Random Sampling for Continuous Streams with Arbitrary Updates

Yufei Tao, Xiang Lian, Dimitris Papadias, and Marios Hadjieleftheriou

Abstract—The existing random sampling methods have at least one of the following disadvantages: they 1) are applicable only to certain update patterns, 2) entail large space overhead, or 3) incur prohibitive maintenance cost. These drawbacks prevent their effective application in stream environments (where a relation is updated by a large volume of insertions and deletions that may arrive in any order), despite the considerable success of random sampling in conventional databases. Motivated by this, we develop several fully dynamic algorithms for obtaining random samples from individual relations, and from the join result of two tables. Our solutions can handle any update pattern with small space and computational overhead. We also present an in-depth analysis that provides valuable insight into the characteristics of alternative sampling strategies and leads to precision guarantees. Extensive experiments validate our theoretical findings and demonstrate the efficiency of our techniques in practice.

Index Terms—Sampling, selectivity estimation.

1 Introduction

A *general stream relation* receives a large number of updates per time unit, which include tuple insertions and deletions that may arrive in any order. Specifically, each *I-command* has the form $\{\mathrm{I}, id, S_A\}$, where the first field is a tag indicating "insertion," the second denotes the id of the tuple being inserted, and S_A corresponds to the set of the remaining attributes of the tuple. Similarly, a *D-command* $\{\mathrm{D}, id\}$ removes the tuple with a specified id. The content of a relation includes all the tuples that were inserted, but have not been removed; the *cardinality* of the relation is the number of such tuples.

We consider that all tuples currently in a relation have distinct ids. Specifically, the id in each I-command should not be identical to the id of any existing tuple in the relation. However, provided that the previous tuple with an id has been deleted, a tuple with the same id can be inserted. In other words, multiple tuples with the same id may be inserted throughout the history, but at any moment, there can be only one tuple with this id.

We address two fundamental problems of approximate processing. Given a single relation T, the first one aims at providing accurate answers to queries of the form:

$$\text{Q1}: \text{ SELECT COUNT}(^*) \text{ FROM } T \text{ WHERE } \theta_{any}.$$

Q1 is a counting query with *arbitrary* conditions θ_{any} in the WHERE clause. We assume *no a priori knowledge about* θ_{any}, which excludes potential solutions [16], [17] that rely on special counting "sketches" (including histograms, wavelets, etc.) for certain attributes (or their combinations). Specifically, although these methods can produce accurate results on the preprocessed columns, they are not useful for general ad hoc queries involving other predicates. *Random-sampling techniques*, on the other hand, constitute a natural methodology for this problem because, by keeping all attributes of the sampled tuples, it is possible to support any predicate θ_{any} with good accuracy guarantees.

The second problem tackled in this paper is to accurately predict the join size of two stream relations T_a and T_b:

$$\begin{aligned}\text{Q2}: \text{ SELECT COUNT}(^*) \text{ FROM } T_a,\ T_b \\ \text{WHERE } \theta_{all} \text{ and } \theta_{any}.\end{aligned}$$

θ_{all} includes a set of *registered* conditions common in all Q2 queries, which differ in their own formulation of θ_{any} (an arbitrary predicate). As an example, assume that T_a has attributes (id, A_a), T_b has (id, A_b), and θ_{all} is $T_a.id = T_b.id$. Then, every possible Q2 query contains this equi-join condition, but can also include a predicate θ_{any} on individual columns of a single table (e.g., $T_a.A_a > 10$), or both tables (e.g., $T_a.A_a + T_b.A_b > 100$). Equivalently, the set of records counted in a Q2 query is a subset of the results of joining T_a and T_b using only the condition $T_a.id = T_b.id$. Our goal is to process Q1 and Q2 accurately using at most M random samples, where M is (by far) lower than the size of the database.

Query types Q1 and Q2 are important to a large number of applications. For example, consider a relation with schema $PR(\textit{stock-id}, \textit{price})$, where each tuple records the current price of a stock. Whenever a stock's price changes, a D-command streams in, removing the obsolete tuple of the stock; then, an I-command immediately follows, inserting a new tuple carrying the updated price. Obviously, unlike the sliding-window stream model [5], the order that the tuples in PR are inserted is most likely not equivalent to the order

- *Y. Tao is with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Sha Tin, New Territories, Hong Kong. E-mail: taoyf@cse.cuhk.edu.hk.*
- *X. Lian and D. Papadias are with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong. E-mail: {xlian, dimitris}@cse.ust.hk.*
- *M. Hadjieleftheriou is with AT&T Labs, 180 Park Avenue, Building 103, Florham Park, NJ 07932. E-mail: marioh@research.att.com.*

Manuscript received 14 May 2005; revised 12 Jan. 2006; accepted 3 Aug. 2006; published online 20 Nov. 2006.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0184-0505.

that they are deleted. A Q1 query in this context may retrieve the number of stocks whose prices qualify a certain predicate.

To motivate Q2 queries, assume that we have another stream relation $TO(\mathit{stock\text{-}id}, \mathit{turnover})$, which captures the current turnovers of the stocks. As with PR, TO is updated with continuously arriving I and D-commands, generated by the buying/selling of any stock. Note that the streams corresponding to PR and TO are typically separate, because they are usually generated by different agents. In this case, to study the statistic relationship between prices and turnovers, a user needs to join PR and TO together with an equality condition on *stock-id*, and then apply other predicates on *price* and *turnover*. Such operations form a Q2 query, where θ_{all} is the equality condition and θ_{any} corresponds to the other predicates.

We are interested in solutions that incur small computational overhead for processing *each* incoming command, as opposed to methods (e.g., the *counting sample* reviewed in the next section) that have low amortized cost but poor worst-case performance for individual updates. For data streams with a high record arrival rate, spending considerable time on *any* tuple necessarily delays a large number of subsequent records, which need to be stored in a system buffer. When the size of this buffer is exceeded, some tuples must be discarded (i.e., load shedding [25]), in which case obtaining a truly random sample set is impossible. For instance, in the stock-trading application mentioned earlier, shedding an update command causes the price of a stock to be inaccurate. Furthermore, shedding a D-command may even lead to duplicate tuples for the same stock.

As elaborated in Section 2, the existing sampling algorithms have at least one of the following disadvantages: 1) they rely on certain assumptions on data updates (e.g., only insertions are allowed, or tuples must be deleted according to the insertion order), 2) they require considerable space (e.g., the underlying relations must be fully materialized for resampling), or 3) they incur prohibitive maintenance overhead (e.g., they must periodically scan the entire sample set). These problems prevent their effective deployment on data stream applications, despite the success of random sampling in conventional databases.

In this paper, we develop sampling algorithms that are *fully dynamic* (supporting any sequence of insertions and deletions), *efficient* (processing each update with very low space and computational overhead), and *accurate* (producing approximate answers for Q1 and Q2 queries with small errors). Specifically, for single relations, our methods significantly improve the well-known *reservoir sampling* and *counting sample* approaches in the presence of intensive updates. For join results, we propose the first sampling algorithm that enables effective approximate processing on streams containing arbitrary update requests (the previous solutions [24] discuss only the special case of sliding windows). In addition, we present an in-depth analysis that provides valuable insight into the characteristics of alternative solutions. Extensive experiments validate our theoretical findings and confirm the efficiency of the proposed techniques in practice.

The rest of the paper is organized as follows: Section 2 reviews previous work that is directly related to ours. Sections 3 and 4 present algorithms for sampling single relations, and analyze their effectiveness on Q1 queries. Section 5 focuses on sampling join results for approximate Q2 processing. Section 6 contains the experimental results, and Section 7 concludes the paper with directions for future work.

2 Related Work

Section 2.1 first reviews approaches for sampling a single relation in the presence of updates, and clarifies their problems. Then, Section 2.2 discusses algorithms for sampling the join result of multiple tables.

2.1 Sampling a Single Relation

Sampling from a static table with n tuples is straightforward. Specifically, a sequence number can be assigned to each record (i.e., the first tuple has number 1, the second 2, and so on). To obtain s samples, we only need to randomly generate s distinct values in the range $[1, n]$, and select the tuples with these sequence numbers. It is more difficult, however, to maintain the randomness of the sample set when new records are inserted into the table, or existing ones are removed. A naive solution would resample the relation as described above whenever its content changes. Obviously, this method is impractical since a resampling process may require accessing a large number of disk pages. *Reservoir sampling* and *counting sample* aim at solving these problems.

2.1.1 Reservoir Sampling

The first *reservoir* algorithms [26], [22] in the database context maintain an array RS with maximum size s (the target sample size), which is initially empty. The first s tuples are directly added into RS, after which the array becomes full. For each subsequent insertion, a random integer x is generated in the range $[1, n]$, where n is the number of insertions handled so far (n continuously grows with time). If x is larger than s, the incoming tuple is ignored; otherwise, $(x \leq s)$, it is recorded at the $x\text{th}$ position of RS, replacing the sample that was originally there. It can be shown [26] that, at any time, the tuples in RS constitute a random sample set of the current content of the data set T. Jermaine et al. [21] present an alternative *reservoir* technique to manage sample sets whose sizes exceed the capacity of the available memory. Similar to [26], [22], this approach supports only insertions.

Gibbons et al. [14] propose an extension, referred to as *delete-at-will* in the sequel, for producing a random sample set in the presence of deletions. Let s be the current size of the sample set RS and n the cardinality of T. If the tuple t being deleted does not belong to RS, the sample set remains unchanged. If t is found in RS, it is removed, after which the size of RS becomes $s - 1$. In either case (whether t appears in RS or not), the cardinality of the relation T changes to $n - 1$ to reflect the removal of t. The handling of insertions is similar to the *reservoir* technique. Assume, for instance, that after a sample is deleted (i.e., the sample size is $s - 1$ and the data cardinality is $n - 1$) a new record arrives. A random integer x is generated in the range $[1, n-1]$ and if x is larger than $s-1$, the incoming tuple is ignored; otherwise, $(x \leq s-1)$, it is recorded at the xth position of RS, replacing the sample that was originally there. Gibbons et al. [14] prove that the resulting RS is still random with respect to the remaining tuples in T.

The problem with *delete-at-will* is that the sample set gradually shrinks with time, and eventually ceases to be useful (e.g., it can no longer provide accurate selectivity estimation). Therefore, T must be scanned so that a sufficient number of samples are retrieved again. This requires retaining all the tuples that have been inserted but not yet deleted, which is impossible in stream environments. In Section 3, we propose R^*, an improved *reservoir* algorithm that supports deletions without decreasing the sample size.

2.1.2 Counting Sample

Counting sample (CS) [13] can produce a random sample set for any sequence of insertions and deletions without resampling the base relation, even in the existence of duplicate tuples (consequently, it is trivially applicable to conventional tables where each record is different). Specifically, CS maintains a list RS (with maximum size s) of elements in the form[1] $\{t, c\}$, where c is a counter that summarizes the number of identical records t in the sample set. Furthermore, a variable τ, initially set to 1, is used to control the probability that a record is sampled.

To handle an incoming tuple t, CS first probes the existing sample set to see if t has been included before. If yes, the counter c in the corresponding pair $\{t, c\}$ is increased by one, and the insertion terminates. Otherwise, (t is not in RS), CS tosses a coin c_1 with probability $1/\tau$ head, and discards t if c_1 tails. Alternatively (c_1 heads), the algorithm includes a new entry $\{t, 1\}$ in RS. If RS does not overflow (i.e., it contains no more than s elements), the insertion is completed. In case of an overflow, a *rejecting* pass is performed to expunge some existing samples from RS.

Specifically, at the beginning of the rejecting pass, a number τ' larger than the current τ (that governs the sampling probability, as mentioned earlier) is chosen. Then, for each element $\{t, c\}$ in RS, a coin c_2 is flipped with probability τ/τ' head. If c_2 tails, the rejecting algorithm moves on to the next element in RS. Otherwise (c_2 heads), it decreases the counter c by 1, and then repeatedly throws a coin c_3 with probability $1/\tau'$ head, until c_3 turns on tail. On each head occurrence of c_3, the counter c is further reduced by 1, until the element $\{t, c\}$ is eliminated when c equals 0. At the end of a rejecting pass (after processing all elements), the value of τ is updated to τ'. If the overflow of RS persists (i.e., no counter became 0 in the previous pass), another pass is executed. This process is repeated until the overflow is remedied.

Deleting a tuple t is much easier. The deletion is ignored if t is not in RS. Otherwise, the counter c in the corresponding pair $\{t, c\}$ is decreased, and the pair is removed from RS if c reaches 0. Note that, unlike *delete-at-will*, the removal of $\{t, c\}$ does not reduce the size of RS, that is, a future sample may still be stored at the original position of the removed entry. It is worth mentioning that the set of tuples in RS is not a random sample set of the current relation T. However, it can be converted into one by scanning the entire RS (see [13] for details).

A disadvantage of CS (that restricts the applicability of this method in practice) is that the entire RS must be scanned (sometimes repeatedly) for expunging existing sample(s) in case of overflows. When the size of RS is large, scanning RS may delay processing a large number of subsequently arriving records, some of which may need to be discarded from the system after the input buffer becomes full. In this case, it is simply impossible to obtain a truly random sample set. Furthermore, the selection of τ' in the rejecting pass is ad hoc. Gibbons et al. [13] suggest that τ' should be 10 percent higher than the current τ, without, however, providing justification on this choice. In Section 4, we present CS^*, an enhanced version of CS that avoids these problems.

Motivated by the shortcomings of *reservoir* and *counting sample*, Babcock et al. [5] propose alternative algorithms[2] for producing random samples in the specific context of "sliding window streams," where tuples are deleted according to the order of their arrival. Since these algorithms are inapplicable to arbitrary sequences of insertions/deletions, we do not discuss them further.

Finally, there exist numerous papers ([4], [7], [18], [20] to mention just some recent ones) on applications of sampling to various estimation tasks (e.g., selectivity estimation, clustering, etc.). The solutions in those papers do not compute random samples, and cannot be used to solve Q1 and Q2 queries formulated in Section 1. Recently, some sketch-based algorithms [9], [10], [11] have been developed to obtain a "probabilistic" random sample set, i.e., the samples may be random with a high probability, but there is no guarantee. We aim at deriving truly random samples in all cases.

2.2 Sampling the Join Results

Let T_1 and T_2 be two relations, and θ be a join predicate. *Naive stream join* (NSJ) is a straightforward method that maintains random sample sets $RS(T_1)$ and $RS(T_2)$ on T_1 and T_2, respectively (e.g., by using the techniques of the previous section). Then, given a Q2 query q, it finds the number n_q of tuple pairs $(t_1, t_2) \in RS(T_1) \times RS(T_2)$ that satisfy q, and estimates the query result as $(n_q \cdot |T_1| \cdot |T_2|) \,/\, (|RS(T_1)| \cdot |RS(T_2)|)$. This estimation, however, is usually not accurate, because the join between the sample sets of T_1 and T_2 typically leads to a very small subset of the actual join result. This phenomenon is caused by the fact [8] that the projection of the join result onto the columns of T_1 (T_2) is not a random sample set of T_1 (T_2).

1. CS adopts a slightly more complex element representation to minimize the space consumption. Since the basic idea is the same, we ignore this difference here for simplicity.

2. It is worth mentioning that the sliding-window algorithms in [5] have lower update cost than the proposed approaches. Specifically, those algorithms require $O(1)$ cost for each incoming tuple, while our approaches incur $O(\log M)$ time, where M is the memory size. This comparison, however, is not fair because the algorithms in [5] utilize the special properties of a sliding-window stream to reduce the update cost, and are not applicable in our problem settings.

```
Algorithm R*-init (RS)
1. $n_I = 0$;
2. for $i = 1$ to $M$
3.     $RS[i].valid$ = FALSE
Algorithm R*-insert (RS, t) /* t is an incoming tuple. RS is an array with maximum size M containing the random samples */
1. $n_I$++ //$n_I$ is the total number of insertions in history
2. $x$ = a random number in [1, $n_I$ ]
3. if $x \le M$
4.     if $RS[x].valid$ = TRUE
5.         remove $RS[x]$ from $I(RS)$ //replace an existing sample
6.     $RS[x] = t$; $RS[x].valid$ = TRUE
7.     insert $RS[x]$ into $I(RS)$
Algorithm R*-delete (RS, t) // t is the tuple to be deleted
1. if $t \in RS$ //this is checked using $I(RS)$
2.     let $x$ be the position number of $t$ in $RS$
3.     $RS[x].valid$ = FALSE
4.     remove $RS[x]$ from $I(RS)$
```

Fig. 1. Adapted reservoir sampling.

To illustrate this, we use an example similar to that in [8]. Consider that T_1 and T_2 have a single attribute A with the following tuples: $T_1 = \{1, 1, \ldots, 1, 2\}$, and $T_2 = \{2, 2, \ldots, 2, 1\}$. Most probably, a random sample $RS(T_1)$ (or $RS(T_2)$) on T_1 (or T_2) will include only records with $A = 1$ (or 2). Therefore, the size of $T_1 \bowtie_\theta T_2$ (where θ is $T_1.A = T_2.A$) is estimated as 0, even though the join actually produces a large number of records.

To solve this problem, Chaudhuri et al. [8] suggest sampling the join result in a two-step manner. The first step joins each tuple $t \in T_1$ with the data in T_2, and records $t.w$ (the *weight* of t) as the number of records in T_2 that qualify θ with t. Then, in the second step, each record $t \in T_1$ is re-examined. This time, a coin is thrown with head probability proportional to $t.w$. If the coin tails, t is ignored and the execution proceeds with the next tuple in T_1. Otherwise (the coin heads), a subset of the records in $t \bowtie_\theta T_2$ (all the join results produced by t) is randomly extracted and included into RS_θ (the sample set over the join results). The expected size of this subset is also proportional to $t.w$ (see [8] for details).

The above method is static because subsequent insertions and deletions on T_1 or T_2 necessarily affect the weights of individual tuples and, hence, change the probabilities that they should appear in RS_θ. Srivastava and Widom [24] develop a similar sampling strategy that can handle updates. Their work, however, is restricted to sliding windows and assumes a priori knowledge about data distributions (records follow either the "age-based" or "frequency-based" model). In Section 5.2, we develop alternative methods without such constraints. Acharya et al. [2] propose the "join synopsis" for obtaining random samples in the special case of foreign-key joins, but their methods are inapplicable to arbitrary join conditions. Other relevant work [3], [9], [19] focuses on estimating the join sizes without computing samples.

3 Dynamic Reservoir Sampling

In this section, we present the R* algorithm, an extension of *reservoir sampling* that supports an arbitrary sequence of insertions and deletions. Section 3.1 discusses the algorithmic details of R* and Section 3.2 analyzes its characteristics.

3.1 Algorithm

R* maintains an array RS with size M, whose value is determined by the amount of available memory. At any time, only a subset of *valid* records in RS belongs to the current sample set. Each element $RS[i]$ $(1 \le i \le M)$ is associated with a tag $RS[i].valid$ that equals TRUE if $RS[i]$ is valid, and FALSE, otherwise. Initially, every $RS[i].valid$ equals FALSE, indicating an empty sample set. Insertions are handled in a way similar to the conventional *reservoir* method. Specifically, for each I-command (let t be the record being inserted), we generate a random integer x in the range $[1, n_I]$, where n_I is the total number of insertions processed so far. If $x \le M$, we place t at the xth position $RS[x]$ of RS, and set $RS[x].valid$ to TRUE. Otherwise, $(x > M)$, no further action is taken and t is ignored.

To handle a D-command $\{\mathrm{D}, id\}$, on the other hand, we first check if the tuple with the requested id belongs to the sample set, namely, whether there exists a number x $(1 \le x \le M)$ such that the id of $RS[x]$ equals id, and $RS[x].valid =$ TRUE. If x is found, deletion is completed by simply modifying $RS[x].valid$ to FALSE, without affecting the other elements in RS. Fig. 1 presents the pseudocode of R*.

We emphasize several differences between R* and the *reservoir* algorithm coupled with *delete-at-will* (reviewed in Section 2.1). First, although both algorithms generate a random number for each incoming I-command, the upper bound of the random number generated by R* equals n_I (Line 2 of R*-*insert* in Fig. 1), as opposed to $|T|$ in *reservoir*.

Second, whenever a sample is deleted, *reservoir* wastes an element in array RS, that is, the element can no longer be used to hold a sample. Accordingly, the maximum possible sample-set size also decreases, thus wasting an increasingly large part of the memory. R*, on the other hand, does not reduce the size of RS, i.e., in the future, the sample set size can still be as large as permitted by the available memory.

Third, after deleting a sample, the positions of the remaining samples in RS are not important for *reservoir* (e.g., the algorithm is still correct, even if two samples switch their positions). For R^*, however, the remaining samples must be fixed to their original positions. The reason for this will be clear in the correctness proof of R^* in the next section.

In order to efficiently retrieve records with particular ids (during deletion), we create an appropriate index $I(RS)$ (e.g., a main-memory B-tree [15]) on the sample ids. $I(RS)$ is updated whenever the content of RS changes. Note that, since $I(RS)$ contains only the ids of the tuples, its size is $1/d$ of the space occupied by the relation, where d is the number of attributes of a tuple.

Given a Q1 query q, we count the number n_q of *valid* records in RS that satisfy q, and report $n_q \cdot |T|/s$ as the approximate answer, where $|T|$ is the current cardinality of T, and s is the number of valid samples. Obviously, both $|T|$ and s can be maintained with trivial overhead—they are simply increased (decreased) whenever a tuple is inserted into (deleted from) T and RS, respectively.

3.2 Analysis

We first show that R^* indeed produces a random sample set. Let U_{seq} be the sequence of all the update requests sorted according to their arrival order, and I_i be the ith I-command; similarly, D_j is the jth D-command. Lemma 1 provides a crucial observation:

Lemma 1. *Let D_j and I_i be a pair of consecutive commands in sequence U_{seq} such that D_j arrives before I_i (i.e., $U_{seq} = \{\ldots D_j I_i \ldots\}$). Denote U'_{seq} as the sequence obtained from U_{seq} by simply swapping the order of I_i and D_j (i.e., $U'_{seq} = \{\ldots I_i D_j \ldots\}$). Then, executing R^* on U_{seq} and U'_{seq} leads to the same RS.*

Proof. Since D_j arrives before I_i in the original sequence U_{seq}, the two updates refer to different records. Let RS_{bfr} (RS'_{bfr}) be the content of RS when all the commands before D_j (I_i) in sequence U_{seq} (U'_{seq}) have been processed. Obviously, $RS_{bfr} = RS'_{bfr}$ since they are the content of RS after processing the same sequence of updates. Similarly, let RS_{aft} (RS'_{aft}) be the content of RS after processing I_i (D_j) in U_{seq} (U'_{seq}). We will show that RS_{aft} is also identical to RS'_{aft}, which establishes the correctness of the lemma, because the remaining parts of U_{seq} and U'_{seq} are exactly the same.

To prove $RS_{aft} = RS'_{aft}$, notice that handling a D-command does not generate any random number. Therefore, the value x produced at Line 2 of R^*-*insert* (Fig. 1) for I_i is the same in both U_{seq} and U'_{seq} (this is why, after deleting a sample, the remaining samples must be kept to their original positions in the array). This indicates that the tuple inserted by I_i will appear in both RS_{aft} and RS'_{aft} simultaneously, or will not appear in any of them. Furthermore, if the tuple requested by D_j exists in RS_{bfr} (RS'_{bfr}), it will disappear in RS_{aft} (RS'_{aft}), which establishes the correctness of the lemma. □

Based on the above lemma, we prove the randomness of the sample set obtained by R^*:

Theorem 1. *Let T be the relation being sampled by R^*. Then, the valid records in RS always constitute a random sample set for the current content of T.*

Proof. Assume that the original update sequence U_{seq} has n_I insertions and n_D deletions. Let us continuously swap pairs of consecutive D and I-commands, if the D-command is before the I-command. Eventually, we obtain a sequence $U'_{seq} = \{I_1 \ldots I_{n_I} D_1 \ldots D_{n_D}\}$, where every I-command is positioned before all the D-commands. Let RS_I be the content of RS after performing all the I-commands using the R^* algorithm, which in this case is the same as the conventional *reservoir*. Hence, RS_I is a random sample set over all tuples that have been inserted. The subsequent execution of R^* on the remaining D-commands is reduced to the *delete-at-will* approach reviewed in Section 2.1, and therefore, the final RS is still a random sample set of U'_{seq}. By Lemma 1, RS thus computed is identical to that obtained by running R^* on the original sequence U_{seq}, completing the proof. □

Note that the above theorem is correct even if the same object is repeatedly inserted and removed from the database. In this case, special care should be taken to understand semantics of the insertions/deletions in U'_{seq} in the above proof. For example, assume that an object is inserted and removed twice by operation sequence I_a, D_b, I_c, D_d, for some integers $a < b < c < d$. These four updates have order I_a, I_c, D_b, D_d in U'_{seq}. Hence, when D_b is processed (according to U'_{seq}), there are two copies of the object in the database (inserted by I_a and I_c, respectively). Then, D_b removes the copy created by I_a, and similarly, D_d removes the one created by I_c.

As a second step, we quantify the number s of valid elements in RS (i.e., the sample size). In particular, our goal is to show that, unlike the *delete-at-will* approach, the sample size of R^* does not decrease with time, but stabilizes at a certain value depending on the total numbers of insertions and deletions already seen.

Lemma 2. *Let s be the number of valid records in RS produced by R^*. Then, the probability $P\{s = v\}$ that s equals a particular value v ($1 \le v \le M$, where M is the maximum size of RS) is given by:*

$$p\{s = v\} = \binom{n_D}{M - v}\binom{n_I - n_D}{v} \Big/ \binom{n_I}{M}, \tag{1}$$

where n_I (n_D) is the total number of insertions (deletions) processed, assuming $n_I \ge M$ (i.e., at least M I-commands have been received).

Proof. In the same way as described in the proof for Theorem 1, we convert the original update sequence U_{seq} into $U'_{seq} = \{I_1 \ldots I_{n_I} D_1 \ldots D_{n_D}\}$ (the result of R^* on U_{seq} is equivalent to that on U'_{seq}). Let RS_I be the content of RS after processing all the I-commands. Since RS_I is a simple random sample set (with M samples) of the relation T' containing the n_I inserted records, the number of possible RS_I equals $\binom{n_I}{M}$. Furthermore, an RS_I leads to a final RS with v valid elements after handling the D-commands in U_{seq} if and only if two conditions are satisfied. First, $M - v$ (out of M) tuples of RS_I are chosen from the set T_D of records deleted by the n_D D-commands. Second, the other v tuples of RS_I are selected from $T' - T_D$ (involving $n_I - n_D$ records). The number of RS_I qualifying the above conditions is $\binom{n_D}{M-v} \cdot \binom{n_I-n_D}{v}$. Therefore, the probability $P\{s = v\}$ can be computed as $\binom{n_D}{M-v} \cdot \binom{n_I-n_D}{v} \Big/ \binom{n_I}{M}$. □

As a corollary, the expected sample size $E(s)$ of R* equals:

$$\mathrm{E(s)} = \sum_{v=0}^{M} (v \cdot P\{s = v\}) = \sum_{v=1}^{M} (v \cdot P\{s = v\}), \tag{2}$$

where $P\{s = v\}$ is represented in (1). Solving this formula results in the following lemma:

Lemma 3. *The expected sample size $E(s)$ equals:*

$$E(s) = M \cdot (n_I - n_D)/n_I, \tag{3}$$

where n_I and n_D are as defined in Lemma 2.

Proof. We prove the lemma by induction. First, for $M = 1$, (2) becomes $1 \cdot P\{s = 1\}$, which, by Lemma 2, is $\binom{n_D}{1-1} \cdot \binom{n_I-n_D}{1} \Big/ \binom{n_I}{1} = (n_I - n_D)/n_I$. Hence, the lemma is correct in this case. Next, assuming that the lemma holds for M equal to any integer k, we show that it also holds for $M = k + 1$.

Let us write $u = v - 1$. Hence, (2) can be rewritten as (setting M to $k + 1$):

$$\sum_{u=0}^{k} (u+1) \cdot \binom{n_D}{k-u} \binom{n_I - n_D}{u+1} \Big/ \binom{n_I}{k+1}.$$

Since $\binom{n_I-n_D}{u+1} = \binom{n_I-n_D+1}{u+1} - \binom{n_I-n_D}{u}$ and

$$\binom{n_I}{k+1} = (k+1) \Big/ \left[(n_I - k)\binom{n_I}{k}\right],$$

the above equation can be transformed to:

$$\begin{aligned} &\frac{k+1}{n_I - k} \sum_{u=0}^{k} (u+1) \cdot \binom{n_D}{k-u} \binom{n_I - n_D + 1}{u+1} \Big/ \binom{n_I}{k} \\ &- \frac{k+1}{n_I - k} \sum_{u=0}^{k} (u+1) \cdot \binom{n_D}{k-u} \binom{n_I - n_D}{u} \Big/ \binom{n_I}{k}. \end{aligned} \tag{4}$$

It holds that $\binom{n_I-n_D+1}{u+1} = \frac{n_I-n_D+1}{u+1} \binom{n_I-n_D}{u}$. Furthermore, by the inductive assumption (Lemma 3 holds for $M = k$), we have $\sum_{u=0}^{k} u \cdot \binom{n_D}{k-u} \binom{n_I-n_D}{u} / \binom{n_I}{k} = k \cdot \frac{n_I-n_D}{n_I}$. Combining these facts, we simplify (4) as:

$$\begin{aligned} &\frac{k+1}{n_I - k} \cdot (n_I - n_D) \cdot \sum_{u=0}^{k} \binom{n_D}{k-u} \cdot \binom{n_I - n_D}{u} \Big/ \binom{n_I}{k} \\ &- \frac{k+1}{n_I - k} \cdot \left(k \cdot \frac{n_I - n_D}{n_I}\right). \end{aligned}$$

Observe that $\sum_{u=0}^{k} \binom{n_D}{k-u} \binom{n_I-n_D}{u} = \binom{n_I}{k}$. Hence, the above equation becomes:

$$\begin{aligned} &\frac{k+1}{n_I - k} \cdot (n_I - n_D) - \frac{k+1}{n_I - k} \cdot \left(k \cdot \frac{n_I - n_D}{n_I}\right) \\ &= (k+1) \cdot \frac{n_I - n_D}{n_I}. \end{aligned}$$

Thus, we complete the proof. □

The above equation has a clear intuition: The percentage of the valid tuples in RS corresponds to the percentage of the records currently in T among all those ever inserted. Indeed, the probability $P\{s = v\}$ given in (1) peaks at $v = M \cdot (n_I - n_D)/n_I$, and quickly diminishes as v drifts away from this value. This, in turn, indicates a small variance for s, thus validating the usefulness of (3). As a corollary of Lemma 3, the actual sample size of R* is expected to remain constant with time provided that the ratio n_I/n_D between the numbers of insertions and deletions is fixed.

We are now ready to quantify the per-tuple processing cost of R*.

Theorem 2. *For each I-command,* R* *performs a deletion from $I(RS)$ with probability $E(s)/n_I$, and an insertion into $I(RS)$ with probability M/n_I, where $E(s)$ is the expected sample size given in (2) or (3), M the memory capacity, and n_I the total number of insertions. For a D-command,* R* *always executes a search in $I(RS)$, and then a deletion from $I(RS)$ with probability $E(s)/(n_I - n_D)$.*

Theorem 2 shows that, for each I-command, R* performs O(1) work with a high probability; when it needs to do something, it performs a single insertion to $I(RS)$. On the other hand, for a D-command, R* always performs a search and, with a small probability, also a deletion on $I(RS)$. Since $I(RS)$ is a main-memory B-tree, the worst-case per-tuple cost of R* is $O(log|RS|) = O(logM)$.

4 Distinct Counting Sample

In this section, we develop an alternative sampling approach CS*, which is motivated by the *counting sample* reviewed in Section 2.1, but improves its update performance considerably. The next section first elaborates the sampling procedures of CS*, and Section 4.2 analyzes its behavior.

4.1 Algorithm

As with R*, CS* maintains an array RS (containing at most M elements), and associates each record $RS[i]$ $(1 \le i \le M)$ with a tag $RS[i].valid$ to indicate its validity in the current sample set. In addition, it uses a stack, denoted as *vacant*, to organize the positions of invalid elements. If RS contains s valid tuples, then *vacant* contains $M - s$ numbers. Let x be the value of $vacant[i]$ $(1 \le i \le M - s)$; it follows that $RS[x].valid$ must be FALSE. Adopting the idea of the original CS method, CS* deploys a variable τ to control the probability of sampling an incoming tuple. Before receiving any record, $RS[i].valid = $ FALSE, $vacant[i]$ is set to i (for all $1 \le i \le M$), and τ is initialized to 1.

Consider that an I-command arrives at the system, inserting tuple t. CS* throws a coin with probability $1/\tau$ head (i.e., $1/\tau$ is the sampling rate), and discards t if the coin tails. Otherwise, it checks whether *vacant* is empty (equivalently, if all the elements in RS are valid). If not, the position $x = vacant[M - s]$ (i.e., the top of the stack) is obtained, and

t is stored at $RS[x]$, which completes the insertion. Now, consider the scenario where *vacant* is empty (i.e., t is not ignored, but RS already contains M valid samples). In this case, a *memory overflow* occurs, and CS* generates a random number x in the range $[1, M+1]$. If $x = M+1$, t is expunged, leaving the content of RS intact; otherwise, ($x \leq M$), t is placed at the xth element of RS, replacing the original sample $RS[x]$. Finally, regardless of the value of x, τ is increased to $\tau \cdot (M+1)/M$ (i.e., the next incoming record is sampled with lower probability).

Processing a D-command (deleting tuple t) is relatively easy. Specifically, we first check whether t exists in RS. If the answer is positive (let $RS[x] = t$), t is removed from RS by 1) marking $RS[x].valid = \text{FALSE}$, and 2) stacking position x into *vacant*. Similar to R*, to efficiently retrieve t by its id, we maintain an index $I(RS)$ on the ids in RS, which is updated with RS.

Fig. 2 summarizes the above procedures. Note that, for each I (D-) command, updating *vacant* can be done in $O(1)$ time. The query processing algorithm for CS* is exactly the same as that for R*. Specifically, we count the number n_q of valid elements in RS satisfying q, and scale up n_q by a factor of $|T|/s$. Finally, recall that the stack *vacant* is not needed in R* because the position $RS[x]$, where an incoming sample should be stored, is randomly generated even though some records in RS are invalid (i.e., the inclusion of t does not necessarily result in a larger sample size). CS*, on the other hand, always uses vacant positions to accommodate new samples (thus increasing the sample size), which requires dedicated structures for recording vacancies.[3]

```
Algorithm CS*-init (RS)
1. $\tau = 1$; $s = 0$
2. for $i = 1$ to $M$
3.     $RS[i].valid$ = FALSE; $vacant[i] = i$
Algorithm CS*-insert (RS, t) // t is an incoming tuple.
1. flip a coin with $1/\tau$ probability head
2. if the coin heads
3.     if $s < M$
4.         $x = vacant[M - s]$; $s{+}{+}$
5.         $RS[x] = t$; $RS[x].valid$ = TRUE
6.         insert $RS[x]$ into $I(RS)$
7.     else
8.         $x$ = a random number in $[1, M + 1]$
9.         if $x = M + 1$ then ignore $t$
10.        else
11.            remove $RS[x]$ from $I(RS)$
12             $RS[x] = t$; insert $RS[x]$ into $I(RS)$;
13         $\tau = \tau \cdot (M + 1)/M$
Algorithm CS*-delete (RS, t) // t is the tuple to be deleted
1. if $t \in RS$      //this is checked using $I(RS)$
2.     let $x$ be the position number of $t$ in $RS$
3.     $RS[x].valid$ = FALSE; remove $RS[x]$ from $I(RS)$
4.     $s - -$; $vacant[M - s] = x$
```

Fig. 2. Adapted counting sample algorithm.

4.2 Analysis

In the sequel, we analyze the performance of CS*, and compare it with the R* algorithm. Similar to R*, CS* guarantees the randomness of the samples:

Theorem 3. *Let T be the stream relation being sampled by* CS*. *The probability for any tuple in T to be a valid record in RS always equals the current sampling rate $1/\tau$. As a result, the set of valid elements in RS is a random sample set of T.*

Proof. We prove the statement by induction. Before the memory overflows for the first time (i.e., $\tau = 1$), every tuple in T appears in RS, in which case the theorem is trivially true. As the inductive step, we assume that the theorem holds after the kth ($k \geq 0$) overflow, and we show that it is still correct after the next overflow. Let t be the arriving record that causes the $(k+1)$st overflow, and t' be any other tuple in T. Then, t (t') appears in RS after the overflow if 1) the coin heads at line 2 of CS*-$insert$ in Fig. 2 (t' was already in RS before t arrived), and 2) it is not discarded (lines 9-12 of CS*-$insert$) during the overflow resolution. The probability for both events to occur equals $(1/\tau) \cdot M/(M+1)$, which is the adjusted sampling rate after the insertion, thus completing the proof. □

At any time, every tuple that has been inserted but not deleted has a probability $1/\tau$ to be a valid sample in CS*. Therefore:

Lemma 4. *The expected sample size of* CS* *is $(n_I - n_D)/\tau$, where τ is the current sampling rate, and n_I (n_D) is the total number of insertions (deletions) that have been processed so far.*

A natural question is "*which is more accurate: R* or* CS*?" Since both methods return random samples, their accuracy depends solely on the cardinality of their sample sets. Furthermore, notice that the sampling rate of R* at any time is essentially M/n_I which, when multiplied with the current database cardinality $n_I - n_D$, gives the expected sample set size as in (3). Therefore, in order to compare the sample set sizes of R* and CS*, it suffices to relate M/n_I to the sampling rate $1/\tau$ of CS*. The comparison result, however, is not definite, i.e., it is possible for either technique to have a larger sample set, depending on the update pattern of the underlying stream. We illustrate this with two concrete examples.

Consider a database whose cardinality $|T|$ remains fixed with time, and assume $|T| > M$. The update sequence consists of $|T|$ initial insertions, after which every subsequent insertion is preceded by a deletion. In this case, the R* sampling rate M/n_I continuously decreases with time due to the growth of the denominator. On the other hand, the sampling rate of CS* remains fixed as soon as the first $|T|$ insertions have been completed, because there is no overflow of array RS after that (recall that CS* decreases its sampling rate by a factor of $M/(M+1)$ only when an overflow occurs). Therefore, eventually, CS* will have a larger sample size than R*.

Assume, on the other hand, that the update sequence contains a large volume n_I of insertions, followed by a certain number n_D of deletions. Then, the sampling rates of both techniques stabilize after all the insertions are processed. At this moment, CS* has incurred $n_I - M$ overflows (in array RS), leading to a final sampling rate $[M/(M+1)]^{n_I - M}$, which can be considerably smaller than

3. It is worth mentioning that, the stack *vacant* could be avoided by reorganizing the positions of valid tuples in RS whenever a sample becomes invalid. We do not follow this approach as it leads to considerable maintenance overhead of $I(RS)$.

the R* sampling rate M/n_I (note that the rate of CS* decays exponentially with n_I). Therefore, after handling all the deletions, R* will end up with a more sizable sample set.

We summarize the update performance of CS* with the following theorem.

Theorem 4. *For each I-command,* CS* *performs a deletion and an insertion on* $I(RS)$ *with probability* $1/\tau$*, where* $1/\tau$ *is the current sampling rate. For each D-command, a deletion is required with probability* $1/\tau$*.*

Similarly to R*, by implementing $I(RS)$ as a main-memory B-tree, the per-command processing cost of CS* is bounded by $O(logM)$.

5 Sampling Methods on Stream Joins

Based on the techniques developed in the previous sections, we proceed to discuss Q2 queries, which return aggregate information about the join of two relations. Section 5.1 presents an algorithm that solves the problem by maintaining random samples on the join results. Then, Section 5.2 proposes an alternative approach with considerably less space consumption and computation overhead.

5.1 Rigorous Join Sampling

Recall that all Q2 formulations (on stream relations T_a and T_b) possess a common set θ_{all} of predicates. We denote $T_{\bowtie}$ as the results of $T_a \bowtie_{\theta_{all}} T_b$. Any concrete Q2 instance can be regarded as a Q1 query with its own condition θ_{any} on a *single* relation $T_{\bowtie}$. This observation establishes a natural reduction from problem Q2 to Q1: Given a random sample set $RS(T_{\bowtie})$ on $T_{\bowtie}$, we can answer any Q2 query in the same way as Q1.

If T_a and T_b are fully preserved in the system, $RS(T_{\bowtie})$ can be maintained using a method (such as R* or CS*) that dynamically computes random samples of a relation. Consider, for example, an arriving I-command that inserts tuple t_a into T_a. This arrival adds a set of tuples to $T_{\bowtie}$ corresponding to the results of $t_a \bowtie_{\theta_{all}} T_b$ (involving t_a and the data in T_b). These tuples are passed to the insertion module of the deployed sampling method for updating $RS(T_{\bowtie})$. A D-command that removes a tuple t_a from T_a can be processed in the reverse manner. Specifically, the deletion eliminates from $T_{\bowtie}$ all tuples $t_a \bowtie_{\theta_{all}} T_b$, which are fed to the deletion module of the sampling method.

To answer a Q2 query q, we count the number n_q of samples in $RS(T_{\bowtie})$ that qualify the condition θ_{any} of q. The query result equals $n_q \cdot |T_{\bowtie}|/|RS(T\bowtie)|$, where $|T_{\bowtie}|$ and $|RS(T_{\bowtie})|$ are the sizes of $T_{\bowtie}$ and $RS(T_{\bowtie})$, respectively. We call this approach the *rigorous join sampling* (RJS). Unfortunately, RJS cannot be implemented in stream environments because it requires 1) keeping the entire T_a and T_b in memory and 2) examining a complete relation for each update.

Therefore, in the sequel, we focus on approximate solutions that do not have theoretical guarantees, but 1) are scalable to the available amount M of memory, 2) have low update overhead, and 3) yet are able to provide accurate answers to Q2 queries.

5.2 Approximate Join Sampling

Approximate join sampling (AJS) aims at "simulating" the behavior of RJS. For example, whereas RJS maintains T_a and T_b completely, AJS preserves only two sets $RS(T_a)$, $RS(T_b)$ of random samples on the two tables (using the R* or CS* algorithm). Furthermore, as opposed to RJS that outputs a sample set $RS(T_{\bowtie})$, AJS approximates $RS(T_{\bowtie})$ with a set $aRS(T_{\bowtie})$, whose elements are *partial pairs* of the form $\{t_a, -\}$ or $\{-, t_b\}$, where "$-$" means NULL, and t_a, t_b are tuples in $RS(T_a)$, $RS(T_b)$, respectively. Assume, for instance, that the $RS(T_{\bowtie})$ of RJS currently contains four joined pairs $\{a_1, b_1\}$, $\{a_1, b_2\}$, $\{a_2, b_2\}$, and $\{a_3, b_3\}$. Then, AJS would produce an $aRS(T_{\bowtie})$ with elements $\{a_1, -\}$, $\{a_1, -\}$, $\{a_2, -\}$, and $\{-, b_3\}$. The first pair, for example, corresponds to a join pair (according to θ_{all}) involving a_1, without indicating the tuple from T_b that produces the result. An alternative interpretation of $\{a_1, -\}$ is that: *any record* (e.g., b_1, b_2) *in* T_b *qualifying* θ_{all} *with* a_1 *can appear in the "$-$" field with an equal probability*. We first explain how to use such incomplete information to answer Q2 queries.

5.2.1 Query Algorithm

As discussed earlier, given a Q2 query (with predicate θ_{any}), RJS obtains the number n_q of samples in $RS(T_{\bowtie})$ that qualify θ_{any}, and then returns $n_q \cdot |T_{\bowtie}|/|RS(T_{\bowtie})|$, where $|RS(T_{\bowtie})|$ and $|T_{\bowtie}|$ are the cardinalities of $RS(T_{\bowtie})$ and $T_{\bowtie}$, respectively. A similar approach is taken by AJS. Specifically, for every partial pair (e.g., $\{t_a, -\}$) in $aRS(T_{\bowtie})$, AJS increases n_q by 1 with the probability that the pair satisfies θ_{any}. The final result equals $n_q \cdot w/|aRS(T_{\bowtie})|$, where $|aRS(T_{\bowtie})|$ is the size of $aRS(T_{\bowtie})$, and w is an estimate for $|T_{\bowtie}|$ (its computation will be clarified later).

Assuming t_b to be any tuple in T_b, the probability that $\{t_a, -\}$ satisfies θ_{any} equals the *conditional probability* $P\{\theta_{any}|t_a, \theta_{all}\}$ that $\{t_a, t_b\}$ satisfies θ_{any}, *given that* $\{t_a, t_b\}$ passes θ_{all} (the "given" condition is needed for $\{t_a, -\}$ to appear in $aRS(T_{\bowtie})$). We consider that θ_{any} and θ_{all} are independent, so that $P\{\theta_{any}|t_a, \theta_{all}\}$ is identical to the probability $P\{\theta_{any}|t_a\}$ that $\{t_a, t_b\}$ satisfies θ_{any}. $P\{\theta_{any}|t_a\}$ equals the percentage of samples in $RS(T_b)$ qualifying θ_{any} with t_a and, hence, can be easily obtained upon the arrival of t_a. Fig. 3 formally presents the query algorithm based on the above discussion.

5.2.2 Insertion

We explain how to update $aRS(T_{\bowtie})$ assuming that the system has received an I-command inserting tuple t_a into T_a (the case of inserting into T_b is symmetric). Recall that in this case, RJS computes $t_a \bowtie_{\theta_{all}} T_b$ (the results of joining t_a with all tuples in T_b). AJS, on the other hand, simply creates a set of $\{t_a, -\}$ pairs to represent the join results. The number of such pairs corresponds to the estimated size of $t_a \bowtie_{\theta_{all}} T_b$. This estimation reduces to a Q1 query: "How many tuples in T_b qualify θ_{all} with t_a?" Thus, using the random sample set $RS(T_b)$, the cardinality of $t_a \bowtie_{\theta_{all}} T_b$ can be predicted as $n_a\,|T_b|/|RS(T_b)|$, where n_a is the number of samples in $RS(T_b)$ satisfying this Q1 query.

Next, RJS will pass the records in $t_a \bowtie_{\theta_{all}} T_b$ to the insertion module of a sampling method for updating $RS(T_{\bowtie})$. Accordingly, AJS passes all the $\{t_a, -\}$ pairs to the same module for modifying $aRS(T_{\bowtie})$, i.e., some of $\{t_a, -\}$ are sampled into $aRS(T_{\bowtie})$, while the rest are

```
Algorithm AJS-query ($q$)
1. $n_q = 0$;
2. for each element in $aRS(T_{\bowtie})$ having the form $\{t_a, -\}$
3.     get the percentage $p$ of tuples $t_b$ in $RS(T_b)$ such that $\{t_a, t_b\}$ qualifies the $\theta_{any}$ of $q$
4.     increase $n_q$ by 1 with probability $p$
5. for each element in $aRS(T_{\bowtie})$ having the form $\{-, t_b\}$
6.     get the percentage $p$ of tuples $t_a$ in $RS(T_a)$ such that $\{t_a, t_b\}$ qualifies the $\theta_{any}$ of $q$
7.     increase $n_q$ by 1 with probability $p$
8. return $n_q \cdot w / |aRS(T_{\bowtie})|$   //the value of $w$ is maintained by the algorithms in Figure 4
```

Fig. 3. AJS query algorithm.

discarded. Processing the I-command is completed after the sampling.

Each $\{t_a, -\}$ added to $aRS(T_{\bowtie})$ corresponds to a complete pair $\{t_a, t_b\}$ incorporated into $RS(T_{\bowtie})$ by RJS. Every record t_b in T_b, which qualifies θ_{all} with t_a, has the same probability to appear in the field "$-$" of this sampled $\{t_a, -\}$. We refer to the probability as "the *appearance probability* of $\{t_a, -\}$." Specifically, let n_{sam} be the number of $\{t_a, -\}$ taken into $aRS(T_{\bowtie})$ (during the processing of the I-command); the appearance probability equals n_{sam} / $[n_a|T_b|/|RS(T_b)|]$, where the term surrounded by the block parentheses describes the expected number of T_b tuples satisfying θ_{all} with t_a (recall that n_a is the number of tuples in $RS(T_b)$ satisfying θ_{all} with t_a, when t_a arrives). We associate this probability with every sampled $\{t_a, -\}$ (its use will be discussed later).

5.2.3 Deletion

AJS handles deletions also by "following" the corresponding actions of RJS. Given a D-command that deletes a record t_a from T_a, RJS will remove from $RS(T_{\bowtie})$ all the elements involving t_a. Motivated by this, AJS eliminates the partial pairs in $aRS(T_{\bowtie})$ that *may* include t_a. Such pairs may have the form $\{t_a, -\}$ or $\{-, t_b\}$. Since the first case is trivial, we focus on the second one.

Let $\{-, t_b\}$ be a member of $aRS(T_{\bowtie})$. Resorting to the analogy with RJS, the corresponding "complete" pair in $RS(T_{\bowtie})$ would include t_a only if *all* the following conditions hold:

1. Tuples t_a and t_b satisfy θ_{all}.
2. Record t_a must have arrived before t_b.
3. When t_b was inserted, t_a was sampled among all the tuples in T_a that qualify θ_{all} with t_b.

Hence, AJS decides to retain $\{-, t_b\}$ if t_a and t_b violate θ_{all}. Otherwise, it removes $\{-, t_b\}$ with probability $P_{second} \cdot P_{third}$, where $P_{second}(P_{third})$ is the probability that the second (third) criterion is satisfied. Next, we explain heuristics for obtaining P_{second} and P_{third}, respectively.

Assume that we know the number $t_b.n_{aft}$ of records in T_a that arrived after t_b, and qualify θ_{all} with t_b. Then, P_{second} can be approximated as $(N_b - t_b.n_{aft})/N_b$, where N_b is the number of tuples *currently* in T_a satisfying θ_{all} with t_b. In particular, N_b can be obtained by a Q1 query on the random sample set $RS(T_b)$, with the query predicate derived from θ_{all} and t_b. Value $t_b.n_{aft}$, on the other hand, can be monitored *precisely* in a simple way. We only have to set it to 0 when $\{-, t_b\}$ is first included in $aRS(T_{\bowtie})$. Then, every time a record from stream T_a is received, we increase $t_b.n_{aft}$ by 1 if the new record qualifies θ_{all} with t_b.

It remains to clarify the computation of P_{third}. In fact, it is exactly the "appearance probability of $\{-, t_b\}$," computed when tuple t_b was inserted. As mentioned earlier, this appearance probability is associated with $\{-, t_b\}$ and, thus, does not need to be recalculated. We summarize the insertion/deletion procedures in Fig. 4, which also includes the modification of the estimated size w of $T_{\bowtie}$ (required for query processing), during updates.

6 Experiments

We empirically evaluate the effectiveness and efficiency of the proposed methods, using a Pentium IV CPU at 3GHz. The experimental results are presented in two parts, focusing on Q1 queries in Section 6.1 and Q2 in Section 6.2.

6.1 Performance of Q1 Processing

The experiments of this section involve stream relations that have two columns (id, A). Specifically, the tuples of a stream T are generated according to two parameters *dist* and λ. The first parameter *dist* determines the distribution of A-values in the domain [0, 10,000]. Unless specifically stated, we use synthetic data, where *dist* can be *Gau* or *Zipf* (we also include real data towards the end of the subsection). *Gau* denotes a Gaussian function with mean 5,000 and variance 2,000, while *Zipf* corresponds to a Zipf distribution skewed towards 0 with coefficient 0.8. The second parameter λ, is an integer that controls the ratio of insertions/deletions in the generated stream. Specifically, each update is an I-command with probability $\lambda/(\lambda + 1)$, or a D-command with probability $1/(\lambda + 1)$, i.e., the chance of receiving an I-command is λ times higher than a D-command. An I-command inserts a tuple into T with a unique id, whose A-value is determined by *dist*. A D-command randomly removes a record from the current T. For both *Gau* or *Zipf*, the entire stream contains four million I-commands, while the number of D-commands is approximately λ times lower than that of I-commands.

A Q1 query selects tuples whose A-values fall in an interval with length *qlen* (i.e., its θ_{any} is a range condition). The query interval is uniformly distributed in the universe [0, 10,000]. A *workload* consists of 1,000 queries obtained with the same *qlen*. The *relative error* of an approximate answer *est* is defined as $|act - est|/act$, where *act* is the actual number of records satisfying θ_{any}. We use two metrics to evaluate accuracy: 1) $rel\varepsilon$ is the average (relative) error of all queries in the workload, and 2) $max\varepsilon_{80}$ (called *80%-max error*), is the 200th largest error, that is, the error of the remaining 800 queries (80 percent of the workload) is bounded by $max\varepsilon_{80}$. While $rel\varepsilon$ indicates the expected accuracy of a technique,

```
**Algorithm AJS-init**
1. $aRS(T_{\bowtie}) = \emptyset$; $RS(T_a) = RS(T_b) = \emptyset$;
2. $S_a = S_b = \emptyset$; $w = 0$
**Algorithm AJS-insert** ($t_a$, $X$)
/* $t_a$ is a tuple being inserted into $T_a$ (the case of inserting a record into $T_b$ is symmetric and omitted); $X$ is a
random sampling algorithm for single relations */
1. $n_a$ = the number of records in $RS(T_b)$ qualifying $\theta_{all}$ with $t_a$
2. $w = w + n_a \cdot |T_b|/|RS(T_b)|$      //$w$ is the estimated size of $T_{\bowtie}$
3. for $i = 1$ to $n_a \cdot |T_b|/|RS(T_b)|$
4.     $X.insert$ ($aRS(T_{\bowtie})$, $\{t_a, -\}$)
5. associate each sampled $\{t_a, -\}$ with its appearance probability
6. for each record $t_b \in S_b$ that satisfies $\theta_{all}$ with $t_a$
7.     $t_b.n_{aft}$ ++;
8. $X.insert$ ($RS(T_a)$, $t_a$)
**Algorithm AJS-delete** ($t_a$, $X$) /* $t_a$ is a tuple being deleted from $T_a$ */
1. $n_a$ = the number of records in $RS(T_b)$ qualifying $\theta_{all}$ with $t_a$
2. $w = w - n_a \cdot |T_b|/|RS(T_b)|$
3. for each $\{t_a, -\} \in aRS(T_{\bowtie})$
4.     $X.delete$ ($aRS(T_{\bowtie})$, $\{t_a, -\}$)
5. for every record $t_b \in RS_b$ that satisfies $\theta_{all}$ with $t_a$
6.     $n_b$ = the number of records in $RS(T_a)$ qualifying $\theta_{all}$ with $t_b$
7.     $N_b = n_b \cdot |T_a|/|RS(T_a)|$
8.     throw a coin $c_1$ with probability $(N_b - t_b.n_{aft})/N_b$ head
9.     if $c_1$ tails then $t_b.n_{aft}$--
10.    else
11.        throw a coin $c_2$ with the appearance probability associated with $\{-, t_b\}$
12.        if $c_2$ heads
13.            $X.delete$ ($aRS(T_{\bowtie})$, $\{-, t_b\}$)
14. $X.delete$ ($RS(T_a)$, $t_a$)
```

Fig. 4. AJS update algorithms.

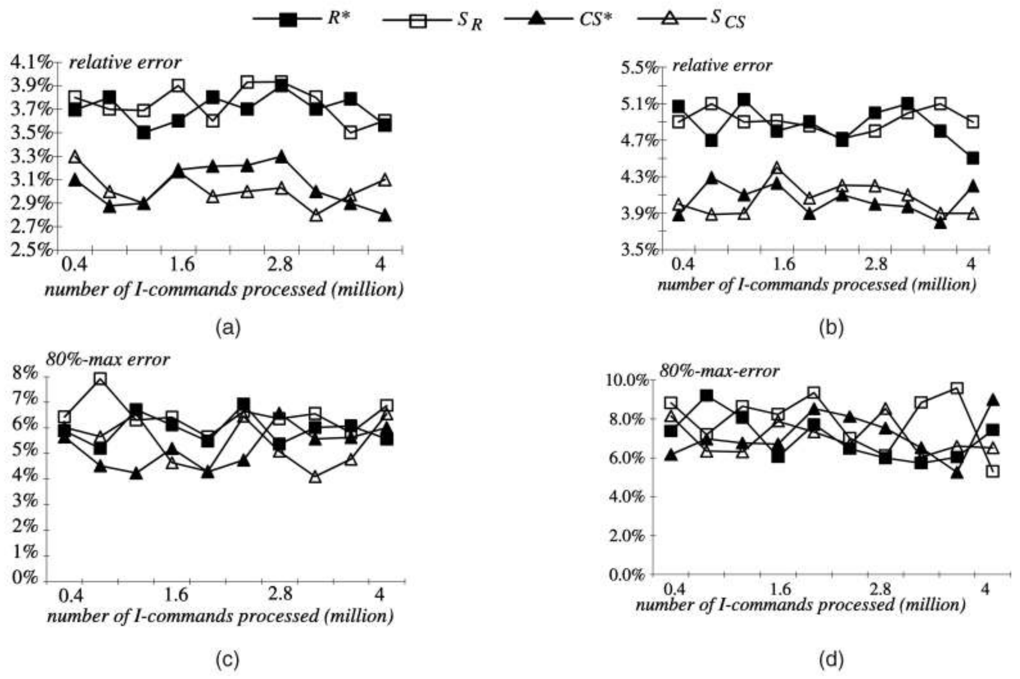

Fig. 5. Query error verification. (a) $rel\varepsilon$ versus time (*Gau*). (b) $rel\varepsilon$ versus time (*Zipf*). (c) $max\varepsilon_{80}$ versus time (*Gau*). (d) $max\varepsilon_{80}$ versus time (*Zipf*).

$max\varepsilon_{80}$ reveals its robustness—small $max\varepsilon_{80}$ means that it is able to capture the results of most queries.

6.1.1 Performance versus Time

The first set of experiments evaluates the randomness of the samples obtained by R* and CS*. For this purpose, we assume memory size $M = 10,000$ (i.e., the sample set can accommodate up to 10k tuples) and streams with $\lambda = 4$. After every 400k I-commands (10 percent of the insertions in the stream), we randomly select two sets S_R and S_{CS} of tuples from the current T, using the *reservoir* and *delete-at-will* algorithms, respectively. The cardinality of S_R (S_{CS}) is

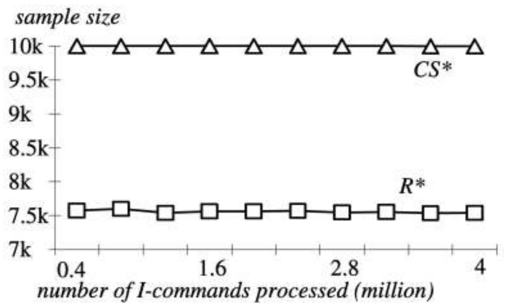

Fig. 6. Sample size changes (*Gau*).

equivalent to the sample size of R^* (CS^*) at this time. The rationale is that, if the samples obtained by R^* (CS^*) are random, they should lead to roughly the same error as S_R (S_{CS}).

Fig. 5a (or 5b) shows the $rel\varepsilon$ as a function of the number of I-commands handled for stream *Gau* (or *Zipf*) using workloads with $qlen = 600$. Figs. 5c and 5d illustrate similar results with respect to $max\varepsilon_{80}$. In all cases, the accuracy of R^* (CS^*) is statistically similar to that of S_R (S_{CS}). Specifically, the maximum deviation between the $rel\varepsilon$ ($max\varepsilon_{80}$) of R^* and S_R is around 0.3 percent (2 percent), while the corresponding value for CS^* and S_{CS} is 0.4 percent (1 percent). Hence, R^* and CS^* indeed produce random sample sets at the presence of insertions and deletions.

Fig. 6 plots the sample size of R^* and CS^* in the experiments of Figs. 5a and 5c (the results for stream *Zipf* are omitted due to their similarity). The (sample) sizes remain (almost) constant at $0.75M = 7.5\mathrm{k}$ tuples for R^* and $M = 10\mathrm{k}$ tuples for CS^*. The phenomenon is expected because, as analyzed in Sections 3.2 and 4.2, the sample set size of R^* only occupies a fraction of the available memory (the fraction depends on the ratio between the numbers of insertions and deletions, as shown in (3), whereas CS^* is able to utilize all the memory for storing samples. It is known [1] that the relative error of random sampling is inversely proportional to the square root of the sample size. Since the size of CS^* is larger than R^* by 33 percent, in theory the relative error of R^* is higher by around 15 percent, which is verified by the results in Fig. 5.

6.1.2 Query Accuracy

As explained in Section 2.1, the (repeated) scanning of the sample set during deletions prevents the application of *counting sample* to our targeted application domain.

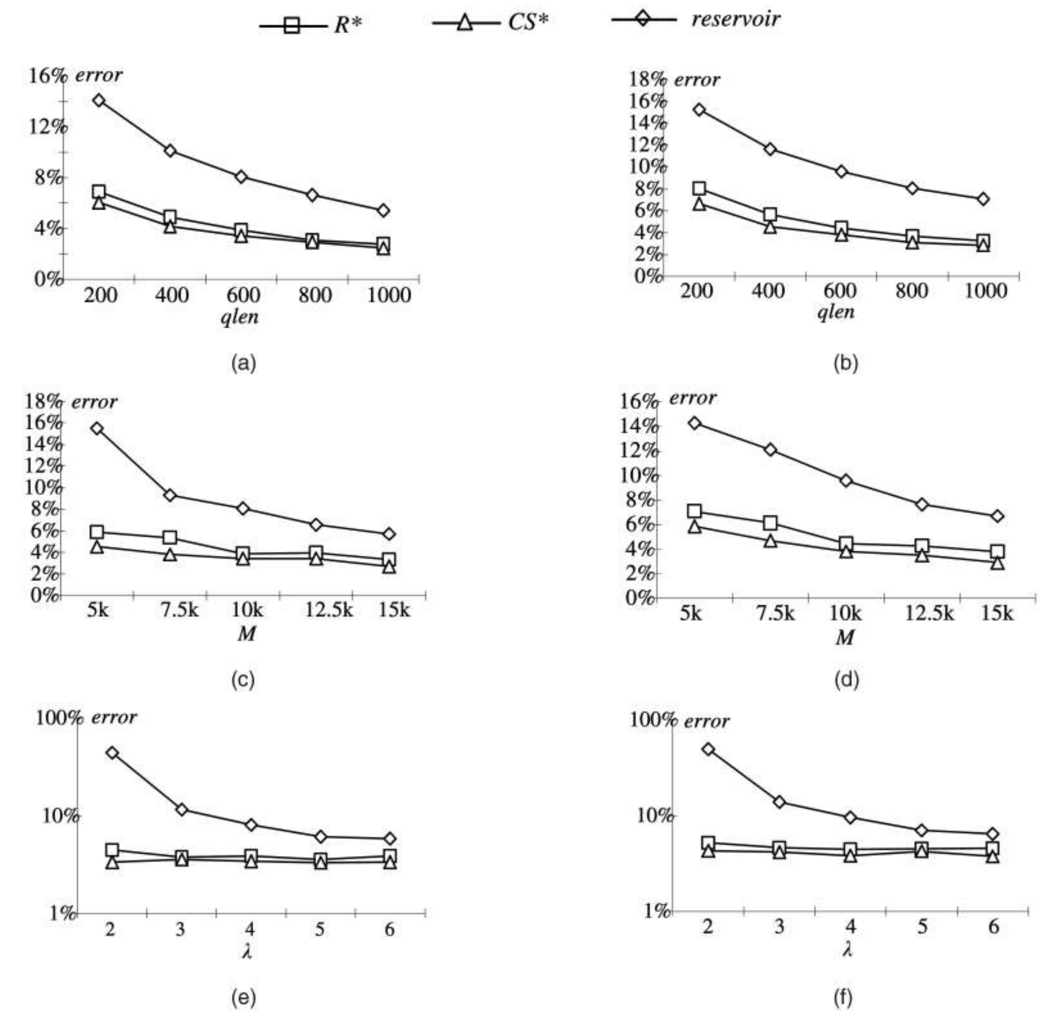

Fig. 7. Query accuracy comparison ($rel\varepsilon$). (a) Error versus *qlen* (*Gau*). (b) Error versus *qlen* (*Zipf*). (c) Error versus M (*Gau*). (d) Error versus M (*Zipf*). (e) Error versus λ (*Gau*). (f) Error versus λ (*Zipf*).

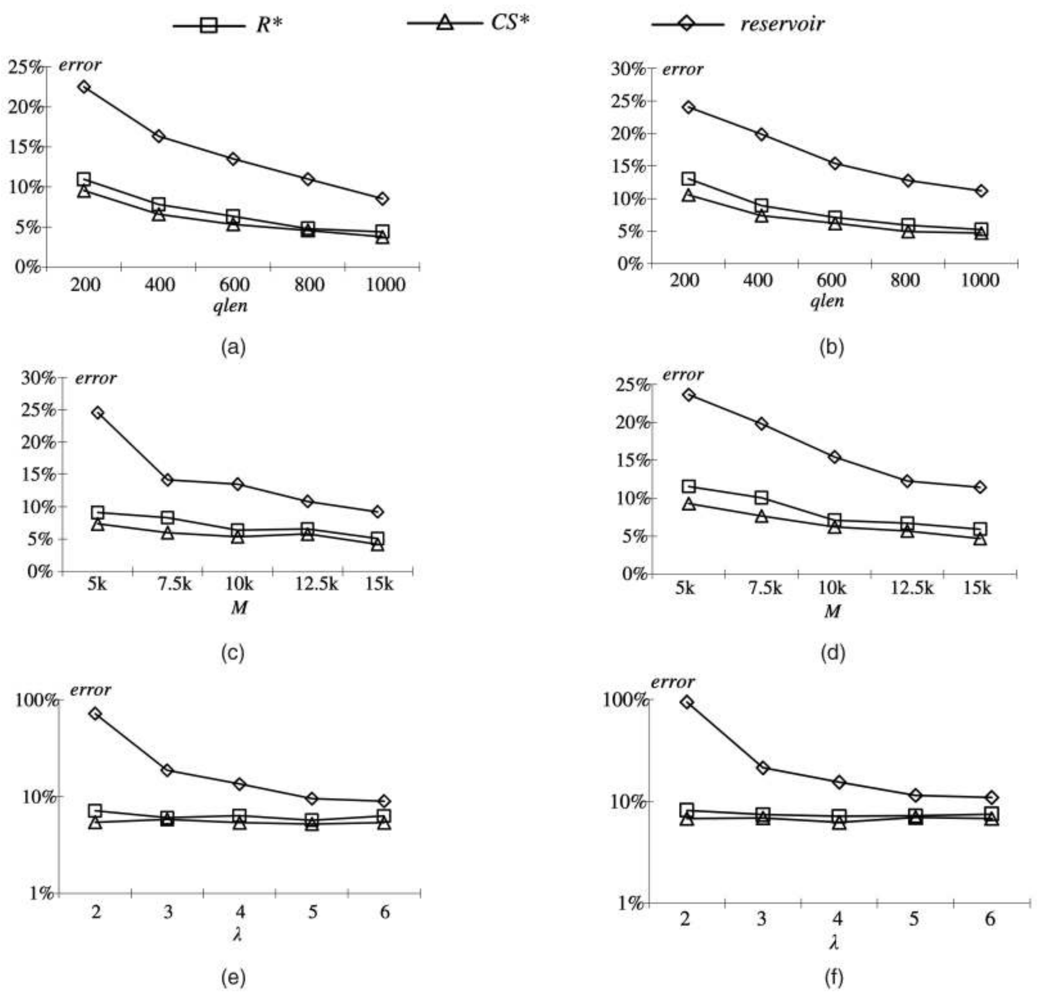

Fig. 8. Query accuracy comparison $(max\varepsilon_{80})$. (a) Error versus *qlen* (*Gau*). (b) Error versus *qlen* (*Zipf*). (c) Error versus M (*Gau*). (d) Error versus M (*Zipf*). (e) Error versus λ (*Gau*). (f) Error versus λ (*Zipf*).

Therefore, we use *reservoir* with *delete-at-will* as a benchmark since it is the only existing sampling method that incurs small per-record update overhead and can support arbitrary insertions/deletions. The following experiments compare the accuracy of R^*, CS^* and *reservoir* at the end of streams (recall that the precision of $\mathrm{R}^*/\mathrm{CS}^*$ remains relatively stable with time).

Figs. 7a and 7b illustrate the $rel\varepsilon$ of all algorithms as a function of *qlen* assuming $M = 10\mathrm{k}$ and $\lambda = 4$. Clearly, the error of $\mathrm{R}^*/\mathrm{CS}^*$ is significantly lower than that of *reservoir*. The better precision for higher *qlen* is expected as sampling is, in general, more accurate for queries with large output sizes. To study the behavior of alternative techniques under different memory constraints, Figs. 7c and 7d, measure their precision as a function of M, setting *qlen* and λ to 600 and 4, respectively. Since a larger memory accommodates more samples, the effectiveness of all algorithms increases with M.

Figs. 7e and 7f evaluate $rel\varepsilon$ for different λ, after fixing *qlen* and M to their median values 600 and 10k, respectively. For $\lambda = 2$, R^* and CS^* outperform *reservoir* by more than an order of magnitude (note the logarithmic scale). The performance gap decreases with λ, because a small number of deletions leads to sample sets with similar cardinalities for all algorithms. Fig. 8 repeats the experiments of Fig. 7 for $max\varepsilon_{80}$. In all cases, the results of $max\varepsilon_{80}$ are similar to $rel\varepsilon$ confirming the robustness of our algorithms.

In order to further verify the generality of our approaches, we also use the real data sets *Stock* and *Tickwise* [27]. The former contains the values of 197 North American stocks during the period from 1993 through 1996, and the latter consists of the exchange rates from Swiss francs to US dollars recorded from 7 August 1990 to 18 April 1991. The series *Stock* (*Tickwise*) has 512k (279k) values in total. Next, we fix M and λ to their median values 10k and 4, respectively, and measure the accuracy of all solutions at the end of the *Stock* and *Tickwise* streams. Figs. 9a and 9b compare the $rel\varepsilon$ of different algorithms as a function of *qlen* for the two distributions, respectively, whereas Figs. 9c and 9d demonstrate results of $max\varepsilon_{80}$. Clearly, R^* and CS^* again outperform *reservoir* considerably in all cases, and their behavior is similar to Figs. 7a, 7b, 8a, and 8b.

6.1.3 Update Overhead

Finally, we examine the maintenance overhead of R^* and CS^*, which, as explained in Sections 3.2 and 4.2, is dominated by the cost for maintaining $I(RS)$ (i.e., the index on the sample set). In order to provide results independent of index implementation, we measure the total number of times that $I(RS)$ is modified throughout the history (in all cases, the per-tuple cost is so small that it is not even measurable). For instance, R^*-I-ins and R^*-I-del denote the

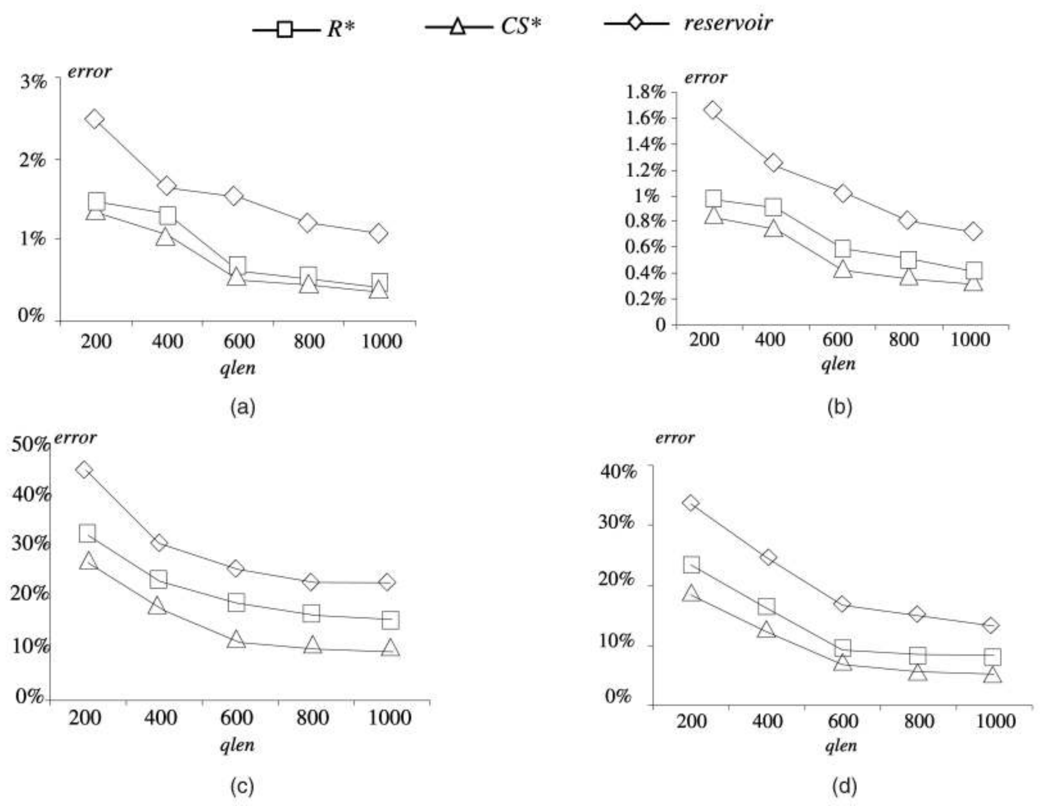

Fig. 9. Query accuracy comparison (real data). (a) $rel\varepsilon$ versus *qlen* (*Stock*). (b) $rel\varepsilon$ versus *qlen* (*Tickwise*). (c) $max\varepsilon_{80}$ versus *qlen* (*Stock*). (d) $max\varepsilon_{80}$ versus *qlen* (*Tickwise*).

number of index insertions and deletions incurred by R* for processing I-commands. R*-*D-del* corresponds to the number of index deletions for handling D-commands. Fig. 10 illustrates these numbers for R* and CS* as a function of the memory size for *Zipf* (the overhead is the same for all distributions). In accordance with Theorems 2 and 4, both methods require higher update costs to produce larger sample sets. R* incurs about 20 percent fewer modifications than CS* for each type of $I(RS)$ updates due to its smaller sample size.

6.2 Performance of Q2 Processing

Having established the effectiveness of R*/CS* for Q1 queries, we proceed to evaluate the efficiency of AJS for Q2 processing. The participating streams T_1 and T_2 contain columns (id, A, B) and (id, A, C), respectively. The domains of attributes A, B, and C are [0, 10,000]. The tuples of each relation are generated in a way similar to the previous section. Specifically, the A-values of T_1 (T_2)

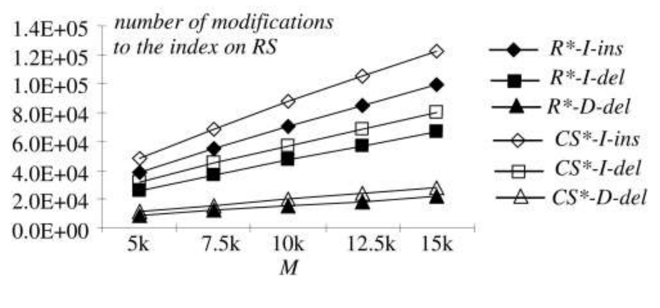

Fig. 10. Update cost comparison (Zipf).

follow the *Zipf* distribution with skew coefficients 0.8, and the $B\,(C)$ values are decided according to a *Gaussian* function with mean 5,000 and variance 2,000. The probability of receiving an I-command is $\lambda = 4$ times higher than that of a D-command. Furthermore, streams T_1 and T_2 are equally fast, i.e., the next tuple belongs to either relation with equal likelihood.

We consider two scenarios that differ in the way the A-values in T_1 and T_2 are skewed. Specifically, in the first case *skew-skew*, the A-values in both T_1 and T_2 are skewed towards 0 in the domain of [0, 10,000]. In the second case *antiskew*, data of T_1 are still skewed toward 0, while those of T_2 are towards 10,000—the dense areas of T_1 and T_2 are opposite.

Each query involves two conditions. The first one θ_{all}, common to all (1,000) queries in the same workload, has the form $|T_1.A - T_2.A| \leq len_{all}$, where len_{all} is a workload parameter. The second condition θ_{any} includes two range predicates on $T_1.B$ and $T_2.C$, extracting the tuples of T_1 and T_2 falling in these intervals, respectively. Specifically, each range has length len_{any} (another parameter), and is uniformly distributed in the universe [0, 10,000]. Queries in a workload have different θ_{any}. Similar to Section 6.1, we quantify the precision of alternative methods by their $rel\varepsilon$ and $max\varepsilon_{80}$ for processing a workload. The size of the available memory is fixed to 10k samples.

6.2.1 Tuning AJS

Recall that AJS consumes a percentage η of the memory for keeping two sample sets on T_1 and T_2, respectively. The next experiment identifies the value of η that achieves the

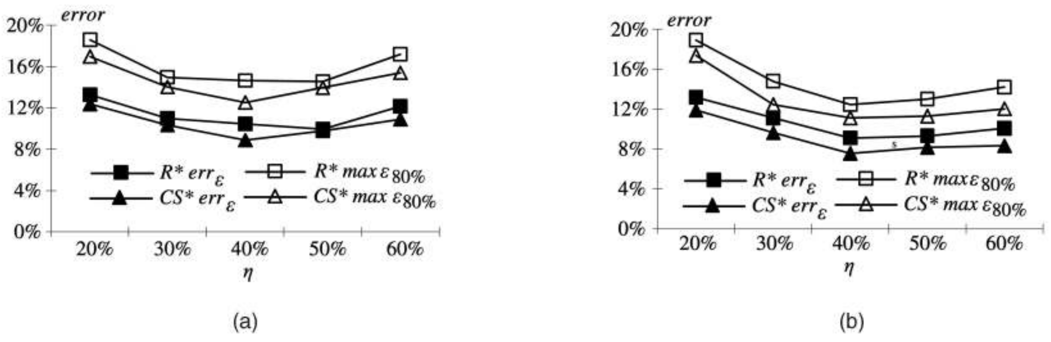

Fig. 11. AJS memory allocation tuning. (a) Error versus η (*skew-skew*). (b) Error versus η (*antiskew*).

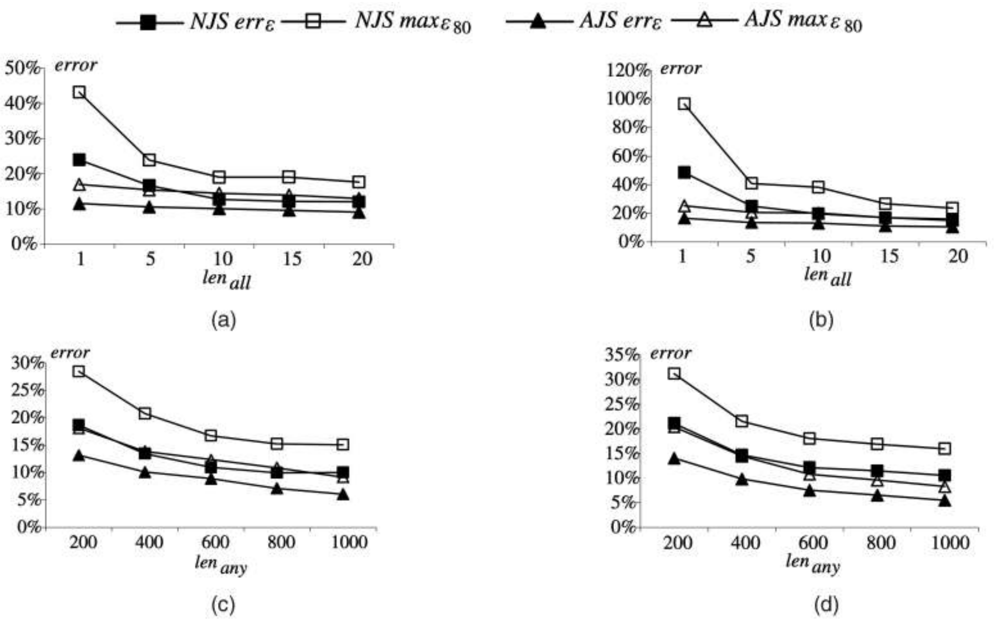

Fig. 12. Query accuracy comparison. (a) Error versus $qlen_{all}$ (*skew-skew*). (b) Error versus $qlen_{all}$ (*antiskew*). (c) Error versus $qlen_{any}$ (*skew-skew*). (d) Error versus $qlen_{any}$ (*antiskew*).

best results. Toward this, we measure the error of AJS in answering a workload with $len_{all} = 10$ and $len_{any} = 600$, after all updates of streams T_1 and T_2 have been processed. Figs. 11a and 11b demonstrates the $rel\varepsilon$ and $max\varepsilon_{80}$ for the *skew-skew* (*antiskew*) distribution, as a function of η, including both R* and CS* as the underlying sampling algorithms (recall that AJS can be integrated with any sampling technique on individual relations).

The accuracy of AJS initially improves as η increases and then deteriorates, i.e., the quality of approximation is compromised when too much or little memory is assigned to the random samples on individual relations. A very small η prevents correct estimation of the result size of $T_1 \bowtie_{\theta_{all}} T_2$ (i.e., variable w in Figs. 4 and 5), leading to biased results. On the other hand, an excessive η leaves limited space for the approximate sample set $aRS(T_{\bowtie})$ on the join results. In the sequel, we set η to 40 percent, which constitutes a good trade-off. Furthermore, we use CS* as the representative single-relation algorithm of AJS.

6.2.2 Query Accuracy.

Due to the absence of previous methods for sampling join results under tight memory footprint, we compare AJS with the NJS method (see Section 2.2), which maintains random sample sets (using CS*) on the base relations and produces approximate answers by joining the two sets. Figs. 12a and 12b compare the error of AJS and NJS as a function of len_{all}, setting len_{any} to 600. Clearly, for both *skew-skew* and *antiskew*, AJS outperforms NJS significantly in terms of accuracy and robustness. To study the influence of len_{any}, we set len_{all} to its median value 10, and repeat the above experiments by varying len_{any} from 200 to 1,000. As shown in Figs. 12c and 12d, AJS again outperforms NJS considerably.

7 Conclusions

This paper presents the first random sampling algorithms, R* and CS*, for efficiently computing aggregate data over streams of arbitrary updates. Going one step further, we propose AJS, a method for sampling join results. We prove, both theoretically and empirically, that our techniques provide accurate answers with small space and computational overhead. While our current focus is on sample sets that fit in the main memory, we plan to investigate situations where part of the sample set is migrated to the disk [21]. Further, it has been observed [6] that the performance of sampling (for approximate aggregate processing) can be improved if query statistics are available in advance. The design of such "workload-aware" methods in stream environments constitutes an interesting topic. Finally, we would like to explore the applicability of random sampling for tracking the positions of continuously moving objects [23].

ACKNOWLEDGMENTS

Yufei Tao was supported CERG Grant 1202/06, and Dimitris Papadias by CERG Grant 6184/05 from the Research Grant Council of Hong Kong.

REFERENCES

[1] S. Acharya, P.B. Gibbons, and V. Poosala, "Congressional Samples for Approximate Answering of Group-By Queries," *Proc. ACM SIGMOD Conf.*, 2000.
[2] S. Acharya, P.B. Gibbons, V. Poosala, and S. Ramaswamy, "Join Synopses for Approximate Query Answering," *Proc. ACM SIGMOD Conf.*, 1999.
[3] N. Alon, P.B. Gibbons, Y. Matias, and M. Szegedy, "Tracking Join and Self-Join Sizes in Limited Storage," *Proc. 18th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, 1999.
[4] B. Babcock, S. Chaudhuri, and G. Das, "Dynamic Sample Selection for Approximate Query Processing," *Proc. ACM SIGMOD Conf.*, 2003.
[5] B. Babcock, M. Datar, and R. Motwani, "Sampling from a Moving Window over Streaming Data," *Proc. Ann. ACM-SIAM Symp. Discrete Algorithms*, 2002.
[6] S. Chaudhuri, G. Das, and V. Narasayya, "A Robust, Optimization-Based Approach for Approximate Answering of Aggregate Queries," *Proc. ACM SIGMOD Conf.*, 2001.
[7] S. Chaudhuri, G. Das, and U. Srivastava, "Effective Use of Block-Level Sampling in Statistics Estimation," *Proc. ACM SIGMOD Conf.*, 2004.
[8] S. Chaudhuri, R. Motwani, and V. Narasayya, "On Random Sampling over Joins," *Proc. ACM SIGMOD Conf.*, 1999.
[9] G. Cormode, S. Muthukrishnan, and I. Rozenbaum, "Summarizing and Mining Inverse Distributions on Data Streams via Dynamic Inverse Sampling," *Proc. 31st Int'l Conf. Very Large Data Bases*, 2005.
[10] G. Frahling, P. Indyk, and C. Sohler, "Sampling in Dynamic Data Streams and Applications," *Proc. Ann. Symp. Computational Geometry*, 2005.
[11] S. Ganguly, M. Garofalakis, and R. Rastogi, "Processing Set Expressions over Continuous Update Streams," *Proc. ACM SIGMOD Conf.*, 2003.
[12] S. Ganguly, P.B. Gibbons, Y. Matias, and A. Silberschatz, "Bifocal Sampling for Skew-Resistant Join Size Estimation," *Proc. ACM SIGMOD Conf.*, 1996.
[13] P.B. Gibbons and Y. Matias, "New Sampling-Based Summary Statistics for Improving Approximate Query Answers," *Proc. ACM SIGMOD Conf.*, 1998.
[14] P.B. Gibbons, Y. Matias, and V. Poosala, "Fast Incremental Maintenance of Approximate Histograms," *ACM Trans. Database Systems*, vol. 27, no. 3, pp. 261-298, 2002.
[15] G. Graefe and P.-A. Larson, "B-Tree Indexes and CPU Caches," *Proc. Int'l Conf. Data Eng.*, 2001.
[16] S. Guha, C. Kim, and K. Shim, "Xwave: Approximate Extended Wavelets for Streaming Data," *Proc. 30th Int'l Conf. Very Large Data Bases*, 2004.
[17] S. Guha, K. Shim, and J. Woo, "Rehist: Relative Error Histogram Construction Algorithms," *Proc. 30th Int'l Conf. Very Large Data Bases*, 2004.
[18] P.J. Haas and C. Konig, "A Bi-Level Bernoulli Scheme for Database Sampling," *Proc. ACM SIGMOD Conf.*, 2004.
[19] P.J. Haas, J.F. Naughton, and A.N. Swami, "On the Relative Cost of Sampling for Join Selectivity Estimation," *Proc. 13th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, 1994.
[20] C. Jermaine, "Robust Estimation with Sampling and Approximate Pre-Aggregation," *Proc. 29th Int'l Conf. Very Large Data Bases*, 2003.
[21] C. Jermaine, A. Pol, and S. Arumugam, "Online Maintenance of Very Large Random Samples," *Proc. ACM SIGMOD Conf.*, 2004.
[22] K.-H. Li, "Reservoir-Sampling Algorithms of Time Complexity $o(n(1 + \log(n/n)))$," *ACM Trans. Math. Software*, vol. 20, no. 4, pp. 481-493, 1994.
[23] M.F. Mokbel, X. Xiong, and W.G. Aref, "Sina: Scalable Incremental Processing of Continuous Queries in Spatio-Temporal Databases," *Proc. ACM SIGMOD Conf.*, 2004.
[24] U. Srivastava and J. Widom, "Memory-Limited Execution of Windowed Stream Joins," *Proc. 30th Int'l Conf. Very Large Data Bases*, 2004.
[25] N. Tatbul, U. Cetintemel, S.B. Zdonik, M. Cherniack, and M. Stonebraker, "Load Shedding in a Data Stream Manager," *Proc. 29th Int'l Conf. Very Large Data Bases*, 2003.
[26] J.S. Vitter, "Random Sampling with a Reservoir," *ACM Trans. Math. Software*, vol. 11, no. 1, pp. 37-57, 1985.
[27] E. Keogh, The UCR Time Series Data Mining Archive, Univ. of California—Computer Science & Eng. Dept., http://www.cs.ucr.edu/~eamonn/TSDMA/index.html, 2006.

Yufei Tao received the PhD degree in computer science from the Hong Kong University of Science and Technology in 2002. After that, he spent a year as a visiting scientist at the Carnegie Mellon University. From 2003 to 2006, he was an assistant professor at the City University of Hong Kong. Currently, he is an assistant professor in the Department of Computer Science and Engineering, the Chinese University of Hong Kong. Professor Tao is interested in research of spatial, temporal databases, data privacy, and security. He is the winner of the Hong Kong Young Scientist Award 2002. He has published extensively in SIGMOD, VLDB, ICDE, the *ACM Transaction on Database Systems*, and the *IEEE Transactions on Knowledge and Data Engineering*.

Xiang Lian received the BS degree from the Department of Computer Science and Technology, Nanjing University, in 2003. He is currently a PhD candidate in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. His research interests include spatio-temporal databases and stream time series.

Dimitris Papadias is a professor in the Computer Science and Engineering Department, Hong Kong University of Science and Technology (HKUST). Before joining HKUST in 1997, he worked and studied at the German National Research Center for Information Technology (GMD), the National Center for Geographic Information and Analysis (NCGIA, Maine), the University of California at San Diego, the Technical University of Vienna, the National Technical University of Athens, Queen's University (Canada), and the University of Patras (Greece). He has published extensively and been involved in the program committees of all major database conferences, including SIGMOD, VLDB, and ICDE. He is an associate editor of the *VLDB Journal*, the *IEEE Transactions on Knowledge and Data Engineering*, and on the editorial advisory board of *Information Systems*.

Marios Hadjieleftheriou received the diploma in electrical and computer engineering in 1998 from the National Technical University of Athens, Greece, and the PhD degree in computer science from the University of California, Riverside, in 2004. He has worked as a research associate at Boston University. Currently, he is working for AT&T Labs—Research. His research interests are in the areas of spatio-temporal data management, data mining, data stream management, time-series databases, and database indexing.