A Stratified Approach to Progressive Approximate Joins

Wee Hyong Tok, Stéphane Bressan, Mong-Li Lee
School of Computing
National University of Singapore
{tokwh,steph,leeml}@comp.nus.edu.sg

ABSTRACT

Users often do not require a complete answer to their query but rather only a sample. They expect the sample to be either the largest possible or the most representative (or both) given the resources available. We call the query processing techniques that deliver such results 'approximate'. Processing of queries to streams of data is said to be 'progressive' when it can continuously produce results as data arrives. In this paper, we are interested in the progressive and approximate processing of queries to data streams when processing is limited to main memory. In particular, we study one of the main building blocks of such processing: the progressive approximate join. We devise and present several novel progressive approximate join algorithms. We empirically evaluate the performance of our algorithms and compare them with algorithms based on existing techniques. In particular we study the trade-off between maximization of throughput and maximization of representativeness of the sample.

1. INTRODUCTION

In data stream applications [3, 2, 4] the amount of data to be processed is generally much larger (potentially infinite) than the available memory. This is particularly true for applications whose processing is running on devices such as handheld computers, for instance. The progressive production of results therefore requires query processing algorithms that can make the best use of main memory and utilize secondary storage cleverly. Representative of this family of algorithms are the progressive joins such as XJoin [19], RPJ [17], HMJ [14], and our own RRPJ [18].

In addition, in many such applications, users are so concerned with rapid production of results that they are ready to give up completeness of the result. In this case, users may prefer results that can be produced in main memory only. In other words, users often do not require a complete answer to their query but rather only a sample. They expect the sample to be either the largest possible -they favor quantity-, the most representative -they favor quality- or both? They may need to seek a compromise between quality and quantity-, given the resources (main memory) available. We call the query processing techniques that deliver such results 'approximate'.

Join algorithms being the keystones of query processing, we are interested here in progressive approximate join algorithms. The reference progressive approximate join is *Prob* introduced in [7] and its extended version [8]. The authors introduce the notion of maximum subset (MAX-Subset) which leads to similar strategies as the ones used by progressive algorithms such as RPJ [17] and RRPJ [18] to maximize the size of the set of results produced, quantity. We show that the performance of *Prob* can be improved by stratifying the memory available. We propose *ProbHash*, a direct extension of *Prob*, in which the memory is hash partitioned and an approximate version of our progressive algorithm RRPJ also using hash partitioning. Interestingly, the authors of [7] have disqualified reservoir sampling based methods based on the extreme scenario given in [5] without further experiments. We show that this disqualification is mistaken. We propose a reservoir sampling-based approximate progressive join, that we call Reservoir Approximate Join (RAJ), and its stratified version $RAJHash$. We show that these algorithms favor the representativeness of the set of results produced and ensure better quality than the other algorithms.

The rest of the paper is organized as follows. In Section 2, we discuss related work. We discuss the notions of quantity and quality of results produced in Section 3. We present the proposed algorithms in Section 4. In Section 5, we empirically evaluate the performance of our four algorithms and compare them with *Prob*. In particular we highlight and discuss the trade-off between quantity and quality. We conclude and discuss future work in Section 6.

2. RELATED WORK

In the data stream literature, various approximate query processing techniques have been proposed for aggregation queries (e.g. *quantile* [10], *heavy hitters* [13] and *distinct counts* [9]) and join queries [7, 8, 1]. In this paper, we focus on progressive, approximate join queries where the results are streamed out to the user as soon as they are produced.

2.1 Progressive Joins

Progressive relational equi-join algorithms [17, 18] studied the problem of producing complete results over data streams. In order to work with limited memory these algorithms need to flush tuples to disk whenever memory is

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT'08, March 25–30, 2008, Nantes, France.
Copyright 2008 ACM 978-1-59593-926-5/08/0003 ...$5.00.

full. These disk-resident tuples are then joined at subsequent phases in order to produce the complete join results. The goal of these algorithms is to maximize results production, as well as ensure high result throughput during join processing. If we retain only the in-memory processing phase, these algorithms are suitable for approximate join processing. In this paper, we modify one of the state-of-art progressive relational equi-join algorithm, RRPJ [18], and show how it can be used for progressive approximate join processing.

2.2 Approximate Joins

In conventional databases, several techniques [16, 15, 5] have been proposed for approximate join processing. These techniques construct a fixed size random sample of the results for a relational query. The underlying assumption is that indices are available for one or both of the datasets, or statistics on the data distribution known apriori. [16, 15] assumes that indexes are available to facilitate efficient random access to the data. Given two relations S_1 and S_2, it randomly chooses a tuple t_{S1} from S_1, and determines whether t_{S1} should be included into the sample by computing its inclusion probability. If t_{S1} is included, a tuple t_{S2}, with the same join attribute, is then randomly chosen from S_2 and joined with t_{S1}. Noting this, [5] proposed a generalized technique for sampling the results of join queries which do not require indices to be pre-constructed. In addition, Surajit et al. noted that for skewed data distributions, the random sampling of results from join queries could cause a worst-case scenario in which no join results are available. None of these works deal with data streams. In our work, we show the impact of the worst-case scenario for data stream processing.

2.3 Progressive Approximate Joins

Several progressive approximate join algorithms [7, 8, 1] have been proposed for data stream applications. In [7] and its extended version [8], the motivation was the maximization of the result subset produced. A reference theoretical algorithm, called *OPT-offline*, was proposed. The algorithm presents an optimal scenario in which the *MAX-subset* error measure is minimized. They cannot be used for online applications. In order to deal with the online case, two heuristics, *PROB* and *LIFE*, were proposed to maximize the expected output size. The focus of the work was on maximizing the result output size of the approximate join, and assumes the availability of a fast CPU for join processing. Given two streams, S_1 and S_2, the priority of a tuple from S_1 is computed based on the arrival probability of tuples from S_2. Priority queues are used for storing the in-memory tuples. Whenever a tuple from S_1 arrives, it will need to scan the entire priority queue of S_2 (and vice-versa). Our work differs in two aspects. Firstly, we show how auxiliary data structures (i.e. hash-based priority queues) can be used to minimize the need to scan all the tuples in memory. Secondly, we show that maximizing the output size of the result does not necessarily ensure good result quality. We quantify the notion of result quality, and propose a technique that is able to deliver good-quality results progressively. Though [1] also studied the use of reservoir sampling over memory-limited join, the focus of the work was on how to balance between the memory allocated for join buffers and the reservoir. In addition, [1] do not progressively output results.

3. MEASURING PERFORMANCE

3.1 What do We Measure?

There are two ways to measures the performance of an approximate algorithm. If we are interested in quantity, the measure of performance for the algorithm is the amount of results produced. If we are interested in quality, we need to measure the similarity between the data distribution of the complete set of results and the data distribution of the set of results produced. Because we are interested in progressive algorithms, performance is not a unique number but a function of time. It is measured in term of throughput, quantity over time, when size matters. It is measured in terms of quality over time (quality throughput), when quality matters. If both quantity and quality matter, we need both functions. Notice that the comparison of both functions by looking at quantity as a function of quality (or vice versa) at given points in time visualizes the compromise realized by a given algorithm.

We considered defining a combined measure of quantity and quality (similarly to the F-measure, which combines recall and precision). Unfortunately, our measure of quality using JS Divergence or any comparable statistical measures is unbounded, and cannot be normalized.

3.2 How do We Measure Quality?

In order to measure quality, we need to compare two data distributions. We can compute, combine and compare any statistics and obtain more or less significant measurements at different level of granularity.

A reasonable yet simple metrics is the Mean-Square Error (MSE) between the normalized histograms of the complete results and result produced by the approximate join. We, however choose a slightly more accurate measurement with the Jensen-Shannon divergence [12], which determines the similarity (or divergence) between two probability distributions.

3.2.1 MSE

We first discuss the MSE measure. The MSE measure measures the error differences between the actual and observed results produced. In the approximate join scenario, the actual results refer to the results produced if the entire join is computed (or when the memory is unlimited and all data fit into main memory). The observed results refer to the results produced by the approximate join method. In order to ensure a fair comparison between the actual and observed result distribution, we compare the normalized frequency instead of the actual frequency for each join attribute value. Let the total number of results produced by the complete and approximate join be |R| and |R'| respectively. For each value $v_i \in V$, where V denotes the domain of the join attribute, and $1 \leq i \leq$ |V|. $|v_i|$ and $|v_i^{'}|$ denotes the number of actual and observed results with value v_i. For each join attribute v_i, the normalized value for the complete and approximate joins is given by $\frac{|v_i|}{|R|}$ and $\frac{|v_i^{'}|}{|R'|}$ respectively. The MSE between the complete join J and approximate join J' is given by

$$MSE(J, J') = \sum_{i=1}^{V} (\frac{|vi|}{|R|} - \frac{|v_i^{'}|}{|R'|})^2 \qquad (1)$$

3.2.2 *Jensen-Shannon Divergence*

In probability and information theory, the Kullback Leibler (KL) and Jensen-Shannon divergence are used to measure the similarity between two probability distributions, P and Q. We use the Jensen-Shannon divergence to measure the similarity between the actual (P) and observed result (Q) distribution. We measure the result quality produced by the approximate join using the Jensen-Shannon divergence. The Jensen-Shannon divergence measures the similarity between the actual result distribution (produced by a join where all tuples fit in memory) and the approximate join result distribution. Let p(v_i) = $\frac{|v_i|}{|R|}$ and q(v_i) = $\frac{|v_i^{'}|}{|R'|}$. Before defining the Jensen-Shannon divergence, we first define the KL divergence, given as follows:

$$D_{KL}(P||Q) = \sum_{i=1}^{V} p(v_i)\log(p(v_i)/q(v_i)) \quad (2)$$

The Jensen-Shannon divergence is given by

$$D_{JS}(P||Q) = \tfrac{1}{2}D_{KL}(P||M) + \tfrac{1}{2}D_{KL}(Q||M) \quad (3)$$

where M = $\frac{1}{2}(P+Q)$

The goal is to minimize either the MSE or the JS divergence. When the value for either MSE or JS divergence is zero, the result distributions from the complete and approximate joins are exactly the same.

Given two approximate join methods, J_1 and J_2, we say that J_1 produces better quality results than J_2 if the *QMeasure*(J_1) < *QMeasure*(J_2). *QMeasure*(Z) refers to either computing MSE(Z) or D_{JS}(Z). Z refers to an arbitrary approximate join method.

4. SOLUTION

In this section, we describe five methods for performing approximate joins: (1) *Approximate RRPJ* (ARRPJ), (2) *Prob*, (3) *ProbHash*, (4) *Reservoir Approximate Join* (RAJ) and (5) *Stratified Reservoir Approximate Join* (RAJHash).

We first present the key idea for an existing progressive approximate join algorithm, *Prob*. Next, we propose the modification of an existing progressive join algorithm for approximate join processing, called *Approx-RRPJ*. Lastly, we propose three new algorithms (*ProbHash*, *RAJ* and *RAJHash*). *ProbHash* aims to maximize the result quantity as well as improve the overall throughput. Both *RAJ* and *RAJHash* are designed to optimize the result quality.

4.1 Approximate Join Framework

We first discuss a general framework for designing approximate join algorithms which explore the tradeoffs between result quantity and quality.

Given two streams S_1(A,B) and S_2(B,C), where A, B and C are attributes of the data streams. Let the *i-th* tuple from S_1 and the *j-th* tuple from S_2 be denoted by $t_{S1}(a_i,b_i)$ and $t_{S2}(b_j,c_j)$ respectively. An approximate join is used to join the tuples from the two streams. The size of the memory available for query processing is small relative to the size of the data streams, which can be unbounded. When a new tuple arrives and memory is full, we will need to selectively discard some tuple(s) from memory. Indeed, an important design criteria for an effective approximate join algorithm is how tuples are discarded.

We first consider approximate join algorithms which maximizes the quantity of the results produced. We call such a algorithm DP_X(k), which discards k tuples whenever memory is full. The goal of the DP_X(k) policy is to maximize the expected size of the result subset. To achieve this, we can model the probability of the join attribute value(s) for tuples arriving on both streams. Let the arrival probabilities be P_{S1}(B) and P_{S2}(B) for streams S_1 and S_2 respectively. Whenever a tuple arrives, we assign a priority to the tuple based on the arrival probabilities from the corresponding stream. For example, when a tuple $t_{S1}(a_i,b_i)$ arrives, its priority value is given by $P_{S2}(b_i)$. Similarly, the priority of a tuple $t_{S2}(b_j,\ c_j)$ can be computed using $P_{S2}(b_j)$. A possible implementation for DP_X(k) is to maintain two priority queue (in ascending priority order) for the data streams. Whenever memory is full, DP_X(k) discards the first k tuples taken from both streams. The intuition is that by keeping in memory tuples which have higher probability of joining with tuples from the other stream, the expected number of results produced will be maximized [17].

Next, we consider approximate join algorithms which are sampling-based. The goal is to optimize the quality of the results produced. We call such a algorithm DP_Y. DP_Y continuously maintains a random uniform sample for each of the data streams. When the memory is not full, tuples are inserted into the respective reservoirs. When memory is full, DP_Y determines whether the newly arrived tuple should be discarded, or be used to replace a tuple from the reservoir. Suppose the size of the memory is M, which is divided equally between the two streams S_1 and S_2. Suppose n_{S_1} and n_{S_2} tuples have arrived for stream S_1 and S_2 respectively. We assume that the number of tuples that have arrived for each stream is greater than the available allocated memory (i.e. n_{S_1} > (M/2), and n_{S_2} > (M/2)). A newly arrived tuple $t_{S1}(a_i,b_i)$ has a $\frac{(M/2)}{n_{S_1}}$ chance of being used to replace a tuple in the reservoir. Similarly, for a tuple from S_2. Even though DP_Y might not maximize the number of results produced, the quality of the results produced could be much better than DP_X(k). This is because DP_Y ensures that the uniformity of the samples for each of the data streams. When a new tuple arrives, it is used to probe the corresponding reservoir. Mindful readers might note that DP_Y might not work well for skewed data streams if the memory is allocated equally between the two reservoirs. In our work, we show how we can tackle this problem by dynamically allocating memory for the reservoirs.

4.2 Approximate RRPJ (ARRPJ)

The Result-Rate Based Progressive Join (RRPJ) [18] was proposed as a progressive join algorithm. It builds statistics on the result distribution of the hash partitions. The goal of RRPJ is to maximize the number of results produced by using the result distribution statistics to determine the non-productive tuples to be flushed to disk whenever memory is full. In RRPJ, when all the tuples have arrived, a cleanup phase is invoked to compute the complete results for the join query.

In order to build a progressive approximate join, we modify RRPJ so that it consists of the in-memory processing phase. We call this join algorithm, *Approximate RRPJ (ARRPJ)*. Whenever memory is full, ARRPJ flushes tuples from memory. The tuples are discarded instead of being flushed to disk partitions.

4.3 Prob

The $PROB$[7, 8] approximate join is an instantiation of $DP_X(1)$. The goal of $PROB$ is to maximize the quantity of results produced. It assigns a priority to each tuple that arrives. *Prob* can make use of either a fixed or variable memory allocation to store tuples from each of the data streams. For fixed allocation, two priority queues are used, one for each of the data streams. For variable allocation, a single priority queue is used for both streams. The priority for a tuple is determined by the arrival probabilities of the partner stream. We describe how *Prob* works. Given two streams S_1 and S_2, a memory size M. Two priority queues, PQ_1 and PQ_2, (one for each stream) are created. Using a fixed memory allocation, the size of each priority queue is $\frac{M}{2}$. In order to deliver results progressively, a probe-and-insert paradigm is used. When a tuple t_{S1} arrives, it needs to probe all the tuples in the PQ_2 in order to determine join matches. Similarly, when a tuple t_{S2} arrives, it needs to probe all the tuples in PQ_1 for join matches. At time τ, given that $|S_1|$ and $|S_2|$ tuples have arrived for S_1 and S_2 respectively. Using a variable memory allocation scheme, the size of the single priority queue is M. Whenever tuples arrive from either stream, it will have to scan all the tuples in the priority queue. The time complexity for both the fixed and variable memory allocation is given by O(M($|S_1|$ + $|S_2|$)).

4.4 ProbHash

In order to reduce the need to probe all in-memory tuples, we propose a progressive join algorithm, *ProbHash*. *ProbHash* relies on hash partitions to organize the in-memory tuples. In essence, *ProbHash* is a CPU-efficient extension of *Prob* [7, 8].

ProbHash organizes the in-memory tuples for each stream by storing the tuples using p priority queues, instead of a single priority queue. The value of p is dependent on the hash function used. The tuples in each priority queue are organized based on a ascending priority order. We denote the set of priority queue for data stream S_i as PQ_{S_i} ($1 \leq i \leq 2$). Figure 1 shows the two sets of priority queues. Whenever a tuple t_{S1} arrives, its hash value is computed by the hash function (denoted by $\oplus$). It is then used to probe one of the priority queues in PQ_{S_2}. If join matches are found, the result is delivered to the user. Afterwhich, t_{S1} is inserted to one of the priority queues of PQ_{S_1}. The set of priority queues, PQ_{S_1} and PQ_{S_2}, are each allocated $\frac{M}{2}$ memory. Within each priority queue set, we make use of a variable memory allocation scheme which allows the size of the priority queues to grow or shrink dynamically. This mitigates the effect of skewed data distribution, and ensure that the memory can be better utilized. Suppose the average length of each priority queue is L (L << M), the time complexity for *ProbHash* is given by O(L($|S_1| + |S_2|$)).

When memory is full ($|S_1| + |S_2| =$ M), and a new tuple arrives, we will need to select a tuple to be discarded from amongst the $2p$ priority queues. We first identify the priority queue PQ_i ($1 \leq i \leq 2p$) which contains the tuple with the smallest priority value. The complexity for finding the queue which contains a tuple with the smallest priority value is given by *O(p)*. This is because we only need to scan the first element of each of the $2p$ priority queues. In the case of a tie (i.e several queues with tuples having the smallest priority value), we randomly pick a tuple from one of these

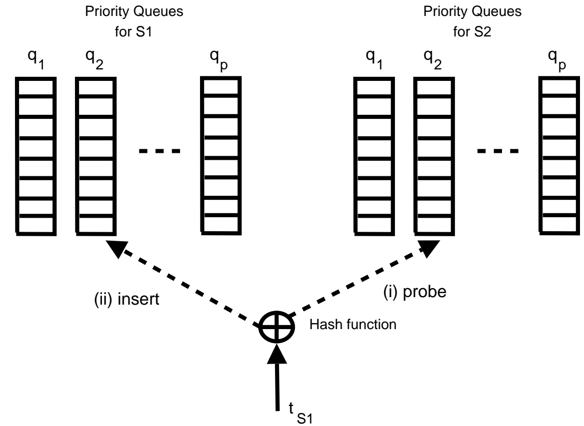


Figure 1: Priority Queue for S_1

queues. Other methods can be used too (e.g. the tuple's age and preferring tuples that are older). We dequeue the tuple with lowest priority. We then compute the hash value for the newly arrived tuple, which is used to determine the priority queue it is inserted into. Due to the variable memory allocation, it is important to note that the size of all the priority queues are not fixed. Hence, if the data distribution is skewed, some priority queues will be longer.

4.5 Reservoir Approximate Join (RAJ)

Conventional reservoir sampling [20] is used to produce a fixed size random sample of data. Algorithm 1 describes the details. While data is arriving (line 2), we get the next tuple from the data stream S (line 3). n denotes the total number of tuples that have arrived so far. If the number of tuples in the reservoir is less than the reservoir size |R|, we insert the tuple into the reservoir (line 5 to 6). Otherwise, the tuple is inserted into the reservoir with probability $\frac{|R|}{n}$ (line 8 to 10).

Algorithm 1: Conventional Reservoir Sampling

```
   Data   : R - Reservoir, |R| - Reservoir Size,
            S - Data Stream
            n - Total numbers tuples that has been inserted
            into R
   begin
1      n = 0 ;
2      while ( !endOfStream(S) ) do
3          Tuple t = getNextTuple(S) ;
4          n = n + 1 ;
5          if ( n < |R| ) then
6              Insert t into R ;
           end
7          else
8              Randomly generate a number ρ between 1 to n ;
9              if ( ρ < |R| ) then
10                 Replace the ρ-th tuple in R with t ;
               end
           end
       end
   end
```

Conventional reservoir sampling can also be used in a progressive approximate join. We call this the Reservoir Approximate Join (RAJ). This is illustrated in Figure 2. Given two streams S_1 and S_2, and memory with size M. Two

reservoirs, $Reservoir_{S1}$ and $Reservoir_{S2}$ are created. Each reservoir is allocated $\frac{M}{2}$ memory. For each reservoir, the conventional reservoir sampling technique is used to manage the reservoir. When a tuple t_{S1} arrives, it is used to probe $Reservoir_{S2}$. Results (if any) are produced. Afterwhich, t_{S1} is inserted into $Reservoir_{S1}$. The problem with this approach is that the entire reservoir needs to be scanned in order to find tuples which can be joined with the newly arrived tuple.

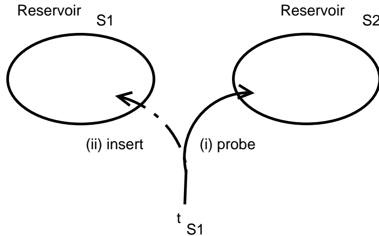


Figure 2: Reservoir Approximate Join

4.6 Stratified Reservoirs Approximate Join (RAJHash)

In statistics, stratified sampling [6] is another effective technique for sampling from a population. In stratified sampling, the population is divided into disjoint k sub-populations of sizes N_1, N_2,...,N_k respectively. Each sub-population is called a stratum, and is mutually exclusive (i.e. every element in the population must be assigned to only one stratum). Hashing is an effective way to assign each element to exactly one stratum. In order to reduce the need to scan the entire reservoir during probing, we adopt the idea of stratified sampling to organize the reservoir for each stream into multiple sub-reservoirs. We call this algorithm the Stratified Reservoirs Approximate Join *(RAJHash)*.

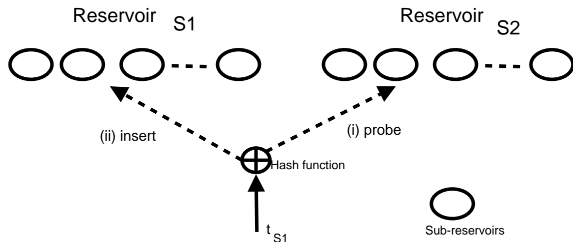


Figure 3: Progressive Approximate Join using Stratified Reservoirs

In the stratified reservoir approach, we allocate $\frac{M}{2}$ memory to each reservoir. Each reservoir consists of k sub-reservoirs. For each reservoir, a variable memory allocation scheme is used to allocate memory for the sub-reservoirs. Given a tuple t, the hash function, f(t) = t.value mod k, is used to assign the tuple to one of the sub-reservoirs. t.value denotes the value of the join attribute. Algorithm 2 describes the insertion of a newly-arrived tuple using the stratified reservoir. In Line 1, h denotes the hashed value of the tuple. If n is less than |R|, then we will just add the tuple to the *h-th* sub-reservoir (Line 4). If n is greater or equal to |R|, then we will need to determine whether to replace a tuple from the reservoir with the newly arrived tuple (Line 6-10). To do this, a random number, ρ (between 1 to n) is generated. If ρ is greater than |R|, we discard t. Otherwise, t is used to replace a tuple from the *h-th* sub-reservoir. In this case, even though ρ is less than |R|, ρ can be greater than the size of the *h-th* sub-reservoir. To find the tuple to be replaced, we compute i = ρ mod S (where S is the size of the *h-th* sub-reservoir). We then replace the *i-th* tuple in the *h-th* sub-reservoir.

Algorithm 2: Stratified Reservoir - Inserting a tuple

```
   Data    : R - Reservoir, |R| - Reservoir Size,
             k - Number of sub-reservoirs
             t - Tuple to be inserted
             n - Total numbers tuples that has been inserted
             into R
   begin
1      h = f(t) ;
2      n = n + 1 ;
3      if ( n < |R| ) then
4          Insert t into the h-th sub-reservoir ;
       end
       else
5          Randomly generate a number ρ
           between 1 to n (inclusive);
6          if ( ρ < |R| ) then
7              S = Get the size of the h-th sub-reservoir ;
8              i = ρ mod S ;
9              Replace the i-th tuple with t ;
           end
           else
10             Discard t ;
           end
       end
   end
```

RAJHash introduces some advantages over *RAJ*. Firstly, it is more CPU-efficient as it reduces the number of in-memory tuples that are probed to identify join matches. Secondly, even in the presence of a skewed distribution, it is able to gracefully allocate more memory for sub-reservoirs which need a large sample, and less memory for sub-reservoirs which contain the skewed values. This is due to the variable memory allocation for the sub-reservoirs. We empirically verify this in Section 5.3.

4.6.1 Example

In this example, we illustrate how Stratified Reservoir works. Given two streams S_1={10, 22, 34, 11, 30, 90, 2, 1, 13, 10} and S_2={ 10, 48, 20, 35, 12, 58, 67, 71, 44, 83 }. In this example, the size of the memory M = 10 tuples. Two reservoirs $Reservoir_{S1}$ and $Reservoir_{S2}$ are created for S_1 and S_2 respectively. Each reservoir can hold 5 tuples. In addition, each reservoir is allocated 10 sub-reservoirs. The hash function *f(t) = t.value mod 10* is used to allocate a tuple to one of the 10 sub-reservoirs. We denote a sub-reservoir for stream S_i as $reservoir_i^j$ ($0 \leq$ j < 9) respectively.

For stream S_1, the first tuple arrives. This is inserted into $reservoir_1^0$. Next, a tuple from S_2 arrives. This is first used to probe $Reservoir_{S1}$, which in turn re-directs it to sub-reservoir $reservoir_1^0$ which produces a result. After 5

tuples have arrived from each of the data streams, we have the following $reservoir_1^0$ = {10, 30}, $reservoir_1^1$ = {11}, $reservoir_1^2$ = {22}, $reservoir_1^4$ = {34}, $reservoir_2^0$ = {10, 20}, $reservoir_2^2$ = {12}, $reservoir_2^5$ = {35} and $reservoir_2^8$ = {48},. When the sixth tuple from S_1 arrives, $Reservoir_{S1}$ is full. We need to decide whether to discard a tuple from $Reservoir_{S1}$. First, we compute the hash value of the sixth tuple to be 0 (i.e. 90 mod 10). To determine whether to discard the tuple, we randomly generate a number ρ between 1 to 6 (inclusive). If $\rho \leq 5$, then we will replace a tuple in the sub-reservoir $reservoir_1^0$ with this newly arrived tuple. Suppose the value of ρ is 4. It is important to note that there are only two tuples in $reservoir_1^0$. To determine which tuple to be replaced, we compute 4 mod 2 = 0. Thus, the first tuple (value=10) is then replaced with the newly arrived tuple. Thus, sub-reservoir $reservoir_1^0$ = {90, 30}. Similarly, when the sixth tuple (value = 58) from S_2 arrives, we need to decide whether to discard or replace a tuple from $reservoir_2^8$. We generate a random number, ρ between 1 to 6 (inclusive). Suppose rho = 6. Thus, we discard the newly arrive tuple. Thus, sub-reservoir $reservoir_2^8$ = {48}.

4.7 Discussion

Approx-RRPJ, *Prob* and *ProbHash* attempt to maximize the quantity (i.e. number of results produced) by sacrificing tuples that do not produce or produce few results. Therefore, they tend to favour results in certain ranges. In contrast, *RAJ* and *RAJHash* strive to maintain a good representative sample. With limited memory, an approximate join algorithm need to effectively make use of the available memory, balancing between quantity and quality of the results produced.

5. PERFORMANCE ANALYSIS

In this section, we perform an extensive performance study, using both synthetic and real-life datasets. 5 algorithms are used in the performance study: (1) *ARRPJ* (2) *Prob* (3) *ProbHash* (4) *RAJ* and (5) *RAJHash*. We implemented all the progressive approximate algorithms in C++, and conduct the experiments on a Pentium 4 2.4 Ghz PC (1GB RAM).

We evaluate the approximation performance by: (1) Visualizing the quality of the results using normalized result histograms, (2) Measuring the percentage of results (Quantity) and ,(3) Measuring the JS Divergence (Quality). We have also conducted experiments to measure the quality of the results using MSE. As the results using MSE show similar trends to JS Divergence, the results are omitted. We also studied the effects of varying memory sizes. In addition, we also studied the throughput and quality throughput of the various progressive approximate join algorithms by taking snapshots of the result distribution at different time epochs.

The experiment parameters are given in Table 1. When the memory allocated to the approximate join is 100%, all tuples fit in memory. Hence, the complete set of join results are produced. We refer to this method as *EXACT*, which we use as a benchmark for comparison for result quality and computing the percentage of results produced by each method.

5.1 Effect of Skewed Distribution

In this experiment, we investigate the performance of the various methods in the presence of a skewed distribution.

Table 1: Experiment Parameters

Parameter	Values
Memory allocated to approximate join, M	Varies between 10% to 100%
Datasets (DS, Dataset Size)	Skewed (100,000 tuples)
	Extreme (100,000 tuples)
	Real-life: Weather (2,074,948 tuples)

The skewed distribution is generated as follows: The frequency of the join attribute values is determined by a Zipfian distribution. The skewness of the Zipfian distribution is determined by a factor η. We set η to be 1.0. We vary the memory allocated to be 10% to 100% of the dataset size (100,000 tuples). The domain of the join attribute value is set to be 1-50.

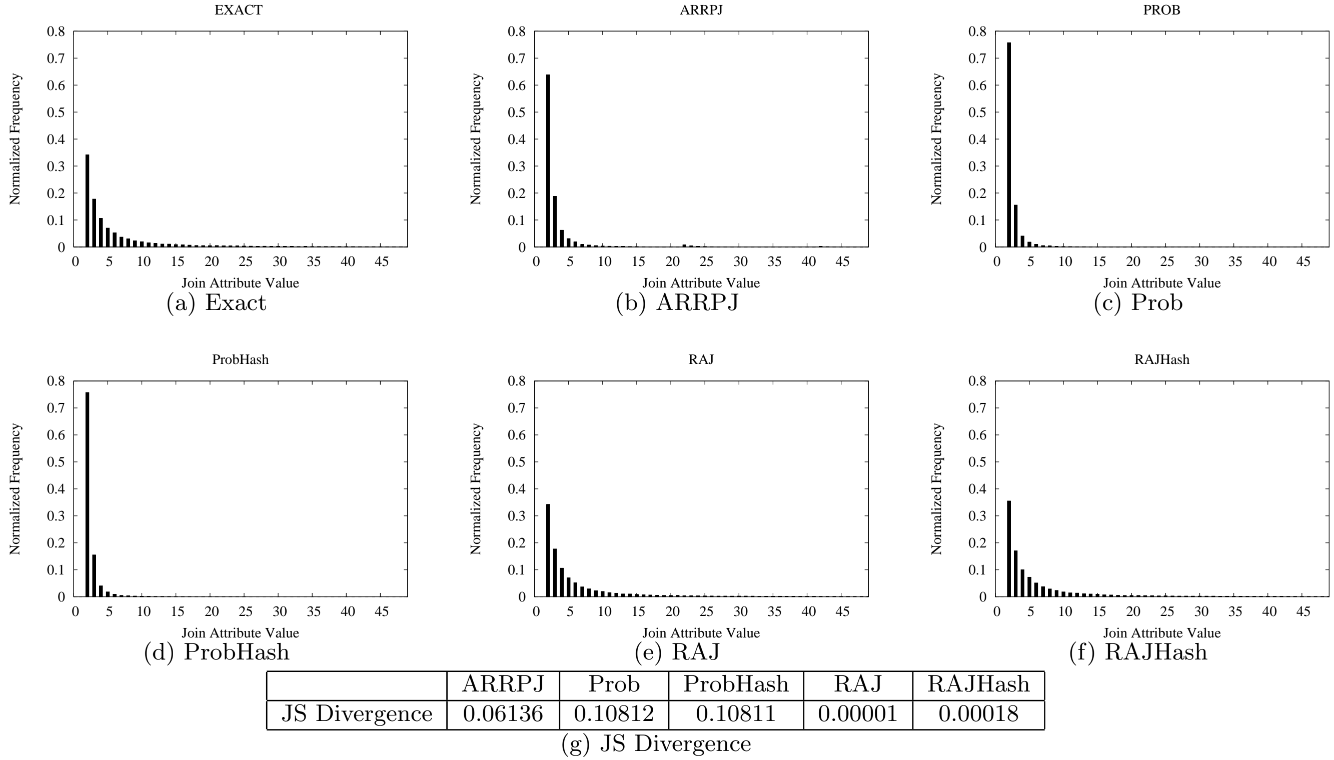
5.1.1 Approximation Performance

We first study the performance of the algorithm w.r.t to approximation only. Therefore, we consider bounded input streams, and look at the quantity and quality after all the tuples have been processed.

The goal of the first experiment is to visualize the quality of the results produced by the various progressive approximate join algorithms. We fixed the memory allocated to the approximate join to be 10% of the dataset. To achieve this, we plot the result histograms for each of the algorithms. In the y-axis, we show the normalized frequency for each join attribute value. Given the number of results produced by an approximate join method J is $|J|$. The number of results with join attribute value v is given by |v|. The normalized frequency is defined as $\frac{|v|}{|J|}$. In the x-axis, we plot the value of the join attributes.

From Figure 4(a)-(c), we can observe that *ARRPJ*, *Prob*, and *ProbHash* favor the production of the most likely results. Hence, the results that are produced are skewed towards the join attribute values that appear more frequently. From Figure 4(e) and (f), it is visually striking that the normalized histograms for *RAJ* and *RAJHash* are almost identical to the distribution of the complete results (Figure 4(a)). For a quantitative comparison of the result quality, we also show the JS Divergence for the various algorithms in Figure 4(g). Thus, we can conclude the quality of results produced by *RAJ* and *RAJHash* is higher than that produced by *ARRPJ*, *Prob*, and *ProbHash*.

In the second experiment, we vary the amount of memory allocated to the progressive approximate join. The x-axis shows the amount of memory allocated as a percentage of the total dataset size. The y-axis shows the percentage of results produced and the JS Divergence respectively for Figure 4(h) and (i). From the figures, we can observe that the quantity and quality improves as the amount of memory allocated increases. In addition, it is consistently observed in all the algorithms. From Figure 4(h), we can observe that the number of results produced by *RRPJ*, *Prob* and *ProbHash* is significantly more than *RAJ* and *RAJHash*. However, from Figure 4(i), we can observe that the JS-divergence of *RAJ* and *RAJHash* is much lower. As noted in Section 4.7, with limited memory, there is always a tradeoff between

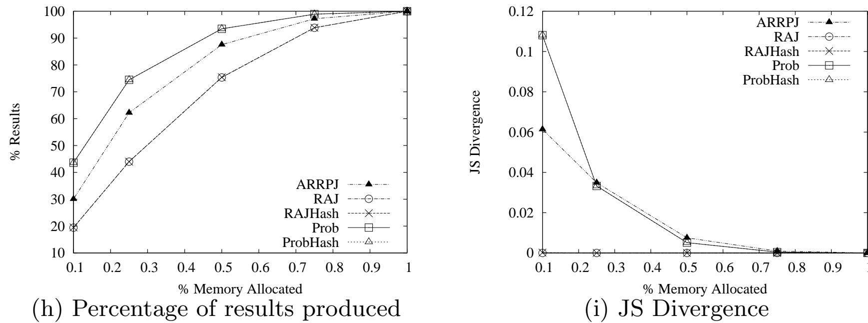


(a) Exact (b) ARRPJ (c) Prob

(d) ProbHash (e) RAJ (f) RAJHash

	ARRPJ	Prob	ProbHash	RAJ	RAJHash
JS Divergence	0.06136	0.10812	0.10811	0.00001	0.00018

(g) JS Divergence



(h) Percentage of results produced (i) JS Divergence

Figure 4: Skewed Dataset

quantity and quality.

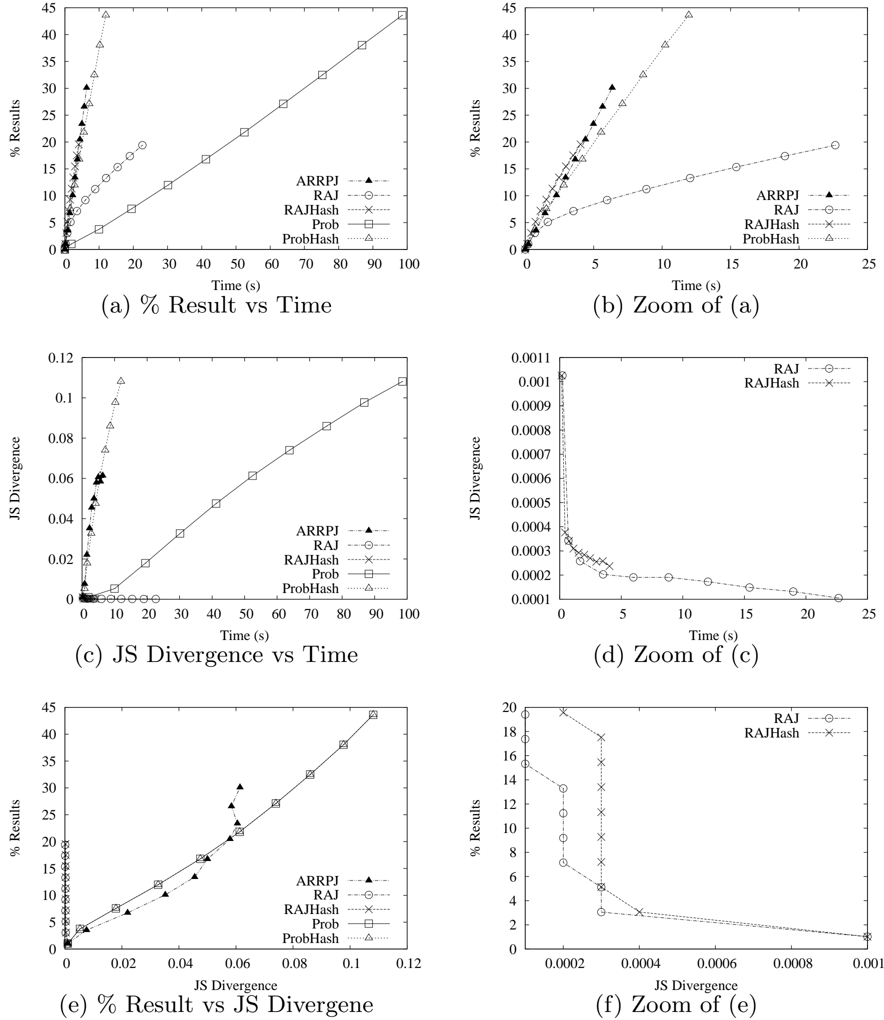
5.1.2 Throughput and Quality Throughput

Next, we measure the quantity and quality of the results over time. We set the amount of memory allocated to the progressive approximate join algorithms to be 10% of the dataset size. We measure the percentage of complete results produced and the JS Divergence over time (x-axis).

From the results presented in Figure 5(a) and (b), we can observe that the throughput of *ARRPJ*, *ProbHash*, and *RAJHash* is significantly better than *RAJ* and *Prob*. However, they produce a lesser percentage of the complete results compared to *Prob*. In Figure 5(c), we can observe that the JS Divergence of *ARRPJ*, *ProbHash*, and *Prob* is significantly higher than *RAJ* and *RAJHash*. In addition, as time progresses, the JS Divergence of *ARRPJ*, *ProbHash*, and *Prob* increases. In contrast, from Figure 5(d), we can observe that the JS Divergence of *RAJ* and *RAJHash* decreases with time. This is because the former three methods aims to maximize quantity. Over an extended period of time, this affects the quality of the results. For the sampling-based algorithms, *RAJ* and *RAJHash*, as time progresses, the quality improves. Hence, the decreasing JS Divergence.

ProbHash has significantly better throughput, compared to *Prob*, due to the partitioning of the data space into multiple priority queues. This reduces the number of scans for join matches. In contrast, for *Prob*, a newly arrived tuple will have to scan all the tuples in the corresponding priority queue, which is inefficient. A similar observation can be made between *RAJHash* and *RAJ*. Most importantly, this is achieved without sacrificing the overall result quality over time.

We also studied the tradeoffs between quantity and quality. The results are presented in Figure 5(e). As observed in earlier graphs, when the percentage of results increases, the JS Divergence for *ARRPJ*, *ProbHash*, and *RAJHash* increases. In contrast, from Figure 5(f), we can observe that for *RAJ* and *RAJHash* the JS Divergence decreases with



(a) % Result vs Time (b) Zoom of (a)

(c) JS Divergence vs Time (d) Zoom of (c)

(e) % Result vs JS Divergene (f) Zoom of (e)

Figure 5: Skewed Dataset : Throughput and Quality Throughput

increasing number of results produced.

5.2 Real Life Dataset

In this experiment, we investigate the performance of the various methods using a real-world dataset, consisting of weather data [11].

The dataset consists of monthly cloud measurements, collected by sensors globally. Similar to [7], we chose the data collected for September 1985 and September 1986 as the inputs to the approximate equijoin. The total size of both datasets is approximately 2 million tuples. For each of the dataset, we extracted the values of the latitude and longitude attributes. These attributes denotes the location of sensors which capture the sensors reading. Next, we partition the data universe using a 18 x 36 square grid. Each grid cell is assigned a unique identifier. Each tuple in the dataset, described by its latitude and longitude, is then assigned the value of the unique identifier. We then perform an equijoin between the 1985 and 1986 datasets.

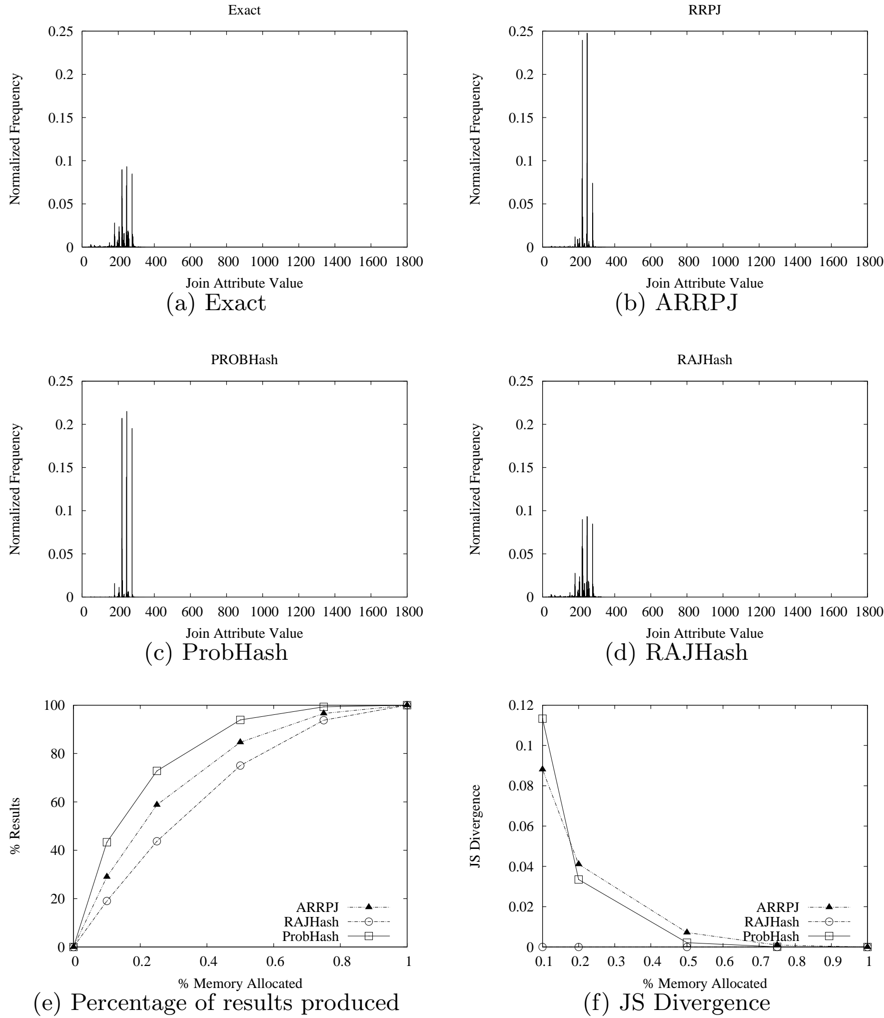
We omit the results for *Prob* and *RAJ*, and show only their more efficient counterparts, *ProbHash* and *RAJHash* respectively.

5.2.1 Approximation Performance

Similar to Section 5.1, we first study the performance of the algorithm w.r.t to approximation only. Thus, we consider bounded input streams, and look at the quantity and quality after all the tuples have been processed.

In the first experiment, we fixed the memory allocated to the approximate join to be 10% of the dataset. We present the result histograms for the various approximate join algorithms in Figure 6(a)-(d), where we can observe the quality of the result distribution. We omit the result histograms for *Prob* and *Reservoir* as they exhibit similar trends to *ProbHash* and *SReservoir* respectively. From Figure 6(a)-(d), we can observe that the normalized result histograms for *RAJHash* is similar to *Exact*. In contrast, we can observe that *ARRPJ* and *ProbHash* indeed maximize the result quantity for the join attribute values which appear more frequently.

In the second experiment, we vary the amount of memory allocated for the progressive approximate join algorithms. The results are presented in Figure 6(e) and (f). From the figures, we can also observe that the quantity and quality improves as the amount of memory allocated increases. Also, we can observe in Figure Figure 6(e) that the number of



(a) Exact (b) ARRPJ

(c) ProbHash (d) RAJHash

(e) Percentage of results produced (f) JS Divergence

Figure 6: Real Life Dataset (WEATHER)

results produced by *RRPJ*, *Prob* and *ProbHash* is significantly more than *RAJ* and *RAJHash*. From Figure 6(f), we can observe that the JS-divergence of *RAJ* and *RAJHash* is much lower.

5.2.2 Throughput and Quality Throughput

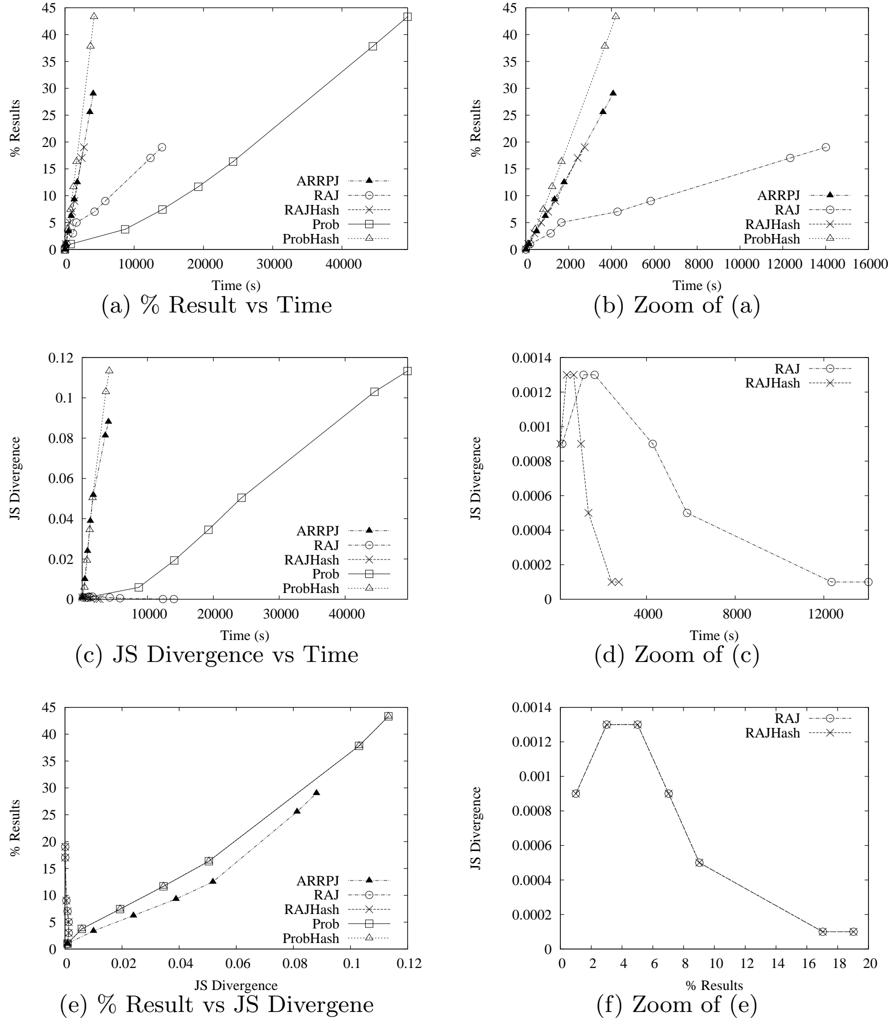
Next, we measure the quantity and quality of the results over time. We set the amount of memory allocated to the progressive approximate join algorithms to be 10% of the dataset size. We measure the percentage of complete results produced and the JS Divergence over time (x-axis).

From Figure 7(a) and (b), we can observe that the throughput of *ARRPJ*, *ProbHash* and *RAJHash* is significantly better than *RAJ* and *Prob*. This is similar to the observation made for the skewed synthetic dataset. From Figure 7(c), we can observe that the JS Divergence of *ARRPJ*, *ProbHash*, and *Prob* is significantly higher than *RAJ* and *RAJHash*. As time progresses, the JS Divergence of *ARRPJ*, *ProbHash*, and *Prob* increases. In contrast, from Figure 7(d), we can observe that the JS Divergence of *RAJ* and *RAJHash* initially increases. This is due the arrival of non-representative tuples in the beginning. Hence when these tuples are used in the join, the results are not representative (hence the increasing JS divergence at the initial stages). However, when time progresses, the JS Divergence decreases with time. The tradeoffs between quantity and quality are presented in 7(e) and (f). We can observe that the JS Divergence increases as the percentage of results produced by *ARRPJ*, *Prob* and *ProbHash* increases. In contrast, the JS Divergence for *RAJ* and *RAJHash* decreases over time. In Figure 7(f), the initial increase in JS Divergence is due to the effects discussed earlier for Figure 7(d).

5.3 Effect of Extreme Dataset

In this experiment, we investigate the performance of the various methods in the presence of the extreme scenario [7, 8]. Due to space constraints, we present only the results for the approximation performance when all tuples have arrived.

The extreme scenario is characterized by having join attribute values that appear less frequently for each dataset. Figure 10 shows the join attributes values used in the extreme scenario for two data streams, R1 and R2. For the experiments, the value of b_1 and b_2 is set to 1 and 2 respectively.



(a) % Result vs Time (b) Zoom of (a)

(c) JS Divergence vs Time (d) Zoom of (c)

(e) % Result vs JS Divergene (f) Zoom of (e)

Figure 7: Real Life Dataset (WEATHER): Throughput and Quality Throughput

Figure 10: Extreme Case

A	B
a_1	b_1
a_2	b_2
a_3	b_2
a_4	b_2
...	...
a_N	b_2

R_1

B	C
b_2	c_1
b_1	c_2
b_1	c_3
b_1	c_4
...	...
b_1	c_N

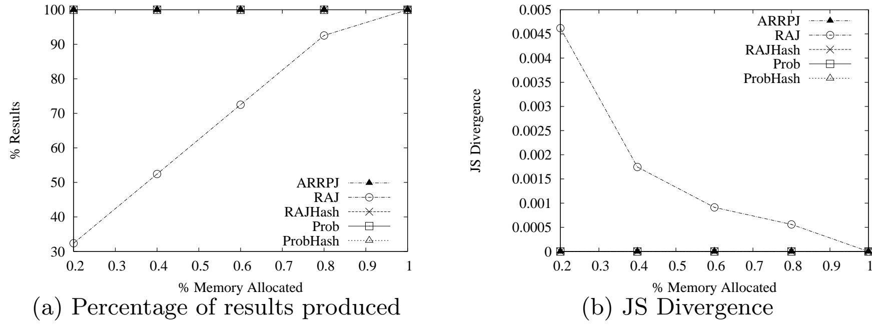
R_2

From Figure 8(a), we can observe that except for *RAJ*, all methods are able to generate 100% of the results. An interesting observation is that *RAJHash* is able to generate 100% of the results, whereas *RAJ* does not. This is because *RAJHash* is able to keep the rare values b_1 (from R_1) and b_2 (from R_2) in the sub-reservoir. When the tuples with join attribute values b_2 (from R_1) and b_1 (from R_2) arrives, they are assigned to the other sub-reservoir. In this way, *RAJHash* was able to maintain a random uniform sample for each of the sub-reservoirs.

[5, 7] noted that using the Reservoir method will not produce any join results for the extreme scenario. The assumption made was that the Reservoir needs to be completely filled for either of the data streams, before join processing can start. In contrast, when a probe-insert paradigm is used to continuously probe the reservoir while it is being built, join results can still be produced since the rare tuples have not been discarded from the reservoir yet. In the experiment, we show that even in the extreme scenario, *RAJ* will still produce results (instead of an empty result set) as it progressive probe the reservoirs for result.

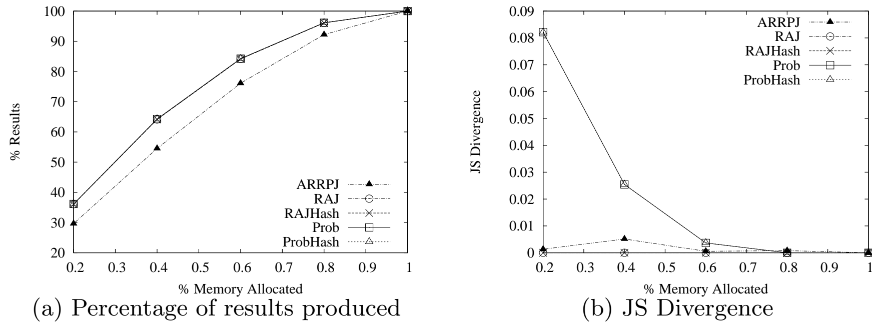
In addition, we also present a softer variant of the extreme scenario. In this variant, we relax the constraints on the appearance of specific join attribute values. In this variant, the values of b_1 (from R_1) and b_2 (from R_2) have a 50% chance of re-appearing in the dataset. From Figure 9(a), we can observe that *Prob*, *ProbHash*, *RAJ* and *RAJHash* produce the same percentage of results. From Figure 9(b), we can observe that the quality of the results produced by *RAJ* and *RAJHash* are significantly better.

6. CONCLUSION



(a) Percentage of results produced (b) JS Divergence

Figure 8: Extreme Scenario : Vary Amount of Memory



(a) Percentage of results produced (b) JS Divergence

Figure 9: Extreme Scenario Variant : Vary Amount of Memory

6.1 Summary

In this paper, we have investigated the problem of progressive approximate join processing using limited memory.

Though several approximate join processing techniques have been proposed, the focus has always been maximization of the size of the set of results. In this work, we have clearly differentiated the notions of quantity and quality. We have shown that algorithms can favor one or the other. We have also empirically demonstrated that there exists a trade-off between the two strategies as they compete for the usage of memory.

We have shown that stratification of memory with hash partitioning can significantly improve the efficiency of progressive approximate joins and therefore improve throughput without sacrificing quantity and quality. We have also shown that reservoir sampling based progressive approximate joins are superior when quality matters.

We propose four new progressive approximate join algorithms: *ARRPJ*, *ProbHash*, *RAJ* and *RAJHash*. The former two, like *Prob*, favor quantity, the latter two favor quality. *ProbHash* improves on *Prob* on every aspects. *RAJ* and *RAJHash* produce results of significantly better quality. Interestingly, although they produces less results, *RAJ* and *RAJHash* are the fastest to produce because of the simplicity of the reservoir data structure and algorithm.

6.2 Future Work

As future work, we are currently studying two problems. Firstly, we are studying a unified framework for approximate join algorithms, which balances between quality and quantity. The framework will allow straightforward generalization to other data models (e.g spatial, high-dimensional, XML). Secondly, we are also studying the design of multi-way approximate join algorithms.

Prob and *ProbHash* cannot be easily generalized to other data models. This is due to the dependence on the arrival probabilities of the partner data stream. While the arrival probabilities for relational data can be computed in a straightforward manner, it is difficult to compute such probabilities for data from other data models. Similarly, *Prob* and *ProbHash* cannot be easily extended for multi-way approximate join, unless the multi-way join query plan is decomposed into a series of binary joins. This is because for a multi-way join, it is not clear which is the partner stream. Decomposing the multi-way join query plan to a series of binary joins would limit the adaptiveness of the join.

One of the advantages of using *RAJ* and *RAJHash* is that they can be easily generalized to other data models. In addition, they can also be easily used in multi-way joins. This is because the decision to discard a tuple from the reservoir (or sub-reservoirs) does not depend on the data model. For multi-way joins, multiple reservoirs can be defined for each of the data streams. We are currently investigating the result quality of the answers that are produced using *RAJ* and *RAJHash* for other data models.

7. REFERENCES

[1] M. Al-Kateb, B. S. Lee, and X. S. Wang. Reservoir sampling over memory-limited stream joins. In *SSDBM*, page 23, 2007.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and

J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[3] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.

[4] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.

[5] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *SIGMOD*, pages 263–274, 1999.

[6] W. G. Cochran. *Sampling Techniques, 3rd Edition*. John Wiley, 1977.

[7] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*, pages 40–51, 2003.

[8] A. Das, J. Gehrke, and M. Riedewald. Semantic approximation of data stream joins. *IEEE Trans. Knowl. Data Eng.*, 17(1):44–59, 2005.

[9] P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, pages 541–550, 2001.

[10] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD*, pages 58–66, 2001.

[11] C. J. Hahn, S. G. Warren, and J. London. Edited synoptic cloud reports from ships and land stations over the globe, 1982-1991, http://cdiac.esd.ornl.gov/ftp/ndp026b, 1996.

[12] J. Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information Theory*, 37(1):145–151, 1991.

[13] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, pages 346–357, 2002.

[14] M. F. Mokbel, M. Lu, and W. G. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *ICDE*, pages 251–263, 2004.

[15] F. Olken. *Random Sampling from Databases*. Ph.D. dissertation, Computer Science, University of California, 1993.

[16] F. Olken and D. Rotem. Simple random sampling from relational databases. In *VLDB*, pages 160–169, 1986.

[17] Y. Tao, M. L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis. RPJ: Producing fast join results on streams through rate-based optimization. In *SIGMOD*, pages 371–382, 2005.

[18] W. H. Tok, S. Bressan, and M.-L. Lee. RRPJ : Result-rate based progressive relational join. In *DASFAA*, pages 43–54, 2007.

[19] T. Urhan and M. J. Franklin. XJoin: Getting fast answers from slow and bursty networks. Technical Report CS-TR-3994, University of Maryland, 1999.

[20] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.