# A SURVEY OF JOIN PROCESSING IN DATA STREAMS

Junyi Xie and Jun Yang
*Department of Computer Science*
*Duke University*
{junyi,junyang}@cs.duke.edu

## 1. Introduction

Given the fundamental role played by joins in querying relational databases, it is not surprising that *stream join* has also been the focus of much research on streams. Recall that relational (theta) join between two non-streaming relations $R_1$ and $R_2$, denoted $R_1 \bowtie_\theta R_2$, returns the set of all pairs $\langle r_1, r_2 \rangle$, where $r_1 \in R_1$, $r_2 \in R_2$, and the join condition $\theta(r_1, r_2)$ evaluates to *true*. A straightforward extension of join to streams gives the following semantics (in rough terms): At any time $t$, the set of output tuples generated thus far by the join between two streams $S_1$ and $S_2$ should be the same as the result of the relational (non-streaming) join between the sets of input tuples that have arrived thus far in $S_1$ and $S_2$.

Stream join is a fundamental operation for relating information from different streams. For example, given two stream of packets seen by network monitors placed at two routers, we can join the streams on packet ids to identify those packets that flowed through both routers, and compute the time it took for each such packet to reach the other router. As another example, an online auction system may generate two event streams: One signals opening of auctions and the other contains bids on the open auctions. A stream join is needed to relate bids with the corresponding open-auction events. As a third example, which involves a non-equality join, consider two data streams that arise in monitoring a cluster machine room, where one stream contains load information collected from different machines, and the other stream contains temperature readings from various sensors in the room. Using a stream join, we can look for possible correlations between loads on machines and temperatures at different locations in the machine room. In this case, we need to relate temperature readings and load data with close, but necessarily identical, spatio-temporal coordinates.

What makes stream join so special to warrant new approaches different from conventional join processing? In the stream setting, input tuples arrive continuously, and result tuples need to be produced continuously as well. We cannot assume that the input data is already stored or indexed, or that the input rate can be controlled by the query plan. Standard join algorithms that use blocking operations, e.g., sorting, no longer work. Conventional methods for cost estimation and query optimization are also inappropriate, because they assume finite input. Moreover, the long-running nature of stream queries calls for more adaptive processing strategies that can react to changes and fluctuations in data and stream characteristics. The "stateful" nature of stream joins adds another dimension to the challenge. In general, in order to compute the complete result of a stream join, we need to retain all past arrivals as part of the processing state, because a new tuple may join with an arbitrarily old tuple arrived in the past. This problem is exacerbated by unbounded input streams, limited processing resources, and high performance requirements, as it is impossible in the long run to keep all past history in fast memory.

This chapter provides an overview of research problems, recent advances, and future research directions in stream join processing. We start by elucidating the model and semantics for stream joins in Section 2. Section 3 focuses on join state management—the important problem of how to cope with large and potentially unbounded join state given limited memory. Section 4 covers fundamental algorithms for stream join processing. Section 5 discusses aspects of stream join optimization, including objectives and techniques for optimizing multi-way joins. We conclude the chapter in Section 6 by pointing out several related research areas and proposing some directions for future research.

## 2. Model and Semantics

**Basic Model and Semantics.** A stream is an unbounded sequence of stream tuples of the form $\langle s, t \rangle$ ordered by $t$, where $s$ is a relational tuple and $t$ is the *timestamp* of the stream tuple. Following a "reductionist" approach, we conceptually regard the *(unwindowed) stream join* between streams $S_1$ and $S_2$ to be a view defined as the (bag) relational join between two append-only bags $S_1$ and $S_2$. Whenever new tuples arrive in $S_1$ or $S_2$, the view must be updated accordingly. Since relational join is monotonic, insertions into $S_1$ and $S_2$ can result only in possible insertions into the view. The sequence of resulting insertions into the view constitutes the output stream of the stream join between $S_1$ and $S_2$. The timestamp of an output tuple is the time at which the insertion should be reflected in view, i.e., the larger of the timestamps of the two input tuples.

Alternatively, we can describe the same semantics operationally as follows: To compute the stream join between $S_1$ and $S_2$, we maintain a *join state* containing all tuples received so far from $S_1$ (which we call $S_1$'s join state) and those from $S_2$ (which we call $S_2$'s join state). For each new tuple $s_1$ arriving in $S_1$, we record $s_1$ in $S_1$'s join state, probe $S_2$'s join state for tuples joining with $s_1$, and output the join result tuples. New tuples arriving in $S_2$ are processed in a symmetrical fashion.

**Semantics of Sliding-Window Joins.** An obvious issue with unwindowed stream joins is that the join state is unbounded and will eventually outgrow memory and storage capacity of the stream processing system. One possibility is to restrict the scope of the join to a recent window, resulting in a *sliding-window stream join*. For binary joins, we call the two input streams *partner stream* of each other. Operationally, a *time-based sliding window* of duration $w$ on stream $S$ restricts each new partner stream tuple to join only with $S$ tuples that arrived within the last $w$ time units. A *tuple-based sliding window* of size $k$ restricts each new partner stream tuple to join only with the last $k$ tuples arrived in $S$. Both types of windows "slide" forward, as time advances or new stream tuples arrive, respectively. The sliding-window semantics enables us to purge from the join state any tuple that has fallen out of the current window, because future arrivals in the partner stream cannot possibly join with them.

*Continuous Query Language*, or *CQL* for short [2], gives the semantics of a sliding-window stream join by regarding it as a relational join view over the sliding windows, each of which contains the bag of tuples in the current window of the respective stream. New stream tuples are treated as insertion into the windows, while old tuples that fall out of the windows are treated as deletions. The resulting sequences of updates on the join view constitutes the output stream of the stream join. Note that deletions from the windows can result in deletions from the view. Therefore, sliding-window stream joins are not monotonic. The presence of deletions in the output stream does complicate semantics considerably. Fortunately, in many situations users may not care about these deletions at all, and CQL provides an `Istream` operator for removing them from the output stream. For a time-based sliding-window join, even if we do not want to ignore deletions in the output stream, it is easy to infer when an old output tuple needs to be deleted by examining the timestamps of the input tuples that generated it. For this reason, time-based sliding-window join under the CQL semantics is classified as a *weak non-monotonic* operator by Golab and Özsu [24]. However, for a tuple-based sliding-window join, how to infer deletions in the output stream timely and efficiently without relying on explicitly generated "negative tuples" still remains an open question [24].

There is an alternative definition of sliding-window stream joins that does not introduce non-monotonicity. For a time-based sliding-window join with duration $w$, we simply regard the stream join between $S_1$ and $S_2$ as a relational join view over append-only bags $S_1$ and $S_2$ with an extra "window join condition": $-w \leq S_1.t - S_2.t \leq w$. As in the case of an unwindowed stream join, the output stream is simply the sequence of updates on the view resulting from the insertions into $S_1$ and $S_2$. Despite the extra window join condition, join remains monotonic; deletions never arise in the output stream because $S_1$ and $S_2$ are append-only. This definition of time-based sliding-window join has been used by some, e.g., [10, 27]. It is also possible to define a tuple-based sliding-window join as a monotonic view over append-only bags (with the help of an extra attribute that records the sequence number for each tuple in an input stream), though the definition is more convoluted. This alternative semantics yields the same sequence of insertions as the CQL semantics. In the remainder of this chapter, we shall assume this semantics and ignore the issue of deletions in the output stream.

**Relaxations and Variations of the Standard Semantics.** The semantics of stream joins above requires the output sequence to reflect the complete sequence of states of the underlying view, in the exact same order. In some settings this requirement is relaxed. For example, the stream join algorithms in [27] may generate output tuples slightly out of order. The XJoin-family of algorithms (e.g., [41, 33, 38]) relaxes the single-pass stream processing model and allows some tuples to be spilled out from memory and onto disk to be processed later, which means that output tuples may be generated out of order. In any case, the correct output order can be reconstruct from the tuple timestamps. Besides relaxing the requirement on output ordering, there are also variations of sliding windows that offer explicit control over what states of the view can be ignored. For example, with the "jumping window" semantics [22], we divide the sliding window into a number of sub-windows; when the newest sub-window fills up, it is appended to the sliding window while the oldest sub-window in the sliding window is removed, and then the query is re-evaluated. This semantics induces a window that is "jumping" periodically instead of sliding gradually.

**Semantics of Joins between Streams and Database Relations.** Joins between streams and time-varying database relations have also been considered [2, 24]. Golab and Özsu [24] proposed a *non-retroactive relation* semantics, where each stream tuple joins only with the state of the time-varying database relation at the time of its arrival. Consequently, an update on the database relation does not retroactively apply to previously generated output tuples. This semantics is also supported by CQL [2], where the query can be interpreted as a join between the database relation and a zero-duration sliding window over the stream containing only those tuples arriving at the current time. We shall assume this semantics in our later discussion on joining streams and database relations.

## 3. State Management for Stream Joins

In this section, we turn specifically to the problem of *state management* for stream joins. As discussed earlier, join is stateful operator; without the sliding-window semantics, computing the complete result of a stream join generally requires keeping unbounded state to remember all past tuples [1]. The question is: What is the most effective use of the limited memory resource? How do we decide what part of the join state to keep and what to discard? Can we mitigate the problem by identifying and purging "useless" parts of the join state without affecting the completeness of the result? When we run out of memory and are no longer able to produce the complete result, how do we then measure the "error" in an incomplete result, and how do we manage the join state in a way to minimize this error?

Join state management is also relevant even for sliding-window joins, where the join state is bounded by the size of the sliding windows. Sometimes, sliding windows may be quite large, and any further reduction of the join state is welcome because memory is often a scarce resource in stream processing systems. Moreover, if we consider a more general stream processing model where streams are processed not just in fast main memory but instead in a memory hierarchy involving smaller, faster caches as well as larger, slower disks, join state management generalizes into the problem of deciding how to ferry data up and down the memory hierarchy to maximize processing efficiency.

One effective approach towards join state management is to exploit "hard" constraints in the input streams to reduce state. For example, we might know that for a stream, the join attribute is a key, or the value of the join attribute always increases over time. Through reasoning with these constraints and the join condition, we can sometimes infer that certain tuples in the join state cannot contribute to any future output tuples. Such tuples can then be purged from the join state without compromising result completeness. In Section 3.1, we examine two techniques that generalize constraints in the stream setting and use them for join state reduction.

Another approach is to exploit statistical properties of the input streams, which can be seen as "soft" constraints, to help make join state management decisions. For example, we might know (or have observed) that the frequency of each join attribute value is stable over time, or that the join attribute values in a stream can be modeled by some stochastic process, e.g., random walk. Such knowledge allows us to estimate the benefit of keeping a tuple in the join state (for example, as measured by how many output tuples it is expected to generate over a period of time). Because of the stochastic nature of such knowledge, we usually cannot guarantee result completeness. However, this approach can be used to minimize the expected error in the incomplete result, or to optimize the organization of the join state in a memory hierarchy to maximize performance. We discuss this approach in Section 3.2.

### 3.1 Exploiting Constraints

***k*-Constraints.** Babu et al. [7] introduced *$k$-constraints* for join state reduction. The parameter $k$ is an *adherence parameter* that specifies how closely a stream adheres to the constraint. As an example of $k$-constraints, consider first a "strict" *ordered-arrival* constraint on stream $S$, which requires that the join attribute values of $S$ tuples never decrease over time. In a network monitoring application, a stream of TCP/IP packets transmitted from a source to a destination should arrive in the order of their source timestamps (denoted by $t_s$ to distinguish them from the tuple timestamps $t$). However, suppose that for efficiency, we instead use UDP, a less reliable protocol with no guarantee on the delivery order. Nonetheless, if we can bound the extent of packet reordering that occur in practice, we can relax the constraint into an ordered-arrival $k$-constraint: For any tuple $s$, a tuple $s'$ with an earlier source timestamp (i.e., $s'.t_s < s.t_s$) must arrive as or before the $k$-th tuple following $s$. A smaller $k$ implies a tighter constraint; a constraint with $k = 0$ becomes strict.

To see how $k$-constraints can be used for join state reduction, suppose we join the packet stream $S$ in the above example with another stream $S'$ using the condition $|S.t_s - S'.t_s| \leq 10$. Without any constraint on $S.t_s$, we must remember all $S'$ tuples in the join state, because any future $S$ tuple could arrive with a joining $t_s$ value. With the ordered-arrival $k$-constraint on $S.t_s$, however, we can purge a tuple $s' \in S'$ from the join state as soon as $k$ tuples have arrived in $S$ following some $S$ tuple with $t_s > s'.t_s + 10$. The reason is that the $k$-constraint guarantees any subsequent $S$ tuples will have source timestamps strictly greater than $s'.t_s + 10$ and therefore not join with $s'$. Other $k$-constraints considered by [7] include generalizations of referential integrity constraints and *clustered-arrival* constraints.

Although $k$-constraints provide some "slack" through the adherence parameter $k$, strictly speaking they are still hard constraints in that we assume the conditions must hold strictly after $k$ arrivals. Babu et al. also developed techniques for monitoring streams for $k$-constraints and determining the value of $k$ at runtime. Interestingly, $k$-constraints with dynamically observed $k$ become necessarily soft in nature: They can assert that the constraints hold with high probability, but cannot guarantee them with absolute certainty.

**Punctuations.** In contrast to $k$-constraints, whose forms are known a priori, *punctuations*, introduced by Tucker et al. [40], are constraints that are dynamically inserted into a stream. Specifically, a punctuation is a tuple of patterns specifying a predicate that must evaluate to *false* for all future data tuples in the stream. For example, consider an auction system with two streams:

$Auction(id, info, t)$ generates a tuple at the opening of each auction (with a unique auction id), and $Bid(auction\_id, price, t)$ contains bids for open auctions. When an auction with id $a_i$ closes, the system inserts a punctuation $\langle a_i, *\rangle$ into the $Bid$ stream to signal that there will be no more bids for auction $a_i$. Also, since auction ids are unique, following the opening of every auction $a_j$, the system can also insert a punctuation $\langle a_j, *\rangle$ into $Auction$ to signal that will be no other auctions with the same id.

Ding et al. [17] developed a stream join algorithm called PJoin to exploit punctuations. When a punctuation arrives in a stream, PJoin examines the join state of the partner stream and purges those tuples that cannot possibly join with future arrivals. For example, upon the arrival of a punctuation $\langle a_i, *\rangle$ in $Bid$, we can purge any $Auction$ tuples in the join state with id $a_i$ (provided that they have already been processed for join with all past $Bid$ tuples). PJoin also propagates punctuations to the output stream. For example, after receiving $\langle a_i, *\rangle$ from *both* input streams, we can propagate $\langle a_i, *, *\rangle$ to the output, because we are sure that no more output tuple with $a_i$ can be generated. Punctuation propagation is important because propagated punctuations can be further exploited by downstream operators that receive the join output stream as their input. Ding and Rundensteiner [18] further extended their join algorithm to work with sliding windows, which allow punctuations to be propagated quicker. For example, suppose that we set the sliding window to 24 hours, and 24 hours have past after we saw punctuation $\langle a_i, *\rangle$ from $Auction$. Even if we might not have seen $\langle a_i, *\rangle$ yet from $Bid$, in this case we can still propagate $\langle a_i, *, *\rangle$ to the output, because future $Bid$ tuples cannot join with an $Auction$ tuple that has already fallen outside the sliding window.

While punctuations are more flexible and generally more expressive than $k$-constraints, they do introduce some processing overhead. Besides the overhead of generating, processing, and propagating punctuations, we note that some past punctuations need to be retained as part of the join state, thereby consuming more memory. For stream joins, past punctuations cannot be purged until we can propagate them, so it is possible to accumulate many punctuations. Also, not all punctuations are equally effective in join state reduction, and their effectiveness may vary for different join conditions. We believe that further research on the trade-off between the cost and the benefit of punctuations is needed, and that managing the "punctuation state" poses an interesting problem parallel to join state management itself.

## 3.2 Exploiting Statistical Properties

Strictly speaking, both $k$-constraints and punctuations are hard constraints. Now, we explore how to exploit "soft" constraints, or statistical properties of input streams, in join state management. Compared with hard constraints, these soft constraints can convey more information relevant to join state management. For example, consider again the UDP packet stream discussed in Section 3.1. Depending on the characteristics of the communication network, $k$ may need to be very large for the ordered-arrival $k$-constraint to hold. However, it may turn out that $99\%$ of the time the extent of packet reordering is limited to a much smaller $k'$, and that $80\%$ of the time reordering is limited to an even smaller $k''$. Soft, statistical constraints are better at capturing these properties and enabling optimization based on common cases rather than the worst case.

Given a limited amount of memory to hold the join state, for each incoming stream tuple, we need to make a decision—not unlike a cache replacement decision—about whether to discard the new tuple (after joining it with the partner stream tuples in the join state) or to retain it in the join state; in the latter case, we also need to decide which old tuple to discard from the join state to make space for the new one. In the following, we shall use the term "cache" to refer to the memory available for keeping the join state.

Before we proceed, we need to discuss how to evaluate a join state management strategy. There are two major perspectives, depending on the purpose of join state management. The first perspective assumes the single-pass stream processing model where output tuples can be produced only from the part of the join state that we choose to retain in cache. In this case, our goal is to minimize the error in (or to maximize the quality of) the output stream compared with the complete result. A number of popular measures have been defined from this perspective:

- *Max-subset.* This measure, introduced by Das et al. [15], aims at producing as many output tuples as possible. (Note that any reasonable stream join algorithm would never produce any incorrect output tuples, so we can ignore the issue of false positives.) Because input streams are unbounded, we cannot compare two strategies simply by comparing the total numbers of output tuples they produce—both may be infinite. The approach taken by Srivastava and Widom [37] is to consider the ratio between the number of output tuples produced up to some time $t$ and the number of tuples in the complete result up to $t$. Then, a reasonable goal is to maximize this ratio as $t$ tends to infinity.

- *Sampling rate.* Like max-subset, this measure aims at producing as many output tuples as possible, but with the additional requirement that the set of output tuples constitutes a uniform random sample of the complete join result. Thus, the goal is to maximize the sampling rate. This measure is first considered in the stream join setting by [37].

- *Application-defined importance.* This measure is based on the notion of importance specific to application needs. For example, *Aurora* [39] allows applications to define *value-based quality-of-service functions* that specify the utilities of output tuples based on their attribute values. The goal in this case is to maximize the utility of the join result.

The second perspective targets expected performance rather than result completeness. This perspective relaxes the single-pass processing model by allowing tuples to be spilled out from memory and onto disk to be processed later in "mop-up" phases. Assuming that we still produce the complete answer, our goal is to minimize the total processing cost of the online and the mop-up phases. One measure defined from this perspective is the *archive metric* proposed by [15]. This measure has also been used implicitly by the XJoin-family of algorithms ([41, 33, 38], etc.). As it is usually more expensive to process tuples that have been spilled out to disk, a reasonable approximation is to try to leave as little work as possible to the mop-up phases; this goal roughly compatible with max-subset's objective of getting as much as possible done online.

In the remainder of this section, we focus first and mostly on the max-subset measure. Besides being a reasonable measure in its own right, techniques developed for max-subset are roughly in line with the archive metric, and can be generalized to certain application-defined importance measures through appropriate weighting. Next, we discuss the connection between classic caching and join state management, and state management for joins between streams and database relations. Finally, we briefly discuss the sampling-rate measure towards the end of this section.

**Max-Subset Measure.** Assuming perfect knowledge of the future arrivals in the input streams, the problem of finding the optimal sequence (up to a given time) of join state management decisions under max-subset can be cast as a network flow problem, and can be solved offline in time polynomial to the length of the sequence and the size of the cache [15]. In practice, however, we need an online algorithm that does not have perfect knowledge of the future. Unfortunately, without any knowledge (statistical or otherwise) of the input streams, no online algorithm—not even a randomized one—can be $k$-competitive (i.e., generating at least $1/k$ as many tuples as an optimal offline algorithm) for any $k$ independent of the length of the input streams [37]. This hardness result highlights the need to exploit statistical properties of the input streams. Next, we review previous work in this area, starting with specific scenarios and ending with a general approach.

**Frequency-Based Model.** In the *frequency-based model*, the join attribute value of each new tuple arriving in a stream is drawn independently from a probability distribution that is stationary (i.e., it does not change over time). This model is made explicit by [37] and discussed as special case by [45], although it has been implicitly assumed in the development of many join state management techniques. Under this model, assuming an unwindowed stream equijoin, we can calculate the *benefit* of a tuple $s$ as the product of the partner stream arrival rate and the probability that a new partner stream tuple has the same join attribute value as $s$. This benefit measures how many output tuples $s$ is expected to generate per unit time in the future. A straightforward strategy is to replace the tuple with the lowest benefit. This strategy, called *PROB*, was proposed by [15], and can be easily shown to be optimal for unwindowed joins. For sliding-window joins, an alternative strategy called *LIFE* was proposed, which weighs a tuple's benefit by its remaining lifetime in the sliding window. Unfortunately, neither PROB nor LIFE is known to be optimal for sliding-window joins. To illustrate, suppose that we are faced with the choice between two tuples $s_1$ and $s_2$, where $s_1$ has a higher probability of joining with an incoming tuple, but $s_2$ has a longer lifetime, allowing it to generate more output tuples than $s_2$ eventually. PROB would prefer $s_1$ while LIFE would prefer $s_2$; however, neither choice is always better, as we will see later in this section.

The frequency-based model is also implicitly assumed by [38] in developing the *RPJ* (*rate-based progressive join*) algorithm. RPJ stores the in-memory portion of each input stream's join state as a hash table, and maintains necessary statistics for each hash partition; statistics for individual join attribute values within each partition are computed assuming local uniformity. When RPJ runs out of memory, it flushes the partition with lowest benefit out to disk. This strategy is analogous to PROB.

Kang et al. [30] assumed a simplified version of the frequency-based model, where each join attribute value occurs with equal frequency in both input streams (though stream arrival rates may differ). With this simplification, the optimal strategy is to prefer keeping the slower stream in memory, because the tuples from the slower stream get more opportunities to join with an incoming partner stream tuple. This strategy is also consistent with PROB. More generally, *random load shedding* [39, 4], or *RAND* [15], which simply discards input stream tuples at random, is also justifiable under the max-subset measure by this equal-frequency assumption.

**Age-Based Model.** The *age-based model* of [37] captures a scenario where the stationarity assumption of the frequency-based model breaks down because of correlated tuple arrivals in the input streams. Consider the $Auction$ and $Bid$ example from Section 3.1. A recent $Auction$ tuple has a much better chance of joining with a new $Bid$ than an old $Auction$ tuple. Furthermore, we may be able to assume that the bids for each open auction follow a similar arrival pattern. The age-based model states that, for each tuple $s$ in stream $S$ (with partner stream $S'$), the expected number of output tuples that $s$ generates at each time step during its lifetime is given by a function $p(\Delta t)$, where $\Delta t$

denotes the age of the tuple (i.e., how long it has been in the join state). The age-based model further assumes that function $p(\Delta t)$ is the same for all tuples from the same stream, independent of their join attribute values. At the first glance, this assumption may appear quite strong: If we consider two tuples with the same join attribute value arriving at two different times $t_1$ and $t_2$, we should have $p(t - t_1) = p(t - t_2)$ for all $t$ when both tuples are in the join state, which would severely limit the form of function $p$. However, this issue will not arise, for example, if the join attribute is a key of the input stream (e.g., $Auction$). Because foreign-key joins are so common, the age-based model may be appropriate in many settings.

An optimal state management strategy for the age-based model, called *AGE*, was developed by [37]. Given the function $p(\Delta t)$, AGE calculates an optimal age $\Delta t_o$ such that the expected number of output tuples generated by a tuple per unit time is maximized when it is kept until age $\Delta t_o$. Intuitively, if every tuple in the cache is kept for exactly $\Delta t_o$ time steps, then we are making the most efficient use of every slot in the cache. This optimal strategy is possible if the arrival rate is high enough to keep every cache slot occupied. If not, we can keep each tuple to an age beyond the optimal, which would still result in an optimal strategy assuming that $p(\Delta t)$ has no local minima. A heuristic strategy for the case where $p(\Delta t)$ has local minima is also provided in [37].

**Towards General Stochastic Models.** There are many situations where the input stream follows neither the frequency-based model nor the age-based model. For example, consider a measurement stream $S_1$ generated by a network of sensors. Each stream tuple carries a timestamp $t_m$ recording the time at which the measurement was taken by the sensor (which is different from the stream timestamp $t$). Because of processing delays at the senors, transmission delays in the network, and a network protocol with no in-order delivery guarantee (e.g., UDP), the $t_m$ values do not arrive in order, but may instead follow a discretized bounded normal distribution centered at the current time minus the average latency. Figure 1.1 shows the pdf (probability density function) of this distribution, which moves right as time progresses. Suppose there is a second stream $S_2$ of timestamped measurements of a different type coming from another network of sensors, which is slower and less reliable. The resulting distribution has a higher variance and looser bounds, and lags slightly behind that of $S_1$. To correlate measurements from $S_1$ and $S_2$ by time, we use an equijoin on $t_m$. Intuitively, as the pdf curve for $S_2$ moves over the join attribute value of a cached $S_1$ tuple, this tuple gets a chance of joining with each incoming $S_2$ tuple, with a probability given by $S_2$'s pdf at that time. Clearly, these streams do not follow the frequency-based model, because the frequency of each $t_m$ value varies over time. They do not follow the age-based model either, because each arriving tuple may have a different $p(\Delta t)$ function, which depends on

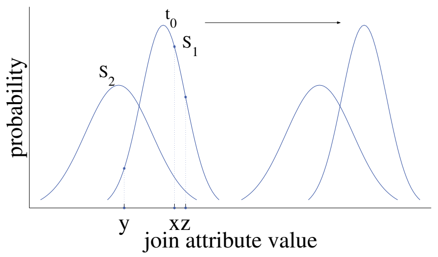

*Figure 1.1.* Drifting normal distributions.

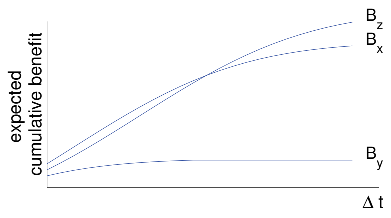

*Figure 1.2.* Example ECBs.

the location of the partner stream's pdf. Blindly applying a specific join state management strategy without verifying its underlying assumption may lead to very poor performance. To illustrate, consider the two tuples $x$ and $y$ from stream $S_2$ in Figure 1.1 currently cached at time $t_0$ as part of the join state. Which tuple should we choose to discard when we are low on memory? Intuitively, it is better to discard $y$ since it has almost already "missed" the moving pdf of $S_1$ and is therefore unlikely to join with future $S_1$ tuples. Unfortunately, if we use the past to predict future, PROB might make the exact opposite decision: $y$ would be kept because it probably has joined more times with $S_1$ *in the past* than $x$.

Work by Xie et al. [45] represents a first step towards developing general techniques to exploit a broader class of statistical properties, without being tied to particular models or assumptions. A general question posed by [45] is, given the stochastic processes modeling the join attribute values of the input stream tuples, what join state management strategy has the best expected performance? In general, the stochastic processes can be non-stationary (e.g., the join attribute value follows a random walk, or its mean drifts over time) and correlated (e.g., if one stream has recently produced a certain value, then it becomes more likely for the other stream to produce the same value).

Knowing the stochastic processes governing the input streams gives us considerable predictive power, but finding the optimal join state management strategy is still challenging. A brute-force approach (called *FlowExpect* in [45]) would be the following. Conceptually, starting from the current time and the current join state, we enumerate all possible sequences of future "state management actions" (up to a given length), calculate the expected number of output tuples for each sequence, and identify the optimal sequence. This search problem can be formulated and solved as a network flow problem. The first action in the optimal sequence is taken at the current time. As soon as any new tuple arrives, we solve the problem again with the new join state, and take the first action in the new optimal sequence. The process then repeats. Interestingly, Xie et al. [45] showed that it is not enough to consider all possible sequences of *unconditional* state management actions; we must also consider strategies that make actions *conditional* upon the join attribute values of future tuples. An example of a conditional action at a future time $t$ might be: “If the new tuple arriving at $t$ has value $100$ as its join attribute, then discard the new tuple; otherwise use it to replace the tuple currently occupying the fifth cache slot.” Unfortunately, searching through the enormous space of conditional action sequences is not practical. Therefore, we need to develop simpler, more practical approaches.

It turns out that under certain conditions, the best state management action is clear. Xie et al. [45] developed an *ECB dominance test* (or *dom-test* for short) to capture these conditions. From the stochastic processes governing the input streams, we can compute a tuple $s$’s *ECB* (*expected cumulative benefit*) with respect to the current time $t_0$ as a function $B_s(\Delta t)$, which returns the number of output tuples that $s$ is expected to generate over the period $(t_0, t_0 + \Delta t]$. As a concrete example, Figure 1.2 plots the ECBs of tuples $x$, $y$, and $z$ from stream $S_2$ in Figure 1.1. Intuitively, we prefer removing tuples with the “lowest” ECBs from the cache. The dom-test states that, if the ECB of tuple $s_1$ *dominates* that of tuple $s_2$ (i.e., $B_{s_1}(\Delta t) \geq B_{s_2}(\Delta t)$ for all $\Delta t > 0$), then keeping $s_1$ is better than or equally good as keeping $s_2$. For example, from Figure 1.2, we see that tuple $y$ is clearly the least preferable among the three. However, because the ECBs of $x$ and $z$ cross over, the dom-test is silent on the choice between $x$ and $z$. To handle “incomparable” ECBs such as these, Xie et al. proposed a heuristic measure that combines the ECB with a heuristic “survival probability” function $L_s(\Delta t)$ estimating the probability for tuple $s$ to be still cached at time $t_0 + \Delta t$. Intuitively, if we estimate that $x$ and $z$ will be replaced before the time when their ECBs cross, then $x$ is more preferable; otherwise, $z$ is more preferable. Although the heuristic strategy cannot guarantee optimality in all cases, it always agrees with the decision of the dom-test whenever that test is applicable.

It is instructive to see how the general techniques above apply to specific scenarios. To begin, consider the simple case of unwindowed stream joins under the frequency-based model. The ECB of a tuple $s$ is simply a linear function $B_s(\Delta t) = b(s)\Delta t$, where $b(s)$ is the number of output tuples that $s$ is expected to generate per unit time, consistent with the definition of “benefit” discussed earlier in the context of the frequency-based model. Obviously, for two tuples $s_1$ and $s_2$, $s_1$’s ECB dominates $s_2$’s ECB if and only if $b(s_1) \geq b(s_2)$. Therefore, the dom-test basically yields PROB, and provides a proof of its optimality. The case of sliding-window joins is considerably more complex, and as discussed earlier, the optimal state management strategy is not known. As illustrated by Figure 1.3, the ECB of a tuple $s$ consists of two connected pieces: The first piece has slope $b(s)$, while the second piece is flat and begins at the time $l(s)$

when $s$ will drop out of the sliding window. While the dom-test does not help in comparing $s_1$ and $s_2$ in Figure 1.3, some insight can still be gained from their ECBs. Suppose we decide to cache $s_1$ and discard $s_2$. If at time $l(s_1)$ when $s_1$ exits the join state, a new tuple will be available to take its place and produce at least $B_{s_2}(l(s_2)) - B_{s_1}(l(s_1))$ output tuples during $(l(s_1), l(s_2)]$, then our decision is justified. Still, the exact condition that guarantees the optimality of the decision is complex, and will be an interesting problem for further investigation.

Finally, let us try applying the ECB-based analysis to the age-based model, for which we know that AGE [37] is optimal. Under the age-based model, every new tuple has the same ECB $B_0(\Delta t)$ at the time of its arrival. As the tuple ages in the cache, its ECB "shifts": The ECB of a tuple at age $t$ is $B_t(\Delta t) = B_0(t + \Delta t) - B_0(t)$. For some shapes of $B_0$, it is possible to have ECBs that are not comparable by the dom-test. Figure 1.4 illustrates one such example; the marks on the ECB curves indicate when the respective tuples reach their optimal ages. Between two tuples $old$ and $new$ in Figure 1.4, the correct decision (by AGE) is to ignore the new tuple and keep caching the old tuple (until it reaches its optimal age). Unfortunately, however, the dom-test is unable to come to any conclusion, for the following two reasons. First, the dom-test actually provides a stronger optimality guarantee than AGE: The dom-test guarantees the optimality of its decisions over *any* time period; in contrast, AGE is optimal when the period tends to infinity. Second, the dom-test examines only the two ECBs in question and does not make use of any global information. However, in order to realize that replacing $old$ tuple is not worthwhile in Figure 1.4, we need to be sure that when we do discard $old$ when it reaches its optimal age, there will be a new tuple available at that time with high enough benefit to make up for the loss in discarding $new$ earlier. Indeed, the age-based model allows us to make this conclusion from its assumption that every incoming tuple has the same ECB. It remains an open problem to develop

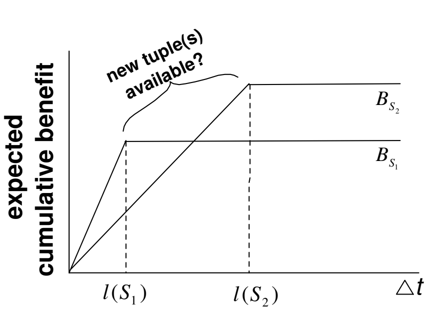

*Figure 1.3.* ECBs for sliding-window joins under the frequency-based model.

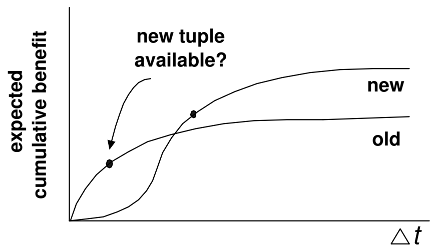

*Figure 1.4.* ECBs under the age-based model.

better, general techniques to overcome the limitations of the dom-test without purposefully "special-casing" for specific scenarios.

An important practical problem is that we may not know in advance the parameter values of the stochastic processes modeling the input streams. One possibility is to use existing techniques to compute stream statistics online, or offline over the history of observations. Another approach proposed by [45] is to monitor certain statistics of the past behavior of the cache and input streams, and use them to estimate the expected benefit of caching. A notable feature of the proposed method is that it considers the form of the stochastic process in order to determine what statistics to monitor. This feature is crucial because, for time-dependent processes, the past is not always indicative of the future. For example, suppose that the join attribute values in a stream follow a distribution whose shape is stationary but mean is drifting over time. Simply tracking the frequency of each value is not meaningful as it changes all the time. Instead, we can subtract the current mean from each observed value, and track the frequency of these offset values, which will remain constant over time.

One direction for future research is to investigate how statistical properties of the input stream propagate to the output of stream joins. This problem is important if we want to apply the techniques in this section to more complex stream queries where the output of a stream join may be the input to another. While there has been some investigation of the non-streaming version of this problem related to its application in query optimization, there are many statistical properties unique to streams (e.g., trends, orderedness, clusteredness) whose propagation through queries is not yet fully understood.

**Relationship to Classic Caching.** A natural question is how the stream join state management problem differs from the classic caching problem. Many cache replacement policies have been proposed in the past, e.g., LFD (longest-forward distance), LRU (least-recently used), LFU (least-frequently used), etc. All seem applicable to our problem. After all, our problem comes down to deciding what to retain in a cache to serve as many "reference requests" by the partner stream as possible. As pointed out by [45], there is a subtle but important difference between caching stream tuples and caching regular objects. When caching regular objects, we can recover from mistakes easily: The penalty of not caching an object is limited to a single cache miss, after which the object would be brought in and cached if needed. In contrast, in the case of stream join state management, a mistake can cost a lot more: If we discard a tuple completely from the join state, it is irrevocably gone, along with all output tuples that it could generate in the future. This difference explains why LFD, the optimal replacement policy for classic caching, turns out to be suboptimal for stream join state management, where references beyond the first one also matter.

Even if we relax the single-pass stream processing model and allow state to be spilled out to disk, join state management still differs from classic caching. The reason is that stream processing systems, e.g., those running the XJoin-family of algorithms, typically recover missing output tuples by processing flushed tuples later offline. In other words, "cache misses" are not processed online as in classic caching—random disk accesses may be too slow for stream applications, and just to be able to detect that there has indeed been a cache miss (as opposed to a new tuple that does not join with any previous arrivals) requires maintaining extra state.

Despite their differences, classic caching and stream join state management can be tackled under the same general analytical framework proposed by [45]. In fact, classic caching can be reduced to stream join state management, and can be analyzed using ECBs, in some cases yielding provably optimal results that agree with or extend classic ones. Such consistency is evidence of the strong link between the two problems, and a hint that some results on classic caching could be brought to bear on the state management problem for stream joins.

**Joining with Database Relation.** Interestingly, unlike the case of joining two streams, state management for joining a stream and a database relation under the non-retroactive relation semantics (Section 2) is practically identical to classic caching [45]. First, it is easy to see that there is no benefit at all in caching any stream tuples, because under the non-retroactive relation semantics they do not join with any future updates to the relation. On the other hand, for tuples in the database relation, their current version can be cached in fast memory to satisfy reference requests by stream tuples. Upon a cache miss, the disk-resident relation can be probed. It would be interesting to investigate whether it makes sense to defer handling of misses to XJoin-style "mop-up" phases. However, care must be taken to avoid joining old stream tuples with newer versions of the database relation.

**The Sampling-Rate Measure.** By design, join state management strategies optimized for max-subset favor input tuples that are more likely to join with the partner stream, causing such tuples to be overrepresented in the result. While this bias is not a problem in many contexts, it can be an issue if a statistically meaningful sample is desired, e.g., to obtain unbiased statistics of the join result. In this case, we should use the sampling-rate measure.

Getting an unbiased random sample of a join result has long been recognized as a difficult problem [34, 12]. The straightforward approach of sampling each input uniformly and then joining them does not work—the result may be arbitrarily skewed and small compared with the actual result. The hardness result from [37] states that for arbitrary input streams, if the available memory is insufficient to retain the entire sliding windows (or the entire history, if the join is unwindowed), then it is impossible to guarantee a uniform random sample for any nonzero sampling rate. The problem is that any tuple we choose to discard may turn out to be the one that will generate all subsequent output tuples.

Srivastava and Widom [37] developed a procedure for generating unbiased random samples of join results under the frequency-based and age-based models. The procedure requires knowledge of the model parameters, and uses them to determine the maximum sampling rate under the constraint that the probability of running out of memory at runtime is sufficiently small. The procedure keeps each stream tuple in the join state until the tuple will not contribute any more result tuples to the sample. Note that not all join result tuples that can be obtained from the join state will actually be output—many may need to discarded in order to keep the sample unbiased. This inefficient use of resources is unavoidable because of the stringent requirement of a truly random sample. A statistically weaker form of sampling called *cluster sampling*, which uses resources more efficiently, was also considered by [37]. Cluster sampling is still unbiased, but is no longer independent; i.e., the inclusion of tuples is not independent of each other. Which type of sampling is appropriate depends on how the join result will be used.

## 4. Fundamental Algorithms for Stream Join Processing

*Symmetric hash join* (*SHJ*) is a simple hashing-based join algorithm, which has been used to support highly pipelined processing in parallel database systems [44]. It assumes that the entire join state can be kept in main memory; the join state for each input stream is stored in a hash table. For each incoming $S$ tuple, SHJ inserts it into the hash table for $S$, and uses it to probe the hash table for the partner stream of $S$ to identify joining tuples. SHJ can be extended to support the sliding-window semantics and the join statement management strategies in Section 3, though SHJ is limited to the single-pass stream processing model. Golab et al. [22] developed main-memory data structures especially suited for storing sliding windows, with efficient support for removing tuples that have fallen out of the sliding windows.

Both *XJoin* [41] and *DPHJ* (*double pipelined hash join*) of *Tukwila* [29] extend SHJ by allowing parts of the hash tables to be spilled out to disk for later processing. This extension removes the assumption that the entire join state must be kept in memory, greatly enhancing the applicability of the algorithm. Tukwila's DPHJ processes disk-resident tuples only when both inputs are exhausted; XJoin schedules joins involving disk-resident tuples whenever the inputs are blocked, and therefore is better suited for stream joins with unbounded inputs. One complication is the possibility of producing duplicate output tuples. XJoin pioneers the use of timestamp marking for detecting duplicates. Timestamps record the period when a tuple was in memory, and the times when a memory-resident hash partition was used to join with the corresponding disk-resident partition of the partner stream. From these timestamps, XJoin is able to infer which pairs of tuples have been processed before.

XJoin is the basis for many stream join algorithms developed later, e.g., [33, 38]. RPJ [38] is the latest in the series. One of the main contributions of RPJ, discussed earlier in Section 3.2, is a statistics-based flushing strategy that tries to keep in memory those tuples that are more likely to join. In contrast, XJoin flushes the largest hash partition; *HMJ* (*hash merge join*) of [33] always flushes corresponding partitions together, and tries to balance memory allocation between incoming streams. Neither XJoin nor HMJ takes tuple join probabilities into consideration. Unlike HMJ, which joins all previously flushed data whenever entering a disk-join phase, RPJ breaks down the work into smaller units, which offer more scheduling possibilities. In particular, RPJ also uses statistics to prioritize disk-join tasks in order to maximize output rate.

There are a number of interesting open issues. First, can we exploit statistics better by allowing flushing of individual tuples instead of entire hash partitions? This extension would allow us to apply the fine-grained join state management techniques from Section 3.2 to the XJoin-family of algorithms. However, the potential benefits must be weighed against the overhead in statistics collection and bookkeeping to avoid duplicates. Second, is it ever beneficial to reintroduce a tuple that has been previously flushed to disk back into memory? Again, what would be the bookkeeping overhead involved? Third, can we develop better statistics collection methods for RPJ? Currently, it maintains statistics on the partition level, but the hash function may map tuples with very different statistics to the same partition.

Sorting-based join algorithms, such as the sort-merge join, have been traditionally deemed inappropriate for stream joins, because sorting is a blocking operation that requires seeing the entire input before producing any output. To circumvent this problem, Dittrich et al. [20] developed an algorithm called *PMJ* (*progressive merge join*) that is sorting-based but non-blocking. In fact, both RPJ and HMJ use PMJ for joining disk-resident parts of the join state. The idea of PMJ is as follows. During the initial sorting phase that creates the initial runs, PMJ sorts portions of both input streams in parallel, and immediately produces join result tuples from the corresponding runs that are in memory at the same time. During the subsequent merge phases that merge shorter runs into longer ones, PMJ again processes both input streams in parallel, and joins them while their runs are in memory at the same times. To ensure that the output contains no duplicates, PMJ does not join tuples from corresponding shorter runs that have been joined in a previous phase; the duplicate avoidance logic is considerably simpler than XJoin. Of course, PMJ pays some price for its non-blocking feature—it does incur a moderate amount of overhead compared to the basic sort-merge join. On the other hand, PMJ also inherits the advantages of sorting-based algorithms over hashing-bashed algorithms, including in particular the ability to handle non-equality joins. A more thorough performance comparison between PMJ and XJoin for equijoins would be very useful.

## 5. Optimizing Stream Joins

**Optimizing Response Time.** Viglas and Naughton [42] introduced the notion of *rate-based optimization* and considered how to estimate the output rate of stream operators. An important observation is that standard cost analysis based on total processing cost is not applicable in the stream setting, because infinite costs resulted from unbounded inputs cannot be compared directly. Even if one can "hack" the analysis by assuming a large (yet bounded) input, classic analysis may produce incorrect estimate of the output rate since it ignores the rate at which inputs (or intermediate result streams) are coming. Specifically, classic analysis assumes that input is available at all times, but in practice operators could be blocked by the input. The optimization objectives considered by [42] are oriented towards response time: For a stream query, how can we produce the largest number of output tuples in a given amount of time, or produce a given number of output tuples in the shortest amount of time?

As an example of response-time optimization, Hammad et al. [28] studied shared processing of multiple sliding-window joins, focusing on developing scheduling strategies aimed at reducing response times across queries. More broadly speaking, work on non-blocking join algorithms, e.g., XJoin and PMJ discussed earlier, also incorporate response-time considerations.

**Optimizing Unit-Time Processing Cost.** Kang et al. [30] were among the first to focus specifically on optimization of stream joins. They made the same observation as in [42] that optimizing the total processing cost is no longer appropriate with unbounded input. However, instead of optimizing response time, they propose to optimize the processing cost per unit time, which is equivalent to the average processing cost per tuple weighted by the arrival rate. Another important observation made by [30] is that the best processing strategy may be *asymmetric*; i.e., different methods may be used for joining a new $S_1$ tuple with $S_2$'s join state and for joining a new $S_2$ tuple with $S_1$'s join state. For example, suppose that $S_1$ is very fast and $S_2$ is very slow. We may index $S_2$'s join state as a hash table while leaving $S_1$'s join state not indexed. The reason for not indexing $S_1$ is that its join state is frequently updated (because $S_1$ is fast) but rarely queried (because $S_2$ is slow).

Ayad and Naughton [4] provided more comprehensive discussions of optimization objectives for stream queries. An important observation is that, given enough processing resources, the steady-state output rate of a query is independent of the execution plan and therefore should be not be the objective of query optimization; a cost-based objective should be used in this case instead.

Another interesting point is that load shedding considerations should be incorporated into query optimization: If we simply shed load from a plan that was originally optimized assuming sufficient resources, the resulting plan may be suboptimal.

**Optimizing Multi-Way Stream Joins.** XJoin can be used to implement multi-way joins in a straightforward manner. For instance, a four-way join among $S_1$, $S_2$, $S_3$, and $S_4$ can be implemented as a series of XJoins, e.g., $((S_1 \text{ XJoin } S_2) \text{ XJoin } S_3) \text{ XJoin } S_4$. Since XJoin needs to store both of its inputs in hash tables, the example plan above in effect materializes the intermediate results $S_1 \text{ XJoin } S_2$ and $(S_1 \text{ XJoin } S_2) \text{ XJoin } S_3$. An obvious disadvantage of this plan is that these intermediate results can become quite large and costly to maintain. Another disadvantage is that this plan is static and fixes the join order. For example, a new $S_3$ tuple must be joined with the materialized $S_1 \text{ XJoin } S_2$ first, and then with $S_4$; the option of joining the new $S_3$ tuple first with $S_4$ is simply not available.

Viglas et al. [43] proposed *MJoin* to combat the above problems. MJoin maintains a hash table for each input involved in the multi-way join. When a tuple arrives, it is inserted into the corresponding hash table, and then used to probe all other hash tables in some order. This order can be different for tuples from different input streams, and can be determined based on join selectivities. Similar to XJoin, MJoin can flush join state out to disk when low on memory. Flushing is random (because of the assumption of a simple statistical model), but for the special case of star joins (where all streams join on the same attribute), flushing is “coordinated”: When flushing one tuple, joining tuples from other hash tables are also flushed, because no output tuples can be produced unless joining tuples are found in all other hash tables. Note that coordinated flushing does not bring the same benefit for binary joins, because in this case output tuples are produced by joining an incoming tuple with the (only) partner stream hash table, not by joining two old tuples from difference hash tables.

Finding the optimal join order in MJoin is challenging. A simple heuristic that tracks selectivity for each hash table independently would have trouble with the following issues: (1) Selectivities can be correlated; e.g., a tuple that already joins with $S_1$ will be more likely to join with $S_2$. (2) Selectivities may vary among individual tuples; e.g., one tuple may join with many $S_1$ tuples but few $S_2$ tuples, while another tuple may behave in the exact opposite way. The first issue is tackled by [5], who provided a family of algorithms for adaptively finding the optimal order to apply a series of filters (joining a tuple with a stream can be regarded as subjecting the tuple to a filter) through runtime profiling. In particular, the *A-Greedy* algorithm is able to capture correlations among filter selectivities, and is guaranteed to converge to an ordering within a constant factor of the optimal. The theoretical guarantee extends to star joins; for general join graphs, though A-Greedy still can be used, the theoretical guarantee no longer holds. The second issue is recently addressed by an approach called *CBR* [9], or *content-based routing*, which makes the choice of query plan dependent on the values of the incoming tuple's "classifier attributes," whose values strongly correlate with operator selectivities. In effect, CBR is able to process each incoming tuple with a customized query plan.

One problem with MJoin is that it may incur a significant amount of recomputation. Consider again the four-way join among $S_1, \ldots, S_4$, now processed by a single MJoin operator. Whenever a new tuple $s_3$ arrives in $S_3$, MJoin in effect executes the query $S_1 \bowtie S_2 \bowtie \{s_3\} \bowtie S_4$; similarly, whenever a new tuple $s_4$ arrives in $S_4$, MJoin executes $S_1 \bowtie S_2 \bowtie S_3 \bowtie \{s_4\}$. The common subquery $S_1 \bowtie S_2$ is processed over and over again for these $S_3$ and $S_4$ tuples. In contrast, the XJoin plan $((S_1$ XJoin $S_2)$ XJoin $S_3)$ XJoin $S_4$ materializes all its intermediate results in hash tables, including $S_1 \bowtie S_2$; new tuples from $S_3$ and $S_4$ simply have to probe this hash table, thereby avoiding recomputation. The optimal solution may well lie between these two extremes, as pointed out by [6]. They proposed an adaptive caching strategy, *A-Caching*, which starts with MJoins and adds join subresult caches adaptively. A-Caching profiles cache benefit and cost online, selects caches dynamically, and allocates memory to caches dynamically. With this approach, the entire spectrum of caching options from MJoins to XJoins can be explored.

A number of other papers also consider multi-way stream joins. Golab and Özsu [23] studied processing and optimization of multi-way sliding-window joins. Traditionally, we eagerly remove (expire) tuples that are no longer part of the sliding window, and eagerly generate output tuples whenever input arrives. The authors proposed algorithms supporting lazy expiration and lazy evaluation as alternatives, which achieve higher efficiency at the expense of higher memory requirements and longer response times, respectively. Hammad et al. [27] considered multi-way stream joins where a time-based window constraint can be specified for each pair (or, in general, subset) of input streams. An interesting algorithm called *FEW* is proposed, which computes a forward point in time before which all arriving tuples can join, thereby avoiding repeated checking of window constraints.

*Eddies* [3] are a novel approach towards stream query processing and optimization that is markedly different from the standard plan-based approaches. Eddies eliminate query plans entirely by routing each input tuple adaptively across the operators that need to process it. Interestingly, in eddies, the behavior of *SteM* [36] mimics that of MJoin, while *STAIRS* [16] is able to emulate XJoin. Note that while eddies provide the *mechanisms* for adapting the processing strategy on an individual tuple basis, currently their *policies* typically do not result in plans that change for every incoming tuple. It would be nice to see how features of CBR can be supported in eddies.

## 6. Conclusion

In this chapter, we have presented an overview of research problems and recent advances in join processing for data streams. Stream processing is a young and exciting research area, yet it also has roots in and connections to well-established areas in databases as well as computer science in general. In Section 3.2, we have already discussed the relationship between stream join state management and classic caching. Now, let us briefly re-examine parts of this chapter in light of their relationship to *materialized views* [25].

The general connection between stream processing and materialized views has long been identified [8]. This connection is reflected in the way that we specify the semantics of stream joins—by regarding them as views and defining their output as the view update stream resulting from base relation updates (Section 2). Recall that the standard semantics requires the output sequence to reflect the exact sequence of states of the underlying view, which is analogous to the notion of *complete and strong consistency* of a data warehouse view with respect to its source relations [46]. The connection does not stop at the semantics. The problem of determining what needs to be retained in the state to compute a stream join is analogous to the problem of deriving auxiliary views to make a join view *self-maintainable* [35]. Just as constraints can be used to reduce stream join state (Section 3.1), they have also been used to help expire data from data warehouses without affecting the maintainability of warehouse views [21]. For a stream join $S_1 \bowtie \cdots \bowtie S_n$, processing an incoming tuple from stream $S_i$ is analogous to maintaining a join view incrementally by evaluating a maintenance query $S_1 \bowtie \cdots \bowtie \Delta S_i \bowtie \cdots \bowtie S_n$. Since there are $n$ different forms of maintenance queries (one for each $i$), it is natural to optimize each form differently, which echoes the intuition behind the asymmetric processing strategy of [30] and MJoin [43]. In fact, we can optimize the maintenance query for each *instance* of $\Delta S_i$, which would achieve the same goal of supporting a customized query plan for each tuple as CBR [9]. Finally, noticing that the maintenance queries run frequently and share many common subqueries, we may choose to materialize some subqueries as additional views to improve query performance, which is also what A-Caching [6] tries to accomplish.

Of course, despite high-level similarities, techniques from the two areas—data streams and materialized views—may still differ significantly in actual details. Nonetheless, it would be nice to develop a general framework that unifies both areas, or, less ambitiously, to apply ideas from one area to the other. Many such possibilities exist. For example, methods and insights from the well-studied problems of *answering query using views* [26] and *view selection* [14] could be extended and applied to data streams: Given a set of stream queries running continuously in a system, what materialized views (over join states and database relations) and/or additional stream queries can we create to improve the performance of the system? Another area is distributed stream processing. Distributed stream processing can be regarded as view maintenance in a distributed setting, which has been studied extensively in the context of data warehousing. Potentially applicable in this setting are techniques for making warehouse self-maintainable [35], optimizing view maintenance queries across distributed sources [31], ensuring consistency of multi-source warehouse views [46], etc. Conversely, stream processing techniques can be applied to materialized views as well. In particular, view maintenance could benefit from optimization techniques that exploit update stream statistics (Section 3.2). Also, selection of materialized views for performance can be improved by adaptive caching techniques (Section 5).

Besides the future work directions mentioned above and throughout the chapter, another important direction worth exploring is the connection between data stream processing and *distributed event-based systems* [19] such as publish/subscribe systems. Such systems need to scale to thousands or even millions of subscriptions, which are essentially continuous queries over event streams. While efficient techniques for handling continuous selections already exist, scalable processing of continuous joins remains a challenging problem. Hammad et al. [28] considered shared processing of stream joins with identical join conditions but different sliding-window durations. We need to consider more general query forms, e.g., joins with different join conditions as well as additional selection conditions on input streams. *NiagaraCQ* [13] and *CACQ* [32] are able to group-process selections and share processing of identical join operations. However, there is no group or shared processing of joins with different join conditions, and processing selections separately from joins limits optimization potentials. *PSoup* [11] treats queries as data, thereby allowing set-oriented processing of queries with arbitrary join and selection conditions. Still, new indexing and processing techniques must be developed for the system to be able to process each event in time sublinear in the number of subscriptions.

## Acknowledgments

This work is supported by a NSF CAREER Award under grant IIS-0238386. We would also like to thank Shivnath Babu, Yuguo Chen, Kamesh Munagala, and members of the Duke Database Research Group for their discussions.

## References

[1] Arasu, A., Babcock, B., Babu, S., McAlister, J., and Widom, J. (2002). Characterizing memory requirements for queries over continuous data streams. In *Proceedings of the 2002 ACM Symposium on Principles of Database Systems*, pages 221–232, Madison, Wisconsin, USA.

[2] Arasu, A., Babu, S., and Widom, J. (2003). The CQL continuous query language: Semantic foundations and query execution. Technical Report 2003–67, InfoLab, Stanford University.

[3] Avnur, R. and Hellerstein, J. M. (2000). Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 261–272, Dallas, Texas, USA.

[4] Ayad, A. and Naughton, J. F. (2004). Static optimization of conjunctive queries with sliding windows over infinite streams. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 419–430, Paris, France.

[5] Babu, S., Motwani, R., Munagala, K., Nishizawa, I., and Widom, J. (2004a). Adaptive ordering of pipelined stream filters. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 407–418, Paris, France.

[6] Babu, S., Munagala, K., Widom, J., and Motwani, R. (2005). Adaptive caching for continuous queries. In *Proceedings of the 2005 International Conference on Data Engineering*, Tokyo, Japan.

[7] Babu, S., Srivastava, U., and Widom, J. (2004b). Exploiting $k$-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions on Database Systems*, 29(3):545–580.

[8] Babu, S. and Widom, J. (2001). Continuous queries over data streams. *ACM SIGMOD Record*.

[9] Bizarro, P., Babu, S., DeWitt, D., and Widom, J. (2005). Content-based routing: Different plans for different data. In *Proceedings of the 2005 International Conference on Very Large Data Bases*, Trondheim, Norway.

[10] Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. B. (2002). Monitoring streams - a new class of data management applications. In *Proceedings of the 2002 International Conference on Very Large Data Bases*, pages 215–226, Hong Kong, China.

[11] Chandrasekaran, S. and Franklin, M. J. (2003). PSoup: a system for streaming queries over streaming data. *The VLDB Journal*, 12(2):140–156.

[12] Chaudhuri, S., Motwani, R., and Narasayya, V. R. (1999). On random sampling over joins. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 263–274, Philadelphia, Pennsylvania, USA.

[13] Chen, J., DeWitt, D. J., Tian, F., and Wang, Y. (2000). NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 379–390, Dallas, Texas, USA.

[14] Chirkova, R., Halevy, A. Y., and Suciu, D. (2001). A formal perspective on the view selection problem. In *Proceedings of the 2001 International Conference on Very Large Data Bases*, pages 59–68, Roma, Italy.

[15] Das, A., Gehrke, J., and Riedewald, M. (2003). Approximate join processing over data streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 40–51, San Diego, California, USA.

[16] Deshpande, A. and Hellerstein, J. M. (2004). Lifting the burden of history from adaptive query processing. In *Proceedings of the 2004 International Conference on Very Large Data Bases*, pages 948–959, Toronto, Canada.

[17] Ding, L., Mehta, N., Rundensteiner, E., and Heineman, G. (2004). Joining punctuated streams. In *Proceedings of the 2004 International Conference on Extending Database Technology*, Heraklion, Crete, Greece.

[18] Ding, L. and Rundensteiner, E. A. (2004). Evaluating window joins over punctuated streams. In *Proceedings of the 2004 International Conference on Information and Knowledge Management*, pages 98–107, Washington DC, USA.

[19] Dingel, J. and Strom, R., editors (2005). *Proceedings of the 2005 International Workshop on Distributed Event Based Systems*, Columbus, Ohio, USA.

[20] Dittrich, J.-P., Seeger, B., Taylor, D. S., and Widmayer, P. (2002). Progressive merge join: A generic and non-blocking sort-based join algorithm. In *Proceedings of the 2002 International Conference on Very Large Data Bases*, pages 299–310, Hong Kong, China.

[21] Garcia-Molina, H., Labio, W., and Yang, J. (1998). Expiring data in a warehouse. In *Proceedings of the 1998 International Conference on Very Large Data Bases*, pages 500–511, New York City, New York, USA.

[22] Golab, L., Garg, S., and Özsu, T. (2004). On indexing sliding windows over on-line data streams. In *Proceedings of the 2004 International Conference on Extending Database Technology*, Heraklion, Crete, Greece.

[23] Golab, L. and Özsu, M. T. (2003). Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of the 2003 International Conference on Very Large Data Bases*, pages 500–511, Berlin, Germany.

[24] Golab, L. and Özsu, M. T. (2005). Update-pattern-aware modeling and processing of continuous queries. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 658–669, Baltimore, Maryland, USA.

[25] Gupta, A. and Mumick, I. S., editors (1999). *Materialized Views: Techniques, Implementations, and Applications*. MIT Press.

[26] Halevy, A. Y. (2001). Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294.

[27] Hammad, M. A., Aref, W. G., and Elmagarmid, A. K. (2003a). Stream window join: Tracking moving objects in sensor-network databases. In *Proceedings of the 2003 International Conference on Scientific and Statistical Database Management*, pages 75–84, Cambridge, Massachusetts, USA.

[28] Hammad, M. A., Franklin, M. J., Aref, W. G., and Elmagarmid, A. K. (2003b). Scheduling for shared window joins over data streams. In *Proceedings of the 2003 International Conference on Very Large Data Bases*, pages 297–308, Berlin, Germany.

[29] Ives, Z. G., Florescu, D., Friedman, M., Levy, A. Y., and Weld, D. S. (1999). An adaptive query execution system for data integration. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 299–310, Philadelphia, Pennsylvania, USA.

[30] Kang, J., Naughton, J. F., and Viglas, S. (2003). Evaluating window joins over unbounded streams. In *Proceedings of the 2003 International Conference on Data Engineering*, pages 341–352, Bangalore, India.

[31] Liu, B. and Rundensteiner, E. A. (2005). Cost-driven general join view maintenance over distributed data sources. In *Proceedings of the 2005 International Conference on Data Engineering*, pages 578–579, Tokyo, Japan.

[32] Madden, S., Shah, M. A., Hellerstein, J. M., and Raman, V. (2002). Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, USA.

[33] Mokbel, M. F., Lu, M., and Aref, W. G. (2004). Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *Proceedings of the 2004 International Conference on Data Engineering*, pages 251–263, Boston, Massachusetts, USA.

[34] Olken, F. (1993). *Random Sampling from Databases*. PhD thesis, University of California at Berkeley.

[35] Quass, D., Gupta, A., Mumick, I. S., and Widom, J. (1996). Making views self-maintainable for data warehousing. In *Proceedings of the 1996 International Conference on Parallel and Distributed Information Systems*, pages 158–169, Miami Beach, Florida, USA.

[36] Raman, V., Deshpande, A., and Hellerstein, J. M. (2003). Using state modules for adaptive query processing. In *Proceedings of the 2003 International Conference on Data Engineering*, pages 353–364, Bangalore, India.

[37] Srivastava, U. and Widom, J. (2004). Memory-limited execution of windowed stream joins. In *Proceedings of the 2004 International Conference on Very Large Data Bases*, pages 324–335, Toronto, Canada.

[38] Tao, Y., Yiu, M. L., Papadias, D., Hadjieleftheriou, M., and Mamoulis, N. (2005). RPJ: Producing fast join results on streams through rate-based optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 371–382, Baltimore, Maryland, USA.

[39] Tatbul, N., Cetintemel, U., Zdonik, S. B., Cherniack, M., and Stonebraker, M. (2003). Load shedding in a data stream manager. In *Proceedings of the 2003 International Conference on Very Large Data Bases*, pages 309–320, Berlin, Germany.

[40] Tucker, P. A., Maier, D., Sheard, T., and Fegaras, L. (2003). Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568.

[41] Urhan, T. and Franklin, M. J. (2001). Dynamic pipeline scheduling for improving interactive query performance. In *Proceedings of the 2001 International Conference on Very Large Data Bases*, pages 501–510, Roma, Italy.

[42] Viglas, S. D. and Naughton, J. F. (2002). Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 37–48, Madison, Wisconsin, USA.

[43] Viglas, S. D., Naughton, J. F., and Burger, J. (2003). Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of the 2003 International Conference on Very Large Data Bases*, pages 285–296, Berlin, Germany.

[44] Wilschut, A. N. and Apers, P. M. G. (1991). Dataflow query execution in a parallel main-memory environment. In *Proceedings of the 1991 International Conference on Parallel and Distributed Information Systems*, pages 68–77, Miami Beach, Florida, USA.

[45] Xie, J., Yang, J., and Chen, Y. (2005). On joining and caching stochastic streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 359–370, Baltimore, Maryland, USA.

[46] Zhuge, Y., Garcia-Molina, H., and Wiener, J. L. (1998). Consistency algorithms for multi-source warehouse view maintenance. *Distributed and Parallel Databases*, 6(1):7–40.