On Joining and Caching Stochastic Streams[*]

Junyi Xie
Department of Computer Science
Duke University
junyi@cs.duke.edu

Jun Yang
Department of Computer Science
Duke University
junyang@cs.duke.edu

Yuguo Chen
Institute of Statistics and Decision Sciences
Duke University
yuguo@stat.duke.edu

ABSTRACT

We consider the problem of joining data streams using limited cache memory, with the goal of producing as many result tuples as possible from the cache. Many cache replacement heuristics have been proposed in the past. Their performance often relies on implicit assumptions about the input streams, e.g., that the join attribute values follow a relatively stationary distribution. However, in general and in practice, streams often exhibit more complex behaviors, such as increasing trends and random walks, rendering these "hardwired" heuristics inadequate.

In this paper, we propose a framework that is able to exploit known or observed statistical properties of input streams to make cache replacement decisions aimed at maximizing the expected number of result tuples. To illustrate the complexity of the solution space, we show that even an algorithm that considers, at every time step, all possible sequences of future replacement decisions may not be optimal. We then identify a condition between two candidate tuples under which an optimal algorithm would always choose one tuple over the other to replace. We develop a heuristic that behaves consistently with an optimal algorithm whenever this condition is satisfied. We show through experiments that our heuristic outperforms previous ones.

As another evidence of the generality of our framework, we show that the classic caching/paging problem for static objects can be reduced to a stream join problem and analyzed under our framework, yielding results that agree with or extend classic ones.

1. INTRODUCTION

There is an increasing need today for applications to be able to process in real-time high-speed data streams such as network traces, financial data feeds, and sensor data streams. Because of large volumes and high arrival rates, streams are often processed online in high-speed main memory in order to meet real-time constraints [1]. Hence, efficient usage of limited resources such as CPU and memory becomes of paramount importance.

[*]The authors are supported by a NSF CAREER Award under grant IIS-0238386.

Join is ubiquitous in both traditional relational database systems and data stream processing systems, and is often one of the most resource-intensive operations. Without any constraints on the input data and its arrival order, accurate join processing requires an unbounded amount of state. To get around this problem, most systems use the *sliding window join*, which restricts tuples participating in the join to a bounded-size window of the most recent input tuples. However, even with a bounded window, it is possible for the input arrival rate to exceed the CPU processing speed, or for the amount of join state to exceed the size of memory allocated for it. In these situations, various forms of *load shedding* (e.g., [14, 12, 8, 17]) are needed to drop tuples from the input or the join state, resulting in incomplete answers.

In this paper, we consider the case where the size of memory is the primary bottleneck. Given a limited amount of memory for maintaining the join state (which we shall also refer to as a cache), we investigate techniques for choosing tuples to discard from the join state to minimize the "error" in the answer. Various error measures are discussed in [8], and we choose to focus on *MAX-subset*, which corresponds to the number of missing join result tuples. A number of cache replacement heuristics for sliding window joins are proposed in [8], including:

- *PROB*, which discards the tuple with the lowest probability to join with a tuple from the other stream;
- *LIFE*, which discards the tuple with the lowest product of its remaining lifetime (w.r.t. the sliding window) and the probability that it joins with another tuple.

Both PROB and LIFE make sense intuitively, but there is little understanding of what conditions make them good (or bad) heuristics. Moreover, implicit in the definitions above is the assumption that the distribution of join attribute values are relatively stationary; otherwise, the probability to join with the other stream could vary over time, making these definitions ambiguous. The most common incarnations of PROB and LIFE, which we call PROB$_p$ and LIFE$_p$, simply assume stationary distributions and use past join probabilities to predict the future. However, in practice, input streams often exhibit more complex behaviors, such as increasing trends and random walks, which may render heuristics with "hardwired" assumptions inadequate. Given known or observed statistical properties of input streams, how do we exploit these properties to manage the cache more intelligently?

Consider the following example. Suppose our system receives a stream S of measurements from a network of sensors. These measurements are timestamped by the sensors at the time of measurement. Ideally, the timestamps would be perfectly ordered if there were only a single sensor and it communicated to the system using a reliable protocol like TCP. However, there are many sensors communicating with the system at the same time using a low-overhead

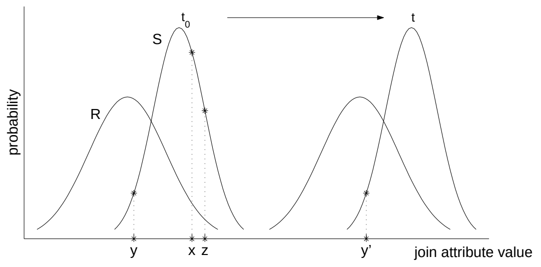

Figure 1: Drifting bounded normal distributions.

connectionless protocol like UDP over a slow and unreliable network. Thus, instead of arriving in order, the timestamps may follow a discretized bounded normal distribution centered at current time minus average latency, which moves right as time progresses. The pdf (probability density function) of this distribution is shown in Figure 1 (discretization is not shown). Suppose there is a second stream R of timestamped measurements of a different type coming from another network of sensors. This network is slower and less reliable, resulting in a distribution with higher variance and looser bound, which lags slightly behind that of S. Suppose we wish to correlate measurements from R and S that were taken at the same time. The query is an equijoin between R and S on the measurement timestamps. Intuitively, when the pdf curve for S moves over the join attribute value of a cached R tuple, this tuple gets a chance of joining with each incoming S tuple, with a probability given by S's pdf at that time.

Consider the three tuples x, y, and z from stream R in Figure 1 currently cached as part of the join state. Which tuples should we choose to discard when we are low on memory? Intuitively, it is best to discard y since it has almost already "missed" the moving pdf of S and is therefore unlikely to join with future S tuples. Unfortunately, using the past to predict future, PROB$_p$ might make the exact opposite decision: y would be the best tuple to keep because it probably has joined more times with S in the past than x and z. Next, how do we choose between x and z? The optimal choice is not clear. On one hand, we can argue that z should be kept because it is expected to generate more result tuples than x over its lifetime. On the other hand, x is closer to the peak of S's pdf and is likely to join immediately. Is it better to reap the immediate benefit of x or to bet on the long-term benefit of z? Another question might be: overall, is it better to discard R tuples or S tuples?

An obvious approach is to make PROB or LIFE aware of any known future statistical properties of the inputs, but how to take advantage of this information is unclear. For example, in PROB, should we use the probability that a tuple x will join with anything in the future, or the average probability that x joins with an arriving tuple? Restricting probability computation to the "lifetime" of x seems reasonable, but should we use the lifetime of x within the sliding window (if one is present), or the estimated time that x will remain in the cache? Instead of relying on intuition, we pursue a more principled approach to answering these questions. We develop a framework to exploit known or observed statistical properties of input streams to make intelligent cache replacement decisions aimed at maximizing the *expected* number of join result tuples that can be generated from the cache. Specifically, we make the following contributions:

- To illustrate the complexity of the solution space, we develop an algorithm which considers all possible sequences of future replacement decisions in order to make the best decision at the current time. Despite its cost and deceivingly complete search space, we show that this algorithm is not optimal.
- We identify a condition between two cached tuples under which an optimal algorithm would always choose one tuple over the other to replace. In several scenarios, testing this condition alone allows us to obtain provably optimal cache replacement policies.
- In case that the optimal algorithm is hard to find, we develop a practical heuristic that behaves consistently with an optimal algorithm whenever the above condition is satisfied. We show through experiments that our heuristic outperforms previously proposed ones.
- We propose techniques for evaluating the practical heuristic efficiently. Furthermore, our approach does not have to know in advance the parameters values of the stochastic processes generating the input streams; we show how to approximate the heuristic through appropriate runtime monitoring.

A natural question is how our problem differs from the classic caching problem. Numerous cache replacement policies have been proposed before, and all seem applicable here. However, 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 caching stream tuples for join, a mistake can cost a lot more: when a tuple is discarded, it is irrevocably gone, along with all result tuples that it could generate in the future. Therefore, an optimal classic cache replacement policy, which simply favors those objects that will be referenced first, may not be optimal for join state management, where references beyond the first one also matter. Another contribution of ours is showing how the two problems can be tackled under the same framework despite their differences:

- We show that classic caching can be reduced to the problem of managing join states for streams, and that it can be analyzed in the same framework, yielding provably optimal results that agree with or extend classic ones. These "coincidences" further evidence the generality of our framework.

Our framework can handle the sliding-window join semantics; details are discussed in the full version version of this paper [11]. For simplicity of presentation, we shall assume the regular join semantics in the rest of paper (although a distribution can still be bounded as in Figure 1, effectively creating a value-based sliding window). We also assume that all streams have the same constant arrival rate; due to space constraints, we discuss how to lift this assumption in [11].

2. RELATED WORK

There is a large body of recent work in data stream processing; general issues are discussed in the survey papers of [1]. Our work can be seen as one form of load shedding, which drops tuples for the purpose of reclaiming memory and with the goal of dropping as few result tuples as possible. Load shedding for joins using random drops is considered in [12]. Sampling is used as a general mechanism for load shedding in [14]. Sampling-based techniques (e.g., [6]) are the methods of choice when we wish to produce a statistical sample of the the query result, but they are less effective when the quality of the join result is measured by the number of result tuples. In addition to random drops, predicate-based load shedding is used in [17] to drop tuples from value ranges with low utilities according to QoS (quality-of-service) specifications; results in our paper can be used to improve the effectiveness of predicate-based load shedding for joins when the loss-tolerance aspect of QoS is considered.

The first paper to consider in detail the join load shedding problem under the MAX-subset measure is [8]. In [8], an optimal offline algorithm based on a min-cost network flow solver is presented. The same solver is used as a building block for our FLOWEXPECT algorithm (Section 4). However, FLOWEXPECT is online: it attempts to maximize the expected number of the result tuples based on the given statistical properties of input streams, without knowing the exact future. Various online heuristics are also proposed in [8]. Using our framework, we can prove that these heuristics are optimal in some scenarios; our heuristic agrees with these heuristics in such scenarios while outperforming them in others.

Our work is complementary to the recent work on reducing memory requirement of stream processing [4] using *k-constraints*, which are relaxed forms of join and arrival constraints. While parameterized by k, they are still *hard* constraints. On the other hand, we exploit *soft* statistical properties that expose more optimization opportunities when result completeness is not required.

A problem orthogonal to ours but essential to the applicability of our framework is how to identify statistical properties of inputs in the first place. Time series data analysis is an established field with many readily applicable techniques. Recent work by the database community addresses efficient online statistical analysis of streams (e.g., [9, 3] and many more); some target specifically at time series (e.g., [7]).

As discussed in Section 1, our problem is different from but closely related to classic caching. There has been extensive work on caching since the 1960's. An optimal offline algorithm, LFD (Longest Forward Distance), is given in [5]. In [2], it is shown that, given a stochastic model of page references, there is an optimal algorithm in terms of expected cost, but this algorithm is infeasible to implement. An alternative A_o is developed and shown to be optimal for references with *almost stationary* distributions. A number of practical algorithms, such as LRU and LRU-k [15], are good approximations of A_o. As we will see in Section 7, straightforward applications of our framework lead naturally to LFD and A_o in scenarios where they are optimal. Beyond these scenarios, analysis in our framework can also reveal optimal algorithms not covered by these classic ones (Section 7.5 and more in [11]).

There is a large body of work on *competitive analysis* of caching algorithms, surveyed in [10]. Competitive algorithms offer much stronger performance guarantees than algorithms that are optimized for the average case, such as A_o and ours. Developing practical algorithms with good competitive ratios is hard in general, although one can often do better by restricting the power of the adversary or by giving the algorithm some knowledge of the input. We believe that average-case analysis provides a good starting point for dealing with streams with good statistical characterizations. Competitive analysis would be a natural direction for future work.

Finally, it is worth noting that MAX-subset is only one of many possible measures for the quality of approximate answers. MAX-subset is appropriate if the goal is to accomplish as much as possible with the cache while (roughly) leaving as little as possible for post-processing, analogous to the *Archive-metric* [8]. However, if a meaningful statistical sample of the result set is more desirable, techniques in this paper and most classic caching techniques are not directly applicable because they are naturally biased towards tuples (or objects) that generate many results (or hits), causing them to be over-represented in the sample. More rigorous sampling-based methods (e.g., [6]) should be used in this case. Most recently, this problem is studied in [16]. MAX-subset is also considered by [16], although they make specific assumptions about the inputs (namely, *frequency*- and *age-based models*) that enable simpler analysis and more efficient solutions. In contrast, our framework is general.

3. PROBLEM SETUP

We model an input stream S as a discrete-time stochastic process $\{X_t^S \mid t = 0, 1, \ldots\}$, where each X_t^S is a random variable representing the value of the join attribute of the tuple produced at time t. In general, the random variables within a process may not be independent, and stochastic processes governing different input streams may not be independent either. For simplicity, we assume that the time domain is $\mathbf{Z}^+$, and that all random variables are discrete. We also assume that all tuples take the same amount of space.

We are interested in both the problem of joining two streams and the problem of joining a stream with a database relation. We shall refer to them as *joining* and *caching* problems, respectively. Although either problem involves both caching and joining, we call the latter problem "caching" because it corresponds exactly to the classic caching scenario.

In the *joining* problem, we perform an equijoin between two streams on a join attribute. A limited amount of memory can be used to cache tuples from either stream to join with future tuples from the other stream. Tuples can have identical join attribute values but are assumed to be distinct from each other; for example, two R tuples with the same join attribute value can join with the same S tuple and produce two distinct result tuples. Assuming that we know the stochastic processes governing the input streams, we want to devise a cache replacement strategy that maximizes the expected number of result tuples that can be computed with the cache.

In the *caching* problem, we perform an equijoin between a *reference stream* and a non-streaming *database relation*. There is limited memory to cache database tuples to join with incoming stream tuples. We assume that every stream tuple joins with exactly one database tuple (i.e., a referential integrity constraint exists from the stream to the relation). As in the joining problem, stream tuples can have identical join attribute values but are considered to be distinct from each other. However, there can be only one database tuple with a given join attribute value. For each incoming stream tuple, we have a cache *hit* if the cache contains the joining database tuple, or a *miss* otherwise. In the case of a miss, the joining tuple is retrieved from the database and we have the option of caching it afterwards. Given the stochastic process governing the reference stream, we want to devise a cache replacement strategy that maximizes the expected number of cache hits (or, equivalently, minimizes the expected number of cache misses).

Note that the joining problem assumes replaced tuples cannot be recovered *online*. Eliminating this assumption mitigates the difference between joining and caching. However, for systems that recover tuples *offline* (e.g., those using the Archive-metric [8]), joining and caching are still distinct problems.

3.1 Reducing Caching to Joining

We now show that the caching problem can be reduced to the joining problem. This reduction allows us to tackle both problems under a unified framework, whose advantage will be apparent later. Given a caching problem with reference stream R, to formulate it as a joining problem, we need to construct a second stream S to be joined with R. For each reference stream tuple r, let S generate the joining database tuple s (one with the same join attribute value as r). Thus we have, for example:

$$\begin{array}{rcl} R & : & r_a^0,\ r_b^1,\ r_a^2,\ r_c^3,\ r_a^4, \ldots \\ S & : & s_a,\ s_b,\ s_a,\ s_c,\ s_a, \ldots \end{array}$$

where the subscripts denote values of the join attributes. Intuitively, we may think of S as a "supply" stream that can be used to populate the cache; indeed, since all cache misses are satisfied by going back to the database, the joining database tuple at each time step is always supplied to the cache regardless of hit or miss.

The above formulation does not quite work yet. Recall from earlier in this section that the joining problem assumes all input tuples to be distinct. Unfortunately, the supply stream S above contains duplicates (e.g., the same database tuple s_a appears three times). Treating them as distinct tuples does not work, because it would allow the unrealistic situation of caching multiple copies of the same S tuple at the same time; furthermore, the number of join result tuples generated would not match the number of cache hits.

The trick is to tweak the join attribute values (and domain) as follows: In R, we replace the i-th occurrence of value v with the pair $(v, i-1)$; in S, we replace the i-th occurrence of value v with (v, i). Two pairs are equal if both components are. Thus, the example streams above are converted into:

$$\begin{aligned} R' &: \; r^0_{(a,0)},\, r^1_{(b,0)},\, r^2_{(a,1)},\, r^3_{(c,0)},\, r^4_{(a,2)}, \ldots \\ S' &: \; s_{(a,1)},\, s_{(b,1)},\, s_{(a,2)},\, s_{(c,1)},\, s_{(a,3)}, \ldots \end{aligned}$$

To see why this transformation works, we make the following observations. (1) Neither stream contains duplicates. (2) Each S' tuple $s_{(v,i)}$ can join with one R' tuple (specifically, the first $r^t_{(v,\_)}$ to come in the future), provided that the S' tuple is still cached at that time. (3) Each R' tuple $r^t_{(v,i)}$ can join with one S' tuple (specifically, the last $s_{(v,\_)}$ seen before the current time t), provided that the S' tuple is still cached now.

Observation (2) implies that each S' tuple $s_{(v,i)}$ can produce no more results after joining for the first time with $r^t_{(v,i)}$. Therefore, any reasonable solution to the joining problem will replace $s_{(v,i)}$ at time t when the next instance of $s_{(v,\_)}$ arrives, thereby avoiding the unrealistic situation of caching multiple copies of the same tuple at the same time. Observation (3) implies that no R' tuple can join with any future S' tuples, so a reasonable solution will not cache any reference stream tuple, which is consistent with the definition of the caching problem.

Excluding the unreasonable solutions (those that cache any reference stream tuples or multiple instances of the same database tuples), which are obviously suboptimal, solutions to the joining problem can be mapped to solutions to the original caching problem straightforwardly. Furthermore, the number of result tuples produced by the joining solution is equal to the number of cache hits produced by the caching solution. A more formal claim about the correctness of this reduction and its proof can be found in [11].

4. THE FLOWEXPECT ALGORITHM

In this section, we give an algorithm, FLOWEXPECT, for making cache replacement decisions that try to maximize the expected benefit of a fixed-size cache. Since we have shown the reduction from caching to joining, we shall focus on joining here.

In each time step, FLOWEXPECT asks the following question: *Given the content of a size-k cache and the two tuples arriving at the current time t_0, which two tuples should we choose to discard now in order to maximize the* expected number *of join result tuples that can be generated from the cache during the interval $[t_0, t_0+l]$?* Here, l specifies how far into the future the algorithm should look ahead. A smaller l makes the algorithm faster, but may result in suboptimal decisions.

The answer to the above question allows us to make a good cache replacement decision (in the expected sense) for the current time only. In the next time step, FLOWEXPECT needs to ask this question again, with the new cache content and the two newly arrived tuples as input. As in OPT-offline [8], we reformulate each instance of the question as a min-cost network flow problem. The main difference between OPT-offline and FLOWEXPECT is that OPT-offline assumes complete knowledge of all future tuples and only needs to be run once. On the other hand, FLOWEXPECT is an online algorithm that has complete knowledge of the past and the present, but only probabilistic knowledge of the future. For this reason, FLOWEXPECT needs to use expected costs in the flow graph; also, FLOWEXPECT has to recompute its decisions when more information becomes available in each new time step. Our main contribution here is not so much FLOWEXPECT per se, but rather the somewhat unexpected observation later in this section regarding the optimality of FLOWEXPECT, which reveals the complexity of the solution space.

We formulate the min-cost network flow problem by constructing a flow graph to capture all possible cache traces from time t_0 up to time $t_0 + l$. The detailed graph construction procedure is given in [11]. In brief, each node in the graph represents the presence of a tuple at a particular time, and each unit-capacity edge represents a replacement decision on a tuple at a particular time. By solving for the min-cost flow of memory size in this graph, we can find a sequence of cache replacement decisions that maximizes the expected benefit.

Suboptimality of FLOWEXPECT As expensive as it may be, FLOWEXPECT is not optimal, even if l is made as large as necessary. The reason is that the min-cost flow problem solved at each time step only considers all possible *predetermined* sequences of future cache replacement decisions. The search space does not include strategies that make future decisions *online* based on the actual join attribute values of tuples when they arrive. In practice, the problem is somewhat alleviated by the fact that FLOWEXPECT recomputes its decision at every time step using the most up-to-date information. However, it is still possible for FLOWEXPECT to make suboptimal decisions.

To illustrate, consider a small but carefully constructed example. Suppose the cache can hold only one tuple at a time. At the current time t_0, the cache contains a tuple with join attribute value of 1 from stream R. We also have some information about what streams R and S will produce in the future (summarized in the table below): some for sure and others probabilistically. The symbol "$-$" represents a tuple that does not join with other tuples or "$-$" tuples themselves.

Time	Join attribute value of new R tuple	Join attribute value of new S tuple
t_0	$-$	2
t_0+1	2	3 with probability 0.5 ($-$ otherwise)
t_0+2	3	1 with probability 0.8 ($-$ otherwise)
t_0+3	2 with probability 0.5 ($-$ otherwise)	1 with probability 0.8 ($-$ otherwise)

The best three sequences of cache replacement decisions considered by FLOWEXPECT are:

- Always keep the currently cached R tuple (1). The expected benefit is $0.8\,(\text{at } t_0+2) + 0.8\,(\text{at } t_0+3) = 1.6$.
- At t_0, replace the cached tuple with the new S tuple (2), and keep it afterwards. This sequence yields an expected benefit of $1\,(\text{at } t_0+1) + 0.5\,(\text{at } t_0+3) = 1.5$.
- At t_0, replace the cached tuple with the new S tuple (2); at t_0+1, replace again with the new S tuple (3 with probability 0.5), and then keep it afterwards. This sequence yields an expected benefit of $1\,(\text{at } t_0+1) + 0.5\,(\text{at } t_0+3) = 1.5$.

FLOWEXPECT would choose the first sequence because of its highest expected benefit, and therefore decides to keep the currently cached R tuple at t_0. However, the flow graph fails to capture the following strategy, which combines the last two sequences in a dynamic way:

– At t_0, replace the cached tuple with the new S tuple (2).
– At $t_0 + 1$, if the new S tuple has value 3, replace the cached tuple with this one; otherwise, keep the cached tuple.
– From $t_0 + 2$ on, keep the currently cached tuple.

If at $t_0 + 1$ the new S tuple has value 3, the above strategy will have an expected benefit of $1\,(\text{at } t_0 + 1) + 1\,(\text{at } t_0 + 2) = 2$; if not, this strategy will have an expected benefit of $1\,(\text{at } t_0 + 1) + 0.5\,(\text{at } t_0 + 2) = 1.5$. Therefore, the overall expected benefit of this strategy is $2 \cdot 0.5 + 1.5 \cdot 0.5 = 1.75$, which is higher than any predetermined sequence of cache replace decisions can generate. Hence, FLOWEXPECT has made the wrong decision of keeping the cached R tuple at time t_0.

The analysis above not only shows the suboptimality of FLOWEXPECT but also reveals how vast the search space is. An optimal algorithm would need to consider all strategies that make conditional decisions based on the join attribute values of new tuples observed at runtime. In general, such a strategy can have a branching factor of $\binom{k+2}{2}$ for every future time step, and the conditional expressions controlling the branches can refer to all values that might have been observed up to that time step. An exhaustive search through this enormous space is clearly impractical.

5. A PRACTICAL FRAMEWORK

Recognizing that the optimal solution might be too expensive to obtain, we now turn to a more practical approach. We shall tackle the joining problem directly, and address the caching problem through reduction to the joining problem.

5.1 Expected Cumulative Benefit

Given the knowledge of stochastic processes governing the input streams, we can calculate the expected total benefit of caching a tuple over a period of time. Formally, at the current time t_0, for each candidate tuple x (either currently cached or just arriving at t_0), we define the *expected cumulative benefit function* (or *ECB* for short) of x w.r.t. t_0, denoted $B_x(\Delta t)$ ($\Delta t \geq 1$), as the expected number of result tuples generated by joining x with tuples from the other stream over the period $[t_0 + 1, t_0 + \Delta t]$. ECB basically indicates how desirable it is to cache x. The following lemma for computing ECB follows directly from the definition of ECB. We use v_x to denote the value of the join attribute for a tuple x, and $\bar{x}_{t_0}$ to denote the history of input streams observed up to t_0.

LEMMA 1. *Suppose that at current time t_0, x is a candidate tuple from stream S (to be joined with R). The ECB for x is given by $B_x(\Delta t) = \sum_{t=t_0+1}^{t_0+\Delta t} \mathbf{Pr}\{X_t^R = v_x \mid \bar{x}_{t_0}\}$.*

For the caching problem, this lemma also applies, although calculation of the ECB should be based on the *transformed streams* after reduction to the joining problem (Section 3). It is straightforward to express the ECB computed by Lemma 1 directly in terms of the statistical properties of the *original reference stream*, as shown by the following corollary to Lemma 1 (see [11] for the proof). Interestingly, the ECB of a database tuple turns out to be the probability that it is referenced at any time in the given period.

COROLLARY 1. *Suppose that at current time t_0, x is a candidate database tuple (to be referenced by stream R). The ECB for x is given by $B_x(\Delta t) = 1 - \mathbf{Pr}\{\bigcap_{t=t_0+1}^{t_0+\Delta t} X_t^R \neq v_x \mid \bar{x}_{t_0}\}$. The ECB for any reference stream tuple y is given by $B_y(\Delta t) = 0$.*

Intuitively, when making a cache replacement decision, we want to remove tuples with the least desirable ECB from the cache. As a concrete example, let us reconsider the three candidate tuples x, y, and z from stream R in the example introduced in Section 1. In

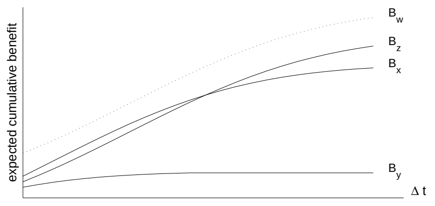

Figure 2: Example ECBs.

Figure 2, we plot their ECBs. Between tuples x and y, we see it is intuitively better to cache x since we expect it to generate consistently more benefit than y over any period of time starting now. The dilemma of comparing x and z is also clearly illustrated by Figure 2. The choice depends on whether z is expected to survive in the cache longer than the cross point of x and z's ECB curves. If z is replaced too soon (perhaps by some tuple with a better ECB arriving later), then it is better to cache x.

The above example shows that not every two ECBs are "comparable." Next, we formalize the notion of comparable ECBs and prove that cache replacement decisions based on comparable ECBs are optimal.

5.2 ECB Dominance Test

We say that an ECB B_x *dominates* another ECB B_y if $B_x(\Delta t) \geq B_y(\Delta t)$ for all $\Delta t \geq 1$. If the inequality is strict for all $\Delta t \geq 1$, we say that B_x *strongly dominates* B_y. Two ECBs are *comparable* if one dominates the other. The main theorem of this section, presented below, states that if tuple x's ECB dominates (or strongly dominates) tuple y's ECB, then discarding x must be no better (or worse) than discarding y.

THEOREM 1. *Suppose there are currently two candidate tuples x and y with ECBs B_x and B_y, respectively. (1) If B_x dominates B_y, then there exists an optimal algorithm that keeps x or discards y at the current time. (2) If B_x strongly dominates B_y, then every optimal algorithm must keep x or discard y at the current time.*

A formal proof can be found in [11], but we will provide the crux for (2) here. Suppose an algorithm A does not meet the above condition, i.e., A discards x at t_0 but keeps y from t_0 to some $t' > t_0$. We construct another algorithm A' that discards y at t_0 and keeps x from t_0 to t'; other than this difference, A' makes the exact same cache replacement decisions as A. It is easy to see that the expected benefits generated by A and A' differ by exactly $B_y(t' - t_0) - B_x(t' - t_0)$. Since B_x strongly dominates B_y, the expected benefit of A' is greater than A. So A cannot be optimal.

Note that ECB is defined w.r.t. to the current time, and dominance tests can tell us optimal decisions at this moment. Dominance relationships may change over time. Even if B_x strongly dominates B_y now, at some later time t' B_y may strongly dominate B_x. There is no conflict: it is still better to keep x now, but if both survive until t', keeping y would be better at that point.

Dominance does not in general induce a total order on the candidate tuples. On the other hand, we do not really need a total order to make an optimal decision. If we can find a set of tuples whose ECBs are all dominated by the ECBs of tuples outside this set, then it is optimal to discard all tuples in this set (provided that the cache needs to discard this many or more tuples). For example, suppose that in Figure 2 there is another tuple w whose ECB dominates all others. If we need to discard three tuples out of w, x, y, and z, the optimal choice would be x, y, and z, even though x and z have incomparable ECBs. If we need to discard only two out of four tuples, it is still safe to discard y, but the choice between x and z is unclear. The following corollary captures this intuition.

COROLLARY 2. *A subset $V \subseteq U$ is called a* dominated subset *(w.r.t. U) if $\forall u \in U - V$, $\forall v \in V$, B_u dominates B_v. Suppose the cache has size k and the set C of candidate tuples has size $k + \Delta k$. If C' is a dominated subset of C with no more than Δk tuples, there exists an optimal algorithm that discards C'.*

5.3 Heuristic of Estimated Expected Benefit

If there is no dominated subset with size equal to the number of tuples to be discarded, we need heuristics to choose among tuples whose ECBs are incomparable. Many heuristics have been proposed in previous work (e.g., PROB and LIFE [8] for joining, LRU and LFU for caching). They may work really well for input streams with certain properties, but they cannot exploit the knowledge about arbitrary statistical properties exhibited by the input streams. For instance, we would not expect these "hardwired" heuristics to work well for the example in Section 1.

In contrast, our *heuristic of estimated expected benefit* (or *HEEB* for short) is based on known or observed statistical properties of stochastic input streams. The advantage of HEEB is its generality. We can apply HEEB to both caching and joining problems with any stochastic input (as we shall do in Section 7) and obtain a cache replacement algorithm that works well for the given input (as we shall see in experiments in Section 8). Furthermore, HEEB agrees with Theorem 1 in that HEEB makes optimal decisions on candidate tuples with comparable ECBs.

For each candidate tuple x, HEEB computes a value H_x. Tuples with lowest such values are discarded. H_x is computed from the ECB $B_x(\Delta t)$ and another function $L_x(\Delta t)$, which estimates the probability that x will still be cached at time $t_0 + \Delta t$. The choice of L_x is fairly flexible and can be fine-tuned for a particular input (more on this topic later). H_x is defined as follows:

$$H_x = B_x(1)L_x(1) + \sum_{\Delta t=2}^{\infty} \left((B_x(\Delta t) - B_x(\Delta t - 1))L_x(\Delta t)\right).$$

In the above, $B_x(1)L_x(1)$ estimates the expected benefit generated by x at time t_0+1, and $(B_x(\Delta t)-B_x(\Delta t-1))L_x(\Delta t)$ estimates the expected benefit at time $t_0+\Delta t$ (since B_x measures *cumulative* benefit, we need to take the difference in order to compute the benefit in a single time step). Summing up these terms, H_x estimates the expected total benefit of caching x. HEEB favors candidate tuples with high estimated expected benefits. Convergence of H_x can be ensured by proper choices of L_x, as discussed next.[1]

Applying Lemma 1 to the definition of H_x, we can obtain the following equivalent definition of H_x for a candidate tuple x to be joined with stream R.

$$H_x = \sum_{\Delta t=1}^{\infty} (\mathbf{Pr}\{X^R_{t_0+\Delta t} = v_x \mid \bar{x}_{t_0}\} \cdot L_x(\Delta t)).$$

For the caching problem, applying Corollary 1 to the definition of H_x, we can obtain the following equivalent definition of H_x for a database tuple x to be referenced by stream R:

$$H_x = \sum_{\Delta t=1}^{\infty} (\mathbf{Pr}\{(X^R_{t_0+\Delta t} = v_x) \cap (\bigcap_{t_0<t<t_0+\Delta t} X^R_t \neq v_x) \mid \bar{x}_{t_0}\} \cdot L_x(\Delta t)).$$

[1]Note that L_x is only an *estimate* of the probability that HEEB will keep x alive in the cache at a given time. It is not possible to use the exact probability to define HEEB; doing so would result in a circular definition. To compute this probability exactly, we need to know how likely a future tuple might be considered by HEEB to be better to cache than x (and therefore replace x). However, HEEB decides what is better based on L_x itself.

Definition of L_x	Resulting H_x	Behavior
$L^{fixed}(\Delta t) =$ 1 for $\Delta t \leq \Delta T$, or 0 otherwise	$H_x^{fixed} = B_x(\Delta T)$, the expected total benefit of x based on the assumption that all tuples are replaced exactly at $t + \Delta T$	Replace the tuple with *least benefit in a fixed amount of time*
$L^{inf}(\Delta t) = 1$ (for caching only)	$H_x^{inf} = \lim_{\Delta t \to \infty} B_x(\Delta t)$, the probability that x will be referenced any time in the future	Replace the tuple *least likely to be used in future*
$L^{inv}(\Delta t) = 1/\Delta t$ (for caching only)	$H_x^{inv} = \sum_{\Delta t=1}^{\infty} (\mathbf{Pr}\{x \text{ is first used at } t_0 + \Delta t \mid \bar{x}_{t_0}\} \ / \ \Delta t)$, the expected value of the inverse of x's *waiting time* (the amount of time before x is first used)	Replace the tuple with the *lowest expected inverse waiting time*
$L^{exp}(\Delta t) = e^{-\Delta t/\alpha}$ ($\alpha > 0$)	H_x^{exp} calculates the expected total benefit of x based on the assumption that the probability for x to remain in the cache decreases exponentially over time	Replace the tuple with the *least benefit assuming exponentially decreasing prob. to remain cached*

Table 1: Several choices of L_x.

Choosing L_x Although the choice of L_x may vary, a good choice should have the following properties:

1. *For all $\Delta t \geq 1$, $0 \leq L_x(\Delta t) \leq 1$.* This property is an obvious must because L_x estimates a probability.
2. *$L_x(\Delta t)$ is a non-increasing function of Δt ($\Delta t \geq 1$).* This property is also a must. Since "x is cached at time $t_0 + \Delta t$" implies "x is cached at $t_0+\Delta t-1$," the estimated probability of the former should be no greater than that of the latter.
3. *The choice of L_x should ensure the convergence of the sum defining H_x.* Note that a sufficient (but not always necessary) condition is that the series $\sum_{\Delta t=1}^{\infty} L_x(\Delta t)$ converges.
4. Suppose both x and y are candidate tuples. *If B_x dominates B_y, then L_x should dominate L_y, i.e., $L_x(\Delta t) \geq L_y(\Delta t)$ for all $\Delta t \geq 1$.* This property captures the intuition that, if caching x is always no worse than caching y, then at any point in time, the probability of x being cached should be no less than that of y being cached. This property ensures that HEEB makes optimal decisions on tuples with comparable ECBs, as Theorem 2 below (proven in [11]) indicates.
5. Suppose x and y are candidate tuples. *If B_x strongly dominates B_y, then $L_x(1) > 0$.* This property rules out constant zero L_x functions, which trivially satisfy other properties.

THEOREM 2. *Suppose the set of L_x functions for all candidate tuples satisfy all five properties above. Then, $H_x \geq H_y$ if B_x dominates B_y; also, $H_x > H_y$ if B_x strongly dominates B_y.*

It is fine to pick the same function L_x for all candidate tuples, which would trivially satisfy Property 4. In fact, we use this simple approach in all our case studies; the details on choosing L_x for each case will be presented in Section 7. A more accurate L_x for each individual x can be derived by studying *both* input streams (not just the one that joins with x) and estimating how likely a future tuple will replace x. However, it is unclear how much additional benefits such accurate estimates can bring in practice, in order to justify the potentially high cost of computing them.

It is interesting to note that particular choices of L_x lead to different instances of HEEB with different assumptions and behaviors. Several examples are summarized in Table 1. Note that L^{inf} and L^{inv} do not guarantee the convergence of H_x in general, but they do guarantee convergence for any caching problem.

Our L_x of choice is L^{exp}, because it guarantees the convergence of H_x^{exp} and makes it incrementally computable, which is useful for an efficient implementation, as we will see in the next subsection. The value of α should be chosen such that the estimated or observed average lifetime of a cached tuple matches $1/(1-\exp(-\frac{1}{\alpha}))$, the average lifetime predicted by L^{exp}. We discuss how to choose α for different scenarios in more detail in Section 7.

6. EFFICIENT IMPLEMENTATION

Although HEEB has simplified the problem of comparing ECBs into the problem of comparing the values of H_x, computing H_x can still be expensive, because in general its value can change over time and therefore needs to be recomputed at every time step. We now discuss two approaches toward practical and efficient implementation of HEEB. First, in Section 6.1, we describe a set of techniques aimed at speeding up the calculation of H_x. Second, in Section 6.2, we propose a histogram-based method that estimates HEEB through appropriate runtime monitoring. This method provides the benefit of HEEB without its computational overhead and without relying on pr knowledge of the parameter values of the input processes for calculation.

6.1 Techniques for Computing HEEB

Time-Incremental Computation Using the value of H_x at the previous time step, sometimes we can compute the value of H_x at the current time incrementally. Certain choices of L_x (e.g., L^{inf} and L^{exp}) make H_x amenable to incremental computation, while others (e.g., L^{inv}) do not. It also helps for input streams to be governed by independent stochastic processes; otherwise, probability calculations at the current time might be conditioned differently from those at the previous time step, making time-incremental computation difficult. Corollaries 3 and 4 below show how to calculate H_x^{exp} (defined using L^{exp}) incrementally over time for joining and caching problems. Corollary 4 also applies to H_x^{inf} (defined using L^{inf}) simply by treating α as ∞. These corollaries, proven in [11], follow directly from the definition of H_x and L^{exp}, and the calculation of B_x (Lemma 1 and Corollary 1, respectively).

COROLLARY 3. *Suppose that x is a candidate tuple from stream S (to be joined with R). Let H_{x,t_0-1}^{exp} denote the value of H_x^{exp} at time t_0-1 and H_{x,t_0}^{exp} the value at time t_0. If all stochastic variables for R and S are independent, then*

$$H_{x,t_0}^{exp} = e^{1/\alpha} H_{x,t_0-1}^{exp} - \mathbf{Pr}\{X_{t_0}^R = v_x\}.$$

COROLLARY 4. *Suppose that x is a candidate database tuple (to be referenced by stream R). Let H_{x,t_0-1}^{exp} denote the value of H_x^{exp} at time t_0-1 and H_{x,t_0}^{exp} the value at time t_0. If all stochastic variables for R are independent, then*

$$H_{x,t_0}^{exp} = \frac{e^{1/\alpha} H_{x,t_0-1}^{exp} - \mathbf{Pr}\{X_{t_0}^R = v_x\}}{1-\mathbf{Pr}\{X_{t_0}^R = v_x\}}.$$

Value-Incremental Computation While time-incremental computation makes it efficient to update H_x for a tuple already in the cache, it does not address how to calculate H_x for a new tuple. Thankfully, we may be able to use the H_x value of an existing tuple x to compute $H_{x'}$ for a new tuple x' incrementally. To explain, we need to review some terminology. A stochastic process $\{X_t\}$ with a deterministic trend are often modelled as follows: $X_t = f(t) + Y_t$, where $f(t)$ is a function that captures the trend of value over time, and all Y_t's, representing noise, are i.i.d. (independently and identically distributed) and have zero mean. Suppose that $f(t)$ either increases or decreases linearly over time. For instance, the motivating example in Section 1 is one such process with bounded normal noise. Consider tuple y at time t_0 and tuple y' at time t in Figure 1, which lie at the same offset relative to the moving R distribution. Intuitively, they should have the same ECB (and therefore H_y^{exp} at time t_0 should be equal to $H_{y'}^{exp}$ at time t), because y and y' see the exact same future from their respective frames of reference. This intuition is formalized in the corollary below, which is proven in [11] and follows directly from the definition of ECB.

COROLLARY 5. *Suppose that x is a candidate tuple to be joined with stream R modelled by $X_t^R = at + b + Y_t^R$, where $a \neq 0$ and Y_t^R's are i.i.d. and have zero mean. Let $B_{v,t}(\Delta t)$ denote the ECB at time t for a tuple with join attribute value v. Then $B_{v,t}(\Delta t) = B_{v+a(t'-t),t'}(\Delta t)$ for all $\Delta t \geq 1$ and any t'.*

To apply Corollary 5 to a new tuple x, we can find a cached tuple x' whose join attribute value is closest to that of x, i.e., $|v_{x'} - v_x|$ is smallest. Let $t' = t_0 + (v_{x'} - v_x)/a$. According to Corollary 5, $H_{x,t_0}^{exp} = H_{x',t'}^{exp}$. $H_{x',t'}^{exp}$ can then be incrementally computed from H_{x',t_0}^{exp} using time-incremental computation.

Precomputation Both time- and value-incremental techniques handle only independent stochastic processes; they do not work for popular models such as AR(1) and random walk, where the value to be generated at the next time step may depend on the value generated at the current time. Thus, an alternative technique based on precomputation is needed for streams of the form $X_t = \phi_0 + \phi_1 X_{t-1} + Y_t$, where ϕ_0 specifies a constant drift at every time step, ϕ_1 controls the value contribution from the previous time step, and Y_t's, representing noise, are i.i.d. and have zero mean. As the following theorem shows, we can precompute a function from which H_x can be calculated at runtime for any tuple at any time.

THEOREM 3. *Suppose that stream R is modelled by $X_t^R = \phi_0 + \phi_1 X_{t-1}^R + Y_t^R$, where $\phi_1 \neq 0$ and Y_t^R's are i.i.d. and have zero mean. Let x_t denote the value of X_t^R observed at t. (1) There exists a function $h_2(\cdot,\cdot)$ such that, for any current time t_0, for any candidate tuple x to be joined with R, $H_x = h_2(v_x, x_{t_0})$, provided that $L_x(\Delta t)$ is a time-independent function $L(\Delta t, v_x, x_{t_0})$. (2) For $\phi_1 = 1$, there exists a function $h_1(\cdot)$ such that $H_x = h_1(v_x - x_{t_0})$, provided that L_x is a time-independent function $L(\Delta t, v_x - x_{t_0})$.*

The good news of Theorem 3 is that both h_2 and h_1 are time-independent, so precomputation is feasible. The requirements on L_x are easily met by all example choices of L_x discussed in Section 5.3. In [11], we give a constructive proof to the above theorem, showing how h_2 and h_1 can be constructed offline. For each input stream, we can precompute its h_2 or h_1 and store a compact, approximate representation online, allowing H_x to be computed efficiently. We will see an example of this approach in Section 7.5. The case of $\phi_1 = 1$, corresponding to a random walk with drift, is relatively simple because h_1 is just a curve to be approximated. An AR(1) model, with $0 < |\phi_1| < 1$, is more complicated because h_2 is a surface. Feasibility of the precomputation approach depends on how compact we make an approximate representation without sacrificing too much accuracy. If an approximation requires lots of space, we might be better off using that space to cache more tuples and switching to a simpler L_x (e.g., L^{fixed} with a small ΔT) that allows H_x to be computed quickly online.

6.2 Estimating HEEB with Histograms

An important practical problem is that we may not know in advance parameter values of the stochastic processes governing the input streams. There exist techniques for computing stream statistics online, or offline over the history of observations; some of these techniques are mentioned in Section 2. However, reliance on such techniques may limit the applicability of HEEB and introduce additional complexity. Therefore, we propose here a method that provides the benefit of HEEB without relying on prior knowledge of the parameter values of the input processes. The method maintains certain statistics of the past behavior of the cache and input streams, and use them to estimate the expected benefit of caching a particular value. A notable feature of this method, which distinguishes it from most existing techniques that collect stream statistics for optimization purposes, is that it considers the form of the stochastic process in order to decide what statistics to monitor. This feature is crucial because, for processes with trends, the past is not always indicative of the future for some statistics.

Although many processes are time-varying, they often have time-invariant components whose effects can be easily isolated. We seek to maintain a time-invariant function h, from which the total expected benefit of caching a tuple can be computed (after adjusting for the effect of time-varying components). We consider two classes of stochastic processes. The first case is a process with linear trend and i.i.d. noise: $X_t = at+b+Y_t$, where Y_t's are i.i.d. with zero mean. By Corollary 5, it is easy to see that although the ECB function $B_{v,t}$ varies over time, the expression $B_{v-at-b,t}$ does not depend on t, meaning that total expected benefit of caching value $v-at-b$ is time-invariant. Hence, instead of tracking the benefit as a function of v, we track the benefit of caching value v by a function $h(\Delta v)$, where Δv is the difference between v and the current mean $(at+b)$. Since $h(\Delta v)$ is time-invariant, it is meaningful to use past observations to predict the future. Though mean is time-varying, it is easy and inexpensive to track online using standard techniques from time series analysis (e.g., moving averages and/or exponential smoothing). The second case we consider is a random walk process with linear trend and i.i.d. steps: $X_t = \phi_0 + X_{t-1} + Y_t$, where Y_t's are i.i.d with zero mean. By Theorem 3, we see that the function $B_{v-x_t,t}$ is time-invariant, where x_t is the current value generated by the random walk. Thus, we can track the benefit of caching value v by a function $h(\Delta v)$, where $\Delta v = v - x_t$.

Next, we discuss how to monitor and maintain the time-invariant function $h(\Delta v)$ approximately at runtime. We use two histograms: $\mathcal{H}_b$ tracks the total number of result tuples generated by a tuple over its lifetime in the cache; $\mathcal{H}_l$ tracks the inverse of the number of time steps during which a tuple remains in the cache. Both histograms are constructed over the domain of Δv. When a tuple enters the cache, we compute Δv from its join attribute value using a procedure appropriate for the process governing the joining input stream, as discussed in the previous paragraph. When this tuple leaves the cache, we update the buckets for Δv in $\mathcal{H}_b$ and $\mathcal{H}_l$ using the total number of hits received by the tuple and the length of its lifetime, which constitute an observation. Each bucket of $\mathcal{H}_b$ and $\mathcal{H}_l$ records the total number of observations, the mean (of hit counts and inverse lifetimes, respectively), and the standard deviation. These three measures can be incrementally updated using a recurrence relation [13]. We exponentially decay these measures periodically, to allow capture of changes in underlying processes. Bucket boundaries are dynamically by splitting buckets with large standard deviations and merging adjacent buckets with low combined standard deviation.

To obtain the total expected benefit for a value v at the current time, we simply calculate Δv from v as discussed earlier and look up the expected benefit from $\mathcal{H}_b$. We also look up the inverse lifetime from $\mathcal{H}_l$ and use it to weigh down the expected benefit. This adjustment is based on the intuition that the longer the wait, the less likely that the benefit will be realized in full. Moreover, this adjustment prevents poor caching decisions from reinforcing themselves. Use of histograms allows us to control the precision-cost tradeoff by varying the number of buckets in the histograms.

While many optimization techniques rely on using the past to predict the future, one distinguishing feature of our method is that we use the form (but not actual parameter values) of stochastic processes to help decide what statistics are appropriate to maintain. The naive approach of simply remembering benefit for each value (without proper offseting) will fail for time-varying distributions. Besides the ability of handling time-varying distributions, our histogram-based method has several other desirable features. First, it uses only the mathematical forms of the stochastic models to decide what information to track; no prior knowledge of the model parameter values is needed. Second, the method can adapt to changes in model parameter values over time. Finally, it has very low computational overhead at runtime, which makes it suitable for streams with high arrival rates.

7. CASE STUDIES

We illustrate the power of our practical framework in five scenarios. For each scenario, we tackle both caching and joining problems. We show when it is possible to make optimal decisions using dominance tests. As we will see, in simple scenarios, dominance tests alone often suffice. When they are insufficient, we show how to choose L_x for HEEB and compute H_x efficiently using the optimizations in Section 6. Except the random walk model in Section 7.5, other models assume no correlation within or between streams. In general, our framework does not assume independence within or between streams; correlations within or between streams only impact the way ECB is computed.

7.1 Offline Streams

In this case, we know in advance the sequence of join attribute values to be produced by input streams (with probability 1). This scenario has little practical significance but nevertheless enables us to compare results with known optimal offline algorithms. For the caching problem, a straightforward application of our general framework leads naturally to the well-known optimal cache replacement policy LFD (Longest Forward Distance) [5]. For the joining problem, HEEB may not be optimal because incomparable ECBs may arise, but FLOWEXPECT, which essentially degenerates to OPT-offline, produces the optimal solution. Details are in [11].

7.2 Stationary, Independent Streams

In the case of a stationary, independent stochastic input stream R, we can define a time-invariant probability distribution function $p_R(v) = \mathbf{Pr}\{X_t^R = v\}$ for all t. This simple scenario is assumed by many of the previously proposed cache replacement algorithms, again allowing us to compare results.

For the caching problem, Corollary 1 tells us that, for a database tuple x at current time t_0, $B_x(\Delta t) = 1 - \prod_{t=t_0+1}^{t_0+\Delta t} \mathbf{Pr}\{X_t^R \neq v_x\} = 1 - (1 - p_R(v_x))^{\Delta t}$. Obviously, for any two database tuples x and y, B_x dominates B_y if $p_R(v_x) \geq p_R(v_y)$. According to Theorem 1, it is optimal to *discard the tuple with the lowest reference probability*. This result agrees with the popular LFU (Least Frequently Used) heuristic and the A_o algorithm [2]. It was shown in [2] that A_o is optimal for a stationary stochastic stream,[2] and that both LRU (Least Recently Used) and WS (Working Set) can be seen as approximations to A_o. In [15], it was shown that LRU-k

[2]A stronger result is proven in [2], which states that A_o is optimal if $p_R(v)$ is *almost stationary*, i.e., the relative ordering of v by $p_R(v)$ does not change over time. From ECB, it is clear that this result holds.

is an optimal approximation to A_o given limited knowledge of past references. Our framework leads us naturally to A_o, and provides an alternative proof that A_o is optimal.

For the joining problem, Lemma 1 tells us that, for a tuple x from stream S (to be joined with stream R), the ECB of x at current time t_0 is $B_x(\Delta t) = \sum_{t=t_0+1}^{t_0+\Delta t} p_R(v_x) = p_R(v_x)\Delta t$. Similarly, if x is from R, $B_x(\Delta t) = p_S(v_x)\Delta t$. Clearly, all ECBs are ranked by how frequently a tuple's join attribute value appears in the other stream. By Theorem 1, it is optimal to *discard the tuple whose join attribute value is least likely to appear* in the other stream. This policy is basically PROB [8]. Thus, our framework provides a proof of its optimality for stationary, independent input streams.

Interestingly, for sliding-window joins, neither PROB nor LIFE is optimal, though HEEB can do better; see [11] for details.

7.3 Linear Trend, Bounded Uniform Noise

Caching Suppose that the reference stream is generated by $X_t^R = f(t)+Y_t^R$, where $f(t)$ is a non-decreasing integer-valued function, and all Y_t^R's follow independent and identical bounded uniform distributions over the interval $[-w_R, w_R]$ of integers, as illustrated in Figure 3 (ignore S for now). The probability of Y_t^R assuming any particular integer value in $[-w_R, w_R]$ is $1/(2w_R+1)$. The non-decreasing trend creates the effect of a "reference window" that moves right over time. Using Corollary 1, we compute the ECB for each database tuple x as follows. Note that it is impossible for a candidate tuple with value greater than $f(t_0)+w_R$ (assuming no prefetching), because it could not have been demand-fetched.

- Category 1: $v_x \in (-\infty, f(t_0) - w_R)$. $B_x(\Delta t) = 0$. Intuitively, tuples in this category have already "missed" the reference window and cannot be referenced in the future. Because a zero ECB is dominated by any ECB, it is optimal to discard any tuple in this category.
- Category 2: $v_x \in [f(t_0) - w_R, f(t_0) + w_R]$. Let t_x be the time when the reference widow moves beyond v_x, i.e., $t_x = \min\{t \mid v_x < f(t) - w_R\}$. Then, $B_x(\Delta t) =$

$$\begin{cases} 1-(1-\frac{1}{2w_R+1})^{\Delta t} & \text{if } \Delta t \in [1, t_x - t_0); \\ 1-(1-\frac{1}{2w_R+1})^{t_x-t_0-1} & \text{otherwise.} \end{cases}$$

  Intuitively, the ECB stops growing once the reference window moves past the tuple. Figure 4 illustrates the ECBs for tuples under this category. Clearly, B_x dominates B_y if $t_x \geq t_y$. Since $f(t)$ is non-decreasing, it is not difficult to see that $v_x \geq v_y$ implies $t_x \geq t_y$. Therefore, among tuples in this category, it is optimal to discard the one with the smallest join attribute value.

Combining the two cases above and noting that tuples in Category 1 have smaller join attribute values than those in Category 2, we have the following algorithm: *discard the tuple with the smallest join attribute value*. Its optimality follows directly from Theorem 1. The analysis holds for any non-decreasing trend function $f(t)$, including nonlinear ones. In fact, the analysis can be generalized to show that this algorithm is optimal for any non-decreasing noise distribution bounded on the right. Interestingly, this case also turns out to be *almost stationary* [2], and indeed, A_o would behave in the exact same way.

Joining Suppose that two input streams R and S have identical increasing linear trend function $f(t)$, but their noise terms follow independent bounded uniform distributions over two different intervals, respectively. For simplicity, we shall assume that $f(t) = t$, and that the noise intervals for R and S, $[-w_R, w_R]$ and $[-w_S, w_S]$, are both centered at 0, with $w_R < w_S$, as illustrated in Figure 3. It is straightforward to generalize the analysis to drop these assumptions. Using Lemma 1, we can compute the ECB for each candidate tuple at current time t_0. All candidate tuples can be divided into five categories below, and representative ECBs are plotted in Figure 5. Due to space constraints, full ECB formulas are omitted here but can be found in [11].

- Category R1: x is from R and $v_x \in (-\infty, t_0 - w_S]$. These tuples have zero ECBs because they will have already missed the window of S at the next time step.
- Category R2: x is from R and $v_x \in (t_0 - w_S, t_0 + w_R]$. These tuples will continue generating benefits at the rate of $\frac{1}{2w_S+1}$ per time step until the window of S moves past them. It is easy to see that, within this category, it is optimal to discard the tuple with the smallest join attribute value, which will fall out of the window the soonest.
- Category S1: x is from S and $v_x \in (-\infty, t_0 - w_R]$. Again, these tuples have zero ECBs because they will have already missed the window of S at the next time step.
- Category S2: x is from S and $v_x \in (t_0 - w_R, t_0 + w_R + 1]$. These tuples will start generating benefits at the rate of $\frac{1}{2w_R+1}$ from the next time step, and will stop when the window of R moves past them. Within this category, it is optimal to discard the tuple with the smallest join attribute value.
- Category S3: x is from S and $v_x \in (t_0 + w_R + 1, t_0 + w_S]$. These tuples lie before the moving R window; they will start generating benefits at the rate of $\frac{1}{2w_R+1}$ once the window moves over them, and will stop when it moves past them.

From the analysis above (and intuitively from Figure 5), we see that ECBs for tuples in the same category are comparable, but ECBs across categories may or may not be comparable. For example, ECBs in Categories R1 and S1 are always dominated by others. However, between tuple x from Category R2 and y from Category S2, B_x dominates B_y if $\frac{v_x-(t_0-w_S)}{2w_S+1} \geq \frac{v_y-(t_0-w_R)}{2w_R+1}$, but they are incomparable otherwise. Therefore, in general, we need HEEB to compare them.

For HEEB, we choose L^{exp}. We use $(w_R + w_S)/2$ as a very crude estimate for the average lifetime of a cached tuple, and choose α accordingly. A more principled technique would be to observe the average lifetime at runtime and adjust α adaptively. We plan to experiment with this technique as future work. For this scenario, since each input process is independent and has linear trend with i.i.d. noise, we can use both value- and time-incremental computation. We report the performance of HEEB in Section 8.

7.4 Linear Trend, Bounded Normal Noise

This scenario corresponds to our motivating example in Section 1. For both caching and joining, it turns out that incomparable ECBs may arise, so HEEB is needed. See [11] for more details.

7.5 Random Walk with Drift

For the caching problem, consider a reference stream R generated by a random walk $X_t^R = \phi_0 + X_{t-1}^R + Y_t^R$, where ϕ_0 represents a constant drift over time and Y_t^R's represent i.i.d. zero-mean steps. Once again, incomparable ECBs may arise (see [11] for details), so HEEB is needed. We use L^{exp} and set α to the size of the cache. Based on Theorem 3 we can precompute a function h^R to facilitate calculating H_x online. As an example, we have precomputed h^R for three cases, where the random walk steps (Y_t^R's) follow normal distributions with variance of 1, and the drift constants (ϕ_0) are 0, 2, and 4, respectively. The three curves are plotted in Figure 6. Intuitively, a larger positive drift tends to make

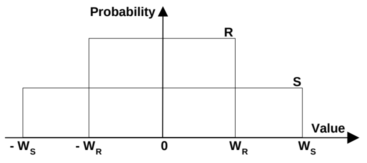

Figure 3: Drifting bounded uniform distributions.

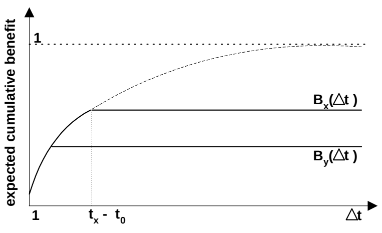

Figure 4: Caching ECBs (linear trend, bounded uniform noise).

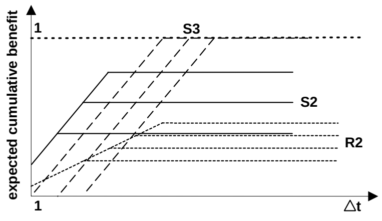

Figure 5: Joining ECBs (linear trend, bounded uniform noise).

it more desirable to cache tuples to the right of the current mean; those that are away by a constant multiple of drift also receive some additional preference because Y_t^R will most likely be 0.

Interestingly, if drift is zero and random walk steps follow symmetric unimodal distributions (e.g., normal), a straightforward analysis of ECBs reveals that in fact all ECBs are comparable, and all candidate tuples can be ranked by their distance from the current position of the random walk (again, see [11] for details). According to Theorem 1, it is therefore optimal to *discard the tuple whose join attribute value is farthest away from the most current reference*. As shown in Figure 6, HEEB agrees with this optimal algorithm in the case of zero drift and normal steps. On the other hand, this scenario is not *0-order* or *almost stationary* [2], so A_o is not applicable. This scenario is an example where our framework is able to derive caching results more general than classic ones.

For the joining problem involving a second stream S generated by another random walk, both analysis and conclusion are similar to the caching problem, and hence omitted for brevity (see [11] for details). Basically, we need to precompute h^R for R and h^S for S according to Theorem 3, and store their approximations online for calculating H_x. We report the performance of HEEB for this scenario in Section 8.

8. EXPERIMENTAL RESULTS

Data and Queries Instead of experimenting with simple configurations (e.g., stationary independent streams, where we have provably optimal results), we focus on more complex and interesting cases. We have five experiment configurations: TOWER, ROOF, FLOOR, WALK, and REAL. The first four use synthetic data and the last one uses a real data set.

For TOWER, ROOF, and FLOOR, input streams R and S are generated by independent stochastic processes with linear trends. Unless otherwise noted, pdf's for R and S drift at the same constant speed of 1, with R lagging one step behind S; noise terms are bounded zero-mean distributions, where the bounds are $[-10, 10]$ for R and $[-15, 15]$ for S. TOWER and ROOF correspond to the scenario of Section 7.4, with bounded normal noises. TOWER's noises have smaller standard deviations (1 for R and 2 for S) than those of ROOF (3.3 for R and 5 for S). FLOOR corresponds to the scenario of Section 7.3, with bounded uniform noises. Figure 7 show the noise pdf's (of S) for all three configurations.

For WALK, input streams are generated by two random walks as discussed in Section 7.5, where the steps follow discretized normal distributions with mean 0 and variance 1. This configuration is rather peculiar in that the two streams do not behave consistently over time: they frequently diverge to the point that their pdf's are far apart and completely disjoint. Thus, the number of result tuples is highly variable between runs and tends to be much lower than other configurations.

For REAL, we use the Melbourne temperature data set available from StatSci.org, which tracks daily temperatures for Melbourne over a period of 10 years (3650 temperatures in total). We feel that self-joins on one data set are somewhat contrived, so we consider the caching problem instead, joining the temperature stream with a synthetic database relation that stores projected energy consumption level for each temperature range (every 0.1 degree Celsius).

Algorithms and Performance Metric We have implemented OPT-offline, FLOWEXPECT, RAND (which simply discards tuples at random), PROB$_p$, LIFE$_p$, and our heuristic HEEB. Please refer to Section 7 for the choice of L_x for HEEB in synthetic data experiments. LIFE$_p$ requires a sliding window to determine tuples' lifetimes. For TOWER, ROOF, and FLOOR configurations, we use the bound on the noise distribution as the sliding window. We make RAND and PROB$_p$ aware of this sliding window, too, so they always discard tuples outside the window first. For the WALK configuration, however, there is no window because of the nature of random walk, so we do not use LIFE$_p$ in WALK experiments.

For each experiment with synthetic data, we conduct 50 runs, each consisting of two streams of 5000 tuples each. For each run, we measure the performance of an algorithm by the total number of result tuples generated after a cache warm-up period (no less than four times the cache size). We then report the average over all 50 runs. Although all runs for the same configuration share the same statistical properties, each run is different. It turns out that differences in performance are small: under 5% in all cases except WALK (for reasons explained earlier) and extremely small caches.

Synthetic Data Results Figure 8 compares the performance of various algorithms across all synthetic data configurations. The size of the cache is 10 tuples, roughly a third of the memory required for most algorithms to generate most join results. The scale is intentionally kept small so that FLOWEXPECT is feasible. OPT-offline is the obvious winner across the board, because of its unfair advantage of knowing the entire input streams in advance; all other algorithms have at best probabilistic knowledge of the future.

We see that HEEB beats RAND, PROB$_p$, and LIFE$_p$ consistently, and even FLOWEXPECT in most cases. The unimpressive results of FLOWEXPECT highlight its suboptimality due to restricted search space (Section 4). PROB$_p$ and LIFE$_p$ (especially PROB$_p$) do not work well when a trend is present. As discussed in Section 1, when the past is used to predict future in a simplistic manner, PROB$_p$ tends to discard new arrivals because they tend to be least frequently joined in the past. LIFE$_p$ fares better because new arrivals gain some advantage by having longer lifetimes, but it still suffers from the incorrect estimation of join probability. RAND, although oblivious, turns out to fairly competitive.

The amount of improvement achieved by exploiting statistical properties varies across configurations. For TOWER, algorithms that correctly exploit its statistical properties have a huge advantage over RAND, PROB$_p$, and LIFE$_p$. The reason is that noises in TOWER have small variances, so the model makes fairly accurate predictions, which can be used effectively in making good cache replacement decisions. As we move to FLOOR, this advan-

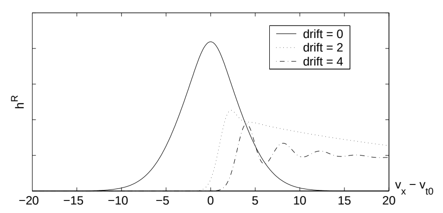

Figure 6: Precomputed h^R (random walk with drift).

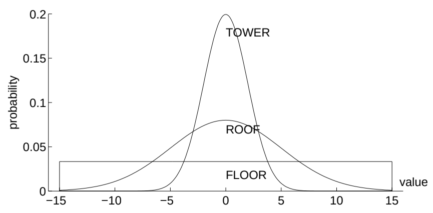

Figure 7: TOWER/ROOF/FLOOR noises.

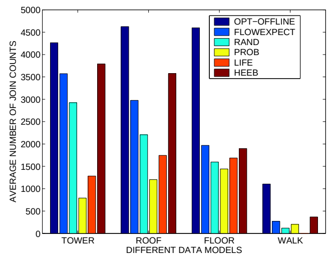

Figure 8: Comparison across synthetic data configurations.

tage diminishes as variances grow larger and future becomes less predictable. For WALK, near future is still predictable, which gives FLOWEXPECT and HEEP an edge over RAND and PROB$_p$; however, variances of future random walk steps cumulate very quickly, so no online algorithm comes close to OPT-offline in its ability to identify joins involving tuples from not-so-near future.

To study the effect of cache size, we vary it from 1 tuple to 50 tuples and show the results in Figures 9–12. In all cases, with more memory, all algorithms perform better and eventually catch up with OPT-offline (except for the case of WALK). For TOWER and ROOF, HEEB converges to OPT-offline much faster than other heuristics. For FLOOR, HEEB still does well but is not as spectacular, for reasons discussed earlier.

The Histogram-Based Method For above experiments, we have computed HEEB using techniques from Section 6.1. Next, we perform experiments to study the performance of our histogram-based method for estimating HEEB from Section 6.2. Because of space constraints, we only show results from two setups; more results can be found in [11]. Figure 13 compares the performance of computed HEEB and the histogram-based method (denoted HIST) with varying cache sizes for input streams with linear trend and low-variance noises ($\sigma = 1$). Figure 14 shows the results for inputs with high-variance noises ($\sigma = 8$). In both experiments, the number of buckets in each histogram is fixed at 10. We also plot the performance of RAND for baseline comparison.

Computed HEEB has the advantage of knowing actual parameter values in advance. On the other hand, HIST has to "learn" the input patterns gradually at runtime, so it is initially prone to poor caching decisions. Because of this initial learning period and the approximate nature of histograms, we do not expect to see the histogram-based approach perform better than computed HEEB, which is indeed the case shown by the figures. For inputs with low-variance noises, we see in Figure 13 that HIST is almost as good as computed HEEB (and much better than RAND), because the inputs exhibit clear patterns that histograms can capture quickly and effectively. For inputs with high-variance noises, as shown in Figure 14, the performance gap between computed HEEB and HIST becomes noticeable, because high noises lead to less predictable patterns, which the histogram-based approach has a hard time learning. Nevertheless, HIST still performs better than RAND and no worse than roughly 90% of computed HEEB. Also note that the advantages of HEEB and HIST over rand are smaller because of the high noise.

Real Data Results We perform a standard MLE procedure offline on REAL and obtained the following AR(1) model: $X_t = 0.72X_{t-1} + 5.59 + Y_t$, where Y_t is a normal distribution with standard deviation 4.22. We compare the performance of LFD [5] (the optimal offline algorithm), RAND, PROB$_p$ (essentially LFU in this case), LRU, and HEEB for memory sizes from 10 to 300. For LFU and LRU, we implement their perfect versions instead of approximations. For HEEB, we use L^{exp} with α set to the size of the cache. The results are summarized in Figure 15. They are from one single run since a real data set is used. Because temperature data exhibits a significant amount of locality (as evidenced by the small gap between RAND and LFD), all heuristics perform reasonably well, with HEEB leading the pack and beating LRU and LFU by as much as 20% for certain memory sizes.

Recall from Theorem 3 that HEEB for an AR(1) model is a surface $h_2(v_x, x_{t_0})$. We precompute and approximate this surface using bicubic interpolation of 25 control points equally spaced over the domain. We have found this simple approximation satisfactory in terms of space, speed, and accuracy (see [11] for the actual and approximated surfaces). Better approximation techniques will likely improve accuracy and/or reduce the number of control points. We also plan to investigate the effect of approximation on the performance of HEEB as future work.

Memory Allocation Recall the question posed in Section 1: when is it better to discard tuples from one stream instead of the other? We conduct a series of experiments with the TOWER configuration to see how HEEB would decide. We start with R and S having identical statistical properties and no lag between them. First, we make R lag behind S for 2 and 4 time steps, while keeping all other parameters unchanged. Second, we double and quadruple the standard deviation of S's noise, again keeping other parameters unchanged. The results are summarized in Figure 16. The vertical axis shows the fraction of cache taken by R tuples. We see that HEEB allocates less memory to streams that lag behind or have large variances. Intuitively, with lag, it is better to cache the "leading" stream, because tuples from the stream behind are unlikely to join with future arrivals, although they can still join with previously cached tuples from the leading stream. For streams with different variances, assuming no lag, tuples from the low-variance stream are always "covered" by the high-variance stream, while many tuples from the high-variance stream fall behind the low-variance stream and thus should be discarded. In both cases, HEEB naturally matches intuitions.

9. CONCLUSION AND FUTURE WORK

The primary goal of this work has been to develop a principled approach to the problem of joining streams with limited cache memory, given known or observed statistical properties of input streams. We have developed a framework which allows us to make optimal cache replacement decisions (in terms of expected number of result tuples produced) based on ECB dominance tests. These tests naturally lead to provably optimal algorithms in a number of scenarios. In case that an optimal algorithm cannot be found efficiently, we have provided a heuristic called HEEB, which agrees with all optimal decisions identified by these tests.

We have shown how to implement HEEB efficiently using incremental computation and precomputation techniques as well as a histogram-based method for estimating HEEB at runtime. The

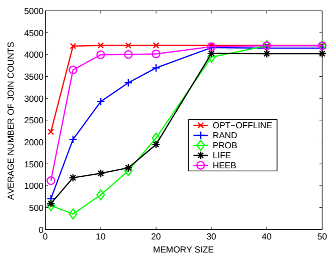

Figure 9: TOWER.

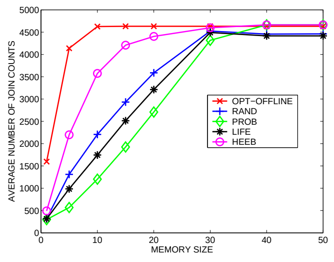

Figure 10: ROOF.

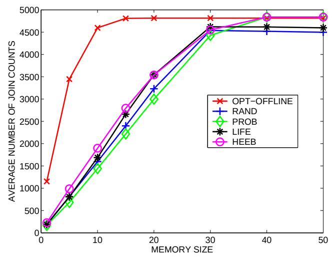

Figure 11: FLOOR.

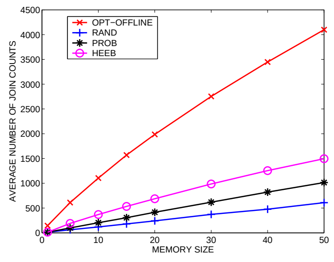

Figure 12: WALK.

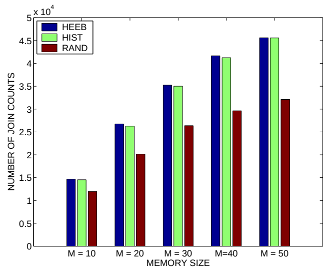

Figure 13: Performance of the histogram-based HEEB; low variance.

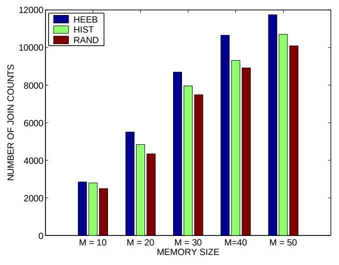

Figure 14: Performance of the histogram-based HEEB; high variance.

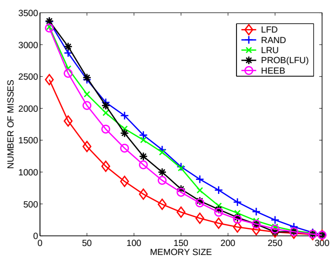

Figure 15: REAL.

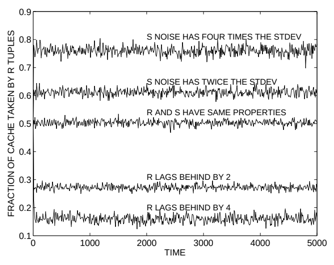

Figure 16: Memory allocation between two streams.

histogram-based method provides the benefit of HEEB without its computational overhead or prior knowledge of the parameter values of the input processes. We have demonstrated the power of the framework in several case studies and verified the effectiveness of HEEB with experiments.

Finally, we have also identified the connection between join state management and the classic caching problem, and shown that, by reducing caching to joining, the two problems can be analyzed within the same framework despite their subtle differences.

As future work, we plan to further investigate the appropriateness of existing caching techniques in the context of join state management. We also plan to consider generalizations to non-equality joins, other stream operators, and metrics other than MAX-subset.

10. REFERENCES

[1] Special issue on data stream processing. *IEEE Data Engineering Bulletin*, 26(1), March 2003.

[2] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of optimal page replacement. *Journal of the ACM*, 18(1):80–93, January 1971.

[3] B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Maintaining variance and k-medians over data stream windows. In *Proc. of the 2003 ACM Symp. on Principles of Database Systems*, San Diego, California, USA, June 2003.

[4] S. Babu, U. Srivastava, and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. on Database Systems*, 29(3), September 2004.

[5] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM System Journal*, 5(2), 1966.

[6] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, Philadelphia, USA.

[7] Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang. Multi-dimensional regression analysis of time-series data streams. In *Proc. of the 2002 Intl. Conf. on Very Large Data Bases*, pages 323–334, Hong Kong, China, August 2002.

[8] A. Das, J. E. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, San Diego, California, USA, June 2003.

[9] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proc. of the 2001 Annual ACM Symposium on Theory of Computing*, pages 471–475, Heraklion, Crete, Greece, July 2001.

[10] S. Irani. Competitive analysis of paging: A survey. In *Proc. of the Dagstuhl Seminar on Online Algorithms*, Dagstuhl, Germany, June 1996.

[11] J.Xie, J. Yang, and Y. Chen. On joining and caching stochastic streams, November 2004. Technical report, Duke University. `http://www.cs.duke.edu/dbgroup/2004-xyc-stream.pdf`.

[12] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Proc. of the 2003 Intl. Conf. on Data Engineering*, Bangalore, India, March 2003.

[13] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley, 1997.

[14] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proc. of the 2003 Conf. on Innovative Data Systems Research*, January 2003.

[15] E. J. O'Neil, P. E. O'Neil, and G. Weikum. An optimality proof of the LRU-k page replacement algorithm. *Journal of the ACM*, 46(1):92–112, January 1999.

[16] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, Toronto, Canada, August 2004.

[17] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, Berlin, Germany, September 2003.