# Adaptive Caching for Continuous Queries

Shivnath Babu, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani
Stanford University
{shivnath,kamesh,widom,rajeev}@cs.stanford.edu

## Abstract

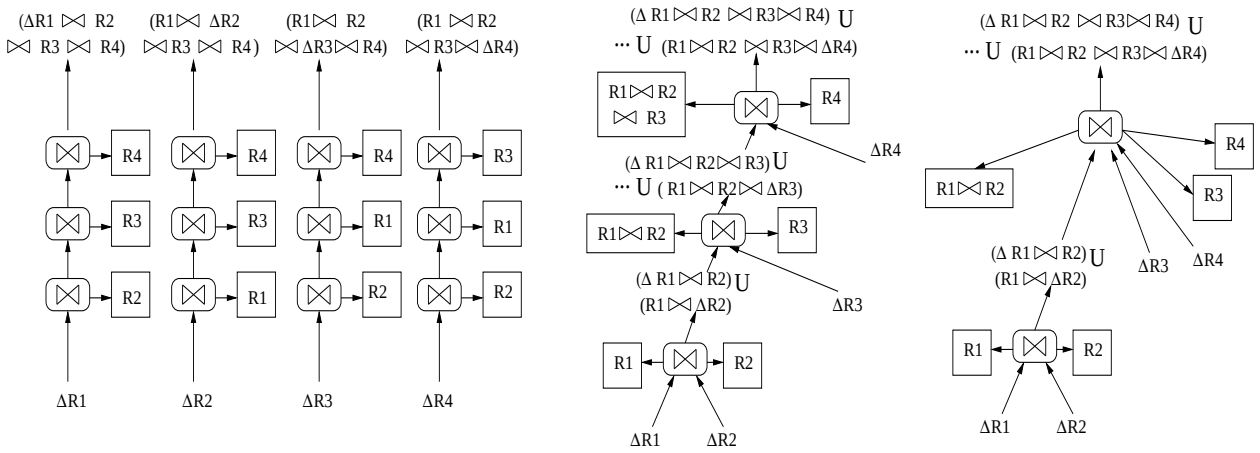
*We address the problem of executing continuous multiway join queries in unpredictable and volatile environments. Our query class captures windowed join queries in data stream systems as well as conventional maintenance of materialized join views. Our adaptive approach handles streams of updates whose rates and data characteristics may change over time, as well as changes in system conditions such as memory availability. In this paper we focus specifically on the problem of adaptive placement and removal of* caches *to optimize join performance. Our approach automatically considers conventional tree-shaped join plans with materialized subresults at every intermediate node, subresult-free* MJoins*, and the entire spectrum between them. We provide algorithms for selecting caches, monitoring their cost and benefits in current conditions, allocating memory to caches, and adapting as conditions change. All of our algorithms are implemented in the STREAM prototype Data Stream Management System and a thorough experimental evaluation is included.*

## 1 Introduction

We consider the problem of processing long-running *continuous queries*, or *CQ*s. CQs are most associated with *Data Stream Management Systems* (*DSMS's*) [9, 10, 21], although *materialized views* are also a type of CQ that operates over streams of data updates. In the data streams or materialized view environment, when a CQ (view) is registered, an execution plan is determined based on current conditions such as stream (update) data and arrival patterns, and system load. If conditions may fluctuate over the lifetime of a CQ, then an *adaptive* approach to execution strategies is essential for good performance [3, 5]: as data and system conditions change, execution strategies must automatically change as well.

In this paper we focus on a specific type of CQ that we call *stream join*: an $n$-way join of relations whose input is a continuous stream of relation updates, and whose output is a continuous stream of resulting updates to the join. Stream joins are quite general, e.g., they capture both windowed join queries in data stream systems [2, 18] and conventional maintenance of materialized join views [2, 24]. The two usual methods to process stream joins are *MJoins* [30] and *XJoins* [28]. In an MJoin, each relation $R$ has a separate query plan, or *pipeline*, describing how updates to $R$ (denoted $\Delta R$) are processed: new tuples in $R$ (or deletes to $R$) are joined with the other $n-1$ relations in some order, generating new tuples (or deleted tuples) in the $n$-way join result. An example MJoin for a four-way stream join $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$ is shown in Figure 1(a). An MJoin does not maintain any intermediate *join subresults*. In contrast, an XJoin, which is a tree of two-way joins, maintains a join subresult for each intermediate two-way join in the plan. Figure 1(b) shows an example XJoin for $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$, which uses a left-deep tree and maintains two join subresults: $R_1 \bowtie R_2$ and $R_1 \bowtie R_2 \bowtie R_3$.

MJoins and XJoins actually lie at two extremes of a spectrum of plans for stream joins. Figure 1(c) shows an example of an intermediate plan in this spectrum. This plan is a tree consisting of two MJoins: an MJoin of $R_1$ and $R_2$, followed by an MJoin of the resulting updates with $R_3$ and $R_4$. It maintains the join subresult $R_1 \bowtie R_2$




(a) MJoin (b) XJoin (c) Tree of MJoins

**Figure 1. Example plans for the stream join $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$**

only. (We will see later in this paper that a large fraction of this spectrum consists of plans that cannot even be represented as a tree of MJoins.) Intermediate plans may be significantly more efficient than any MJoin or XJoin. For example, if the update rates for $R_3$ and $R_4$ are much higher than that for $R_1$ and $R_2$, then the plan in Figure 1(c) may outperform the MJoin in Figure 1(a), the XJoin in Figure 1(b), and any other MJoin or XJoin. An MJoin may unnecessarily recompute the joining $R_1 \bowtie R_2$ tuples for incoming $R_3$ and $R_4$ updates. An XJoin may incur high overhead to maintain a join subresult involving $R_3$, $R_4$, or both. Thus, it is important to consider the entire spectrum of stream join plans.

Choosing an efficient plan for the current stream and system conditions is a difficult problem as we have just motivated. In addition, it is important to adapt as these conditions change, and that is challenging as well. For example, if $\Delta R_1$ has a long burst of updates, then we may want to switch to the MJoin in Figure 1(a) from the plan in Figure 1(c). In volatile stream environments, the cost incurred in switching plans must be kept low to enable fast adaptivity. Furthermore, system conditions, e.g., the amount of memory available to a plan, may change significantly over time. Plan switching costs and memory availability must be accounted for in plan selection and decisions to adapt.

### 1.1 Our Contributions

We propose a new approach for stream joins that addresses the above problems by starting with MJoins and adding join subresult *caches* adaptively. With this approach, we are able to capture the entire spectrum from MJoins to XJoins. Caches can be added, populated incrementally, and dropped with little overhead. The benefit of each cache is dependent entirely on conditions such as stream rates, data characteristics, and memory availability, so we use caching opportunistically. We have devised an algorithm called *A-Caching* (*Adaptive-Caching*) to adapt the caches used in an MJoin when conditions change. A-Caching estimates cache benefit and cost online, selects caches dynamically, and allocates memory to caches dynamically.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 formalizes stream joins, MJoins, and caches. Section 4 presents A-Caching's online cache benefit estimation and cache selection. Section 5 presents A-Caching's dynamic memory allocation scheme. Section 6 extends A-Caching by relaxing a restriction introduced earlier in Section 4. Section 7 presents experimental results, and Section 8 outlines future work.

## 2 Related Work

Multiway windowed stream joins have received a great deal of attention recently [5, 12, 30], although no previous work has considered adaptive caching in this environment to the best of our knowledge. MJoins were proposed

| Notation | Description |
|---|---|
| $n$ | Number of joining relations |
| $R_i$ | $i$th relation |
| $\Delta R_i$ | $i$th update stream |
| $\bowtie_{i_1}, \ldots, \bowtie_{i_{n-1}}$ | $i$th pipeline processing $\Delta R_i$; the join operator $\bowtie_{i_j}$ joins its input with $R_{i_j}$ |
| $C_{ijk}$ | Cache of segment $\bowtie_{i_j}, \bowtie_{i_{j+1}}, \ldots, \bowtie_{i_k}$ |
| $K_{ijk}$ | Cache key of $C_{ijk}$ |
| *benefit*($C_{ijk}$) | Avg. processing cost per unit time when $C_{ijk}$ is not used minus when it is used |
| *cost*($C_{ijk}$) | Avg. processing cost per unit time to maintain $C_{ijk}$ on updates to $R_{i_j}, \ldots, R_{i_k}$ |
| $W$ | Our online estimate for any statistic is the avg. of its $W$ most recent measurements |

**Table 1. Notation used in this paper**

in [30], which studies their advantages over XJoins [28]. Reference [12] considers lazy window maintenance, [4] considers plan selection under resource constraints, and [33] considers plan switching between XJoins. Our own previous work [5] considers adaptive ordering without caching.

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 these accurate statistics to generate better plans for future queries [8, 27]. Another approach [16, 17, 20] re-optimizes parts of a query plan following a blocking *materialization point*, based on accurate statistics on the materialized subexpression. *Convergent query processing* is proposed in [15]: a query is processed in stages, each stage leveraging its increased knowledge of input statistics from the previous stage to improve the query plan. Reference [29] introduces techniques for moving execution to different parts of a query plan adaptively when remote input relations incur high latency.

The novel *Eddies* architecture [3] enables very fine-grained adaptivity by eliminating query plans entirely, instead *routing* each tuple adaptively across the operators that need to process it. Reference [11] shows the performance benefits of XJoins (*STAIRs*) over MJoins (*SteMs* [23]) in Eddies, and studies the interaction between join subresults and adaptivity. Reference [11] does not consider caching or the spectrum between MJoins and XJoins.

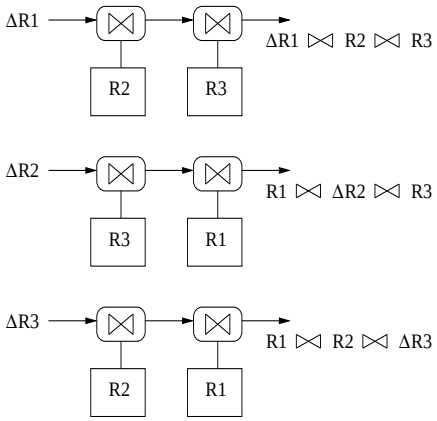
Caches are used widely in database systems, e.g., to avoid recomputing expensive predicates [13], as join indexes [32], view indexes [25], and view caches [26] to balance update costs and query speedup, and to make views self-maintainable [22]. Previous cache selection algorithms do not address one or more issues relevant in the CQ context, such as adaptivity, plan switching, cache sharing, and ease of collecting statistics during query execution. Previous work on optimizing incremental view maintenance plans [19, 24, 31] do not consider the spectrum between MJoins and XJoins, and are non-adaptive.

## 3 Preliminaries

### 3.1 Stream Joins and MJoins

Notation used throughout the paper is summarized in Table 1. We are given a *stream join*—a continuous $n$-way join query $R_1 \bowtie R_2 \cdots \bowtie R_n$ over relations $R_1, \ldots, R_n$. For clarity of presentation let us assume that all joins are equijoins of the form $R_i.attr_j = R_k.attr_l$. $\Delta R_i$ denotes the continuous stream of insertions and deletions to $R_i$. The result of the stream join is the stream of insertions and deletions to the $n$-way join based on the insertions and deletions in $\Delta R_1, \ldots, \Delta R_n$.

An *MJoin* [30] for this $n$-way stream join consists of $n$ *pipelines*, where the $i$th pipeline processes $\Delta R_i$. When an insertion or deletion $r$ arrives in $\Delta R_i$, $r$ is joined then with the other $n-1$ relations in some order $R_{i_1}, \ldots, R_{i_{n-1}}$




(a) MJoin for R1 ⋈ R2 ⋈ R3

| R1 |
|---|
| A |
| 0 |

| R2 | |
|---|---|
| A | B |
| 1 | 2 |
| 1 | 3 |
| 2 | 3 |

| R3 |
|---|
| B |
| 2 |
| 4 |
| 6 |

(b) Sample data

**Figure 2. Plan/data used in Examples 3.1–3.5**

to generate the corresponding insertions and deletions to the $n$-way join. $\Delta R_i$'s pipeline is represented as $\bowtie_{i_1}$, $\bowtie_{i_2}, \ldots, \bowtie_{i_{n-1}}$ where $\bowtie_{i_j}$ denotes the join operator that performs the join with $R_{i_j}$. A tuple $r$ input to $\bowtie_{i_j}$ is the concatenation of a tuple each from $R_i, R_{i_1}, \ldots, R_{i_{j-1}}$ such that $r$ satisfies all the join predicates among $R_i, R_{i_1}, \ldots, R_{i_{j-1}}$. $\bowtie_{i_j}$ joins $r$ with $R_{i_j}$ enforcing all join predicates between $R_{i_j}$ and $R_i, R_{i_1}, \ldots, R_{i_{j-1}}$, using indexes on $R_{i_j}$ whenever applicable. For each joining tuple $r_j \in R_{i_j}$, $\bowtie_{i_j}$ forwards $r \cdot r_j$ to the next operator in the pipeline.

We assume that the updates in $\Delta R_1, \ldots, \Delta R_n$ have a global ordering on input, e.g., based on arrival time. (The system could break ties if needed.) Updates are processed strictly in this order, and each update is processed to completion before the next update is processed. Update processing for $r \in \Delta R$ consists of the join computation using $\Delta R$'s pipeline and the update of $R$.

**Example 3.1** Consider a three-way stream join $R_1 \bowtie_{R_1.A=R_2.A} R_2 \bowtie_{R_2.B=R_3.B} R_3$. Figure 2(a) shows an MJoin for this stream join. Suppose the contents of the relations are as shown in Figure 2(b) and an insertion $\langle 1 \rangle$ on $\Delta R_1$ is processed next. The first join operator in $R_1$'s pipeline joins $\langle 1 \rangle$ with $R_2$, producing two intermediate tuples $\langle 1, 1, 2 \rangle$ and $\langle 1, 1, 3 \rangle$. These tuples are joined with $R_3$ by the second join operator, producing the output tuple $\langle 1, 1, 2, 2 \rangle$. Finally, $\langle 1 \rangle$ is inserted into $R_1$. □

### 3.2 Caches in MJoin Pipelines

In our environment, a *cache* is an associative store used during join processing that corresponds to a contiguous segment of join operators in a pipeline. We use the notation $C_{ijk}$ to represent a cache in $R_i$'s pipeline corresponding to the operator segment $\bowtie_{i_j}, \ldots, \bowtie_{i_k}$. Logically, $C_{ijk}$ contains *key-value* pairs $(u, v)$, where the key $u$ is the set of join attributes, denoted $K_{ijk}$, between a relation in $\{R_i, R_{i_1}, R_{i_2}, \ldots, R_{i_{j-1}}\}$ and a relation in $\{R_{i_j}, \ldots, R_{i_k}\}$. In other words, $K_{ijk}$ is the set of join attributes between the relations in the $i$th pipeline that appear before the cached segment and the relations in the cached segment. $K_{ijk}$ is called the *cache key* for $C_{ijk}$. $C_{ijk}$ is required to satisfy a *consistency invariant*, which guarantees that all cached entries are current and correct. However, we do not assume or make any guarantees on completeness, i.e., a cache may contain only a subset of the corresponding join subresult.

**Definition 3.1** ***(Consistency Invariant)*** *Cache $C_{ijk}$ satisfies the consistency invariant if for all $(u, v) \in C_{ijk}$, $v = \sigma_{K_{ijk}=u}(R_{i_j} \bowtie \ldots \bowtie R_{i_k})$.* □

Each cache $C_{ijk}$ supports the following operations:

- *create*$(u, v)$ for a key-value pair $(u, v)$, which adds the key-value pair to the cache.

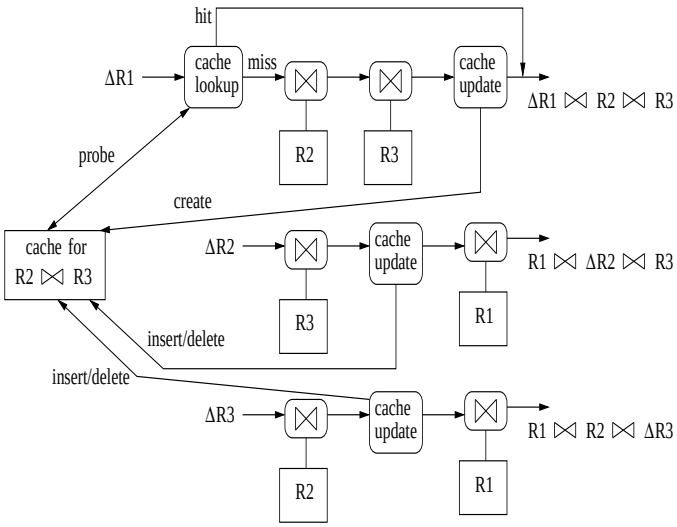



**Figure 3. Plan with a $R_2 \bowtie R_3$ cache**

- *probe*$(u)$ for a key $u$, which returns a *hit* with value $v$ if $(u, v) \in C_{ijk}$, and a *miss* otherwise.
- *insert*$(u, r)$ and *delete*$(u, r)$ for a key $u$ and a tuple $r \in R_{i_j} \bowtie \cdots \bowtie R_{i_k}$. If $(u, v) \in C_{ijk}$, *insert*$(u, r)$ adds $r$ to set $v$, otherwise *insert*$(u, r)$ is ignored. Similarly, if $(u, v) \in C_{ijk}$, *delete*$(u, r)$ removes $r$ from set $v$, otherwise *delete*$(u, r)$ is ignored.

Intuitively, a cache $C_{ijk}$ is placed in $R_i$'s pipeline just before $\bowtie_{i_j}$. When a tuple $r$ reaches $\bowtie_{i_j}$ during join processing, we first probe $C_{ijk}$ with $\pi_{K_{ijk}}(r)$. If we get a hit, then we directly have the tuples in $R_{i_j} \bowtie \cdots \bowtie R_{i_k}$ that join with $r$, and we save the work that would otherwise have been performed to generate them. If we miss, then we continue regular join processing and add the computed result (which could be empty) to $C_{ijk}$ for later probes. Because caches need not provide any guarantee on completeness, caches can be added at any time without stalling the pipelines, then populated incrementally. They can also be dropped at any time. In more general terms relating back to Section 1, plan switching costs are negligible.

Specifically, to use a cache $C_{ijk}$ during join processing, a *CacheLookup* operator $L$ and a *CacheUpdate* operator $U$ are placed in $R_i$'s pipeline. $L$ is placed just before $\bowtie_{i_j}$ and $U$ is placed just after $\bowtie_{i_k}$. Recall that $\bowtie_{i_j}$ is the first operator in the segment corresponding to $C_{ijk}$ and $\bowtie_{i_k}$ is the last operator in this segment. When $L$ receives a (possibly composite) tuple $r$, it probes the cache with $u = \pi_{K_{ijk}}(r)$. If the cache returns a hit with a value $v$, then $L$ bypasses operators $\bowtie_{i_j}, \ldots, \bowtie_{i_k}$, and $U$, and forwards $r \cdot s$ for each $s \in v$ to the operator following $U$. If there is a cache miss, $L$ sends $r$ on to $\bowtie_{i_j}$ to continue with regular join processing through $\bowtie_{i_j}, \ldots, \bowtie_{i_k}$, resulting in the computation of $v = \sigma_{K_{ijk}=u}(R_{i_j} \bowtie \cdots \bowtie R_{i_k})$, and $U$ adds $(u, v)$ to $C_{ijk}$ using *create*$(u, v)$.

**Example 3.2** Consider the three-way join from Example 3.1 and the current contents of relations $R_1$, $R_2$, and $R_3$ in Figure 2(b). Figure 3 shows an MJoin with a cache, currently empty, for the $R_2, R_3$ segment in $\Delta R_1$'s pipeline. Suppose tuple $\langle 1 \rangle \in \Delta R_1$ is processed next, so the *CacheLookup* operator probes the cache with $\langle 1 \rangle$. This probe misses and $\langle 1 \rangle$ is forwarded to the join operators. The joining $\langle 1, 2, 2 \rangle$ tuple is inserted into the cache by the *CacheUpdate* operator in $\Delta R_1$'s pipeline. If the tuple processed next is also $\langle 1 \rangle \in \Delta R_1$, then the cache probe is a hit and the join result is output immediately. □

Next we describe the maintenance of $C_{ijk}$ on updates to $R_{i_l} \in \{R_{i_j}, \ldots, R_{i_k}\}$. Suppose $r$ is inserted into $R_{i_l}$. If $C_{ijk}$ contains an entry $(u, v)$ for $u \in \pi_{K_{ijk}}(R_{i_j} \bowtie \cdots R_{i_{l-1}} \bowtie r \bowtie R_{i_{l+1}} \bowtie \cdots \bowtie R_{i_k})$, then the tuples $\sigma_{K_{ijk}=u}(R_{i_j} \bowtie \cdots R_{i_{l-1}} \bowtie r \bowtie R_{i_{l+1}} \bowtie \cdots \bowtie R_{i_k})$ must be added to $v$ to maintain the consistency invariant; similarly for a deletion. If $C_{ijk}$ does not contain an entry $(u, v)$ for a $u \in \pi_{K_{ijk}}(R_{i_j} \bowtie \cdots R_{i_{l-1}} \bowtie r \bowtie R_{i_{l+1}} \cdots \bowtie R_{i_k})$, then the consistency invariant is not affected and nothing needs to be done.

**Example 3.3** We continue with Example 3.2. Recall that the cache currently contains one entry $(u, v)$ with $u = \langle 1 \rangle$ and $v = \{\langle 1, 2, 2 \rangle\}$. Suppose tuple $\langle 3 \rangle$ is inserted into $R_3$ next. We must update $(u, v)$ by adding $\sigma_{R_2.B=u}(R_2 \bowtie \{\langle 3 \rangle\}) = \langle 1, 3, 3 \rangle$ to $v$ so that a new tuple $\langle 1 \rangle \in \Delta R_1$ will correctly produce two output tuples, $\langle 1, 1, 2, 2 \rangle$ and $\langle 1, 1, 3, 3 \rangle$. □

To maintain $C_{ijk}$'s consistency when any of $R_{i_j}, \ldots, R_{i_k}$ is updated, we must compute the corresponding updates to $R_{i_j} \bowtie \cdots \bowtie R_{i_k}$. If these updates are not computed as part of regular join processing, then we must compute them separately. We specify a *prefix invariant* (below) that is both necessary and sufficient to ensure that all updates to $R_{i_j} \bowtie \cdots \bowtie R_{i_k}$ are computed as part of regular join processing. We limit the discussion in the rest of this section to caches that satisfy the prefix invariant. This restriction is revisited in Section 4 and relaxed in Section 6.

**Definition 3.2** ***(Prefix Invariant)*** *Cache $C_{ijk}$ satisfies the prefix invariant if the first $k-j$ join operators in $\Delta R_{i_l}$'s pipeline, for each $R_{i_l} \in \{R_{i_j}, \ldots, R_{i_k}\}$, correspond to joins with one of $\{R_{i_j}, \ldots, R_{i_k}\} - R_{i_l}$.* □

In other words, we require $\Delta R_{i_l}$'s pipeline to join incoming tuples with the other relations in $R_{i_j}, \ldots, R_{i_k}$ in some order, before joining with the remaining relations in $R_1, \ldots, R_n$. Then, the tuples produced by the segment consisting of the first $k - j$ join operators in $\Delta R_{i_l}$'s pipeline are the updates to $R_{i_j} \bowtie \cdots \bowtie R_{i_k}$ for each update to $R_{i_l}$.

**Example 3.4** The plan in Figure 3 satisfies the prefix invariant for the cache corresponding to the $R_2, R_3$ segment in $\Delta R_1$'s pipeline: $\Delta R_2$'s pipeline contains $R_3$ for the first join, and vice-versa. However, a cache corresponding to the $R_2, R_1$ segment in $\Delta R_3$'s pipeline would not satisfy the prefix invariant because the join with $R_1$ is not the first join in $\Delta R_2$'s pipeline. □

To maintain the consistency of $C_{ijk}$, we place $k - j + 1$ *CacheUpdate* operators: $U_l$, $j \leq l \leq k$, is placed just before the $(k - j + 1)$st join operator in $\Delta R_{i_l}$'s pipeline. Because of the prefix invariant, $U_l$ has access to all updates to $R_{i_j} \bowtie \cdots \bowtie R_{i_k}$ caused by an update to $R_{i_l}$. $U_l$ extracts the cache key from each tuple and updates $C_{ijk}$ by making the required *insert* or *delete* call.

**Example 3.5** Let us revisit Example 3.3. When the insertion $\langle 3 \rangle$ is processed by $\Delta R_3$'s pipeline, the intermediate tuples $\langle 1, 3, 3 \rangle$ and $\langle 2, 3, 3 \rangle$ pass through the *CacheUpdate* operator, which makes the corresponding *insert* calls to the cache. Since the cache key $\langle 1 \rangle$ for $\langle 1, 3, 3 \rangle$ is present, $\langle 1, 3, 3 \rangle$ is added to its associated value. Since the cache key $\langle 2 \rangle$ for $\langle 2, 3, 3 \rangle$ is not present, this insert call will be ignored by the cache, maintaining consistency. □

### 3.3 Cache Implementation

In our system (Section 7), each cache is implemented as a hash table probed on the cache key. The number of hash buckets is chosen based on expected cache size, which is estimated online as described in Section 4.3. The cached values are sets of references to tuples in relations, so actual tuples are never copied into the caches. The cached values are stored in dynamically-allocated memory pages separately from the hash bucket pointers. Dynamic memory allocation to caches is described in Section 5. When an *insert*$(u, r)$ is invoked and the cache contains an entry $(u, v)$, we add $r$ to $v$. Deletes are the converse. *create*$(u, v)$ adds a new entry, potentially replacing an existing entry. We use a simple *direct-mapped* cache replacement scheme to keep its run-time overhead low: If a new key hashes to a bucket that already contains another key (i.e., a *collision*), then we simply replace the existing entry with the new one, without violating consistency. In the future we plan to experiment with other low-overhead cache replacement schemes.

## 4 Adaptive Caching

The performance of our stream join plans depends on join ordering as well as caching, where caching includes both cache selection and memory allocation to caches. Instead of optimizing both ordering and caching in concert, we take a modular approach as a first step in attacking this complex problem:

1. Previous work on join ordering in MJoins (without caching) provides efficient adaptive algorithms, often with optimality guarantees [5]. We use *A-Greedy* from [5] for adaptive join ordering in our implementation, but the benefits of our approach should be independent of the ordering algorithm used.

2. For a given join ordering, we then focus on adaptive selection of caches to optimize performance for that ordering.

3. Given an ordering and caching scheme, we allocate memory adaptively to the selected caches.

Apart from making the overall problem easier to handle, our modular approach is motivated by the observation that the ordering and caching problems, which are NP-Hard individually, when considered separately permit efficient near-optimal approximation algorithms. (See [5] and Section 4.4.) Full treatment of the integrated ordering-caching problem is the subject of future work.

Now let us focus on the problem of choosing caches adaptively for a given join ordering. We first restrict our plan space by considering only caches that satisfy the prefix invariant (Definition 3.2), and later drop this restriction in Section 6. There are compelling reasons to consider this restricted plan space:

1. The cache selection problem is NP-Hard even in this restricted plan space. However, the prefix invariant enables efficient algorithms to find solutions with quality guarantees (Section 4.4).

2. The prefix invariant ensures that all updates to caches are computed at no extra cost as part of regular join processing (Section 3.2).

3. This plan space subsumes the space of stream join plans that are *trees of MJoins*. (See [30] for a detailed motivation of this plan space.) As we will see in Section 7, plans from this space are very effective at improving basic MJoin performance.

For presentation, we will first assume that we have enough memory for all selected caches. In Section 5 we describe our algorithm for allocating memory adaptively to caches.

### 4.1 Cache Cost Model

Intuitively, the *benefit* we get from using a cache $C$, denoted *benefit*$(C)$, is the processing cost without the cache minus the processing cost with the cache. We use the *unit-time cost metric* [18], commonly applied for continuous queries. This metric considers the average cost in terms of processing time to perform the computations resulting from all updates arriving in a single time unit. For motivation and details of this metric see [18].

Let $d_{ij}$ be the average number of tuples processed per unit time by the join operator $\bowtie_{i_j}$, and let $c_{ij}$ be the average processing cost per tuple for $\bowtie_{i_j}$. When we do not use cache $C_{ijk}$, the average processing cost per unit time for the segment $\bowtie_{i_j}, \ldots, \bowtie_{i_k}$ is $\sum_{l=j}^{k} d_{il}c_{il}$. When we use $C_{ijk}$, it is probed for each of the $d_{ij}$ tuples that reach $\bowtie_{i_j}$. If there is a cache hit, the result of the join with $R_{i_j}, \ldots, R_{i_k}$ is available in the cache and no work needs to be done by $\bowtie_{i_j}, \ldots, \bowtie_{i_k}$. In case of a cache miss, regular pipeline processing continues in $\bowtie_{i_j}, \ldots, \bowtie_{i_k}$ and the joining tuples are inserted into the cache. Thus:

$$\begin{aligned} \textit{benefit}(C_{ijk}) \;=\; & \textstyle\sum_{l=j}^{k} d_{il}c_{il} - d_{ij} \times \textit{probe\_cost}(C_{ijk}) - \\ & \textit{miss\_prob}(C_{ijk}) \times (\textstyle\sum_{l=j}^{k} d_{il}c_{il} + d_{i,k+1} \times \textit{update\_cost}(C_{ijk})) \end{aligned}$$

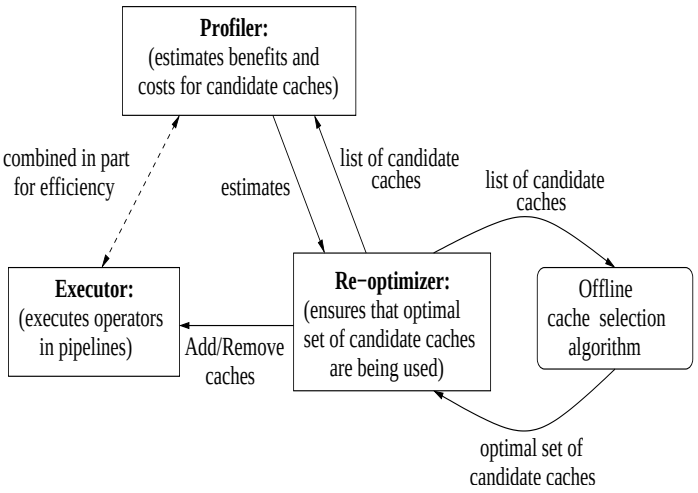



**Figure 4. A-Caching**

$d_{i,k+1}$ is the average number of tuples processed by the join operator following $\bowtie_{i_k}$, so it is the average number of tuples output by $\bowtie_{i_j}, \ldots, \bowtie_{i_k}$ per unit time. (The $d_{i,k+1}$ notation is slightly different from the regular $d_{ij}$ notation for clarity.) On average, *miss_prob*$(C_{ijk}) \times d_{i,k+1}$ tuples will need to be inserted into $C_{ijk}$ because of cache misses.

The cost of using $C_{ijk}$, denoted *cost*$(C_{ijk})$, is the cost of maintaining $C_{ijk}$ on updates to $R_{i_j}, \ldots, R_{i_k}$. Recall that because of the prefix invariant, the updates themselves are computed as part of the regular join processing. Thus:

$$\textit{cost}(C_{ijk}) = \textit{update\_cost}(C_{ijk}) \times \textstyle\sum_{l=j}^{k} d_{l,k-j+1}$$

### 4.2 Processing Stream Joins Adaptively

We are now ready to describe A-Caching—our adaptive algorithm for adding and dropping caches from MJoin pipelines. At any point in time we have a set of pipelines determined by a join ordering algorithm. From these pipelines, we identify a set of *candidate caches*, which are those that satisfy the prefix invariant (Definition 3.2). The goal of A-Caching is to ensure that the set of caches being used in the pipelines at any point of time is the subset $X$ of *nonoverlapping* candidate caches that maximizes $\sum_{C \in X}$ *benefit*$(C)$ − *cost*$(C)$, where caches are nonoverlapping if they have no join operators in common. Although there may be certain situations where overlapping caches are beneficial, we do not consider them because they complicate cache management and optimization considerably.

A-Caching consists of two of the components shown in Figure 4: the *Profiler*, which maintains online estimates of the benefits and costs of candidate caches, and the *Re-optimizer*, which ensures that the optimal set of caches is being used at any point of time. The third component in Figure 4 is the *Executor*, which executes the operators in the pipelines. The Profiler/Re-optimizer/Executor triangle is characteristic of our general approach to adaptivity throughout the STREAM prototype data stream system [5, 6, 21].

In addressing re-optimization, we first consider the offline version of the cache selection problem, which is to find the optimal nonoverlapping subset of candidate caches given stable benefits and costs for all caches. We develop an *offline cache selection algorithm* for this problem, then use it to derive the adaptive algorithm employed by the Re-optimizer. Next, we describe how the Profiler performs online cache benefit and cost estimation (Section 4.3), then our offline cache selection algorithm (Section 4.4), and finally the overall adaptive algorithm (Section 4.5).

### 4.3 Estimating Cache Benefits and Costs Online

Let us consider how we can estimate the cost and benefit of a candidate cache online. Since caches can be added and dropped with little overhead, one way to compute *benefit*$(C_{ijk})$ and *cost*$(C_{ijk})$ is to force $C_{ijk}$ to be

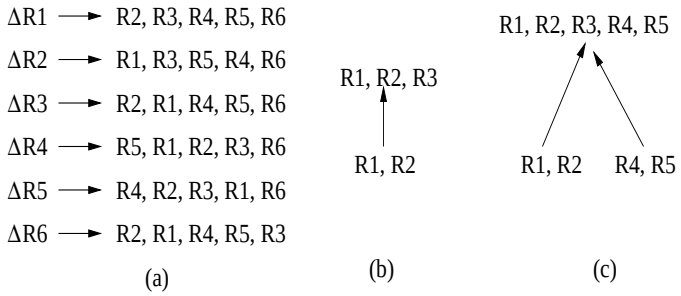



**Figure 5. Plan for $R_1 \bowtie R_2 \cdots \bowtie R_6$**

used for some time and see how it performs. Alternatively, *benefit*$(C_{ijk})$ and *cost*$(C_{ijk})$ can be estimated from online estimates of the corresponding $d_{ij}$, $c_{ij}$, *probe_cost*$(C_{ijk})$, *update_cost*$(C_{ijk})$, and *miss_prob*$(C_{ijk})$. Our implementation of A-Caching currently uses the second technique because online estimation of most of these parameters can be combined with that done for adaptive join ordering [5]. We provide a summary of our methods here; details are in Appendix A.

$\mathbf{d_{ij}}, \mathbf{c_{ij}}$: We maintain online estimates of $d_{ij}$ and $c_{ij}$ by tracking the complete processing of a sample of tuples entering the $i$th pipeline [5]. For each *profiled* tuple, we measure the number of tuples processed by each join operator $\bowtie_{i_j}$ in the pipeline and the total time spent in $\bowtie_{i_j}$. We keep track of the last $W$ measurements of these values per operator. (In general, our estimate for any statistic is the average of its $W$ most recent measurements.) $d_{ij}$ and $c_{ij}$ can be estimated from these measurements.

***probe_cost***$(C_{ijk})$, ***update_cost***$(C_{ijk})$: Based on our cache implementation described in Section 3.3, we can derive expressions for *probe_cost*$(C_{ijk})$ and *update_cost*$(C_{ijk})$ in terms of the cache key size, which is constant, and the average number of tuples per cached entry. The average number of tuples in a $C_{ijk}$ entry is $d_{i,k+1}/d_{ij}$, both estimated as described above.

***miss_prob***$(C_{ijk})$: When $C_{ijk}$ is being used, *miss_prob*$(C_{ijk})$ can be observed directly. Otherwise, we use a technique that applies a *Bloom filter* [7] over the stream of probe values to estimate the number of distinct join attribute values, which is then used to estimate *miss_prob*$(C_{ijk})$. The estimate of the number of distinct values also enables us to estimate the total memory requirement for the cache.

### 4.4 Offline Cache Selection Algorithm

Because of the prefix invariant, if two candidate caches in a pipeline overlap, then one of them is a superset of the other. Thus, overlapping caches in a pipeline have a hierarchical relationship, and they can be organized into a tree with each cache $C$ having as its parent the smallest candidate cache in the pipeline that is a strict superset of $C$.

**Example 4.1** Consider a 6-way equijoin $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_6$ on attribute $A$. Suppose the pipelines are ordered as shown in Figure 5(a). The prefix property holds for $\{R_1, R_2\}$, $\{R_4, R_5\}$, $\{R_1, R_2, R_3\}$, and $\{R_1, R_2, R_3, R_4, R_5\}$, which together give rise to the set of candidate caches with at least one join. For example, there are two candidate caches in $\Delta R_4$'s pipeline—one corresponding to the $R_1, R_2$ segment and the other corresponding to the overlapping $R_1, R_2, R_3$ segment—which can be organized into the tree in Figure 5(b). Similarly, there are three candidate caches in $\Delta R_6$'s pipeline, which can be organized into the tree in Figure 5(c). □

The following theorem states that the optimal set of nonoverlapping candidate caches can be chosen for a single pipeline in linear time.

**Theorem 4.1** *Given $m$ candidate caches on a single pipeline, the nonoverlapping subset $X$ that maximizes $\sum_{C \in X}$ benefit$(C)$ − cost$(C)$ can be found in $O(m)$ time.* □

**Proof 4.1** Since the caches are on the same pipeline, they can be organized into a forest where each tree corresponds to a maximal overlapping subset of caches. The union of the optimal solutions over all trees in this forest gives the optimal solution for the forest. Within a tree, we compute the optimal solution recursively. We traverse the tree bottom-up and derive the optimal set of caches for each subtree $T$ rooted at a cache $C$. The optimal solution for $T$ is either $C$ or the union of optimal solutions for $C$'s children. This algorithm is $O(m)$. □

The maximum number of candidate caches in a single pipeline is linear in the number of joining relations $n$, so the above algorithm is $O(n)$. Unfortunately, Theorem 4.1 does not extend to candidate caches in multiple pipelines because caches can be *shared* across pipelines.

**Definition 4.1** ***(Shared Caches)*** *Candidate caches $C_{ijk}$ and $C_{pqr}$ in different pipelines ($i \neq p$) are shared caches if $R_{i_j}, \ldots, R_{i_k}$ is some permutation of $R_{p_q}, \ldots, R_{p_r}$ and $K_{ijk} = K_{pqr}$.* □

That is, two caches are shared if they cache the same join and have the same key. Sharing is not restricted to pairs of caches—we can have larger groups of shared caches.

**Example 4.2** Consider again the 6-way equijoin $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_6$ with the plan shown in Figure 5(a). The candidate cache corresponding to $R_1, R_2$ is shared in the pipelines for $\Delta R_3$, $\Delta R_4$, and $\Delta R_6$. The candidate cache corresponding to $R_1, R_2, R_3$ is shared in the pipelines for $\Delta R_4$ and $\Delta R_5$. Note that the key for each cache is attribute $A$, which is equated across all relations in the join. □

The obvious advantage of using shared caches is that their maintenance costs can be shared. The total benefit of using a group of shared caches is the sum of their individual benefits, but the total cost is simply the cost of a single cache. Thus, from the perspective of a single cache, it is always advantageous to use it in multiple pipelines so that the benefits add up while the cost is fixed. However, the combination of sharing and the need to choose nonoverlapping caches makes the offline cache selection problem NP-Hard. The proof of the following theorem is given in Appendix B.

**Theorem 4.2** *Given $m$ candidate caches not all on the same pipeline, finding the nonoverlapping subset $X$ that maximizes $\sum_{C \in X} \text{benefit}(C) - \text{cost}(C)$ is NP-Hard. However, if there are no shared caches, then the optimal solution can be found in $O(m)$ time.* □

Our objective of picking the optimal nonoverlapping subset $X$ of candidate caches that maximizes $\sum_{C \in X}$ *benefit*$(C) -$ *cost*$(C)$ can be stated alternatively as picking the subset that minimizes $\sum_{C \in X}$ *proc*$(C)$ + *cost*$(C)$, where *proc*$(C_{ijk})$ is the average cost per unit time of using $C_{ijk}$ in $\Delta R_i$'s pipeline. Note that *proc*$(C_{ijk})$ does not include the cost of maintaining $C_{ijk}$ on updates to $R_{i_j}, \ldots, R_{i_k}$. From Section 4.1:

$$\textit{proc}(C_{ijk}) = d_{ij} \times \textit{probe\_cost}(C_{ijk}) + \textit{miss\_prob}(C_{ijk}) \times (\textstyle\sum_{l=j}^{k} d_{il} c_{il} + d_{i,k+1} \times \textit{update\_cost}(C_{ijk}))$$

In this alternative formulation, in addition to the set of candidate caches, we treat each operator $\bowtie_{i_j}$ as a cache of zero length with *cost*$(\bowtie_{i_j}) = 0$ and *proc*$(\bowtie_{i_j}) = d_{ij} c_{ij}$. We can state the following theorem in terms of this formulation. The proof and algorithms are given in Appendix B.

**Theorem 4.3** *There exists a greedy polynomial-time algorithm for finding the nonoverlapping subset $X$ minimizing $\sum_{C \in X} \text{proc}(C) + \text{cost}(C)$ that gives an $O(\log n)$ approximation, where $n$ is the number of joining relations. Furthermore, there exists a randomized polynomial-time algorithm that is also an $O(\log n)$ approximation.* □

Our offline cache selection algorithm is the optimal recursive algorithm if there are no shared candidate caches, otherwise it is the greedy approximation algorithm from Theorem 4.3 (Appendix B). While the total number of candidate caches $m = O(n^2)$, our experiments indicate that the overhead of exhaustively searching over the $2^m$ possible combinations of the candidate caches is typically negligible for $n \leq 6$, even in an adaptive setting.

### 4.5 Adaptive Cache Selection Algorithm

For presentation, we first describe a simplified version of our adaptive algorithm. We then identify the inefficiencies in this simplified version and describe how we address them. Recall that the goal of the adaptive algorithm is to ensure that the optimal nonoverlapping subset of candidate caches is being used as conditions change. A candidate cache $C$ is in one of three states at any point in time:

- **Used**: $C$ is being used in join processing as described in Section 3.2.
- **Profiled**: Although $C$ is not being used in join processing, we want to estimate *benefit*$(C)$ and *cost*$(C)$ if it were it to be used. The estimation is performed as described in Section 4.3.
- **Unused**: $C$ is neither being profiled nor chosen for use by our adaptive algorithm.

The simplified version of our adaptive algorithm manipulates the states of candidate caches as follows:

1. All candidate caches start out in the *profiled* state.
2. We proceed with regular join processing until each profiled cache has collected at least $W$ observations for each estimated statistic. Recall from Section 4.3 that our online estimate of any statistic is the average over the $W$ most recent observations.
3. We run our offline cache selection algorithm to choose the optimal set of caches to use among the *profiled* caches. The chosen caches are moved to the *used* state and the rest to the *unused* state. No new data updates are processed during this step. Each *used* cache is empty initially and populated incrementally as join processing continues.
4. After a *re-optimization interval* $I$, we move all candidate caches back to the *profiled* state and repeat Steps 2 and 3. The interval $I$ can be specified in terms of time or number of tuples processed, and is typically much larger than the interval required to collect $W$ observations for each estimated statistic.
5. If the ordering of join operators in a pipeline is changed at any point by the join ordering algorithm, we remove all caches used in that pipeline, recompute its candidate caches, and start each new candidate cache in the *profiled* state. These caches will be considered for placement in the next re-optimization step.

Next we point out the inefficiencies in the simplified version and how we address them.

a. The simplified algorithm does not react to changes within the re-optimization interval $I$. This problem is fundamentally hard to solve because faster adaptivity requires higher run-time profiling overhead [5, 10]. A complete solution to this problem falls in the realm of *adapting adaptivity* [10], which is beyond the scope of this paper. Our current approach is to react immediately to changes that make a *used* cache inefficient, but react gradually (at the next re-optimization step) to changes that make an *unused* cache useful. We monitor *benefit*$(C) -$ *cost*$(C)$ continuously for all used caches, which can be done with little overhead, and move $C$ immediately to the *unused* state if the difference becomes negative.

b. During re-optimization, the simplified algorithm always moves a *used* cache $C$ to the profiled state to estimate its benefit and cost. In reality, we need move $C$ only if it has an *unused* subset cache $C'$, so that we can access the full probe stream for $C'$ required to estimate *miss_prob*$(C')$.

c. Even if statistics are stable, the simplified algorithm periodically invokes the offline algorithm. We reduce this overhead by invoking the offline algorithm only when the benefit or cost of at least one *used* or *profiled* cache has changed beyond a certain percentage $p$. Our experiments indicate that a value of $p = 20\%$ is very effective at reducing run-time overhead without affecting adaptivity significantly.

## 5 Adaptive Memory Allocation to Caches

Since the memory in a DSMS must be partitioned among all active continuous queries [9, 21], it may be that we do not have sufficient memory to store all caches selected by our cache selection algorithm. Continuing with our

modular approach, we first select caches assuming infinite memory, then allocate pages of memory dynamically to the chosen caches so that we can adapt to changes in the amount of memory available. We use a greedy allocation scheme based on the *priority* of a cache $C$, which is defined as the ratio of *benefit*$(C)$ − *cost*$(C)$ to the expected memory requirement of $C$. Intuitively, the priority of a cache is its net benefit per unit memory used. The cache memory requirement is the product of the expected number of entries in the cache and the expected size of each cache entry, both of which are estimated before $C$ is used (Section 4.3).

## 6 Globally-Consistent Caches

The biggest simplification of A-caching as described so far is that it restricts candidate caches to those that satisfy the prefix invariant. We address this drawback by relaxing the consistency invariant to create a larger space of candidate caches—called *globally-consistent caches*—to choose from. The following example illustrates the basic idea of globally-consistent caches.

**Example 6.1** Let the pipelines for the six-way join $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_6$ be as shown in Figure 5(a), except suppose $\Delta R_6$'s pipeline is now $R_2, R_3, R_4, R_5, R_1$. Suppose we want to cache $R_2 \bowtie R_3$ in $\Delta R_6$'s pipeline. $R_2, R_3$ does not satisfy the prefix invariant because updates to $R_2 \bowtie R_3$ when $R_2$ changes are not computed as part of regular join processing. However, $R_1, R_2, R_3$ satisfies the prefix invariant. Therefore, if we were to cache $(R_2 \bowtie R_3) \ltimes R_1$ instead of $R_2 \bowtie R_3$, then all updates to this cache will be computed as part of regular join processing. Informally, $(R_2 \bowtie R_3) \ltimes R_1$ contains the subset of tuples in the $R_2 \bowtie R_3$ join subresult which have a joining tuple in $R_1$. □

Globally-consistent caches cache joins of the form $X \rhd\!\lhd Y$, where $X$ and $Y$ are themselves joins of disjoint subsets of $R_1, \ldots, R_n$, and $X \cup Y$ satisfies the prefix invariant. These caches satisfy the following *global-consistency invariant*, which generalizes the consistency invariant.

**Definition 6.1** ***(Global-Consistency Invariant)*** *Cache $C_{ijk}$ satisfies the global-consistency invariant if for all $(u, v) \in C_{ijk}$, $(\sigma_{K_{ijk}=u}(R_{i_j} \bowtie \cdots \bowtie R_{i_k})) \ltimes (R_i \bowtie R_{i_1} \bowtie \cdots \bowtie R_{i_{j-1}} \bowtie R_{i_{k+1}} \bowtie \cdots \bowtie R_{i_{n-1}}) \subseteq v \subseteq \sigma_{K_{ijk}=u}(R_{i_j} \bowtie \cdots \bowtie R_{i_k})$.* □

Globally-consistent caches $X \rhd\!\lhd Y$ actually capture a spectrum. Our original caches are at one end of this spectrum, where no relations are in $Y$. At the other end are caches where $X \cup Y = \{R_1, \ldots, R_n\}$, which can be used irrespective of the join ordering since the prefix invariant always holds for $R_1, \ldots, R_n$. In other words, we can always use a globally-consistent cache $C_{ijk}$ based on $X \rhd\!\lhd Y$ where $X = R_{i_j} \bowtie \cdots \bowtie R_{i_k}$ and $Y$ is the join of all relations not in $X$.

The use, maintenance, and online estimation algorithms for globally-consistent caches are similar to those outlined in Section 3.2. However, with globally-consistent caches, the offline cache selection problem becomes as hard as the NP-Hard *independent set* problem, which cannot be approximated efficiently [14]. Without a good approximation algorithm, we must resort to exhaustive search, with pruning heuristics for efficiency. Our approach is to fix a maximum number $m$ of candidate caches and use exhaustive search over the $2^m$ possible combinations.

Let $p$ be the number of caches that satisfy the prefix invariant. If $p \geq m$, then we ignore globally-consistent caches and use the cache selection algorithm from Section 4.5. If $p < m$, the $m$ caches we consider are:

1. the $p$ caches that satisfy the prefix invariant, and
2. $m - p$ additional caches: We start with up to $n$ globally-consistent caches $X \rhd\!\lhd Y$ where $X$ is all but one relation, then if we have not used up our quota, globally-consistent caches $X \rhd\!\lhd Y$ where $X$ is all but two relations, and so on. When the quota is reached within one of the iterations, we select from that group of caches arbitrarily.

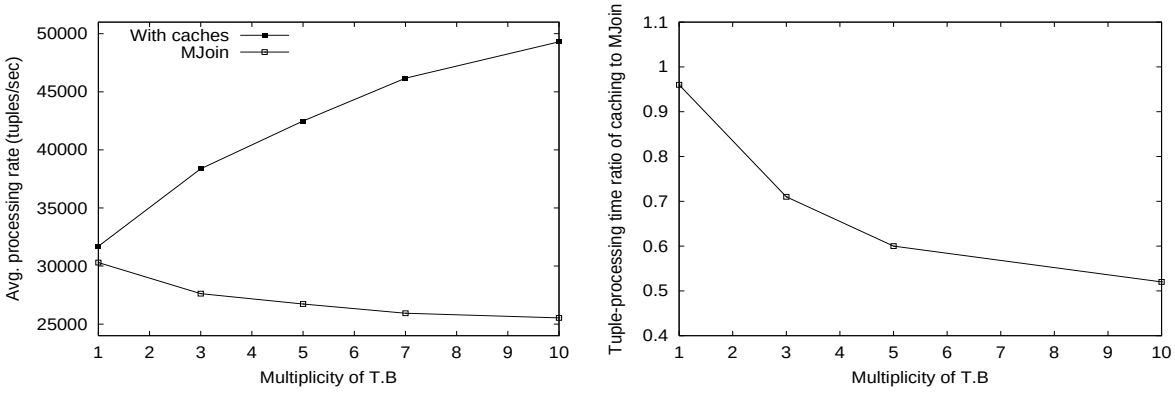



**Figure 6. Varying cache hit probability**

## 7 Experimental Evaluation

In this section we present experimental results from an implementation of our algorithms in the *StreaMon* adaptive processing engine [6] of the STREAM prototype Data Stream Management System [21].

### 7.1 Experimental Setup

We used two equijoin queries in our experiments—a three-way join $R(A) \bowtie_A S(A,B) \bowtie_B T(B)$, and a general $n$-way join of the form $R_1(A) \bowtie_A R_2(A) \bowtie_A \cdots \bowtie_A R_n(A)$. All relations are sliding windows [18, 21] on append-only streams, with window sizes set appropriately to get the desired join selectivity. The update stream corresponding to each relation is the stream of insertions and deletions to the window—generated by a *window operator* in the STREAM prototype [1]—over the corresponding append-only stream. We used a synthetic data generator to produce multiple append-only streams with specified data characteristics and relative arrival rates. In each experiment we report the maximum load the system can handle, in terms of the number of tuples processed per second, for each of the algorithms that we analyze. Stream arrival rates are uniform by default. All input tuples are 32 bytes long. Default values of the re-optimization interval $I$ (Section 4.5) and the window size $W$ (Section 4.3) are 2 seconds and 10 respectively. All joins use hash indexes by default.

### 7.2 Performance of Caches in MJoins

First we consider the performance of caches in MJoins with respect to a variety of input parameters such as join selectivity and stream rate. For all experiments in this section, we show an *absolute graph* and a *relative graph*. The absolute graph shows the average tuple-processing rates (average number of tuples processed per second) of the best plan with caches and the best MJoin plan without caches. The relative graph shows the ratio of the average tuple-processing rate of MJoin to that of the plan with caches. For example, a $y$ value of $0.6$ on the relative graph means that if the plan with caches processes $Y$ tuples per second, then the underlying MJoin processes $0.6Y$ tuples per second. Note that these numbers include all types of overheads, including profiling and re-optimization overhead. Most experiments in this section use the $R \bowtie_A S \bowtie_B T$ query. The join attributes $R.A, S.A, S.B$, and $T.B$ draw values from the same domain in the same order. The multiplicity of these values is 1 in $R$ and $S$ and a variable $r$ in $T$ (default $r = 5$), and the rate of $\Delta T$ correspondingly is $r$ times that of $\Delta R$ and $\Delta S$. Globally-consistent caches were not used in the experiments reported here, but they exhibit similar performance.

Figure 6 compares plans with caching against MJoins when we vary the cache hit probability. These plans are similar to Figures 3 and 2(a). There is only one candidate cache, which we force to be chosen. This cache is probed using $T.B$, so by varying the multiplicity of $T.B$ we vary the cache hit probability. As the hit probability

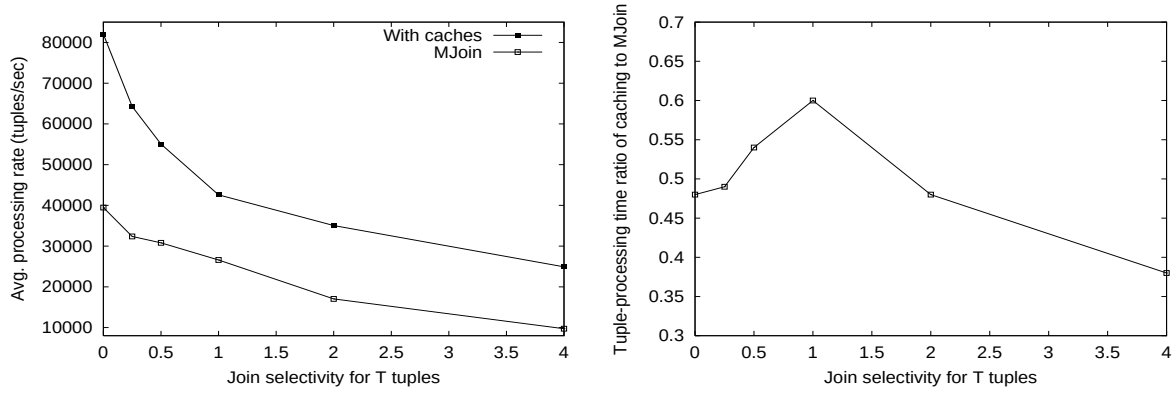



**Figure 7. Varying join selectivity**

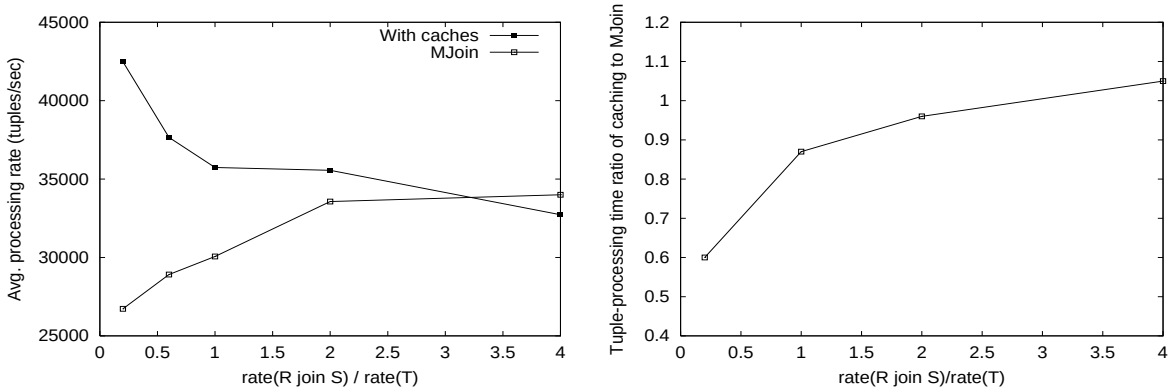



**Figure 8. Varying update to probe ratio**

increases, using the cache becomes significantly better than the best MJoin. Interestingly, using the cache is better even when all multiplicities are one. Because we are using sliding windows, every inserted tuple in the input stream subsequently appears as a deleted tuple, so we have one opportunity for a cache hit.

Figure 7 varies join selectivity (or multiplicity) in $\Delta T$'s pipeline, that is, the number of $R \bowtie S$ tuples joining with $\Delta T$ tuples. Notice that caching improves performance over the entire join selectivity range. The relative improvement is least when selectivities are close to 1, which is explained by the two opposing effects at work: As join selectivity increases, each cache hit saves more work, but each cache miss requires more work to update the cache.

Figure 8 varies the ratio of the cache update rate to the cache probe rate. Once again, we force the one candidate cache to be used. As expected, the performance of caching deteriorates with a higher update rate. However, the update cost of the cache is low with respect to the work saved per cache hit, so using the cache is better even when the update rate is higher than the probe rate.

Figure 9 considers the $n$-way join $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$, varying $n$. The multiplicity of the join attributes is 1 for $\lfloor \frac{n}{2} \rfloor$ of the streams and 5 for the others. Notice that the improved join performance using caches is maintained throughout the range. As an example, the 7-way join used 6 caches out of 15 available candidate caches.

In Figure 10, we consider varying join costs. Returning to the three-way join query, we drop the hash index on $S.B$ so that the join with $S$ in $\Delta T$'s pipeline is forced to use a nested-loop join. The join cost is then proportional to the number of tuples in (the window) $S$, which is varied in Figure 10. As expected, the relative performance of caching improves significantly with increasing join cost.

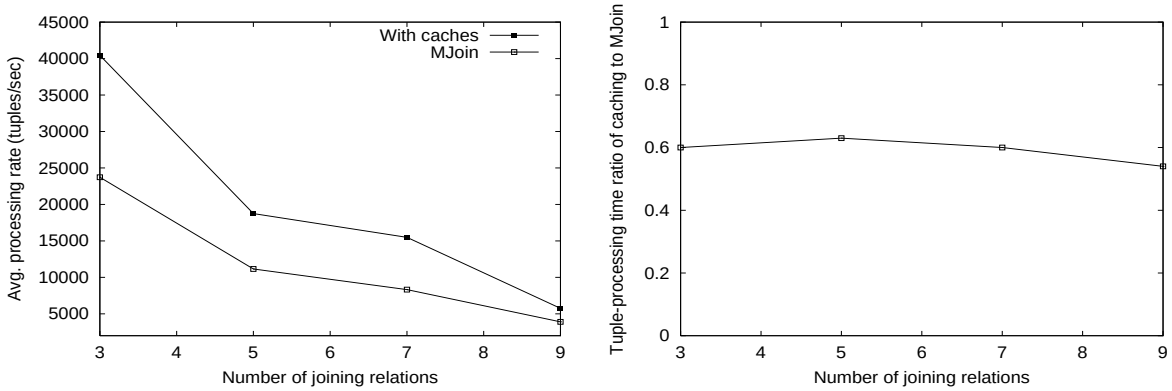



**Figure 9. Varying number of joins**

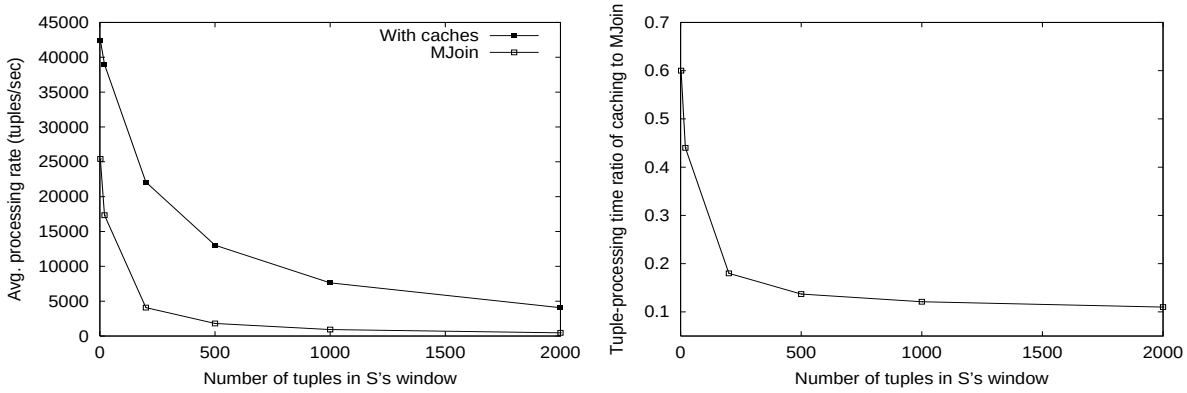



**Figure 10. Varying join cost**

### 7.3 Performance of Stream-Join Plans

We experimented extensively with different input stream characteristics, namely, update rates and join selectivities, to study the performance of the four different types of plans that we consider in this paper: (i) MJoins, (ii) XJoins, (iii) Caching-based plans from Section 4 satisfying the prefix invariant, and (iv) Caching-based plans from Section 6 which may include globally-consistent caches. Figure 11 summarizes eight of our experiments, to illustrate the trends that we observed, discussed below.

Our experiments used the 4-way join $R_1(A) \bowtie_A R_2(A) \bowtie_A R_3(A) \bowtie_A R_4(A)$. Update rates and join selectivities were varied in our experiments. Figure 11 shows plan performance for eight sample points $D_1$–$D_8$ from the spectrum of input characteristics. The relative stream arrival rates and pairwise join selectivities at these sample points are shown in Table 2. As usual, these numbers represent the maximum input load the system can handle in each case, so they include all types of overheads as well. For each point, we report the performance of the best MJoin ($M$), the best XJoin ($X$), the best caching-based plan with the prefix invariant ($P$), and the best caching-based plan with globally-consistent caches ($G$). $M$ is chosen using the *A-Greedy* algorithm from [5], which is the adaptive join ordering algorithm used in caching-based plans as well. $X$ is chosen by exhaustive search. For uniformity, both $P$ and $G$ are chosen by exhaustive search over the set of candidate caches, with $m = 6$ for the algorithm in Section 6. The overhead of optimization accounts for a negligible fraction of the total processing time in all cases in Figure 11. All plans were given as much memory as they required to maintain join subresults; memory constraints are considered in Section 7.4.

Our experiments showed the following trends:

1. $X$, $P$, and $G$ almost always outperform $M$; see $D_1$–$D_8$ in Figure 11. (The experiments in [11] seem to indicate a similar trend between $X$ and $M$ in Eddies.) This trend is aided by our efficient implementation

| Sample Point | Arrival Rates | | | | Pairwise Join Selectivities | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $R$ | $S$ | $T$ | $U$ | $R \bowtie S$ | $R \bowtie T$ | $R \bowtie U$ | $S \bowtie T$ | $S \bowtie U$ | $T \bowtie U$ |
| $D_1$ | 10 | 1 | 1 | 1 | 0.004 | 0.005 | 0.005 | 0.007 | 0.0045 | 0.005 |
| $D_2$ | 8 | 1 | 1 | 8 | 0.004 | 0.005 | 0.005 | 0.007 | 0.0045 | 0.005 |
| $D_3$ | 10 | 15 | 1 | 5 | 0.003 | 0.005 | 0.007 | 0.0045 | 0.006 | 0.008 |
| $D_4$ | 1 | 1 | 1 | 1 | 0.003 | 0.004 | 0.0067 | 0.002 | 0.0023 | 0.0027 |
| $D_5$ | 4 | 1 | 1 | 4 | 0.005 | 0.007 | 0.005 | 0.006 | 0.005 | 0.002 |
| $D_6$ | 1 | 1 | 1 | 1 | 0.005 | 0.0033 | 0.0025 | 0.0067 | 0.005 | 0.0075 |
| $D_7$ | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| $D_8$ | 1 | 1 | 1 | 1 | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 |

**Table 2. Relative stream arrival rates and pairwise join selectivities for the eight sample points in Figure 11. The stream arrival rates are relative to the rate of stream $T$.**

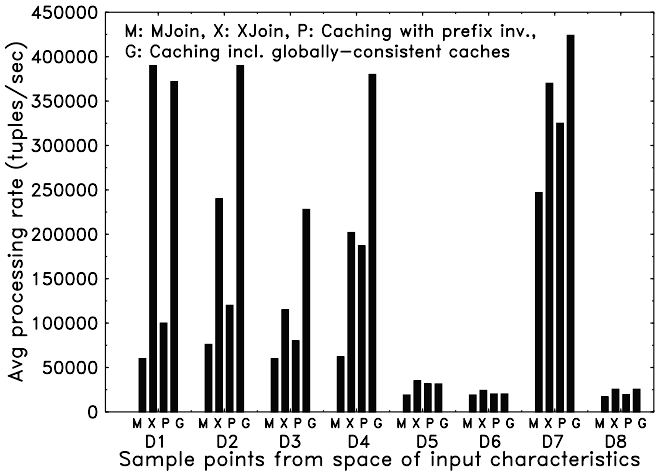



**Figure 11. Performance of stream-join plans**

of join subresult materialization and caching (Section 3.3) on top of MJoin, especially minimizing memory-to-memory copying and using a low-overhead cache management scheme.

2. $P$ usually significantly outperforms $M$, showing the benefit of adding caches to MJoins; see $D_1$–$D_8$.

3. There are cases where $X$ significantly outperforms $P$; see $D_1$, $D_2$, and $D_3$. The best MJoin ordering for the input characteristics in these experiments was such that the prefix invariant was not satisfied for one or more high-benefit caches. This problem is alleviated when our algorithm considers globally-consistent caches. That is, $G$ is almost always as good as $X$. Even when $X$, $P$, and $G$ all maintain the same join subresults, $X$ may be slightly better than $P$ and $G$ because a probe into a (fully materialized) join subresult in an XJoin never "misses": If no joining tuples are found, then no joining tuples exist.

4. $G$ can significantly outperform $X$; see $D_2$, $D_3$, $D_4$, and $D_7$. Any XJoin for a 4-way join can materialize at most two join subresults, with at most one 3-way join subresult. $G$ removes this restriction by considering plans from the spectrum between MJoins and XJoins. $G$ outperforms $X$ at $D_2$, $D_3$, $D_4$, and $D_7$ by caching more than one 3-way join subresult.

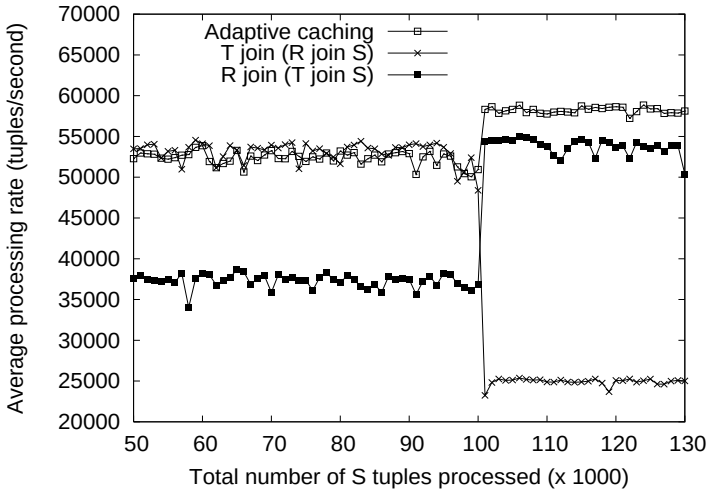



**Figure 12. Adaptivity to changing stream rate**

### 7.4 Adaptivity Experiments

Next we consider adaptivity to changing input characteristics. We use the 3-way join query and consider the case where stream $\Delta R$ is bursty. The rate of $\Delta R$ during a burst is 20 times its normal rate. The re-optimization interval $I$ was set to 10,000 tuples.

In Figure 12 we consider three plans for the 3-way join query: a static plan, denoted $T \bowtie (R \bowtie S)$, which always uses a single $R \bowtie S$ cache in $\Delta T$'s pipeline, another static plan, denoted $R \bowtie (T \bowtie S)$, which always uses a $T \bowtie S$ cache in $\Delta R$'s pipeline, and our adaptive cache selection algorithm from Section 6. The $x$-axis in Figure 12 shows the progress of time in terms of the number of $\Delta S$ tuples that have arrived so far. The $y$-axis shows the current tuple-processing rate (average number of tuples processed per unit time). The initial stream characteristics correspond to our default experimental setup from Section 7.2, so that using the $R \bowtie S$ cache in $\Delta T$'s pipeline is optimal as evident from the performance of the static plan $T \bowtie (R \bowtie S)$. We see that our adaptive algorithm converges to this plan. Also note that the performance of our adaptive algorithm is almost equal to that of the static plan, indicating very low run-time overhead for adaptivity. A burst in $\Delta R$ starts when 100,000 $\Delta S$ tuples have arrived and continues through the reminder of the run. As expected, $T \bowtie (R \bowtie S)$ performs poorly during the burst, while static $R \bowtie (T \bowtie S)$ becomes the high-performer. Our adaptive algorithm converges quickly to the new best plan, which uses a globally-consistent $(T \bowtie S) \ltimes R$ cache in $\Delta R$'s pipeline.

Figure 13 shows how our caching-based plans adapt smoothly to the amount of memory available for storing join subresults. This experiment used the same setup as in point $D_8$ in Figure 11 (Section 7.3), except now we vary on the $x$-axis the amount of memory available for storing join subresults. MJoins are insensitive to extra memory since they do not store join subresults. For the input that we used, any XJoin requires at least 25.6 KB of memory to store its join subresults, so the region before $x = 25.6$ KB is infeasible for XJoins. The XJoin curve following this infeasible region should actually be a step function, with the steps corresponding to optimal plans for different memory availability. Since our XJoin optimizer does not take the amount of available memory as a parameter, Figure 13 reports only the best XJoin, which requires roughly 32 KB memory to store its join subresults in our system.

## 8 Conclusions and Future Work

We studied the problem of using caches to improve performance and adaptivity in continuous multiway joins. Our algorithms consider the spectrum between stateless MJoins and cache-rich XJoins, and adapt as stream and

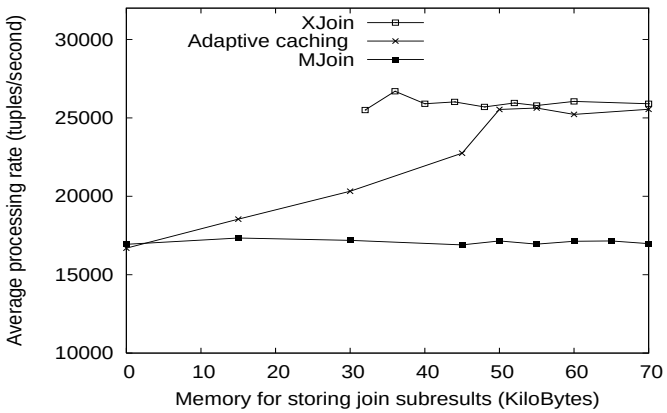



**Figure 13. Adaptivity to memory availability**

system conditions change. Our experiments clearly indicate the benefits of this approach. Although we focused on joins, our algorithms generalize to query plans composed of one or more operator pipelines. Given a set of pipelines with (possibly shared) cacheable segments, our algorithms can monitor cache cost and benefits and place caches adaptively to improve performance. Avenues for future work include:

1. Exploring the tradeoff between run-time overhead and adaptivity outlined in Section 4.5 as it applies to our adaptive cache selection algorithm.

2. Our cache selection algorithm begins from scratch each time it is invoked, and it is invoked whenever any input statistic changes significantly. Two potential improvements are: (i) Develop an incremental algorithm that adds or drops caches based solely on the statistics that have changed. (ii) Identify over time "unimportant statistics" whose significant changes tend not to produce new cache selections.

## References

[1] A. Arasu, S. Babu, and J. Widom. An abstract semantics and concrete language for continuous queries over streams and relations. *VLDB Journal*. (To appear).

[2] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. Technical report, Stanford University Database Group, Oct. 2003. Available at http://dbpubs.stanford.edu/pub/2003-67.

[3] 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.

[4] A. Ayad and J. Naughton. Static optimization of conjunctive queries with sliding windows over infi nite streams. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 419–430, June 2004.

[5] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream fi lters. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 407–418, June 2004.

[6] S. Babu and J. Widom. StreaMon: An adaptive engine for stream query processing. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, June 2004. Demonstration proposal.

[7] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[8] 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.

[9] 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.

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

[11] A. Deshpande and J. Hellerstein. Lifting the burden of history from adpative query processing. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, Aug. 2004.

[12] 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.

[13] J. Hellerstein and J. Naughton. Query execution techniques for caching expensive methods. In *Proc. of the 1996 ACM SIGMOD Intl. Conf. on Management of Data*, pages 423–434, June 1996.

[14] D. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, Boston, MA, 1997.

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

[16] 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.

[17] N. Kabra and D. DeWitt. Effi cient 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.

[18] 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.

[19] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom. Performance issues in incremental warehouse maintenance. In *Proc. of the 2000 Intl. Conf. on Very Large Data Bases*, pages 461–472, Aug. 2000.

[20] V. Markl, V. Raman, D. Simmen, G. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 659–670, June 2004.

[21] 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.

[22] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. of the 1996 Intl. Conf. on Parallel and Distributed Information Systems*, pages 158–169, Dec. 1996.

[23] 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.

[24] K. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proc. of the 1996 ACM SIGMOD Intl. Conf. on Management of Data*, pages 447–458, June 1996.

[25] N. Roussopoulos. View indexing in relational databases. *ACM Trans. on Database Systems*, 7(2):258–290, 1982.

[26] N. Roussopoulos. An incremental access method for ViewCache: Concept, algorithms, and cost analysis. *ACM Trans. on Database Systems*, 16(3):535–563, 1991.

[27] 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.

[28] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, June 2000.

[29] 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.

[30] 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.

[31] D. Vista. Integration of incremental view maintenance into query optimizers. In *Proc. of the 1998 Intl. Conf. on Extending Database Technology*, pages 374–388, Mar. 1998.

[32] Z. Xie and J. Han. Join index hierarchies for supporting effi cient navigations in object-oriented databases. In *Proc. of the 1994 Intl. Conf. on Very Large Data Bases*, pages 522–533, Aug. 1994.

[33] Y. Zhu, E. Rundensteiner, and G. Heineman. Dynamic plan migration for continuous queries over data streams. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 431–442, June 2004.

## A Estimating Cache Benefits and Costs Online

From the equations for *benefit*$(C_{ijk})$ and *cost*$(C_{ijk})$ in Section 4.3, it follows that they can be estimated from online estimates of the appropriate $d_{ij}$, $c_{ij}$, *probe_cost*$(C_{ijk})$, *update_cost*$(C_{ijk})$, and *miss_prob*$(C_{ijk})$. We discuss each of these in turn.

$\mathbf{d_{ij}}, \mathbf{c_{ij}}$ : To maintain online estimates of $d_{ij}$ and $c_{ij}$, we profile the complete processing of a fraction of tuples entering the $i$th pipeline [5]. Before we start to process a tuple $r \in R_i$ using its pipeline $\bowtie_{i_1}, \ldots, \bowtie_{i_{n-1}}$, we will choose to profile it with a probability $p_i$. If we choose to profile $r$, then as $r$ and the intermediate tuples generated by it are processed by $\bowtie_{i_1}, \ldots, \bowtie_{i_{n-1}}$, we maintain two values $\delta_j$ and $\tau_j$ per join operator $\bowtie_{i_j}$. $\delta_j$ is the number of tuples processed by $\bowtie_{i_j}$ as part of the processing for $r$, and $\tau_j$ is the corresponding processing time spent in $\bowtie_{i_j}$. Furthermore, we will not use any caches in this pipeline throughout the processing of $r$. Once the processing of $r$ is complete, we insert $\delta_j$ and $\tau_j$ respectively into sliding windows that maintain the last $W$ observations in each case. (Recall from Table 1 that our estimate for any statistic is the average of the corresponding $W$ most recent observations.) We use the sum of values in these windows, denoted *sum*$(\delta_j)$ and *sum*$(\tau_j)$, to estimate $d_{ij}$ and $c_{ij}$ as: $d_{ij} = \textit{rate}(R_i) \times \text{sum}(\delta_j)/W, c_{ij} = \text{sum}(\tau_j)/\text{sum}(\delta_j)$. Note that we measure processing costs in terms of processing time. *rate*$(R_i)$ is the number of $R_i$ tuples processed per unit time, which is maintained in a straightforward manner.

***probe_cost***$(C_{ijk})$, ***update_cost***$(C_{ijk})$ : Based on our cache implementation described in Section 3.3, we can derive expressions for *probe_cost*$(C_{ijk})$ and *update_cost*$(C_{ijk})$ in terms of the cache key size, which is constant, and the average number of tuples in a cached entry. The details are straightforward. The average number of tuples in a $C_{ijk}$ entry is $d_{i,k+1}/d_{ij}$.

***miss_prob***$(C_{ijk})$ : When $C_{ijk}$ is being used, *miss_prob*$(C_{ijk})$ can be observed directly. When $C_{ijk}$ is not being used and we need to estimate *miss_prob*$(C_{ijk})$, we place a *CacheLookup* operator $L$ just before $\bowtie_{i_j}$ so that $L$ can see the complete stream of tuples processed by $\bowtie_{i_j}$. For each nonoverlapping window of $W_d$ tuples in this stream, $L$ hashes each tuple on $K_{ijk}$ into a *Bloom filter* [7] with $\alpha W_d$, $\alpha \geq 1$, bits before forwarding the tuple. If $b$ bits in the bloom filter are set after a sequence of $W_d$ tuples have been processed, then an estimate of *miss_prob*$(C_{ijk})$ is $b/W_d$. Intuitively, $b$ distinct keys are present among the $W_d$ tuples, and each distinct key will cause a cache miss when it appears for the first time after which it will be cached. As usual, we maintain the last $W$ such observations, and our online estimate of *miss_prob*$(C_{ijk})$ is the average of these $W$ observations.

## B NP-Hardness and Approximation Algorithms

We show that the problem of choosing the nonoverlapping subset $X$ of candidate caches that minimizes $\sum_{C \in X}$ *proc*$(C)$ +*cost*$(C)$, denoted *total cost*, when the caches satisfy the prefix invariant can be formulated as an integer program. We present an NP-Hardness proof and several approximation algorithms.

We group together candidate caches which share the same update cost (Definition 4.1). For the current join ordering, let $G_1, G_2, \ldots, G_k$ denote all maximal groups of shared candidate caches given by Definition 4.1. That is, each candidate cache $C$ belong to exactly one of $G_1, G_2, \ldots, G_k$, say $G_i$, such that $G_i$ also includes all other candidate caches $C'$ that can be shared with $C$.

**Integer Program:** We can formulate the offline cache selection problem (Section 4.4) as an integer program. We have a variable $x_c$ for every candidate cache $c$, which is set to 1 if the cache is chosen, and 0 otherwise. Note that the set of candidate caches forms a tree under containment (Section 4.4); we denote this tree $T$. We also have a variable $z_r$ for every group $G_r$, $1 \leq r \leq k$, which is set to 1 if at least one cache from that group is chosen, and 0 otherwise. Let $B_c =$ *proc*$(c)$ denote the processing cost of cache $c$ (Section 4.4), and $L_r =$ *cost*$(c \in G_r)$ denote the (common) update cost of using caches in group $G_r$. Recall from Section 4.4 that we treat the operators themselves as caches of length zero.

We need to choose caches to minimize: $\sum_c B_c x_c + \sum_r L_r z_r$. We then need to enforce the constraint that every operator $p$ is covered by one and only one chosen cache: $\sum_{c:p \in c} x_c = 1 \ \forall$ operators $p$. The variable $z_r$ is constrained to 1 if at least one cache from $G_r$ is chosen: $z_r = \max_{c \in G_r} x_c \ \forall G_r$. We finally enforce the integrality constraints: $x_c, z_r \in \{0, 1\}$.

To solve the formulation in polynomial time, we relax the final integrality constraints to linear constraints: $0 \leq x_c, z_r \leq 1$. We solve this formulation, which yields a lower bound on the value of the optimum integer solution. We then convert this *fractional* solution to a feasible integer solution, which is a provably good approximation.

**Theorem B.1** *There is a randomized polynomial time $O(\log n)$ approximation to minimizing the total cost.* □

**Proof B.1** We begin by solving the linear relaxation described above. We consider each group $G_r$ in turn, and independently choose $\alpha_r$ uniformly at random in $[0, 1]$. For every $c \in G_r$, if $x_c \geq \alpha_r$, we choose cache $c$, else we drop it. At the end of this process, we have chosen a certain set of possibly overlapping caches. If a set of caches overlap, we choose the one that covers the most operators, and drop the rest.

Note that the probability that cache $c$ is chosen is exactly $x_c$. Similarly, the probability that group $G_r$ is chosen is exactly $z_r$. Therefore, the expected cost of this solution is at most the cost of the optimal fractional solution.

However, note that not all operators need be covered by some cache. For operator $p$, each cache $c$ covering it was chosen independently with probability $x_c$. Also, we have: $\sum_{c:p \in c} x_c = 1$. Therefore the probability that the operator is not covered is: $\Pi_{c:p \in c}(1 - x_c) \geq \frac{1}{e}$

Therefore, the expected fraction of operators left uncovered at the end of the first step is $(1 - 1/e)$. We repeat the randomized rounding $3 \log m$ times and take the union of these solutions, where $m = n(n-1)$ is the number of operators. The probability that an operator is still uncovered is now $\frac{1}{m^3}$, which means all operators are covered with high probability. Since each solution has expected cost at most the cost of the optimal fractional solution, the expected approximation ratio is $O(\log m) = O(\log n)$. □

**Poly-time Special Case:** The linear program actually produces an integer optimum solution for the case any operator is present in at most two non-zero length caches. The proof is omitted.

**Greedy Algorithm:** We now present a simple greedy algorithm for this problem which achieves the same logarithmic approximation guarantee.

For every group $G_r$, we find the smallest *cost-rate*, $D_r$ for using that group. Suppose cache $c$ covers $n_c$ operators.

$$D_r = \min_{S \subseteq G_r} \frac{L_r + \sum_{c \in S} B_c}{\sum_{c \in S} n_c}$$

The subset $S_r$ that yields this minimum value can be computed in linear time. We sort the caches $c$ in $G_r$ in increasing order of $\frac{B_c}{n_c}$.

**Claim:** The optimum set $S$ is a prefix sequence in this sorted order of caches.

*Proof of Claim:* We split cache $c$ into $n_c$ caches that cover one operator each and have cost $\frac{B_c}{n_c}$. For these unit length caches, it is clear that the optimal set $S$ would pick a prefix of the cache sequence sorted in increasing order of cost. If we replace the original lengths, the last cache could be picked to a fractional value. The claim now is that either dropping this cache or picking it fully yields the same value of $D_r$, which follows from this fact:

$$\forall A, B, a, b \geq 0 : \ \frac{A + \alpha a}{B + \alpha b} \geq \min\left(\frac{A}{B}, \frac{A + a}{B + b}\right) \ \forall \alpha \in [0, 1]$$

Therefore, the optimal set $S$ is a prefix sequence in the sorted order of caches, where the sorting is in increasing order of $\frac{B_c}{n_c}$. □

We find that group which yields the smallest $D_r$ value, and pick all the caches in the corresponding set $S_r$. We now delete all the covered operators from the system. Note that this possibly reduces the number of operators covered by the other caches in the system.

We repeat this process on the remaining caches (by recomputing the $D_r$ values and picking the best group), until all operators are covered by some cache. If this process ends up choosing overlapping caches, we simply pick the cache that covers the most operators and delete the rest.

We will now show that this algorithm is a $O(\log n)$ approximation to the problem of minimizing total cost. The proof follows the same lines as the proof showing the greedy algorithm for set cover is a logarithmic approximation. At iteration $i$, suppose there are $m_i$ operators remaining. The optimal solution yields a feasible solution on these operators because of the tree structure of the caches – any cache we pick is either completely contained in one of OPT's caches, or contains some of OPT's caches. This means there exists one group $G_s$ in OPT's solution with $D_s \leq \frac{V}{m_i}$, where $V$ is the cost of the optimal solution. We therefore pick a group $G_r$ with $D_r \leq \frac{V}{m_i}$, which implies the cost of the greedy solution is over the $k$ iterations is at most:

$$\sum_{i=1}^{k}(m_i - m_{i+1})\frac{V}{m_i} \leq \sum_{i=1}^{n}\frac{V}{i} \leq V \times O(\log n)$$

This shows that the greedy algorithm is a logarithmic approximation. This ratio is best possible unless $P = NP$. We omit the proof of this claim.

**NP-Hardness:** We now show that the $K$ level case is NP-Hard. We will consider the formulation that maximizes the *net benefit* $\sum_{C \in X}$ *benefit*$(C)$ − *cost*$(C)$, and show it is as hard as the independent set problem on a general graph with $K$ vertices. This would also show that the formulation that minimizes the total cost is NP-Hard.

**Theorem B.2** *The cache placement problem on a tree with $K$ levels is NP-Hard.* □

**Proof B.2** Given a graph with $K$ vertices, we construct a set of cache groups $G_1, G_2, \ldots, G_K$ such that $G_i$ is contained in $G_{i+1}$. The tree is therefore a path of length $K$. The update cost and benefit of the groups are such that we get a net benefit of 1 from a group iff we choose *all* caches in that group, else we get no benefit at all. This is simple to enforce. Given a graph with $K$ vertices, $v_1, v_2, \ldots, v_K$, we place a pipeline for each vertex. Suppose $v_i$ has neighbors $v_{i1}, v_{i2}, \ldots, v_{ij}$, then we have caches from groups $G_i, G_{i1}, G_{i2}, \ldots, G_{ij}$ in pipeline $i$.

Any feasible solution can pick only one cache from each pipeline. Suppose we pick a cache corresponding to group $G_i$. The cost structure forces us to pick *all* caches corresponding to this group. This immediately precludes choosing any cache from a group $G_j$ if $v_i$ and $v_j$ are neighbors in the graph.

Therefore, the groups corresponding to any feasible set of chosen caches forms an independent set in the graph, and the value of the solution is the number of groups picked, which is the size of the independent set. This proves the theorem. □