On Joining and Caching Stochastic Streams

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

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

Duke CS Technical Report 2003-XX

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, which render 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 Related Work

There is 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 [13]. 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 3). 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., [10, 3] and many more); some target specifically at time series (e.g., [7, 18]).

As discussed in Section **??**, 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 [14], are good approximations of A_o. As we will see in Section 5, 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 5.5).

There is a large body of work on *competitive analysis* of caching algorithms, surveyed in [11]. Competitive algorithms offer much stronger performance guarantees than algorithms that are optimized for the average case, such as A_o and ours. In general, developing practical algorithms with good competitive ratios is hard, 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 their models make specific assumptions about the inputs that allow for more efficient solutions; on the other hand, our model is general.

2 Problem Setup

We model each 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 by S 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.

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:

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

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:

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

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 $r_{(v,\_)}^t$ 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 these 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.

Let $\mathbb{D}$ the value domain of join attribute v for all tuples in reference sequence $R = \{r^t_{(v,k(v))}\}$ time stamped on t and pair $(v, k(v))$ represents the $(k(v)+1)$-th occurrence of value v, and we construct another stream S as introduced above, then we formalize above argument by following theorem,

Theorem 1 [1] *Given cache C with initial state C_0 at time 0, suppose $R = \{r^t_{(v,k(v))}\}$, where $v \in \mathbb{D}$, $t \in \mathbb{Z}^+$, is a reference sequence. Let $H(C_0, R, P)$ be the number of hits generated by cache C for reference sequence R under a reasonable replacement policy P, and $J(C_0, R, S, P)$ is the number of tuples resulting from joining two streams R and S given the same cache C_0 and replacement policy P. Then,*

$$H(C_0, R, P) = J(C_0, R, S, P)$$

3 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 the joining problem 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. Ideally, if we know that we are going to compute the join only up to a specific time t_{ref}, or if we only care about maximizing the number of result tuples up to t_{ref}, we can choose $l = t_{ref} - t_0$. As we will see later in this section, a large l slows down the algorithm. In practice, a smaller l might suffice, assuming that the current decision has little influence far into the future. In general, however, a decision made based on a smaller look-ahead may not be optimal.

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

[1] In this report, the proof of all theorems, corollaries and lemma can be found in the appendix section.

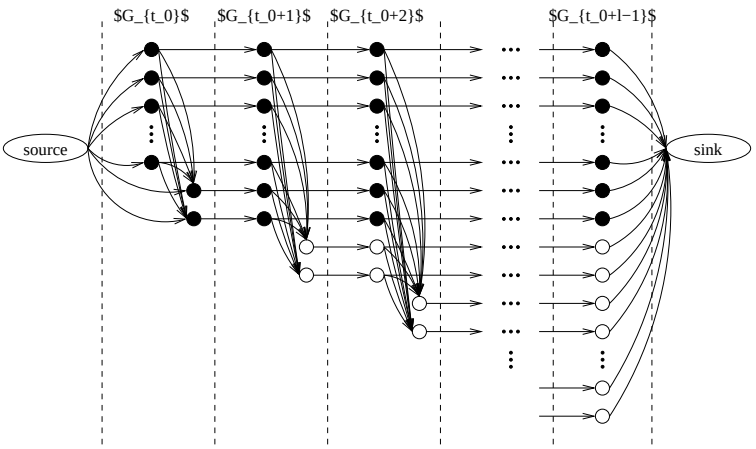


Figure 1: A network flow graph.

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 in Section 3.4 regarding the optimality of FLOWEXPECT, which reveals the complexity of the solution space. Readers familiar with OPT-offline [8] can skim Section 3.1 while focusing on the difference between the two algorithms.

3.1 Constructing the Flow Graph

Intuitively, the network flow graph captures all possible cache traces from time t_0 up to time $t_0 + l$. The general form of the graph is shown in Figure 1. Between source and sink, the graph consists of l vertical *slices* $G_{t_0}, G_{t_0+1}, \ldots G_{t_0+l-1}$, where slice G_t corresponds to the state of the cache at time t.

The first slice, G_{t_0}, contains $k + 2$ nodes: k nodes represent the k tuples currently in the cache at time t_0, and the remaining two represent the tuples produced by the two input streams at time t_0. We call these nodes *determined nodes* (denoted by a black dot) since they represent tuples that are known at the current time.

The second slice, G_{t_0+1}, contains copies of the $k + 2$ nodes from the previous slice, connected to the respective originals with horizontal arcs. Intuitively, a flow along one of these horizontal arcs means that we keep the corresponding tuple in cache from time t_0 to $t_0 + 1$. In addition, two new nodes represent the tuples produced by the input streams at time $t_0 + 1$. We call these nodes *undetermined nodes* (denoted by a white dot), because we do not know their join attribute values (although we assume that we know how the two random variables are distributed). The two new nodes are connected to the duplicate nodes (those copied from the previous slice) with non-horizontal arcs. Intuitively, a flow along one of these non-horizontal arcs means that we replace an existing tuple in cache with a newly arrived tuple at time $t_0 + 1$.

In general, slice G_{t+1} is constructed from slice G_t by copying all nodes in G_t and adding two new nodes for the two tuples arriving at time $t + 1$. Horizontal arcs connect original nodes in G_t to their duplicates in G_{t+1}, and non-horizontal arcs connect duplicate nodes to the two new nodes. Finally, all nodes in the first

and last slices are connected to source and sink, respectively.

All arcs have unit capacity, and we push a flow of size k through the graph. Assume for now that the flow is integral (we will explain why later), i.e., the flow on each arc is either 1 or 0. Except source and sink, each node has either in-degree of 1 or out-degree of 1, so it can admit either a flow of 1 or no flow at all. Therefore, a flow of size k through the graph consists of k paths connected only at source and sink, but disjoint otherwise. Each path carries a flow of size 1 and represents the sequence of cache replacement decisions made during $[t_0, t_0 + l - 1]$ for a particular cache slot.

We assign costs to arcs according to the expected benefits generated by joining cached tuples with incoming tuples. The various cases are discussed below. Since we are solving for a flow with minimum cost, the costs are negated expected benefits.

- A horizontal arc connecting a node n in G_t to its copy in G_{t+1} represents the action of keeping the tuple from time t to $t + 1$. This action can potentially generate one result tuple by joining with the new tuple from the partner stream at time $t + 1$. Suppose that n represents a tuple from stream S, which joins with partner stream R.

  If n is a determined node with join attribute value v, then this tuple will join with the incoming R tuple at time $t + 1$ with probability $\Pr\{X^R_{t+1} = v \mid \bar{x}_{t_0}\}$. This probability is conditioned on $\bar{x}_{t_0}$, the history of all streams observed by the algorithm up to the current time t_0. Therefore, we assign a cost of $(-\Pr\{X^R_{t+1} = v \mid \bar{x}_{t_0}\})$ to this arc.

  If n is an undetermined node representing a tuple arrived at time t' ($t_0 < t' \leq t$), then this tuple will join with the incoming R tuple at time $t + 1$ with probability $\sum_v \Pr\{X^S_{t'} = v \cap X^R_{t+1} = v \mid \bar{x}_{t_0}\}$. Therefore, we assign a cost of $(-\sum_v \Pr\{X^S_{t'} = v \cap X^R_{t+1} = v \mid \bar{x}_{t_0}\})$ to the arc.
- A non-horizontal arc connecting two nodes in G_t represents the action of replacing a tuple in the cache with a newly arrived tuple at time t. This action does not generate any result tuple at time t; the newly cached tuple might generate a result tuple at time $t + 1$, but that benefit is attributed to the arc leaving the newly cached tuple. Therefore, we assign a cost of 0 to all non-horizontal arcs.
- For arcs leaving source, we assign a cost of 0.
- For arcs entering sink, we assign their costs by treating them as horizontal arcs out from G_{t+l-1}.

We ignore the benefit at the current time (t_0) because it is independent of current and future replacement decisions. We also ignore the benefit generated by joining tuples arriving at the same time because they will be joined regardless of replacement decisions.

3.2 Solving the Min-Cost Flow Problem

By construction of the graph, each feasible integral flow of size k corresponds to a sequence of cache replacement decisions during $[t_0, t + l - 1]$. Furthermore, it is not difficult to see that there is a one-to-one correspondence between the set of all possible sequences of cache replacement decisions and the set of all feasible size-k integral flows through the graph. The following theorem states that the cost of a feasible size-k integral flow correctly reflects the expected benefit of the corresponding sequence of cache replacement decisions.

Theorem 2 *The cost of a feasible integral flow of size k through the graph constructed in Section 3.1 is the negated expected benefit generated by the cache during $[t_0 + 1, t_0 + l]$ by following the corresponding sequence of cache replacement decisions.*

Therefore, by solving for the min-cost flow, FLOWEXPECT finds the sequence that maximizes the expected benefit, and then follows the decision made by this sequence at time t_0.

The integral flow assumption is justified by the fact that there always exists an optimal flow which is integral, which follows from the observation that both the flow target and capacities are integers.

Using an algorithm from Goldberg [9], we can solve the min-cost flow problem in $O(n^2 m \log n)$ time, where m is the number of arcs and n is the number of nodes in the graph. In our case, this bound translates to $O((k+l)^3 l^3 \log((k+l)l))$ for each step of FLOWEXPECT (see **Appendix** for detailed derivation), not even considering the work of constructing the graph. Clearly, a larger l further exacerbates the problem. Because of its complexity, FLOWEXPECT is simply not practical, although we can still use it as a yardstick for measuring the effectiveness of other algorithms.

3.3 Definition of Optimality

An algorithm A for making cache replacement decisions takes four inputs: K, the set of tuples currently in the cache, N, the set of newly arrived tuples, H, the sequence of all arrivals up to now, and p, a joint probability function of future tuples' join attribute values (conditioned on H). The algorithm outputs the new state of the cache, a subset of $K \cup N$ of size $|K|$. Given p and initial K_0, N_0, and H_0, consider a sequence of subsequent arrivals $\mathcal{N} = N_1, N_2, \ldots, N_l$. We define $f(A, \mathcal{N})$, the *performance of* A on $\mathcal{N}$, inductively as follows:

- $f_0 = 0$.
- $K_{i+1} = A(K_i, N_i, H_i, p)$.
- $f_{i+1} = f_i + \bowtie(K_{i+1}, N_{i+1})$, where $\bowtie(K, N)$ denotes the number of result tuples produced by joining N with K.
- $H_{i+1} = H_i N_{i+1}$.
- $f(A, \mathcal{N}) = f_l$.

Again, note that we are ignoring the result tuples generated by N_0 and those generated by joining N_i with itself, because they are produced regardless of A's behavior. The *expected performance of* A over sequences of l subsequent arrivals is defined by $\sum_{\mathcal{N}} \Pr\{\mathcal{N} \mid H_0\} f(A, \mathcal{N})$, where $\mathcal{N}$ is any sequence of l arrivals. An *optimal algorithm* (w.r.t. a given l) is one with the highest expected performance over sequences of l arrivals.

3.4 Suboptimality of FlowExpect

As expensive as FLOWEXPECT may be, it 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, we consider a small but carefully constructed example. Suppose that 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): We know some for sure and others only probabilistically. The symbol "$-$" represents tuples that do 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 min-cost flow problem 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. Therefore, we need to develop simpler, more practical approaches.

4 A Practical Framework

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

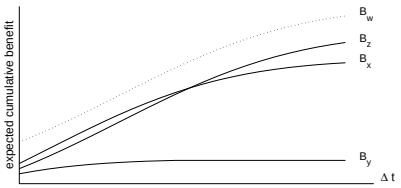


Figure 2: Example ECBs.

4.1 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 (with proof in appendix) shows how to compute the ECB. Here and thereafter, we use v_x to denote the value of the join attribute for a tuple x. Recall that $\bar{x}_{t_0}$ denotes 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 $B_x(\Delta t) = \sum_{t=t_0+1}^{t_0+\Delta t} \Pr\{X_t^R = v_x \mid \bar{x}_{t_0}\}$.*

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 2). 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 (again, see appendix 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 $B_x(\Delta t) = 1 - \Pr\{\bigcap_{t=t_0+1}^{t_0+\Delta t} X_t^R \neq v_x \mid \bar{x}_{t_0}\}$. 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 **??**. In 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.

4.2 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 3 *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 appendix, 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'.*

4.3 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 **??**.

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 5) and obtain a cache replacement algorithm that works well for the given input (as we shall see in experiments in Section 6). Furthermore, HEEB agrees with Theorem 3, i.e., it 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.[2]

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.

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

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:

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

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. we formally state and prove this claim in Appendix.

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 4 below (proven in Appendix) indicates.

[2]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.

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 4 *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 5. 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 the following table. 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.

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)	$H_x^{inv} = \sum_{\Delta t=1}^{\infty}(\Pr\{x \text{ is first used at } t_0 + \Delta t \mid \bar{x}_{t_0}\} \ / \ \Delta t)$, 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*

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 5.

4.4 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. In this subsection, we describe a set of optimization techniques aimed at making HEEB more practical. In Section 5, we will see how they are applied in different scenarios.

4.4.1 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} simply by treating α as ∞. These corollaries, proven in Appendix, 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*

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

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*

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

4.4.2 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 **??** is one such process with bounded normal noise. Consider tuple y at time t_0 and tuple y' at time t in Figure **??**, 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 Appendix and follows directly from the definition and calculation 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.

4.4.3 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 5 *Suppose that stream R is modelled by $X_t^R = \phi_0 + \phi_1 X_{t-1} + Y_t^R$, 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 5 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 4.3. In the full version Appendix, 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 5.5. 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.

5 Case Studies

In this section 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 4.4.

5.1 Offline Streams

If we know the sequence of join attribute values $\{a_0, a_1, \ldots\}$ to be produced by a stream in advance, we may analyze the sequence as an independent stochastic process $\{X_t\}$ where $\Pr\{X_t = a_t\} = 1$. This scenario has little practical significance but nevertheless enables us to compare results with known optimal offline algorithms.

For the caching problem, Corollary 1 tells us that, at the current time t_0, the ECB of a database tuple x is given by $B_x(\Delta t) = 1 - \prod_{t=t_0+1}^{t_0+\Delta t} \Pr\{X_t \neq v_x\}$. Let t_x denote the first time after t_0 when v_x appears in the reference stream. It is not hard to see that $B_x(\Delta t)$ is just a single-step function that jumps from 0 to 1 at $\Delta t = t_x - t_0$. Clearly, dominance induces a total order on the candidate tuples: B_x strongly dominates B_y if x is referenced earlier than y. Therefore, according to Theorem 3, it is optimal to *discard the tuple that will not be referenced for the longest time*; heuristics are unnecessary in this case. This result agrees perfectly with the well-known optimal offline cache replacement policy LFD (Longest Forward Distance) [5]. It is

nice to see that a straightforward application of our general framework leads naturally to an optimal policy for offline caching.

For the joining problem, Lemma 1 implies that each tuple's ECB is a step function with potentially multiple steps. Each step corresponds to an occurrence of a joining tuple form the other stream, and increases the value of the function by 1. In general, these ECBs are not comparable. Since streams are offline, FLOWEXPECT is a better option. Because we know the entire input streams deterministically, edge weights in the flow graph constructed by FLOWEXPECT will be either 0 or 1, and there is no need to recompute the min-cost flow at every time step. In this case, FLOWEXPECT degenerates into OPT-offline [8], which is optimal.

5.2 Stationary, Independent Streams

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

For the caching problem, Corollary 1 tells us that, for a database tuple x at current time t_0, $B_x(\Delta t) = 1 - \prod_{t=t_0+1}^{t_0+\Delta t} \Pr\{X_t^R \neq v_x\} = 1 - (1 - p_R(v_x))^{\Delta t}$. Obviously, for any two database tuples x and y, B_x dominates B_y if $p_R(v_x) \geq p_R(v_y)$. Therefore, according to Theorem 3, 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,[3] and that both LRU (Least Recently Used) and WS (Working Set) can be seen as approximations to A_o. In [14], it was shown that LRU-k 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 3, it is optimal to *discard the tuple whose join attribute value is least likely to appear* in the other stream. This policy is basically the PROB heuristic of [8]. Thus, our framework provides a proof of its optimality for stationary, independent stochastic input streams.

5.3 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 can compute the ECB for each database tuple x as follows. Note that it is impossible for a candidate tuple with join attribute value greater than $f(t_0) + w_R$ (assuming no prefetching), because it would 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

[3] Actually, 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 the ECB, it is clear that this result holds.

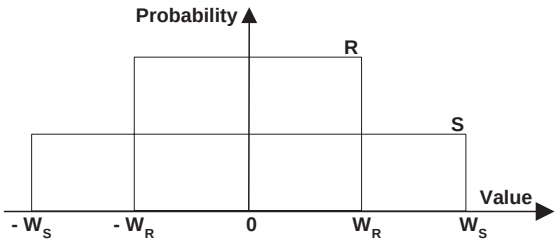


Figure 3: Drifting bounded uniform distributions.

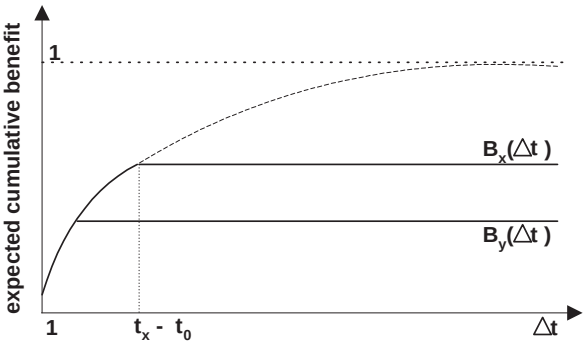


Figure 4: Caching ECBs (linear trend, bounded uniform noise).

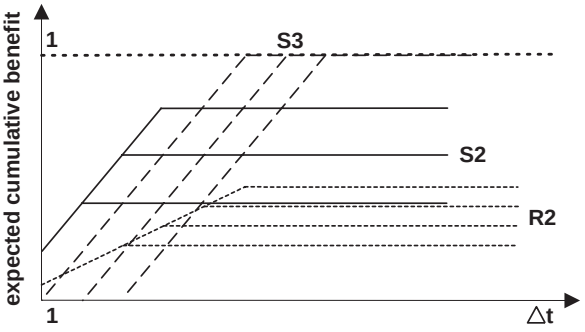


Figure 5: Joining ECBs (linear trend, bounded uniform noise).

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 3. Note the above 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 Appendix.

- 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 6.

5.4 Linear Trend, Bounded Normal Noise

For the joining problem, consider input streams R and S generated by $X_t^R = f^R(t) + Y_t^R$ and $X_t^S = f^S(t) + Y_t^S$ respectively, with the increasing linear trends $f^R(t)$ and $f^S(t)$. Suppose that the noise terms (Y_t^R and Y_t^S) follow independent discretized bounded normal distributions, which are identical over time for each stream but possibly different from the other stream. This scenario corresponds to our motivating example in Section **??**. Sometimes, candidate tuples are comparable. For instance, for tuples x and y both from R, B_x strongly dominates B_y if v_y lies to the left of $f^S(t)$ and is farther away from $f^S(t)$ than v_x (e.g. Figure **??**). Because of space constraint, we leave the detailed analysis to Appendix. In general, not all candidate tuples are comparable, as we have already seen in Section 4.1. Therefore, HEEB is needed.

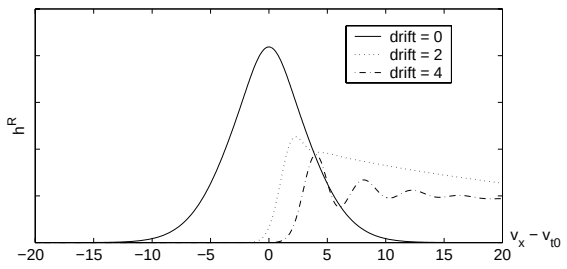


Figure 6: Precomputed h^R (random walk with drift).

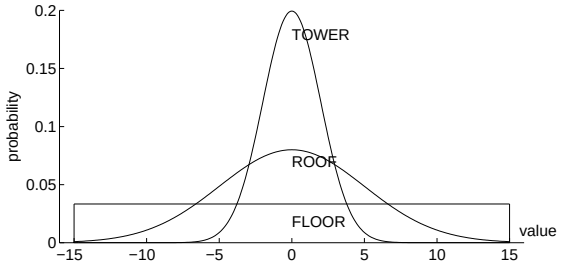


Figure 7: TOWER/ROOF/FLOOR noises.

For the caching problem (with a single reference stream R), it turns out that there might be incomparable candidate tuples as well (see Appendix for details). Hence, HEEB is also needed. It is easy to see that this case is not *almost stationary* [2], so A_o does not apply.

As in the scenario of Section 5.3, HEEB with L^{exp} is easy to implement in this scenario thanks to both time- and value-incremental computation (Section 4.4). We roughly estimate the average lifetime of a cached tuple to be the time it takes for $f(t)$ to increase by twice the standard deviation of the noise distribution, and we choose α accordingly. The performance of HEEB for this scenario is reported in Section 6.

5.5 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 Appendix for details), so HEEB is needed. We use L^{exp} and set α to the size of the cache. Based on Theorem 5 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 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, seeAppendix for details). According to Theorem 3, 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 Appendix for details). Basically, we need to precompute h^R for R and h^S for S according to Theorem 5, and store their approximations online for calculating H_x. We report the performance of HEEB for this scenario in Section 6.

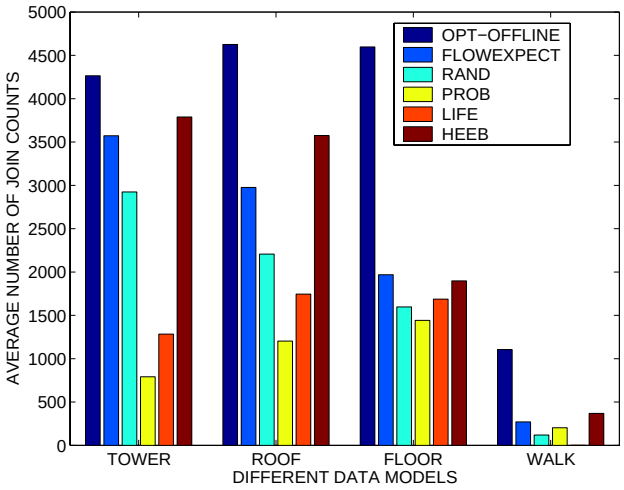


Figure 8: Comparison across synthetic data configurations.

6 Experimental Results

6.1 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 5.4, 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 5.3, 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 5.5, 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 join 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).

6.2 Algorithms and Performance Metric

We have implemented OPT-offline [8], FLOWEXPECT, RAND (which simply discards tuples at random), PROB [8], LIFE [8], and our heuristic HEEB. Please refer to Section 5 for the choice of L_x for HEEB in synthetic data experiments.

LIFE 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 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 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 count over all 50 runs. Although all runs for the same configuration share the same statistical properties, each run is different. It turns out that variances in performance are small: under 5% in all cases except WALK (for reasons explained earlier) and experiments with extremely small cache sizes.

6.3 Synthetic Data Results

Figure 8 compares the performance of various algorithms across all synthetic data configurations. The size of the cache is 10, 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, and LIFE consistently, and even FLOWEXPECT in most cases. The unimpressive results of FLOWEXPECT highlight its suboptimality due to restricted search space (Section 3.4). PROB and LIFE (especially PROB) do not work well when a trend is present. As discussed in Section **??**, when the past is used to predict future in a simplistic manner, PROB tend to discard new arrivals because they tend to be least frequently joined in the past. LIFE 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, and LIFE. The reason is that noises in TOWER have relatively small variances, so the model can make fairly accurate predictions, which can be used effectively in making good cache replacement decisions. As we move to FLOOR, this advantage 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; 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 between a present tuple and one from not-so-near future.

To study the effect of cache size, we vary it from 1 to 50 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 certainly not as spectacular, for reasons discussed

earlier.

6.4 Look-ahead Distance

To study the effect of look-ahead of FLOWEXPECT, we conduct experiments on different look-ahead distance. Due to the constraint of computation abilities, we limit the stream length as 500 and memory size 20. And the streams are generated with linear trend with bound uniform noise, as described in section 5.3. Figure 19 shows the effect of look-ahead distance against the performance of FLOWEXPECT. Obviously performance of other heuristics does not depend on the look-ahead distance. In general, effect of look-ahead distance depends on the statistical property of streams. Interestingly, the experiment shows that look-ahead with limited distance ($\Delta T = 5$) brings apparent performance improvement but after that, the effect on performance improvement becomes indistinguishable and unattractive given the exponentially increasing cost with longer look-ahead distance. It is interesting to seek the reasonable look-ahead distance for FLOWEXPECT, which is beyond the scope of this paper and we leave it for future work due to space constraint.

6.5 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 13. 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 5 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 Figures 15 and 16 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.

6.6 Memory Allocation

Recall the question we posed in Section **??**: 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 14. The vertical axis shows the 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 any future arrivals, although they can still join with previously cached tuples from the leading stream. In

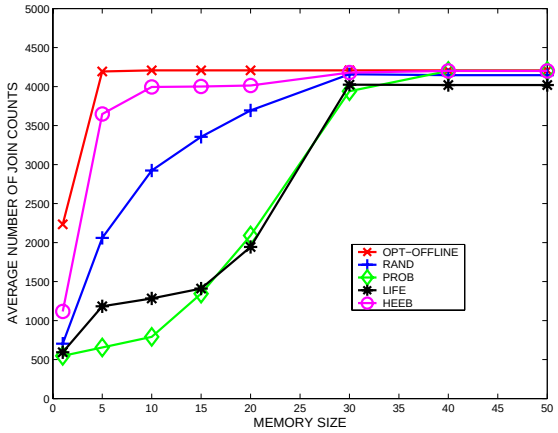


Figure 9: TOWER.

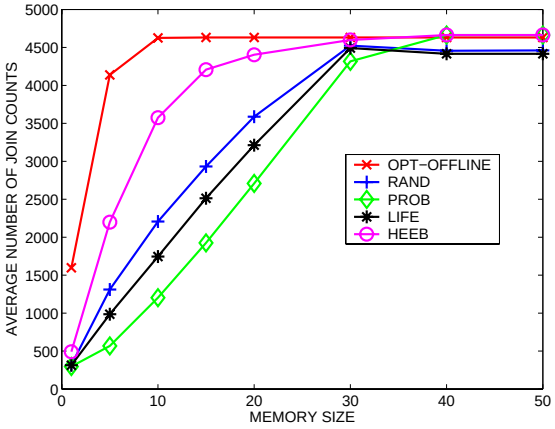


Figure 10: ROOF.

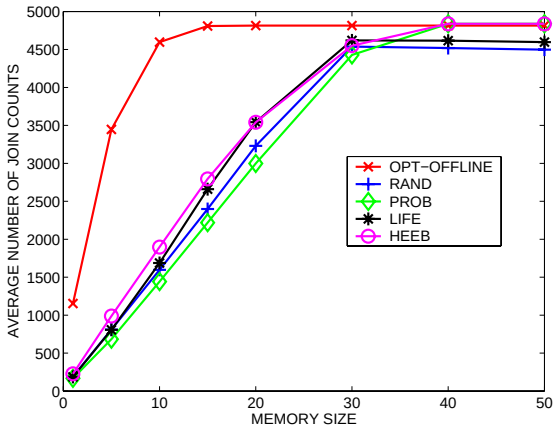


Figure 11: FLOOR.

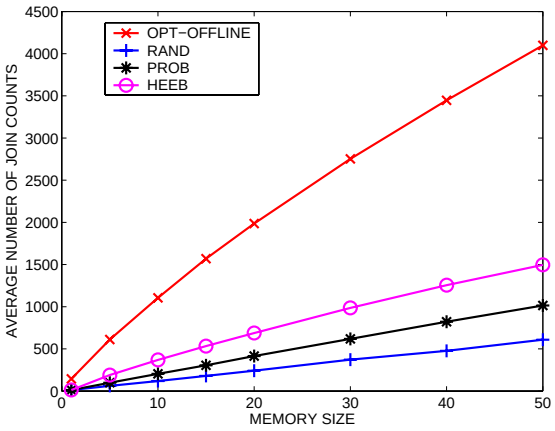


Figure 12: WALK.

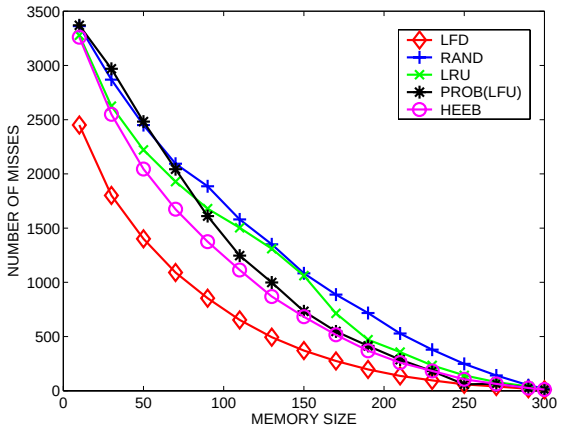


Figure 13: REAL.

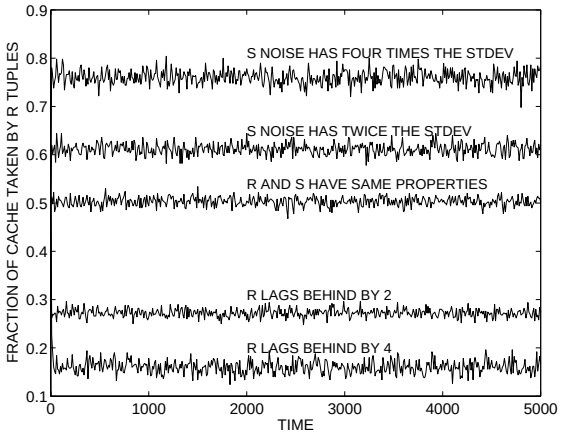


Figure 14: Memory allocation.

the case of streams of 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 our intuitions.

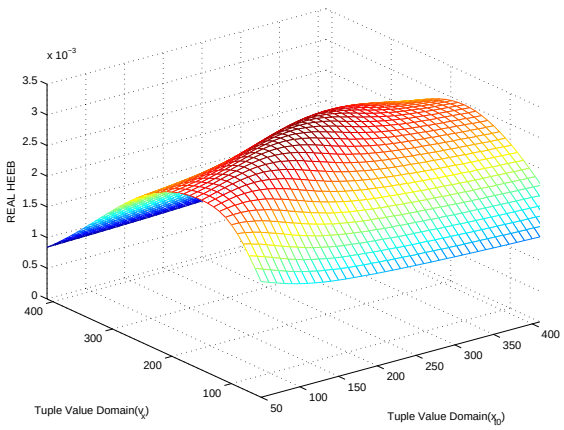


Figure 15: HEEB for REAL (actual).

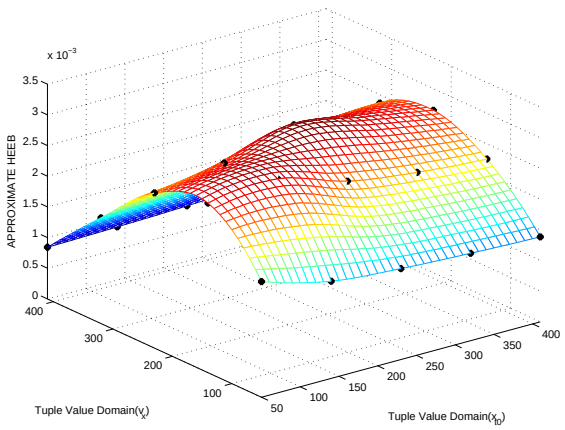
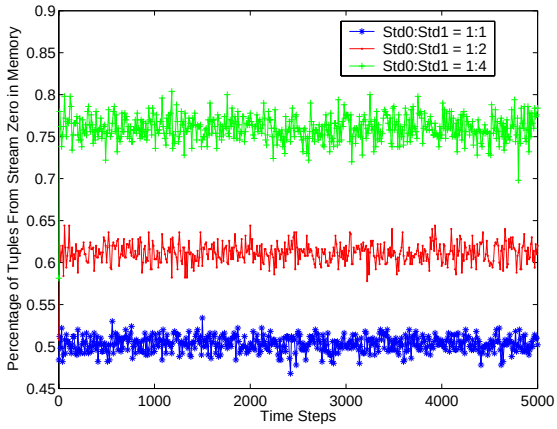


Figure 16: HEEB for REAL (approximated; control points shown by dots).



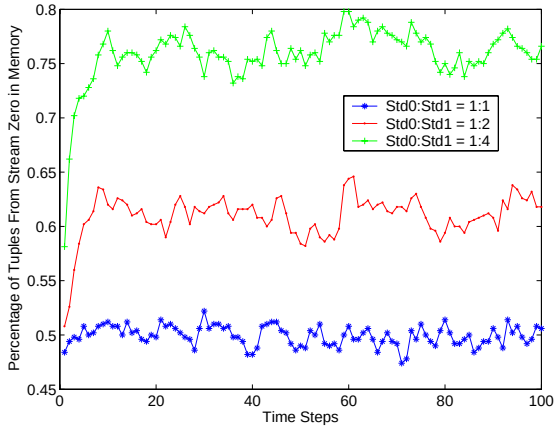
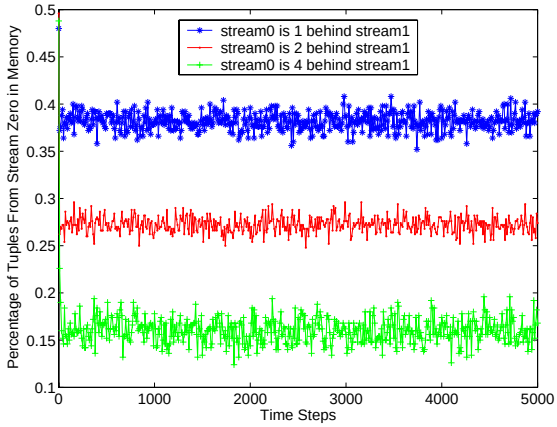


Figure 17: Streams w/ Different Variance



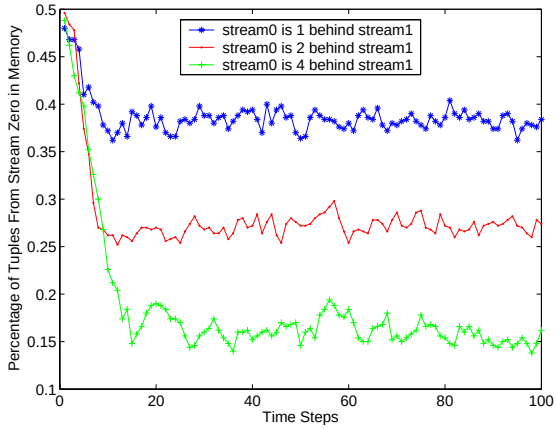


Figure 18: Streams w/ Different Lags

7 Handling the Sliding-Window Semantics

We have assumed regular join semantics so far for simplicity of presentation. However, recall from Section **??** that most stream processing systems use the sliding-window semantics, restricting tuples participat-

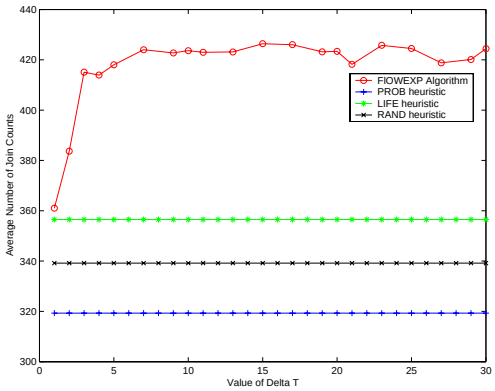


Figure 19: Look-ahead Effect of FlowExpect

ing in the join to those that arrived during $[t_0 - w, t_0]$, where t_0 is the current time. Incorporating sliding windows into our framework is straightforward. Intuitively, as soon as a tuple falls outside the window, it stops generating benefits and its ECB becomes flat. More precisely, for a tuple x that arrived at time t_x, its "sliding-window ECB" is 0 if $t_x \leq t_0 - w$, or $\min\{B_x(\Delta t), B_x(t_x + w - t_0)\}$ otherwise, where B_x is the regular ECB defined by Lemma 1. All other results on joining in Sections 4.1–4.3 remain valid. Among the implementation techniques in Section 4.4, time-incremental computation requires very little modification, while value-incremental computation and recomputation become more complicated because of the dependency of HEEB on tuple arrival time.

As we have done in Section 5 for regular join semantics, we can analyze various scenarios and heuristics under the sliding-window semantics using our framework. To illustrate, consider the case of stationary, independent streams. We have shown that PROB is optimal under regular join semantics in Section 5.2. Unfortunately, under the slide-window semantics, neither PROB nor LIFE is optimal. For example, consider the three candidate tuples below, where $p(x)$ denotes the (time-invariant) probability that an incoming tuple from the other stream has join attribute value v_x, and $l(x)$ denotes the remaining lifetime of x w.r.t. the sliding window (i.e., the number of time steps that x will remain in the sliding window).

- Tuple x_1: $p(x_1) = 0.50$, $l(x_1) = 1$.
- Tuple x_2: $p(x_2) = 0.49$, $l(x_2) = 50$.
- Tuple x_3: $p(x_3) = 0.01$, $l(x_3) = 51$.

PROB prefers x_1 to x_2 because $p(x_1) > p(x_2)$. However, this decision seems rather short-sighted because x_2 has a decent probability to join and will likely continue to be productive long after x_1 has expired. On the other hand, LIFE prefers x_3 to x_1 because $p(x_3)l(x_3) > p(x_1)l(x_1)$. However, this decision seems rather pessimistic since it assumes that there will not be any better tuple to replace x_3 in the next 50 time steps (otherwise x_1 would have been better). Since ECBs are incomparable in this case, we need to resort to HEEB. A good choice of L_x would be a modified version of L^{exp} that sets L_x to 0 once x falls outside the sliding window. This HEEB instance makes a more reasonable assumption about tuple lifetimes in the cache than PROB and LIFE. It weighs short-term benefits more, yet it does not ignore long-term benefits. For the example above, it will likely rank the three tuples as follows: x_2, x_1, x_3, which is arguably the most reasonable order.

8 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 demonstrated the power of the framework in several case studies and verified the effectiveness of HEEB with experiments. 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. Coping with changes (either permanent or transient) in input characteristics is also important. Finally, we plan to consider generalizations to non-equality joins, other stream operators, and metrics other than MAX-subset.

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. Technical report, Stanford University, Stanford, California, USA, November 2002.

[5] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM System Journal*, 5(2):78–101, 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*, pages 263–274, Philadelphia, Pennsylvania, USA, June 1999.

[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] A. V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *Journal of Algorithms*, 22(1):1–29, 1997.

[10] 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.

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

[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] 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.

[14] 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.

[15] Alexander Schrijver. *Combinatorial Optimization, Volumn A*. Spring-Verlag Berlin Heidelberg, 2003.

[16] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. Technical report, Stanford University, Stanford, California, USA, March 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.

[18] Y. Zhu and D. Shasha. StatStream: Statistical monitoring of thousands of data streams in real time. In *Proc. of the 2002 Intl. Conf. on Very Large Data Bases*, pages 358–369, Hong Kong, China, August 2002.

A Basic Notations

We model each stream as a sequence of tuples generated over time. At every time step, one tuple is produced by the stream, and the value of its join attribute follows a certain probability distribution. The table below summarizes the basic notations used in appendix.

notation	meaning
R, S,	streams
$V_S(t)$	random variable of join attribute value of S tuple at time t
$f_S(x,t)$	PDF of the random variable $V_S(t)$
$P_S(x,t)$	$\mathbf{Pr}\{V_S(t) = x\}$
$M_S(t)$	mean of $V_S(t)$
t_0	current time

B Proof of Theorem 1

To prove theorem 1, we first introduce the notation of *reasonable* replacement policy and then we prove that under any *reasonable* policy, the join count in the reduced joining problem is always equal to number of hits in original caching problem.

Definition 1 For any cache state at any time t, if there exists a tuple in cache which will never be referenced in future, then replacement policy P is *reasonable* if and only if P either does not evict any tuple, or evicts a tuple which will never be referenced in future.

Based on above definition, we prove the theorem 1.

Proof:

Suppose at time t, the cache state in caching case for stream is C_t^c and cache state of reduced joining case is C_t^j. It is easy to see that if $C_t^j = C_t^c$ for $\forall t$, then the number of cache hit is equal to the number of joining counts.

Assume at time 0, both cases share the same initial cache state, that is, $C_t^c = C_t^j = C_0$. And assume at time t, $C_t^c = C_t^j = C_t$. Then at time $t+1$, let the reference tuple from stream R is $r_{(v,k(v))}^{t+1}$. Two cases can take place,

1. If $r_{(v,k(v))}^{t+1} \notin C_t^c$, then there does not exist a matching tuple in C_t^j either. Thus both suffer from a miss, leading to no cache hit and join count produced. For the caching case, a corresponding tuple is fetched from the next level of memory, and from transition procedure the fetched tuple is the same as the arriving S tuple $s_{(v,k(v)+1)}$ in the reduced joining case. Since both cache share the same replacement policy, obviously,
$$C_{t+1}^c = C_{t+1}^j$$

2. If $r_{(v,k(v))}^{t+1} \in C_t$, then both cases benefit from a cache hit and a join result tuple respectively. It is easy to see for the caching case, $C_{t+1}^c = C_t^c = C_t$. From the transition procedure, in joining case, the new arrived S tuple is $s_{(v,k(v)+1)}$. Since for the joining case every S tuple is referenced once at most, cached tuple $s_{(v,k(v))}$ will never be used after time t. From definition 1 it is not hard to see that under any *reasonable* replacement policy P, $s_{(v,k(v))}$ is the only tuple that will be never referenced in future because at any time, at most one S tuple in cache is expired due to the join hit. Then P must replace cached tuple $s_{(v,k(v))}$ with the $s_{(v,k(v)+1)}$ and thus the new cache state is,
$$C_{t+1}^j = C_t^j - \{s_{(v,k(v))}\} + \{s_{(v,k(v)+1)}\}$$
Recall that from section 2 tuple $s_{(v,k(v))}$ and $s_{(v,k(v)+1)}$ are identical tuples except the different labels, thus,
$$C_{t+1}^j = C_t^j - \{s_{(v,k(v))}\} + \{s_{(v,k(v)+1)}\} = C_t^j = C_t = C_t^c = C_{t+1}^c$$
Thus, given $C_t^c = C_t^j$ at time t, the caching and reduced joining share the same cache state at time $t+1$,
$$C_{t+1}^c = C_{t+1}^j$$

Therefore, by induction $C_t^c = C_t^j, \forall t \in \mathbb{Z}^+$ and the number of cache hits and then number of join results are equivalent.
$$H(C_0, R, P) = J(C_0, R, S, P)$$

□

C Proof of Theorem 2

In C.1, we prove some lemmas and then in C.2 we prove theorem 2 based on the lemmas in C.1.

C.1 Lemmas

Lemma 2 *Suppose $c(i,j)$ is the cost of the horizontal arc $e(i,j)$ in the flow graph G, and t_i, t_j are time stamps with node i and j, and suppose the benefit at time t_j by following arc $e(i,j)$ is $B(t_j)$, then $\mathbf{E}(B(t_j)) = -c(i,j)$.*

Proof: Without generality we assume tuple(s) associated with node i is(are) from stream R. Since $e(i,j)$ is a horizontal arc, $t_j = t_i + 1$. Thus, if i is a determined node associated tuple value x

$$\mathbf{E}(B(t_j)) = \mathbf{E}(B(t_i+1)) = \mathbf{Pr}\{V_S(t_{i+1}) = x\} = P_S(x, t_i+1) = P_S(x, t_j) = -c(i,j)$$

Otherwise, if i is a non-determined node represented by random variable X, then $\mathbf{E}(B(t_j)) = \mathbf{E}(B(t_i + 1)) = E_X(\mathbf{Pr}\{V_S(t_i+1) = X\})$. Assume t_k is the arrival time of X, it is easy to see that $\mathbf{Pr}\{X = x\} = \mathbf{Pr}\{V_R(t_k) = x\}$, therefore,

$$\begin{aligned}
& \mathbf{E}(B(t_j)) = \mathbf{E}_{\mathbf{X}}(\mathbf{Pr}\{V_S(t_i+1) = X\}) \\
= & \sum \mathbf{Pr}\{X = x\} \times \mathbf{Pr}\{V_S(t_i+1) = x\} \\
= & \sum \mathbf{Pr}\{V_R(t_k) = x\} \times \mathbf{Pr}\{V_R(t_i+1) = x\} \\
= & \sum P_R(x, t_k) \times P_S(x, t_i+1) \\
= & \sum P_R(x, t_k) \times P_S(x, t_j) \\
= & -c(i,j)
\end{aligned}$$

□

Lemma 3 *If the k flow paths $\{p_1, p_2, \ldots, p_k\}$ are the solution of the min-cost max flow problem over graph G, then each path p_i contains no partial flow.*

Proof: If we add an extra arc $e(sink, source)$ with capacity k and cost 0, min-cost max flow problem over graph G becomes a min-cost circulation. Simply assign each arc with demand 0, from the Corollary 12.2a in page 181 of [15], each path p_i is integral. □

Lemma 4 *If the k flow paths $\{p_1, p_2, \ldots, p_k\}$ are the solution of the min-cost max flow problem over graph G, then any two paths p_i and $p_j$of solution do not share any arc with each other.*

Proof: Each arc in flow graph G has capacity 1 and by Lemma 3, each path p_i is integral thus one arc is dedicated one path at most. Thus no two paths can share one arc. □

C.2 Theorem 2

Proof: Assume $\{p_1, p_2, \ldots, p_k\}$ is k integral flow paths from source to sink in flow graph G. And path p_i is $\{se, a_{i1}, a_{i2}, \ldots, a_{in}, sk\}$ where a_{ij} is the j_{th} node along path p_i except the source se and sink sk. Let

$p'_i = \{h_i(t_0, t_0+1), h_i(t_0+1, t_0+2), \ldots, h_i(t_0+l-1, t_0+l)\}$ a set of all horizontal arcs in path p_i in which $h_i(j, j+1)$ is the horizontal arc between time j and $j+1$ which represents keeping the corresponding tuple from time j to $j+1$. It is easy to see that $h_i(t_0, t_0+1) = \{a_{i1}, a_{i2}\}$ and $h_i(t_0+l-1, t_0+l) = \{a_{i(n-1)}, a_{in}\}$. Let $c_i(j, j+1)$ the cost associated with arc $h_i(j, j+1)$. Since all non-horizontal arc of p_i has associated cost 0, thus the cost $C(i)$along path p_i is simply the summarized cost along all its horizontal arcs, $C(i) = \sum_{j=t_0}^{t_0+l-1} c_i(j, j+1)$. That is, the summarized cost along the k paths is

$$\sum_{i=1}^{k} C(i) = \sum_{i=1}^{k} \sum_{j=t_0}^{t_0+l-1} c_i(j, j+1)$$

Assume $B(t)$ the benefit obtained at time t by caching k tuples. Obviously the expected total benefit by keeping k tuples from time t_0 to t_0+l is the summary of the expected benefits during that period, that is,

$$\mathbf{E}(\sum_{t=t_0+1}^{t_0+l} B(t)) = \sum_{t=t_0+1}^{t_0+l} \mathbf{E}(B(t))$$

Since we keep k tuples in cache all the time, the benefit $B(t) = \sum_{i=1}^{k} B_i(t)$ where $B_i(t)$ is the benefit by keeping tuple i from $t-1$ to t. Therefore, the total expected benefit is

$$\begin{aligned} &\mathbf{E}(\sum_{t=t_0+1}^{t_0+l} B(t)) = \sum_{t=t_0+1}^{t_0+l} \mathbf{E}(B(t)) \\ = &\sum_{t=t_0+1}^{t_0+l} \mathbf{E}(\sum_{i=1}^{k} B_i(t)) = \sum_{t=t_0+1}^{t_0+l} \sum_{i=1}^{k} \mathbf{E}(B_i(t)) \end{aligned}$$

By Lemma 2, we know $\mathbf{E}(B_i(t)) = -c_i(t-1, t)$, thus

$$\begin{aligned} &\mathbf{E}(\sum_{t=t_0+1}^{t_0+l} B(t)) = \sum_{t=t_0+1}^{t_0+l} \sum_{i=1}^{k} \mathbf{E}(B_i(t)) \\ = &\sum_{t=t_0+1}^{t_0+l} \sum_{i=1}^{k} -c_i(t-1, t) = -\sum_{i=1}^{k} \sum_{t=t_0+1}^{t_0+l} c_i(t-1, t) \\ = &-\sum_{i=1}^{k} \sum_{j=t_0}^{t_0+l-1} c_i(j, j+1) = -\sum_{i=1}^{k} C(i) \end{aligned}$$

Therefore, the negated expected total benefit by keeping k tuples from time t_0 to t_0+l is the cost along all of the k paths in flow graph G, which is the cost of a feasible integral flow of size k through G. □

By Lemma 3, 4 we know each flow path p_i in the solution is integral and disjointed, which does not share any arc. And from Theorem 2, the solution of min-cost flow problem yields the maximized expected total benefit, which leads to following Corollary.

Corollary 6 *Suppose the k paths $\{p_1, p_2, \ldots, p_k\}$ are the optimal solution of the min-cost max flow problem over graph flow graph G, and path $p_i = \{se, a_{i1}, a_{i2}, \ldots, a_{in}, sk\}$ for $i = 1, 2, \ldots, k$. then keeping the tuple set $\{a_{i1}\}, i = 1, 2, \ldots, k$ yields the maximized expected total benefit from time t_0 to t_0+l.*

We make the correctness proof of model in the scenario of one binary join query over two probabilistic streams. However, we can extend the proof to the general scenario in which multiple binary join queries over multiple probabilistic streams. The only difference in the proof lies in computation of expected benefit of the horizontal arc. In single binary join case, this benefit only depends on its partner stream while in the case of multiple binary joins, this expected benefit is a summary of each expected benefit of the binary join with one parter stream. We omit the proof due to space limitation.

D Complexity of FLOWEXPECT

At time t_0, there $k+2$ nodes, and at each time point afterwards, the two incoming tuples from two streams will add two nodes for replacement tuples and one connecting node. All nodes in the last time point t_0-1 are kept to t_0. Thus the total number of nodes in graph including $source$ and $sink$ is

$$\begin{aligned} & 2+\sum_{i=0}^{l}(k+2+3i) \\ = \;& 2+l(k+2)+\frac{3}{2}l(l+1) \\ = \;& \frac{3}{2}l^2+l(k+\frac{7}{2})+2 \\ = \;& O((k+l)l) \end{aligned}$$

At time $t_i=t_0+i, i=1,2,\ldots,l$, all nodes with time stamp t_i have $k+2+2(i-1)$ incoming arcs, and there are $k+2+2i+2$ non-horizontal arcs within these nodes at time t_i, and there are $k+2$ outgoing arc of $source$ and $k+2+2l$ incoming arcs of $sink$, thus the total number of arcs is

$$\begin{aligned} & \sum_{i=1}^{l}[k+2+2(i-1)+(k+2+2(i-1)+2)]+(k+2)+(k+2+2l) \\ = \;& 2l^2+2(2k+4)l+2k+4 \\ = \;& O(l^2+kl+k) \\ = \;& O((k+l)l) \end{aligned}$$

Therefore the the complexity bound of algorithm in Goldberg[9] is translated into

$$O(n^2 m\log n)=O((k+l)^3 l^3\log((k+l)l))$$

E Proof of Lemma 1

Proof: Assume tuple x is from stream S and let $b_x(t)$ be the number of result tuples generated by joining x with tuples from its partner stream R at time $t>t_0$. At any time t, if x matches the incoming S tuple, then one join result result is generated, otherwise no result tuple is generated by joining x with the incoming

tuple. Thus,

$$
\begin{aligned}
& B_x(\Delta t) = \sum_{t=t_0+1}^{t_0+\Delta t} \mathbf{E}(b_x(t)) \\
= & \sum_{t=t_0+1}^{t_0+\Delta t} \Big\{ 1 \times \Pr\{X_t^R = v_x | \bar{x}_{t_0}\} + 0 \times (1 - \Pr\{X_t^R = v_x | \bar{x}_{t_0}\}) \Big\} \\
= & \sum_{t=t_0+1}^{t_0+\Delta t} \Pr\{X_t^R = v_x | \bar{x}_{t_0}\}
\end{aligned}
$$

□

F Proof of Corollary 1

Proof: Similarly as above, let $b_x(t)$ be the number of result tuples generated by joining x with tuples from its partner stream R at time $t > t_0$. And tuple x is from *transformed stream* S. From section 2 each *transformed stream* S tuple can only join with one reference stream R tuple, equivalently speaking, if tuple x joins with R tuple at t, x is unable to join with any R tuple from t_0 up to t which indicates there is no reference R tuple arrived at t_0 through $t-1$. Thus,

$$
\begin{aligned}
& \mathbf{E}(b_x(t)) = \Pr\{x \text{ is referenced at } t\} \\
= & \begin{cases} \Pr\{X_t^R = x | \bar{x}_{t_0}\}, & t = t_0 + 1; \\ \Pr\{\{\bigcap_{t=t_0+1}^{t-1} X_t^R \neq v_x\} \cap X_t^R = x | \bar{x}_{t_0}\}, & t > t_0 + 1. \end{cases}
\end{aligned}
$$

Therefore,

$$
\begin{aligned}
& B_x(\Delta t) = \sum_{t=t_0+1}^{t_0+\Delta t} \mathbf{E}(b_x(t)) \\
= & \Pr\{X_{t_0+1}^R = x | \bar{x}_{t_0}\} + \sum_{t=t_0+2}^{t_0+\Delta t} \Big\{ \Pr\{\{ \bigcap_{t=t_0+1}^{t-1} X_t^R \neq v_x\} \cap X_t^R = x | \bar{x}_{t_0}\} \Big\} \\
= & \Pr\{x \text{ is referenced during } [t_0+1, t+\Delta t] \} \\
= & 1 - \Pr\{x \text{ is not referenced during } [t_0+1, t+\Delta t] \} \\
= & 1 - \Pr\{ \bigcap_{t=t_0+1}^{t_0+\Delta t} X_t^R \neq v_x \mid \bar{x}_{t_0}\}
\end{aligned}
$$

Obviously any tuple y in reference stream will never be used in future, thus,

$$B_y(\Delta t) = 0$$

□

G Proof of Theorem 3

G.1 Proof of (1)

Proof: Assume there does not exist any optimal algorithm that keeps x or discards y at the current time t_0, that is, all optimal algorithms discard x and keep y at t_0. Suppose A is an optimal algorithm, then at t_0, A discards x and keeps y up to time $t' > t_0$, when y is replaced by A. Suppose a non-optimal algorithm A' which merely differs from A by replacing y instead of x at t_0 and keeping x up to time t', when x is replaced by A' as A does. Otherwise, A' makes the exactly same decision as A. Obviously, the expected expected benefit produced by A' will never be less than A since B_x dominates B_y for all the time. Thus A' is also optimal since A is optimal. This is contradicted with the assumption A' is non-optimal. Therefore, there exists an an optimal algorithm that keeps x or discards y. □

G.2 Proof of (2)

Proof: Suppose an algorithm A that 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. Therefore, all optimal algorithms must keep x or discard y at time t_0 □

H Proof of Corollary 2

Proof: Assume there does not exist an optimal algorithm that discards C'. Suppose A is an optimal algorithm, and thus $\exists x \in C'$ and $\exists y \notin C'$ such that A keeps x up to t' while discarding y at current time t_0. Construct another algorithm A' which merely differs from A by keeping y up to t' and discarding x at t_0, and A' behaves exactly the same as A otherwise. Since B_y dominates B_x, expected benefit generated by A' will never be less than A. In consequence, A' is also optimal since A is optimal. And A' discards C', which is contradicted with the assumption of thee is no optimal algorithm that discards C'. Therefore, there exists an optimal algorithm that discards C'. □

I Proof of Property 3 in Choosing L_x

Proof: It is easy to see that $\Pr\{X^R_{t_0+\Delta t} = v_x \mid \bar{x}_{t_0}\} \leq 1$, thus

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

Since the series $\sum_{\Delta t=1}^{\infty} L_x(\Delta t)$ converges, $H_x \leq \sum_{\Delta t=1}^{\infty} L_x(\Delta t)$ also converges. □

J Proof of Theorem 4

Proof: Since $H_x = \lim_{\Delta t \to \infty} G_x(\Delta t)$ where,

$$\begin{aligned} G_x(\Delta t) &= B_x(1)L_x(1) + \sum_{t=2}^{\Delta t}((B_x(t) - B_x(t-1))L_x(t)) \\ &= \sum_{t=1}^{\Delta t-1} B_x(t)(L_x(t) - L_x(t+1)) + B_x(\Delta t)L_x(\Delta t) \end{aligned}$$

From property 1 and 2, $L_x(t) \geq 0$ and $L_x(t) - L_x(t+1) \geq 0$, and since B_x dominates B_y,

$$\begin{aligned} G_x(\Delta t) &= \sum_{t=1}^{\Delta t-1} B_x(t)(L_x(t) - L_x(t+1)) + B_x(\Delta t)L_x(\Delta t) \\ &\geq \sum_{t=1}^{\Delta t-1} B_y(t)(L_x(t) - L_x(t+1)) + B_y(\Delta t)L_x(\Delta t) \\ &= B_y(1)L_x(1) + \sum_{t=2}^{\Delta t}((B_y(t) - B_y(t-1))L_x(t)) \end{aligned}$$

From property 4, $L_x(\Delta t) \geq L_y(\Delta t)$, thus,

$$\begin{aligned} G_x(\Delta t) &= B_y(1)L_x(1) + \sum_{t=2}^{\Delta t}((B_y(t) - B_y(t-1))L_x(t)) \\ &\geq B_y(1)L_y(1) + \sum_{t=2}^{\Delta t}((B_y(t) - B_y(t-1))L_y(t)) \\ &= G_y(\Delta t) \end{aligned}$$

From property 3 and proof I, H_x and H_y converge. Therefore,

$$H_x = \lim_{\Delta t \to \infty} G_x(\Delta t) \geq \lim_{\Delta t \to \infty} G_y(\Delta t) = H_y$$

It is easy to see that if B_x strongly dominates B_y, strict inequality holds,

$$H_x > H_y$$

. □

K Proof of Corollary 3

Proof: Since $L_x = L^{exp}(\Delta t) = e^{-\Delta t/\alpha}, \alpha > 0$, and the streams are governed by independent stochastic processes, then for joining problem,

$$H^{exp}_{x,t_0} = \sum_{\Delta t=1}^{\infty}(\Pr\{X^R_{t_0+\Delta t} = v_x\} \cdot L_x(\Delta t)) = \sum_{\Delta t=1}^{\infty}(\Pr\{X^R_{t_0+\Delta t} = v_x\} \cdot e^{-\Delta t/\alpha})$$

Similarly,

$$\begin{aligned} H^{exp}_{x,t_0-1} &= \sum_{\Delta t=1}^{\infty}(\Pr\{X^R_{t_0-1+\Delta t}=v_x\}\cdot L_x(\Delta t)) = \sum_{\Delta t=1}^{\infty}(\Pr\{X^R_{t_0-1+\Delta t}=v_x\}\cdot e^{-\Delta t/\alpha}) \\ &= e^{-1/\alpha}\sum_{\Delta t=1}^{\infty}(\Pr\{X^R_{t_0+\Delta t-1}=v_x\}\cdot e^{-(\Delta t-1)/\alpha}) \\ &= e^{-1/\alpha}\Bigg\{\sum_{\Delta t=2}^{\infty}(\Pr\{X^R_{t_0+\Delta t-1}=v_x\}\cdot e^{-(\Delta t-1)/\alpha}) + \Pr\{X^R_{t_0}=v_x\}\Bigg\} \end{aligned}$$

Let $\Delta t' = \Delta t - 1$, then

$$\begin{aligned} H^{exp}_{x,t_0-1} &= e^{-1/\alpha}\Bigg\{\sum_{\Delta t'=1}^{\infty}(\Pr\{X^R_{t_0+\Delta t'}=v_x\}\cdot e^{-\Delta t'/\alpha}) + \Pr\{X^R_{t_0}=v_x\}\Bigg\} \\ &= e^{-1/\alpha}\Bigg\{H^{exp}_{x,t_0} + \Pr\{X^R_{t_0}=v_x\}\Bigg\} \end{aligned}$$

Therefore,

$$H^{exp}_{x,t_0} = e^{1/\alpha}H^{exp}_{x,t_0-1} - \Pr\{X^R_{t_0}=v_x\}.$$

□

L Proof of Corollary 4

Proof: Since $L_x = L^{exp}(\Delta t) = e^{-\Delta t/\alpha}, \alpha > 0$, and the streams are governed by independent stochastic processes, then for caching problem,

$$\begin{aligned} H^{exp}_{x,t_0} &= \sum_{\Delta t=1}^{\infty}(\Pr\{(X^R_{t_0+\Delta t}=v_x)\cap(\bigcap_{t_0<t<t_0+\Delta t}X^R_t\neq v_x)\mid \bar{x}_{t_0}\}\cdot L_x(\Delta t)) \\ &= \sum_{\Delta t=1}^{\infty}\Bigg\{\Pr\{X^R_{t_0+\Delta t}=v_x\}\prod_{t'=1}^{\Delta t-1}(1-\Pr\{X^R_{t_0+t'}=v_x\})\cdot e^{-\Delta t/\alpha}\Bigg\} \end{aligned}$$

Similarly,

$$\begin{aligned} H^{exp}_{x,t_0-1} &= \sum_{\Delta t=1}^{\infty}\Bigg\{\Pr\{X^R_{t_0-1+\Delta t}=v_x\}\prod_{t'=1}^{\Delta t-1}(1-\Pr\{X^R_{t_0-1+t'}=v_x\})\cdot e^{-\Delta t/\alpha}\Bigg\} \\ = e^{-1/\alpha}&\sum_{\Delta t=1}^{\infty}\Bigg\{\Pr\{X^R_{t_0-1+\Delta t}=v_x\}\prod_{t'=1}^{\Delta t-1}(1-\Pr\{X^R_{t_0-1+t'}=v_x\})\cdot e^{-(\Delta t-1)/\alpha}\Bigg\} \\ = e^{-1/\alpha}&\Bigg\{\Pr\{X^R_{t_0}=v_x\}+\sum_{\Delta t=2}^{\infty}\Bigg\{\Pr\{X^R_{t_0-1+\Delta t}=v_x\}\prod_{t'=1}^{\Delta t-1}(1-\Pr\{X^R_{t_0-1+t'}=v_x\})\cdot e^{-(\Delta t-1)/\alpha}\Bigg\}\Bigg\} \end{aligned}$$

Let $\Delta t' = \Delta t - 1$, then

$$\begin{aligned} H^{exp}_{x,t_0-1} &= e^{-1/\alpha}\left\{ \Pr\{X^R_{t_0} = v_x\} + \sum_{\Delta t'=1}^{\infty} \left\{ \Pr\{(X^R_{t_0+\Delta t'} = v_x)\} \prod_{t'=1}^{\Delta t'} (1 - \Pr\{(X^R_{t_0-1+t'} = v_x)) \cdot e^{-(\Delta t')/\alpha}) \right\} \right\} \\ &= e^{-1/\alpha}\left\{ \Pr\{X^R_{t_0} = v_x\} + \sum_{\Delta t'=1}^{\infty} \left\{ \Pr\{X^R_{t_0+\Delta t'} = v_x\} \prod_{t''=0}^{\Delta t'-1} (1 - \Pr\{X^R_{t_0+t''} = v_x\}) \cdot e^{-\Delta t'/\alpha} \right\} \right\} \end{aligned}$$

$\Rightarrow$

$$\begin{aligned} & e^{1/\alpha} H^{exp}_{x,t_0-1} - \Pr\{X^R_{t_0} = v_x\} = \sum_{\Delta t'=1}^{\infty} \left\{ \Pr\{X^R_{t_0+\Delta t'} = v_x\} \prod_{t''=0}^{\Delta t'-1} (1 - \Pr\{X^R_{t_0+t''} = v_x\}) \cdot e^{-\Delta t'/\alpha} \right\} \\ = \;& (1 - \Pr\{X^R_{t_0} = v_x\}) \sum_{\Delta t'=1}^{\infty} \left\{ \Pr\{X^R_{t_0+\Delta t'} = v_x\} \prod_{t''=1}^{\Delta t'-1} (1 - \Pr\{X^R_{t_0+t''} = v_x\}) \cdot e^{-\Delta t'/\alpha} \right\} \\ = \;& (1 - \Pr\{X^R_{t_0} = v_x\}) H^{exp}_{x,t_0} \end{aligned}$$

Therefore,

$$H^{exp}_{x,t_0} = \frac{e^{1/\alpha} H^{exp}_{x,t_0-1} - \Pr\{X^R_{t_0} = v_x\}}{1 - \Pr\{X^R_{t_0} = v_x\}}$$

□

M Proof of Corollary 5

Proof: Since $X^R_t = at + b + Y^R_t$ and Y^R_t's are iid and have zero mean,

$$\begin{aligned} \Pr\{X^R_t = v\} &= \Pr\{Y^R_t = v - (at + b)\} = \Pr\{Y^R_t = v + (t' - t)a - (at' + b)\} \\ &= \Pr\{Y^R_{t'} = v + (t' - t)a - (at' + b)\} \\ &= \Pr\{X^R_{t'} = v + (t' - t)a\} \\ &= \Pr\{X^R_{t+(t'-t)} = v + (t' - t)a\} \end{aligned}$$

where $v' = v + (t' - t)a$. Thus for joining problem,

$$\begin{aligned} B_{v,t}(\Delta t) &= \sum_{k=t+1}^{t+\Delta t} \Pr\{X^R_k = v\} \\ &= \sum_{k=t+1}^{t+\Delta t} \Pr\{X^R_{k+(t'-t)} = v + a(t' - t)\} \\ &= \sum_{q=t'+1}^{t'+\Delta t} \Pr\{X^R_q = v + a(t' - t)\} \\ &= B_{v+a(t'-t),t'}(\Delta t) \end{aligned}$$

Similarly, for caching problem,

$$\begin{aligned} B_{v,t}(\Delta t) &= 1 - \prod_{q=t+1}^{t+\Delta t-1} (1 - \Pr\{X_q^R = v\}) \\ &= 1 - \prod_{q=t+1}^{t+\Delta t-1} (1 - \Pr\{X_{q+(t'-t)}^R = v + a(t'-t)\}) \\ &= 1 - \prod_{r=t'+1}^{t'+\Delta t-1} (1 - \Pr\{X_r^R = v + a(t'-t)\}) \\ &= B_{v+a(t'-t),t'}(\Delta t) \end{aligned}$$

□

N Proof of Theorem 5

N.1 Proof of (1)

Proof:

$$\begin{aligned} & X_{t_0+\Delta t}^R \\ =\ & \phi_0 + \phi_1 X_{t_0+\Delta t-1}^R + Y_{t_0+\Delta t}^R \\ =\ & \cdots \\ =\ & x_{t_0}\phi_1^{\Delta t} + \phi_0 \sum_{i=0}^{\Delta t-1} \phi_1^i + \sum_{i=1}^{\Delta t} \phi_1^{\Delta t-i} Y_{t_0+i}^R. \end{aligned}$$

Therefore,

$$\begin{aligned} & \Pr\{X_{t_0+\Delta t}^R = v_x \mid \bar{x}_{t_0}\} \\ =\ & \Pr\{\sum_{i=1}^{\Delta t} \phi_1^{\Delta t-i} Y_{t_0+i}^R = v_x - x_{t_0}\phi_1^{\Delta t} - \phi_0 \sum_{i=0}^{\Delta t-1} \phi_1^i \mid \bar{x}_{t_0}\} \\ =\ & \Pr\{\sum_{i=1}^{\Delta t} \phi_1^{\Delta t-i} Y_i^R = v_x - x_{t_0}\phi_1^{\Delta t} - \phi_0 \sum_{i=0}^{\Delta t-1} \phi_1^i\}. \end{aligned}$$

The last step is due to the fact that Y_t^R's are i.i.d. Clearly, the above probability is independent of t_0 and past history, and thus can be expressed by a function $y(\Delta t, v_x, x_{t_0})$. If $L_x(\Delta t)$ is also a time-independent function $L(\Delta t, v_x, x_{t_0})$, then:

$$\begin{aligned} H_x &= \sum_{\Delta t=1}^{\infty} (\Pr\{X_{t_0+\Delta t}^R = v_x \mid \bar{x}_{t_0}\} \cdot L_x(\Delta t)) \\ &= \sum_{\Delta t=1}^{\infty} (y(\Delta t, v_x, x_{t_0}) \cdot L(\Delta t, v_x, x_{t_0})). \end{aligned}$$

The result is simply a function of v_x and x_{t_0} because Δt disappears after the summation.

□

N.2 Proof of (2)

Proof: If $\phi_1 = 1$, starting with the derivation of $\Pr\{X^R_{t_0+\Delta t} = v_x \mid \bar{x}_{t_0}\}$ above, we have:

$$\begin{aligned} & \Pr\{X^R_{t_0+\Delta t} = v_x \mid \bar{x}_{t_0}\} \\ = \; & \Pr\{\sum_{i=1}^{\Delta t} \phi_1^{\Delta t - i} Y_i^R = v_x - x_{t_0}\phi_1^{\Delta t} - \phi_0 \sum_{i=0}^{\Delta t-1} \phi_1^i\} \\ = \; & \Pr\{\sum_{i=1}^{\Delta t} Y_i^R = v_x - x_{t_0} - \Delta t \phi_0\} \end{aligned}$$

Clearly, the above probability can be expressed by a function $y(\Delta t, v_x - x_{t_0})$. If $L_x(\Delta t)$ is also a time-independent function $L(\Delta t, v_x - x_{t_0})$, then:

$$H_x = \sum_{\Delta t=1}^{\infty} (y(\Delta t, v_x - x_{t_0}) \cdot L(\Delta t, v_x - x_{t_0})),$$

which is simply a function of $v_x - x_{t_0}$.

□

O ECB of Joining Tuples in 5.3

- Category R1: x is from R and $v_x \in (-\infty, t_0 - w_S]$.

$$B_x(\Delta t) = 0$$

.

- Category R2: x is from R and $v_x \in (t_0 - w_S, t_0 + w_R]$.

$$B_x(\Delta t) = \begin{cases} \frac{\Delta t}{2w_S+1}, & \text{when } \Delta t \in [1, v_x - (t_0 - w_S)]; \\ \frac{v_x - (t_0 - w_S)}{2w_S+1}, & \text{afterwards.} \end{cases}$$

- Category S1: x is from S and $v_x \in (-\infty, t_0 - w_R]$.

$$B_x(\Delta t) = 0$$

.

- Category S2: x is from S and $v_x \in (t_0 - w_R, t_0 + w_R + 1]$.

$$B_x(\Delta t) = \begin{cases} \frac{\Delta t}{2w_R+1}, & \text{when } \Delta t \in [1, v_x - (t_0 - w_R)]; \\ \frac{v_x - (t_0 - w_R)}{2w_R+1}, & \text{afterwards.} \end{cases}$$

- Category S3: x is from S and $v_x \in (t_0 + w_R + 1, t_0 + w_S]$. Thus,

$$B_x(\Delta t) = \begin{cases} 0 & \text{if } \Delta t \in [1, v_x - (t_0 + w_R)); \\ 1 & \text{if } \Delta t \in (v_x - (t_0 - w_R), \infty); \\ \frac{\Delta t - (v_x - (t_0 + w_R)) + 1}{2w_R+1} & \text{otherwise.} \end{cases}$$

P Detailed Analysis of Tuple Dominance in 5.4

The procedure to derive ECB for tuples and conduct the dominance test among tuples is similar to the above case. Here we construct one example of tuple dominance and the other of incomparable tuples for joining problem. Caching problem is similar.

- In Figure **??**, for any two tuples x and y from R, B_x strongly dominates B_y if v_y lies to the left of $f^S(t)$ and is farther away from $f^S(t)$ than v_x.

  Proof:The ECB for tuple x is

  $$B_x(\Delta t) = \sum_{t=t_0+1}^{t_0+\Delta t} \Pr\{X_t^S = v_x \mid \bar{x}_{t_0}\}$$

  Since $f^S(t)$ is monotonically increasing(moving rightward in the figure) and $v_y < v_x$, if at time t_0+1, $\Pr\{X_{t_0+1}^S = v_y \mid \bar{x}_{t_0}\} < \Pr\{X_{t_0+1}^S = v_x \mid \bar{x}_{t_0}\}$, thus at any time $t \geq t_0 + 1$

  $$\Pr\{X_t^S = v_y \mid \bar{x}_{t_0}\} < \Pr\{X_t^S = v_x \mid \bar{x}_{t_0}\}$$

  Therefore,

  $$B_y(\Delta t) = \sum_{t=t_0+1}^{t_0+\Delta t} \Pr\{X_t^S = v_y \mid \bar{x}_{t_0}\} < B_x(\Delta t) = \sum_{t=t_0+1}^{t_0+\Delta t} \Pr\{X_t^S = v_x \mid \bar{x}_{t_0}\}$$

  □

- For tuple x and z, it is easy to see that in "recent future", $\Pr\{X_t^S = v_z \mid \bar{x}_{t_0}\} > \Pr\{X_t^S = v_x \mid \bar{x}_{t_0}\}$ since x is closer to $f^S(t)$ when they are both on the right side of $f^S(t)$. However, when $f^S(t) > z$, that is, both are on the left side of $f^S(t)$, $\Pr\{X_t^S = v_z \mid \bar{x}_{t_0}\} < \Pr\{X_t^S = v_x \mid \bar{x}_{t_0}\}$. Thus there does not exist a consistent dominance relation between x and z for all $t > t_0$, in other words, x and z are not comparable.

Q Detailed Analysis of Tuple Dominance in 5.5

- Non-zero constant drift
  Without generality, we assume drift $\phi_0 > 0$. It is easy to see that

  $$X_t^R = x_{t_0} + \phi_0(t - t_0) + \sum_{t'=t_0+1}^{t} Y_{t'}^R$$

  Assume i.i.d. $Y_{t'}^R \sim N(0, \sigma^2)$ then

  $$X_t^R = x_{t_0} + \phi_0(t - t_0) + \sum_{t'=t_0+1}^{t} Y_{t'}^R \sim N(x_{t_0} + \phi_0(t - t_0), (t - t_0)\sigma^2)$$

For two S tuples s_1 and s_2, such that, at time t_1, $x_{t_0} + \phi_0(t_1 - t_0) < s_1 < s_2$ however at time $t_2 > t_1$, $s_1 < s_2 < x_{t_0} + \phi_0(t_2 - t_0)$, it is easy to see that s_1 is closer to the mean at t_1 and s_2 is closer to the mean at t_2, thus at t_1

$$\Pr\{X_{t_1}^R = v_{s_2} \mid \bar{x}_{t_0}\} < \Pr\{X_{t_1}^R = v_{s_1} \mid \bar{x}_{t_0}\}$$

and t_2

$$\Pr\{X_{t_1}^R = v_{s_2} \mid \bar{x}_{t_0}\} > \Pr\{X_{t_1}^R = v_{s_1} \mid \bar{x}_{t_0}\}$$

Therefore, $B_{s_1}(t)$ dominates $B_{s_2}(t)$ if $t_0 < t \leq t_1$ but the dominance may break when $t > t_1$.

- Zero-drift
  From above, if drift $\phi_0 = 0$, then,

$$X_t^R = x_{t_0} + \sum_{t'=t_0+1}^{t} Y_{t'}^R \sim N(x_{t_0}, (t - t_0)\sigma^2)$$

  Mean of X_t^R is x_{t_0} and does not change over time. Thus, for two tuples s_1 and s_2 such that $|s_1 - x_{t_0}| \leq |s_2 - x_{t_0}|$,

$$\Pr\{X_t^R = v_{s_1} \mid \bar{x}_{t_0}\} > \Pr\{X_t^R = v_{s_2} \mid \bar{x}_{t_0}\}, \forall t > t_0$$

  which leads to $B_{s_1}(t)$ dominates $B_{s_2}(t)$ for all $t > t_0$. Therefore, all candidate tuples can be ranked by their distance from the current position of the random walk.

Note that above derivation can be extended directly to caching case and we omit it for brevity.