input sequence shown, for different orderings. Note that all of the incoming tuples except $e = 1$ are dropped by some filter. For $O_1 = F_1, F_2, F_3, F_4$, the total number of probes for the eight $I$ tuples shown is 20. (For example, $e = 2$ requires three probes—$F_1$, $F_2$, and $F_3$—before it is dropped by $F_3$.) The corresponding number for $O_2 = F_3, F_2, F_4, F_1$ is 18, while $O_3 = F_3, F_1, F_2, F_4$ is optimal for this example at 16 probes. □

# 4. THE *A-GREEDY* ADAPTIVE FILTER ORDERING ALGORITHM

In this section we develop A-Greedy, the core adaptive ordering algorithm for pipelined filters that we propose in this paper. Let

[2] We use $f$ throughout the paper to denote the mapping from positions in the filter ordering to the indexes of the filters at those positions.

us first consider a greedy algorithm based on stable statistics. Using our cost metric introduced in Section 3 and assuming for the moment uniform times $t_i$ for all filters, a greedy approach to filter ordering proceeds as follows:

1. Choose the filter $F_i$ with highest unconditional drop probability $d(i|0)$ as $F_{f(1)}$.
2. Choose the filter $F_j$ with highest conditional drop probability $d(j|1)$ as $F_{f(2)}$.
3. Choose the filter $F_k$ with highest conditional drop probability $d(k|2)$ as $F_{f(3)}$.
4. And so on.

To factor in varying filter times $t_i$, we replace $d(i|0)$ in step 1 with $d(i|0)/t_i$, $d(j|1)$ in step 2 with $d(j|1)/t_j$, and so on. We refer to this ordering algorithm as *Static Greedy*, or simply *Greedy*. We will see later that this algorithm has strong theoretical guarantees and very good performance in practice. Greedy maintains the following *Greedy Invariant* (*GI*):

DEFINITION 4.1. *(***Greedy Invariant***)* $F_{f(1)}, F_{f(2)}, \ldots, F_{f(n)}$ *satisfies the Greedy Invariant if:*

$$\frac{d(i|i-1)}{t_{f(i)}} \geq \frac{d(j|i-1)}{t_{f(j)}}, 1 \leq i \leq j \leq n \qquad \square$$

The goal of our adaptive algorithm A-Greedy is to maintain, in an online manner, an ordering that satisfies the GI. A-Greedy has two logical components: a *profiler* and a *reoptimizer*. The profiler continuously collects and maintains statistics about input characteristics. These statistics are used by the reoptimizer to detect and correct violations of the GI in the current filter ordering.

## 4.1 The A-Greedy Profiler

To maintain the GI, the reoptimizer needs continuous estimates of (conditional) filter selectivities, as well as tuple-processing times. We consider each in turn.

### *4.1.1 Selectivity Estimates*

Our greedy approach is based on $d(i|j)$ values, which denote the conditional probability that $F_{f(i)}$ will drop a tuple $e$, given that $e$ was not dropped by any of $F_{f(1)}$, $F_{f(2)}$, ..., $F_{f(j)}$. *Selectivity* generally refers to the inverse of drop probability—that is, the selectivity of filter $F_{f(i)}$ executed after $F_{f(1)}, F_{f(2)}, \ldots, F_{f(j)}$ is $1 - d(i|j)$, or the probability of a tuple passing the filter. For $n$ filters, the total number of conditional selectivities is $n2^{n-1}$. Clearly it is impractical for the profiler to maintain online estimates of all these selectivities. Fortunately, to check whether a given ordering satisfies the GI, we need to check $(n+2)(n-1)/2 = O(n^2)$ selectivities only. Thus, by monitoring $O(n^2)$ selectivities, we can detect when statistics change sufficiently that the GI is violated by the current ordering.

Once a GI violation has occurred, to find a new ordering that satisfies the GI we may need $O(n^2)$ new selectivities in the worst case. Furthermore, the new set of required selectivities depends on the new input characteristics, so it cannot be predicted in advance. Our approach is to balance run-time monitoring overhead and the extra computation required to correct a violation: The profiler does not estimate filter selectivities directly. Instead, it maintains a *profile* of tuples dropped in the recent past. The reoptimizer can compute any selectivity estimates that it requires from this profile. As we will see in Section 4.2, the reoptimizer maintains a view over this profile for efficient incremental monitoring of conditional selectivities.

The profile is a sliding window of *profile tuples* created by sampling tuples from input stream $I$ that get dropped during filter processing. A profile tuple contains $n$ boolean attributes $b_1, \ldots, b_n$ corresponding to filters $F_1, \ldots, F_n$. Profile tuples are created as follows: When a tuple $e \in I$ is dropped during processing, $e$ is *profiled* with probability $p$, called the *drop-profiling probability*. If $e$ is chosen for profiling, processing of $e$ continues artificially to determine whether any of the remaining filters unconditionally drop $e$. The profiler then logs a tuple with attribute $b_i = 1$ if $F_i$ drops $e$ and $b_i = 0$ otherwise, $1 \leq i \leq n$.

The profile is maintained as a *sliding window* so that older input data does not contribute to statistics used by the reoptimizer. This *profile window* could be time-based, e.g., tuples in the last 5 minutes, or tuple-based, e.g., the last 10,000 tuples. The window must be large enough so that:

1. Statistics can be estimated with high confidence.
2. Reoptimization is robust to transient bursts.

In our current implementation, we fix a single drop-profiling probability and profile window size for each pipelined-filter query. There may be advantages to modifying these parameters adaptively, which we plan to explore in future work (Section 10).

*Example 3.* Figure 2(b) shows the profile window for the example in Figure 2(a) when the drop-profiling probability is 1, i.e., all dropped tuples are profiled, and the profile window is a tuple-based window of size 6. $\square$

### *4.1.2 Processing-Time Estimates*

In addition to selectivities, the GI is based on expected time $t_i$ for each filter $F_i$ to process a tuple. Since overhead is incurred to measure and record processing times, we want to measure processing times for only a sample of tuples. Furthermore, so that old statistics are not used by the reoptimizer, we maintain a sliding window of processing-time samples. While the sampling rate and sliding window for processing-time estimates need not be the same as for selectivity estimates, currently we use the same *profile tuples* as for selectivity estimates, and record the times as $n$ additional columns in the profile window (not shown in Figure 2(b)). We use $a_i$ to denote the average time for $F_i$ to process a tuple, computed as the running average of processing-time samples for $F_i$.

## 4.2 The A-Greedy Reoptimizer

The reoptimizer's job is to ensure that the current filter ordering satisfies the GI. Specifically, the reoptimizer maintains an ordering $O$ such that $O$ satisfies the GI for statistics estimated from the tuples in the current profile window. For efficiency, the reoptimizer incrementally maintains a specific view over the profile window. We describe this view next, then explain how the view is used to maintain the GI.

### *4.2.1 The Matrix View*

The view maintained over the profile window is an $n \times n$ upper triangular matrix $V[i,j], 1 \leq i \leq j \leq n$, so we call it the *matrix view*. The $n$ columns of $V$ correspond in order to the $n$ filters in $O$. That is, the filter corresponding to column $c$ is $F_{f(c)}$. Entries in the $i^{th}$ row of $V$ represent the conditional selectivities of filters $F_{f(i)}, F_{f(i+1)}, \ldots, F_{f(n)}$ for tuples that are not dropped by $F_{f(1)}, F_{f(2)}, \ldots, F_{f(i-1)}$. Specifically, $V[i,j]$ is the number of tuples in the profile window that were dropped by $F_{f(j)}$ among tuples that were not dropped by $F_{f(1)}, F_{f(2)}, \ldots, F_{f(i-1)}$. Notice that $V[i,j]$ is proportional to $d(j|i-1)$.

The reoptimizer maintains the ordering $O$ such that the matrix view for $O$ always satisfies the condition:

$$\frac{V[i,i]}{a_{f(i)}} \geq \frac{V[i,j]}{a_{f(j)}}, 1 \leq i \leq j \leq n \qquad (2)$$

```
1. /** Input: Profile tuple ⟨b_1, ..., b_n⟩ inserted into the profile window */
2. /** Find the first filter F_{f(dc)} that dropped the corresponding tuple */
3. dc = 1;
4. while (b_{f(dc)} == 0)
5.     dc = dc + 1;
6. /** Increment V entries for filters that dropped the tuple */
7. for (c = dc; c ≤ n; c = c + 1)
8.     if (b_{f(c)} == 1)
9.         for (r = 1; r ≤ dc; r = r + 1)
10.            V[r, c] = V[r, c] + 1;
```

**Figure 3: Updating $V$ on an insert to the profile window**

```
1. /** old, new: Values of V[i, j]/a_{f(j)} before and after an update */
2. if (new > old and i < j and V[i, i]/a_{f(i)} < new)
3.     Violation at position i, so invoke Figure 5 with input i;
4. if (new < old and i == j)
5.     for (c = i + 1; c ≤ n; c = c + 1)
6.         if (new < V[i, c]/a_{f(c)}) {
7.             Violation at position i, so invoke Figure 5 with input i;
8.             return; }
```

**Figure 4: Detecting a violation of the Greedy Invariant**

By the definition of $V$, Equation (2) is equivalent to saying that $O$ satisfies the GI for the statistics estimated from the tuples in the profile window. If the estimated $V[i, j]$ and $a_i$ represent the real $d(j|i-1)$ and $t_i$ respectively, then $O$ satisfies the GI.

*Example 4.* Consider the scenario in Figure 2 and assume that all measured processing times are equal (not unreasonable for hash-probes into similarly-sized tables), so $a_i = 1, 1 \leq i \leq n$. Figure 2(c) shows the matrix view over the profile window in Figure 2(b). Ordering $O = F_3, F_1, F_2, F_4$ satisfies the GI. Recall from Example 2 that we computed this ordering as having the lowest number of hash probes to process the example input stream. □

Matrix view $V$ must be updated as profile tuples are inserted into and deleted from the profile window. Figure 3 contains the pseudocode for updating $V$ when a new profile tuple is recorded. The same code is used for deletion of a profile tuple, with line 10 replaced by $V[r, c] = V[r, c] - 1$.

### 4.2.2 Violation of the Greedy Invariant

Suppose an update to the matrix view or to a processing-time estimate causes the following condition to hold:

$$\frac{V[i,i]}{a_{f(i)}} < \frac{V[i,j]}{a_{f(j)}}, 1 \leq i < j \leq n \tag{3}$$

Equation (3) states that $F_{f(j)}$ has a higher ratio of drop probability to processing time for tuples that were not dropped by $F_{f(1)}, F_{f(2)}, \ldots, F_{f(i-1)}$ than $F_{f(i)}$ has. Thus, $O = F_{f(1)}, \ldots, F_{f(i)}, \ldots, F_{f(j)}, \ldots, F_{f(n)}$ no longer satisfies the GI. We call this situation a *GI violation at position $i$.* An update to $V$ or to an $a_i$ can cause a GI violation at position $i$ either because it reduces $V[i, i]/a_{f(i)}$, or because it increases some $V[i, j]/a_{f(j)}, j > i$. Figures 4 and 5 contain pseudocode to detect and to correct, respectively, violations of GI at position $i$.

*Example 5.* From Example 4 and Figure 2(c), $O = F_3, F_1, F_2, F_4$ satisfies the GI. Let the next two tuples arriving in $I$ be 3 and 6 respectively, both of which will be dropped. Since the drop-profiling probability is 1, both 3 and 6 are profiled, expiring the earliest two profile tuples in the 6-tuple window. Figure 6(a) shows the resulting profile window. Figure 6(b) shows the updated matrix view, which indicates a GI violation at position 1. (Recall that processing times are all 1.) The new ordering $F_4, F_3, F_1, F_2$ satisfying the GI and the corrected matrix view are shown in Figure 6(c). □

```
1. /** Input: O violates the Greedy Invariant at position i */
2. maxr = i;
3. for (r = i; r ≤ maxr; r = r + 1) {
4.     /** Greedy Invariant holds at positions 1, ..., r − 1 and at
5.      *  maxr + 1, ..., n. Compute V entries for row r */
6.     for (c = r; c ≤ n; c = c + 1)
7.         V[r, c] = 0;
8.     for each profile tuple ⟨b_1, b_2, ..., b_n⟩ in the profile window {
9.         /** Ignore tuples dropped by F_{f(1)}, F_{f(2)}, ..., F_{f(r−1)} */
10.        if (b_{f(1)} == 0 and b_{f(2)} == 0 and ··· and b_{f(r−1)} == 0)
11.            for (c = r; c ≤ n; c = c + 1)
12.                if(b_{f(c)} == 1) V[r, c] = V[r, c] + 1;
13.    }
14.    /** Find the column maxc with maximum V[r,maxc]/t_{f(maxc)} */
15.    maxc = r;
16.    for (c = r + 1; c ≤ n; c = c + 1)
17.        if (V[r, c]/a_{f(c)} > V[r, maxc]/a_{f(maxc)}) {
18.            maxc = c;
19.            if (maxc > maxr)  maxr = maxc;
20.        }
21.    if (r ≠ maxc) {
22.        /** Current filter F_{f(maxc)} becomes the new F_{f(r)}.
23.         *  We swap the filters at positions maxc and r */
24.        for (k = 0; k ≤ r; k = k + 1)
25.            Swap V[k, r] and V[k, maxc]; }
```

**Figure 5: Correcting a violation of the Greedy Invariant**

### 4.2.3 Discussion

From the algorithm in Figure 5 we see that if we reorder the filters such that there is a new filter at position $i$, then we may need to reevaluate the filters at positions $> i$ because their conditional selectivities may have changed. Each reevaluation requires a full scan of the profile window to compute the required conditional selectivities. Thus, the algorithm in Figure 5 may require anywhere from 1 to $n - i$ scans of the profile window, depending on the changes in input characteristics.

The adaptive ordering can thrash if both sides of Equation (2) are almost equal for some pair of filters. To avoid thrashing, we flag a violation and invoke the algorithm in Figure 5 only if:

$$\frac{V[i,i]}{a_{f(i)}} < \alpha \frac{V[i,j]}{a_{f(j)}}, 1 \leq i < j \leq n \tag{4}$$

Parameter $\alpha \leq 1$, called the *thrashing-avoidance parameter*, is a constant to reduce the possibility of thrashing. $\alpha$ has a theoretical basis as we show in the next section.

We now analyze the behavior of A-Greedy with respect to the three important properties introduced in Section 1: guaranteed convergence to a good plan in the presence of a stable environment, run-time overhead to achieve this guarantee, and speed of adaptivity as statistics change.

## 4.3 Convergence Properties

We say the stream and filter characteristics have *stabilized* when the following three conditions hold over a long interval of time:

$C_1$: The data distribution of stream tuples is constant.

$C_2$: For every subset of filters $F_1, F_2, \ldots, F_n$, the data distribution of input tuples passing all of the filters in the subset is constant.

$C_3$: The expected processing time $t_i$ for each filter $F_i$ is constant.

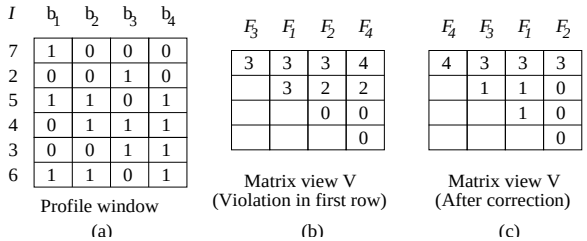


**Figure 6: Violation of the Greedy invariant**

When stream and filter characteristics stabilize, A-Greedy soon produces the same filter ordering that would be produced by the Static Greedy algorithm (Section 4), namely the ordering that guarantees the GI (Definition 4.1). We prove that this ordering is within a small constant factor of the optimal ordering according to our cost metric.

THEOREM 4.1. *When stream and filter characteristics are stable, the cost of a filter ordering satisfying the GI is at most four times the cost of the optimal filter ordering.* □

A complete proof is provided in the online technical report [2]. The key to the proof is showing that the filter ordering problem is equivalent to the *pipelined set cover* problem [10, 15, 27]. The proof establishes the equivalence of our problem to pipelined set cover and shows that Static Greedy is equivalent to a greedy 4-approximation algorithm for pipelined set cover [10, 15, 27].

The equivalence of pipelined filters and pipelined set cover brings out some interesting characteristics of our problem. Pipelined set cover is *MAX SNP-hard* [27], which implies that any polynomial-time ordering algorithm can at best provide a constant-factor approximation guarantee for this problem. Reference [15] shows that the factor 4 in Theorem 4.1 is tight. However, a nice property is that the constant factor depends on problem size, e.g., for 20, 100, and 200 filters the bound is 2.35, 2.61, and 2.8 respectively; the theoretical constant 4 requires an infinite number of filters. Irrespective of these theoretical bounds, our experiments in Section 9 show that A-Greedy usually finds the optimal ordering.

Recall from the end of Section 4 that we introduce a constant $\alpha \leq 1$ to avoid thrashing behavior, replacing equation (3) with equation (4) in determining whether the GI is violated. When this parameter is incorporated, the constant 4 in Theorem 4.1 is replaced with $\frac{4}{\alpha}$ [27]. Here too, the constant depends on the number of filters, and generally the optimal ordering is found in practice.

## 4.4 Run-time Overhead and Adaptivity

The overhead of adaptive ordering using A-Greedy is the run-time cost of the profiler and the reoptimizer. It can be divided into four components:

1. *Profile-tuple creation.* The profiler creates profile tuples for a fraction of dropped tuples equal to the drop-profiling probability. To create a profile tuple for a tuple $e$ dropped by $F_{f(i)}$, the profiler needs to additionally evaluate the $n-i$ filters following $F_{f(i)}$, recording whether the filter is satisfied and measuring the time for each filter to process $e$.

2. *Profile-window maintenance.* The profiler must insert and delete profile tuples to maintain a sliding window. It also must maintain the running averages $a_i$ of filter processing times as tuples are inserted into and deleted from the profile window.

3. *Matrix-view update.* The reoptimizer updates $V$ (Figure 3) whenever a profile tuple is inserted into or deleted from the profile window, accessing up to $n^2/4$ entries.

4. *Detection and correction of GI violations.* The reoptimizer must detect violations (Figure 4) caused by changes in input characteristics, and correct them (Figure 5). To detect a violation on an update, up to $n$ entries in $V$ may need to be accessed, while it may require up to $n-i$ full scans of the profile window to correct a GI violation at position $i$.

A change in stream or filter characteristics that causes a GI violation will cause a violation of Equation (2) and will be detected immediately. Furthermore, A-Greedy can correct the violation immediately (Figure 5), because any additional conditional selectivities required to find the new filter ordering can be computed from the existing profile window. Thus, A-Greedy is a very rapidly adapting algorithm.

# 5. THE SWEEP ALGORITHM

As we have seen in the previous section, A-Greedy has very good convergence properties and extremely fast adaptivity, but it imposes nonnegligible run-time overhead. Thus, it is natural to consider whether we can sacrifice some of A-Greedy's convergence properties or adaptivity speed to reduce its run-time overhead. It is possible to do so, as we demonstrate through several variants of A-Greedy presented in this and the next two sections.

A-Greedy detects every violation of GI in an ordering $O$ as soon as it occurs by continuously verifying the relative ordering for each pair of filters in $O$. Suppose instead that at a given point in time we only check for violations of GI involving the filter at one specific position $j$, with respect to any filter at a position $k < j$. By rotating $j$ over $2, \ldots, n$, we can still detect all violations, although not as quickly as A-Greedy. This approach is taken by our first variant of A-Greedy, which we call *Sweep*.

Sweep proceeds in stages. During a stage, only the filter at a position $j$ in $O$, i.e., $F_{f(j)}$, is profiled. (Details of profiling are given momentarily.) A stage ends either when a violation of GI at position $j$ is detected, or when a prespecified number of profile tuples are collected with no violation. Sweep maintains a weaker invariant than the GI.

DEFINITION 5.1. *(**Sweep Invariant**) Let $F_{f(j)}$ be the currently profiled filter. $F_{f(1)}, \ldots, F_{f(n)}$ satisfies the Sweep Invariant if:*

$$\frac{d(i|i-1)}{t_{f(i)}} \geq \frac{d(j|i-1)}{t_{f(j)}}, 1 \leq i < j$$ □

Within each stage, Sweep works like A-Greedy: A profiler collects profile tuples and a reoptimizer detects and corrects violations of the Sweep Invariant efficiently by maintaining a matrix view $V$ over the profile tuples. To maintain the Sweep Invariant when $F_{f(j)}$ is being profiled, the reoptimizer needs to maintain $V[i,i], 1 \leq i \leq j$ and $V[i,j], 1 \leq i < j$ only, as shown in Figure 7(a). Furthermore, to maintain these entries, only attributes $b_{f(1)}, b_{f(2)}, \ldots, b_{f(j)}$ in the profile window are required. The Sweep profiler collects profile tuples by profiling dropped tuples with the same drop-profiling probability as the A-Greedy profiler. However, for each profiled tuple, the Sweep profiler needs to additionally evaluate $F_{f(j)}$ only.

## 5.1 Convergence Properties, Run-time Overhead, and Adaptivity

Like A-Greedy, if characteristics stabilize then Sweep provably converges to an ordering that satisfies the GI. Furthermore, the run-time overhead of profiling and reoptimization is lower for Sweep than for A-Greedy: For a profiled tuple that is dropped by $F_{f(i)}$,

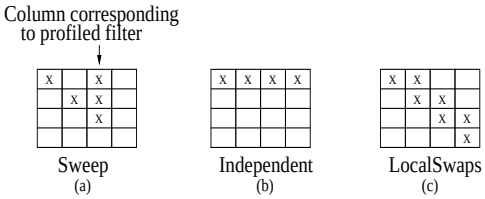


**Figure 7: View entries maintained by A-Greedy variants**

the A-Greedy profiler evaluates $n - i$ additional filters, while the Sweep profiler evaluates at most one additional filter. Similarly, the worst-case cost of updating $V$ for profile-tuple insertions and deletions is $O(n^2)$ for A-Greedy, but $O(n)$ for Sweep.

The lower run-time overhead of Sweep comes at the cost of reduced adaptivity speed compared to A-Greedy. Since only one filter is profiled in each stage of Sweep, an ordering that violates the GI can remain undetected for a relatively long time (up to $n - 2$ stages of Sweep), and can have arbitrarily poor performance. Even after the violation is detected, Sweep may take longer to converge to the new GI ordering than A-Greedy would.

# 6. THE INDEPENDENT ALGORITHM

A-Greedy is designed specifically to perform well in the presence of nonindependent filters. Suppose we assume the filters are independent, an assumption made frequently in database literature although it is seldom true in practice [9, 30]. Under the independence assumption, conditional selectivities reduce to simple one-filter selectivities, and the GI reduces to the following invariant:

DEFINITION 6.1. *(**Independent Invariant**) $F_{f(1)}$, $F_{f(2)}$, ..., $F_{f(n)}$ satisfies the Independent Invariant if:*

$$\frac{d(i|0)}{t_{f(i)}} \geq \frac{d(j|0)}{t_{f(j)}}, 1 \leq i \leq j \leq n$$ □

The *Independent* algorithm maintains the above invariant irrespective of whether the filters are actually independent or not. Like Sweep, Independent can be implemented as a variation of A-Greedy. To maintain the Independent Invariant, profile tuple creation is the same as in A-Greedy, but since conditional selectivities are not needed, the drop-profiling probability can be lower than in A-Greedy. Furthermore, the reoptimizer needs to maintain the entries in the first row of $V$ only, as shown in Figure 7(b). The original *Eddies* algorithm for adaptive ordering [1], ordering algorithms proposed recently for multiway stream joins [16, 32], and certain ordering algorithms for relational joins [24], all are similar to the Independent algorithm.

## 6.1 Convergence Properties, Run-time Overhead, and Adaptivity

Convergence properties of the Independent algorithm depend on whether the assumption of filter independence holds or not. If the filters are all independent, then Independent converges to the optimal ordering (not just the GI ordering) [18], and so does A-Greedy. However, if the filters are not independent, then the cost of the ordering produced by Independent can be $O(n)$ times worse than the GI ordering, as we show in the following example.

*Example 6.* Consider the pipelined filters in Example 2, but with $n$ filters instead of four. Let input characteristics stabilize such that tuples arriving in $I$ are distributed uniformly in [1, 2, ..., 100]. Let $F_1$–$F_{n-1}$ contain {1, 2, ..., 49}, let $F_n$ contain {50, 51, ..., 100}, and assume uniform processing times ($t_i = 1$, say) for all filters. Since $F_1, \ldots, F_{n-1}$ all have a higher probability of dropping a tuple than $F_n$, Independent will converge to an ordering $O_i$ that is a permutation of $F_1, \ldots, F_{n-1}$ followed by $F_n$. The expected number of filters processed per input tuple is $0.49n+0.51$. On the other hand, A-Greedy will converge to an ordering $O_g$ starting with one of $F_1, \ldots, F_{n-1}$, then $F_n$, then the remaining filters in some order. The expected number of filters processed per input tuple is 1.49, which is optimal. □

Independent cannot guarantee the good convergence properties of A-Greedy or Sweep, as we have just seen. However, like A-Greedy it adapts to changes instantaneously, and with a lower drop-profiling probability and maintenance of $V$ limited to the first row, it has lower run-time overhead.

# 7. THE LOCALSWAPS ALGORITHM

The three algorithms we have seen so far all detect invariant violations involving filters $F_i$ and $F_j$ that are arbitrarily distant from one another in the current ordering. An alternative approach is to monitor "local" violations only. The *LocalSwaps* algorithm maintains the following invariant.

DEFINITION 7.1. *(**LocalSwaps Invariant**) $F_{f(1)}$, $F_{f(2)}$, ..., $F_{f(n)}$ satisfies the LocalSwaps Invariant if:*

$$\frac{d(i|i-1)}{t_{f(i)}} \geq \frac{d(i+1|i-1)}{t_{f(i+1)}}, 1 \leq i < n$$ □

Intuitively, LocalSwaps detects situations where a swap between adjacent filters in $O$ would improve performance. To maintain the LocalSwaps Invariant, the reoptimizer needs to maintain only the entries in two diagonals of $V$, as shown in Figure 7(c). This reduces the cost of profiling: For each profiled tuple dropped by $F_{f(i)}$, the LocalSwaps profiler needs to additionally evaluate $F_{f(i+1)}$ only. Furthermore, when a profile tuple is inserted into or deleted from the profile window, only a constant number of $V$ entries need be updated.

## 7.1 Convergence Properties, Run-time Overhead, and Adaptivity

Unlike our other algorithms, the convergence behavior of LocalSwaps depends on how the stream and filter characteristics change. In the best cases, LocalSwaps converges to the same ordering as A-Greedy. However, in some cases the LocalSwaps ordering can have $O(n)$ times higher cost than the GI ordering, as we show in the following example.

*Example 7.* Consider the pipelined filters in Example 2, but with $n$ filters instead of four. Let input characteristics stabilize such that tuples arriving in $I$ are distributed uniformly in [1, 2, ..., 100]. Let $F_i$ contain $\{1, 2, \ldots, 100\} - \{(i-1)\delta, \ldots, i\delta\}$, where $\delta = 100/n$, and , and assume uniform processing times ($t_i = 1$, say) for all filters. Suppose initially A-Greedy and LocalSwaps are both using ordering $O = F_1, F_2, \ldots, F_n$ satisfying their respective invariants. Suppose the tuples in $F_n$ change so that $F_n$ now contains $\{(n-2)\delta, \ldots, (n-1)\delta\}$ and this is the only change before stabilization. The LocalSwaps Invariant still holds for $F_1, F_2, \ldots, F_n$, so LocalSwaps does not modify $O$. However, A-Greedy will modify $O$ to $O' = F_n, F_{n-1}$, then the remaining filters in some order, which is optimal. The expected number of filters processed per input tuple for $O$ and $O'$ are $50(n+1)$ and $1+200/n$ respectively.

Clearly, LocalSwaps has lower run-time overhead than A-Greedy because of its lower profiling and view-maintenance cost. However, as with any algorithm that is restricted to local moves, LocalSwaps may take more time to converge to the new plan when stream and filter characteristics change, and it may get stuck in a local optima as shown in Example 7.

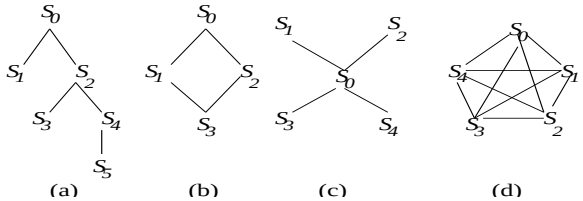


**Figure 8: Acyclic, cyclic, and star join graphs**

# 8. ADAPTIVE ORDERING OF MULTIWAY JOINS OVER STREAMS

In this section we show how pipelined filters and the A-Greedy algorithm (along with its variants) apply directly to a large class of multiway stream joins. We focus on the *MJoin* operator [32] for joining multiple windowed streams. Previous work [16, 32] has shown that MJoins have many advantages over alternate execution strategies in the streams environment. We assume that join windows are stored in their entirety in memory, which is typical in stream systems [11, 26].

An MJoin $M$ computes the join $\sigma_P(S_0[W_0] \times S_1[W_1] \times \cdots \times S_{n-1}[W_{n-1}])$ of $n$ windowed streams. $P$ is a conjunction of join predicates. Each windowed stream $S_i[W_i]$ itself produces a stream of inserts and deletes to the window, which drive the MJoin computation. (In fact MJoin can be applied in any situation where a stream of inserts and deletes to *synopses* are processed [3].) For each stream $S_i$, $M$ maintains a *join order* $O_i$, which is an ordering of $\{S_0, S_1, \ldots, S_{n-1}\} - \{S_i\}$, specifying the order in which window inserts and deletes from $S_i$ are joined with the other stream windows.

Let us consider undirected *join graphs*, with streams as vertices and an edge between $S_i$ and $S_j$ if and only if there is a join predicate between them. Join graphs can be *acyclic* as in Figure 8(a), or *cyclic* as in 8(b). A specific type of acyclic join seen frequently in practice is a *star join*, as shown in Figure 8(c). Note that a star join can have additional join predicates which can be eliminated because of predicate transitivity; for example, a natural join of $S_0[W_0], \ldots, S_4[W_4]$ on an attribute $A$, which originally has the fully-connected join graph in Figure 8(d), can be reduced to the star join in Figure 8(c) because of predicate transitivity.

## 8.1 Adaptive Ordering of Star Joins

First consider processing window inserts and deletes for the *root* (center) stream of the star join. For presentation we will abuse terminology and refer to these inserts and deletes as "tuples in stream $S$." Let the root be $S_0$ and suppose we have ordering $O = S_{f(1)}, \ldots, S_{f(n-1)}$. An incoming tuple $s \in S_0$ can be processed along ordering $O$ using one of two common techniques derived from relational query processing. The first technique treats $O$ as a *binary linear processing tree* (*BLPT*) [24], which materializes the complete join output for each prefix of $O$ before proceeding to the next stream. BLPTs are used in [32]. The second technique treats $O$ as a *pipelined processing tree* (*PPT*) [24], which uses a multiway nested loop to generate the join output without materializing any intermediate results. PPTs are used in [16]. Both BLPTs and PPTs are suboptimal for star joins because they may do more processing than necessary for $O$ tuples that get dropped eventually and do not produce any output. Thus, we introduce a more efficient technique that processes each tuple $s_0 \in S_0$ in two phases—*drop probing* and *output generation*—applied sequentially. For a given ordering, our technique does the minimum amount of processing for every tuple.

In the drop-probing phase, each $S_i$ window is probed with $s_0$ to find whether $s_0$ joins with a tuple in the window or not. The probes are done independently but in the order specified by $O$. If $s_0$ does not join with any tuple in $S_{f(i)}$'s window, i.e., $S_{f(i)}$ drops $s_0$, then we do not further process $s_0$. Otherwise, once we know that $S_{f(i)}$ does not drop $s_0$, we move on to probe $S_{f(i+1)}$'s window without generating any more matches in $S_{f(i)}$. In each case we save enough state so that the join computation performed for drop-probing need not be redone if we get to the output generation phase. If none of $S_1, S_2, \ldots, S_{n-1}$ drop $s_0$, then we go to the output generation phase, which uses a PPT to output $s_0 \bowtie S_1[W_1] \bowtie \cdots \bowtie S_{n-1}[W_{n-1}]$.

For a tuple $s_i$ in a nonroot stream $S_i$, the join with the root stream $S_0$ will always be done first to avoid Cartesian products. Then, drop-probing and output generation (if required) will be invoked separately for each tuple in $s_i \bowtie S_0[W_0]$. Because PPTs do not materialize intermediate results and all processing happens in memory, we have the following property.

PROPERTY 8.1. *Our execution technique for star joins ensures that the processing required for input tuples that are not dropped is independent of the order used to process them.* □

By Property 8.1, only the cost of the drop-probing phase affects the performance of a given ordering in a star join. Since drop-probing for star joins is clearly an instance of pipelined filters, the A-Greedy algorithm can be used to order the joins adaptively, and Theorem 4.1 applies directly.

## 8.2 Adaptive Ordering of Acyclic and Cyclic Joins

Consider the processing of $S_0$ tuples in the acyclic join depicted in Figure 8(a). We can think of the edges in this join graph as specifying a set a precedence constraints on ordering costs. For example, $O = S_1, S_3, S_2, S_4, S_5$ will never have a cost lower than $O = S_1, S_2, S_3, S_4, S_5$. Furthermore, $S_3$ will drop an $s_0 \in S_0$ only if it drops every $S_2$ tuple in $s_0 \bowtie S_2[W_2]$.

Because of these complications, processing of general acyclic (as opposed to star) joins is not equivalent to pipelined filters, so we do not get the same theoretical guarantees. Nevertheless, the A-Greedy algorithm and its variants, and the two-phase execution strategy for star joins from the previous subsection, can be extended to acyclic joins in a relatively straightforward manner. To provide an ordering of stream joins for $S_0$ tuples, A-Greedy will consider each of the $n-1$ distinct paths from $S_0$ to another $S_i$ as a single entity, monitoring the probability that this path will drop an $S_0$ tuple and recording the processing time. Details of profiling and maintaining a matrix-view extend from Section 4 directly. The two-phase execution strategy from Section 8.1 can be extended to acyclic joins so that Property 8.1 holds, provided the ordering satisfies the precedence constraints imposed by the join. Cyclic joins also can be handled: We choose a spanning tree of the join graph, then treat the join as the corresponding acyclic join as in [1, 24, 28].

Full details of join handling are omitted due to space constraints, but as we have seen, most of our techniques and results apply to a wide class of multiway joins.

# 9. EXPERIMENTAL EVALUATION

We have implemented the four adaptive ordering algorithms—A-Greedy, Sweep, Independent, and LocalSwaps—in the STREAM prototype Data Stream Management System [26]. The algorithms are used to order pipelined filters over a single stream and for join ordering in MJoins (Section 8). The experimental results presented in this section are for $n$ synthetic filters $F_1, \ldots, F_n$ on a stream $S_0$, where each $F_i$ probes a hash index on a 10,000-tuple sliding window over a stream $S_i$ for matches, simulating one pipeline of

| Parameter | Default Value |
|---|---|
| Drop-profiling probability | 0.01 (A-Greedy), 0.005 (Independent) |
| Profile-window size | 500 profile tuples (Sweep), 1000 profile tuples (others) |
| Thrash-avoidance parameter | $\alpha = 0.9$ |
| Correlation factor | $\Gamma = 2$ |
| Unconditional filter selectivity | 50% |
| Filter processing cost | 1 hash probe on 10,000-entry index |

**Table 1: Default values used in experiments**

an MJoin. Each filter evaluation probes the memory-resident hash index, then makes a randomized selection decision with the appropriate conditional selectivity. All experiments were performed on a 700 MHz Linux machine with 1024 KB processor cache and 2 GB memory.

## 9.1 Summary of Experimental Results

1. Under stable stream and filter characteristics, A-Greedy's ordering is usually optimal, while Independent's ordering almost always has higher cost.

2. Sweep always converges on A-Greedy's ordering under stable stream and filter characteristics. LocalSwaps usually also finds this ordering.

3. As expected, Sweep, Independent, and LocalSwaps have lower run-time overhead than A-Greedy. As the number of filters increases, or as some filters become much more expensive than the others, the run-time overhead of both Independent and A-Greedy increases, while that of Sweep and LocalSwaps remains stable.

4. As expected, Independent and A-Greedy adapt faster to changes than Sweep or LocalSwaps. As the frequency of changes increases, the relative performance of A-Greedy over the other three algorithms improves because it adapts faster, usually finding the optimal ordering immediately.

## 9.2 Convergence Experiments

Our first set of experiments study the convergence behavior of our algorithms when stream and filter characteristics stabilize. The factors affecting performance during convergence are the number, selectivity, and cost of the filters, and the correlation among them. We use a relatively straightforward model to capture correlation among filters. The $n$ filters are divided into $\lceil n/\Gamma \rceil$ groups containing $\Gamma$ filters each, where $\Gamma$ is called the *correlation factor*. Two filters are independent if they belong to different groups, otherwise they are positively correlated such that they produce the same result on 80% of input tuples. The filters are completely independent when $\Gamma = 1$ and are most correlated when $\Gamma = n$. For each experiment, we fix $\Gamma$ and an unconditional selectivity for each filter group, which implies the conditional selectivities based on $\Gamma$ and our 80% rule. The default values of all parameters used in the experiments are shown in Table 1.

Figure 9 shows the performance of A-Greedy and Independent for different values of $n$. (Sweep and LocalSwaps are considered later.) The $y$-axis shows the average processing rate in tuples per second, measured over a large interval after convergence. For small $n$, Figure 9 also shows the performance of the *Optimal* algorithm, which uses the optimal ordering computed from the input statistics by an offline exhaustive search, thereby showing the maximum attainable performance. A-Greedy is better than Independent

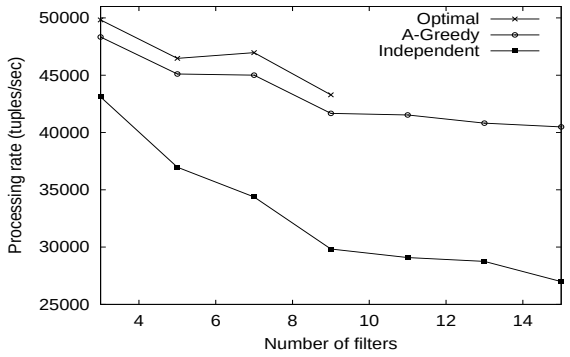


**Figure 9: Effect of the number of filters under convergence**

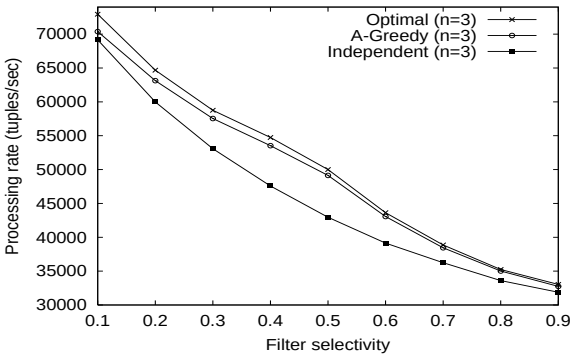


**Figure 10: Effect of selectivity under convergence**

throughout, and this performance gap widens with $n$. A-Greedy's performance is near-optimal for small $n$, but it degrades as $n$ increases. As we will see in Section 9.3, this degradation is because of A-Greedy's increasing run-time overhead.

Figure 10 shows the performance of A-Greedy and Independent for different filter selectivities. The relative performance of A-Greedy with respect to Independent is best for intermediate values of filter selectivities. For low selectivities, most of the tuples get dropped by the first filter, which is the same for both algorithms. For high selectivities, very few tuples get dropped, so costs do not vary significantly across orderings.

Figure 11 shows the performance of A-Greedy and Independent for different values of the correlation factor $\Gamma$. As expected, the relative performance of A-Greedy initially improves with respect to Independent as the filters become more correlated. Because of our simple model of correlation, when $\Gamma$ is close to $n$ all the filters are similar and costs do not vary much across orderings, so the performance improvement of A-Greedy is less significant.

## 9.3 Run-time Overhead Experiments

Table 2 breaks down the time spent by A-Greedy for $n = 3$ and $n = 8$ under stable stream and filter characteristics. In each case we consider two values of the drop-profiling probability $p$ (Section 4.1.1): the default $p = 0.01$ and a higher $p = 0.05$. The listed tasks refer to Section 4.4. More than 98% of A-Greedy's time is spent either performing useful tuple processing or creating profile tuples. The overhead of creating profile tuples increases both with $n$ and with $p$, going from less than 1% for $n = 3$ and $p = 0.01$, to 13.34% for $n = 8$ and $p = 0.05$.

Figures 12 and 13 show the processing rate and run-time breakdown of all our adaptive algorithms under convergence, varying $n$.

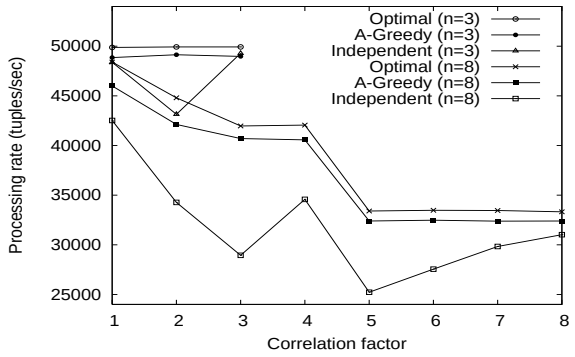


**Figure 11: Effect of correlation under convergence**

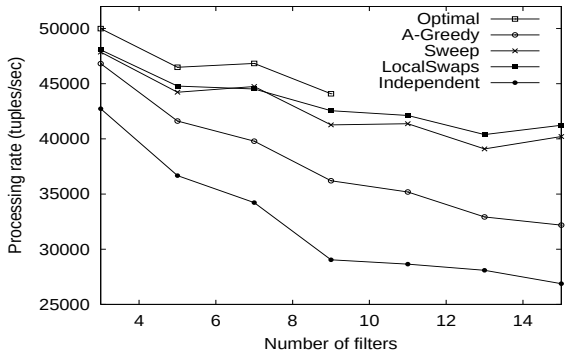


**Figure 12: Performance of all algorithms**

| Component | n=3, p=0.01 | 3, 0.05 | 8, 0.01 | 8, 0.05 |
|---|---|---|---|---|
| Tuple proc. | 98.77 | 94.17 | 96.62 | 84.77 |
| Profile-tuple | 0.88 | 4.17 | 2.92 | 13.34 |
| Profile-window | 0.20 | 0.93 | 0.20 | 0.91 |
| View update | 0.15 | 0.73 | 0.22 | 0.98 |
| View violations | 0 | 0 | 0.04 | 0 |

**Table 2: Run-time percentage breakdown for A-Greedy**

We use a drop-profiling probability of $0.05$ in this experiment to better illustrate the differences between these algorithms. Now we see the benefits of the lower run-time overhead of Sweep and LocalSwaps. Figure 12 shows that the gap between A-Greedy and Sweep and LocalSwaps increases with $n$. Figure 13 shows that this widening gap is solely because of A-Greedy's run-time overhead. Also note from Figure 13 that the time spent processing tuples is similar for Optimal, A-Greedy, Sweep, and LocalSwaps, indicating convergence to a similar ordering.

Figures 14 and 15 show the processing rate and run-time breakdown of our adaptive algorithms for different values of the filter costs. We used $n = 8$, with four filters having cost one and the other four filters having cost $c$ for $c \in \{1, 10, 50, 100\}$, where a filter with cost $c$ performs $c$ random probes on the memory-resident index per input tuple. Figure 15 shows that the percentage run-time overhead of A-Greedy is much higher than with uniform filter costs; around $25\%$ for $c = 100$. Both Sweep and LocalSwaps perform better than A-Greedy because of their lower run-time overhead. Although LocalSwaps has slightly lower run-time overhead than Sweep, Sweep performs better because LocalSwaps does not find the optimal ordering for $c = 50$ and $c = 100$.

## 9.4 Adaptivity Experiments

Each of Figures 16–18 shows a timeline from experiments where we varied the stream and filter characteristics over time for $n = 8$. We use filters with different selectivities and periodically permute the filter selectivities to change filter characteristics. For example, Figure 16 shows that a change was made after the system had processed 600,000 input tuples. The $y$-axis in Figures 16–18 shows the number of filters evaluated per 2000 input tuples. As expected, Figures 16 and 17 show that A-Greedy converges to the new Greedy ordering much faster than Sweep or LocalSwaps. Also note that the cost of LocalSwaps's ordering changes more smoothly compared to Sweep. Figure 18 shows that Independent also adapts quickly, but its ordering is worse than A-Greedy's ordering both before and after the change.

In general, A-Greedy and Independent have higher run-time overhead but faster adaptivity to changes than Sweep or LocalSwaps.

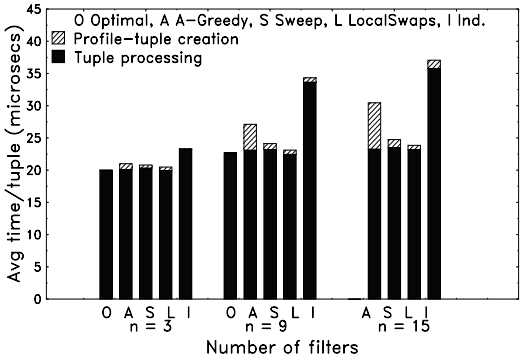


**Figure 13: Run-time breakdown for Figure 12**

Figure 19 shows this tradeoff by plotting on the $y$-axis the total time to process a workload where filter characteristics are varied periodically, with the period shown on the $x$-axis. The lower this period, the higher the rate of change. We used $n = 8$, with four filters having cost one and the other four having cost ten. Each change permutes the selectivities randomly. When the rate of change is high, the faster adaptivity of A-Greedy and Independent enable them to significantly outperform Sweep (and LocalSwaps also, which is not shown in Figure 19 to avoid clutter). This advantage diminishes as we reduce the rate of change, and the overall behavior approaches the convergence behavior of Figures 12 and 14.

Figures 20 and 21 further explore the tradeoff between run-time overhead and adaptivity using the same setup as in Figure 19. Figure 20 fixes the period of change at 100,000 tuples, where A-Greedy performs better than Sweep (Figure 19), and Figure 21 fixes the period of change at 1,000,000 tuples, where Sweep performs better. In each case we vary on the $x$-axis the cost of the expensive filters, which varies the run-time overhead (Figure 15). In Figure 20 we see that even when the filter costs are high, so A-Greedy has high run-time overhead, A-Greedy continues to perform better than (actually improves over) Sweep. The reason is that as filter costs increase, the performance gaps between different orderings also increase, so Sweep gets penalized more for its slower adaptivity. Figure 21 shows that given sufficient time between changes, Sweep's performance with respect to A-Greedy improves as A-Greedy's percentage run-time overhead increases.

# 10. CONCLUSION AND FUTURE WORK

We have shown that A-Greedy provides strong guarantees, both theoretically and experimentally, for ordering pipelined filters adaptively. A-Greedy handles correlated filters and joins in environments where changes may be rapid and unpredictable. We also identified a three-way tradeoff among provable convergence to good

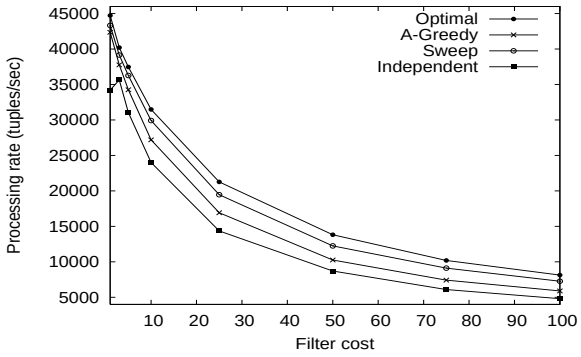


**Figure 14: Effect of varying filter costs**

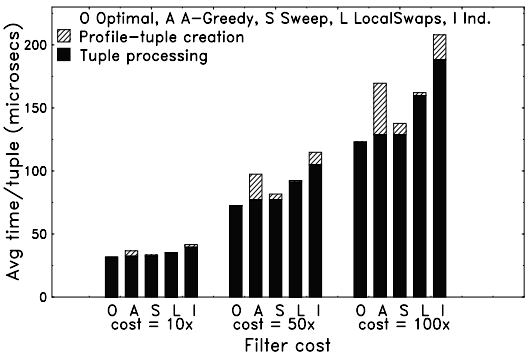


**Figure 15: Run-time breakdown for Figure 14**

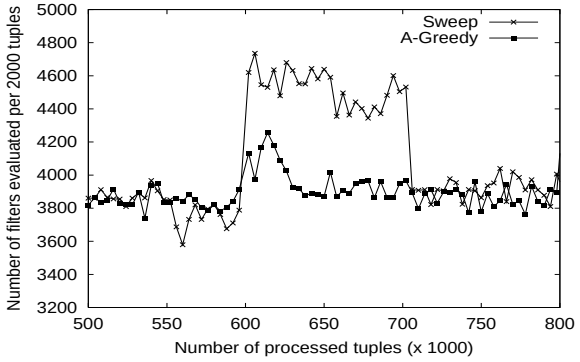


**Figure 16: Adaptivity of A-Greedy versus Sweep**

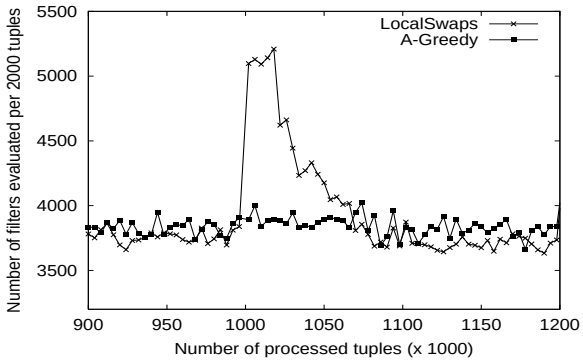


**Figure 17: Adaptivity of A-Greedy versus LocalSwaps**

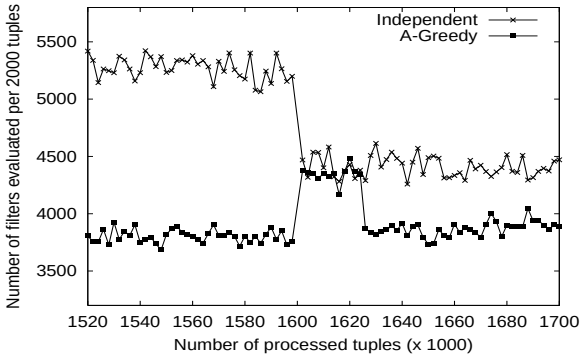


**Figure 18: Adaptivity of A-Greedy versus Independent**

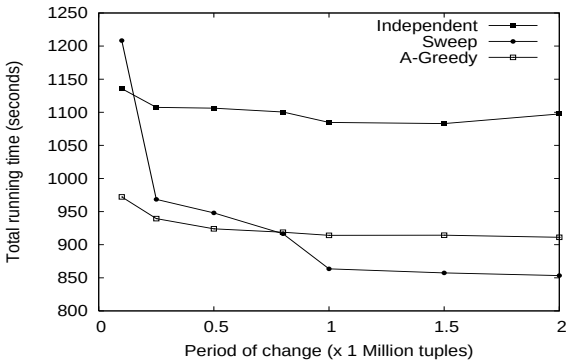


**Figure 19: Varying the rate of change**

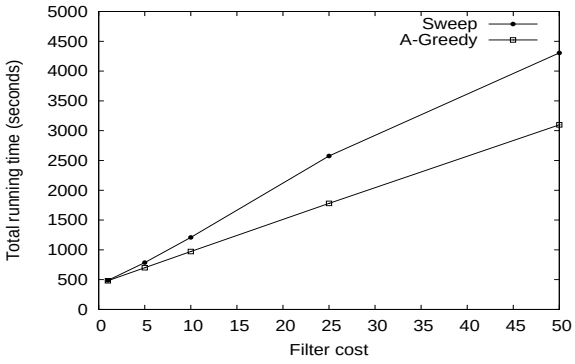


**Figure 20: Varying filter cost at $x = 10^5$ from Figure 19**

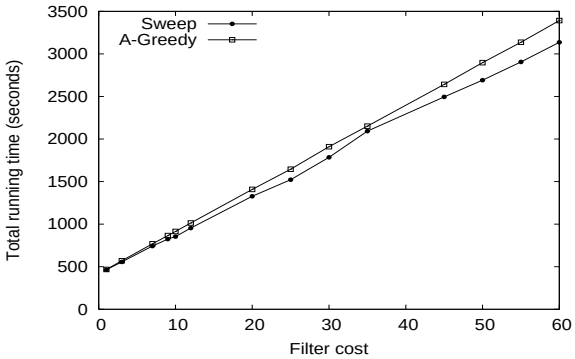


**Figure 21: Varying filter cost at $x = 10^6$ from Figure 19**

orderings, run-time overhead, and speed of adaptivity, and we developed variants of A-Greedy that lie at different points along this tradeoff spectrum.

A disadvantage of the fully pipelined MJoin algorithm, considered here as an application of pipelined filters, is that overall performance can suffer due to excessive recomputation of intermediate results. In follow-up work [3], we have developed an algorithm called *A-Caching* (for *Adaptive-Caching*) that places *subresult caches* adaptively in MJoins to minimize recomputation. With A-Caching, our pipelined multiway stream joins can adapt over the

entire spectrum between stateless MJoins and cache-rich join trees, as stream and system conditions change.

An interesting avenue for future work is the problem of balancing query execution against profiling and reoptimization overhead for optimal performance in an online adaptive setting. For example, Sweep overtakes A-Greedy in Figure 19 because A-Greedy incurs the same profiling overhead, irrespective of the rate of change of input characteristics. Furthermore, Figure 20 suggests that the cost structure of the underlying plan space also affects the tradeoff.

## 11. ACKNOWLEDGMENTS

We are grateful to David DeWitt and the anonymous referees for helpful suggestions to improve the paper.

## 12. REFERENCES

[1] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 261–272, May 2000.

[2] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. Technical report, Stanford University Database Group, Nov. 2003. Available at http://dbpubs.stanford.edu/pub/2003-69.

[3] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. Technical report, Stanford University Database Group, Mar. 2004. Available at http://dbpubs.stanford.edu/pub/2004-14.

[4] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 263–274, June 2002.

[5] D. Carney et al. Monitoring streams–a new class of data management applications. In *Proc. of the 2002 Intl. Conf. on Very Large Data Bases*, Aug. 2002.

[6] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. First Biennial Conf. on Innovative Data Systems Research*, Jan. 2003.

[7] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Trans. on Database Systems*, 24(2):177–228, 1999.

[8] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, May 2000.

[9] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Trans. on Database Systems*, 9(2):163–186, 1984.

[10] E. Cohen, A. Fiat, and H. Kaplan. Efficient sequences of trials. In *Proc. of the 2003 Annual ACM-SIAM Symp. on Discrete Algorithms*, Jan. 2003.

[11] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 647–651, June 2003.

[12] A. Deshpande. An initial study of overheads of eddies. *SIGMOD Record*, 32(4), Dec. 2003.

[13] N. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. In *Proc. of the 2000 ACM SIGCOMM*, pages 271–284, Sept. 2000.

[14] F. Fabret et al. Filtering algorithms and implementation for very fast publish/subscribe. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, June 2001.

[15] U. Feige, L. Lovász, and P. Tetali. Approximating min-sum set cover. In *Proc. of the 5th Intl. Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX)*, Sept. 2002.

[16] L. Golab and T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, Sept. 2003.

[17] M. Hammad, W. Aref, and A. Elmagarmid. Stream window join: Tracking moving objects in sensor-network databases. In *Proc. of the 2003 Intl. Conf. on Scientific and Statistical Database Management*, June 2003.

[18] J. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Trans. on Database Systems*, 23(2):113–157, 1998.

[19] Z. Ives. *Efficient Query Processing for Data Integration*. PhD thesis, University of Washington, Seattle, WA, USA, Aug. 2002.

[20] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 299–310, June 1999.

[21] N. Kabra and D. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 106–117, June 1998.

[22] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proc. of the 2003 Intl. Conf. on Data Engineering*, Mar. 2003.

[23] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing boolean expressions in object-bases. In *Proc. of the 1992 Intl. Conf. on Very Large Data Bases*, pages 79–90, Aug. 1992.

[24] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. of the 1986 Intl. Conf. on Very Large Data Bases*, pages 128–137, Aug. 1986.

[25] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, June 2002.

[26] R. Motwani, J. Widom, et al. Query processing, approximation, and resource management in a data stream management system. In *Proc. First Biennial Conf. on Innovative Data Systems Research (CIDR)*, Jan. 2003.

[27] K. Munagala, S. Babu, R. Motwani, and J. Widom. The pipelined set cover problem. Technical report, Stanford University Database Group, Oct. 2003. Available at http://dbpubs.stanford.edu/pub/2003-65.

[28] V. Raman, A. Deshpande, and J. Hellerstein. Using state modules for adaptive query processing. In *Proc. of the 2003 Intl. Conf. on Data Engineering*, Mar. 2003.

[29] K. Ross. Conjunctive selection conditions in main memory. In *Proc. of the 2002 ACM Symp. on Principles of Database Systems*, June 2002.

[30] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, pages 9–28, Sept. 2001.

[31] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 130–141, June 1998.

[32] S. Viglas, J. Naughton, and J. Burger. Maximizing the output rate of multi-join queries over streaming information sources. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, Sept. 2003.