RPJ: Producing Fast Join Results on Streams through Rate-based Optimization

Yufei Tao
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
taoyf@cs.cityu.edu.hk

Man Lung Yiu
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
mlyiu2@csis.hku.hk

Dimitris Papadias
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
dimitris@cs.ust.hk

Marios Hadjieleftheriou
Department of Computer Science
University of California, Riverside,
Riverside, CA, USA
marioh@cs.ucr.edu

Nikos Mamoulis
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
nikos@csis.hku.hk

ABSTRACT

We consider the problem of "progressively" joining relations whose records are continuously retrieved from remote sources through an unstable network that may incur temporary failures. The objectives are to (i) start reporting the first output tuples as soon as possible (before the participating relations are completely received), and (ii) produce the remaining results at a fast rate. We develop a new algorithm RPJ (Rate-based Progressive Join) based on solid theoretical analysis. RPJ maximizes the output rate by optimizing its execution according to the characteristics of the join relations (e.g., data distribution, tuple arrival pattern, etc.). Extensive experiments prove that our technique delivers results significantly faster than the previous methods.

1. INTRODUCTION

Data streams have received considerable attention [8, 13, 14] in the past few years, due to their importance in numerous applications (e.g., sensor data analysis, network monitoring, etc.) that manipulate records transmitted from remote sources. Unlike traditional databases where all the tuples are available *before* a query is raised, query processing on streams is performed as records arrive through the underlying network. This property renders many "classical" algorithms inefficient or inapplicable [2]. Consider, for example, two finite stream relations R_1, R_2, and a join $R_1 \bowtie R_2$ with an equality condition $R_1.A^{join} = R_2.A^{join}$ on their common attribute A^{join}. A "textbook" algorithm such as hash/sort-merge join is inadequate because its partitioning/sorting step requires one or both relations to be available in advance. Therefore, it cannot produce any result until R_1 and/or R_2 have been completely received. Several stream algorithms [12, 15, 17] have been proposed recently to process joins in a "progressive" manner. The objectives are to (i) generate the first result as early as possible (soon after data transmission begins), and (ii) output the remaining results at a fast rate (as tuples continuously arrive).

The major difficulty in progressively joining stream relations lies in the fact that each arriving tuple must be handled very efficiently in order to cope with the large volume of incoming data. In particular, the amortized processing cost per record should be smaller than the time interval between two consecutive arrivals. Otherwise, the number of tuples that have been received but not yet processed will continuously grow, eventually exceeding the memory capacity. Another challenge is that the underlying network may incur *unpredictable* failures (e.g., congestion, packet loss, etc.), causing delays of tuple transmission. In this case, a good algorithm should be able to "hide" the delays by continuing to output join results (using the records that have been received earlier).

The existing algorithms consider that the memory is not large enough to accommodate all the tuples received from the input streams, such that part of the data must be migrated to the disk. As reviewed in the next section, these approaches adopt the architecture illustrated in Figure 1. Tuples of R_1 that have already arrived are stored in two separate structures R_1^{mem}, R_1^{disk}, which reside in memory and disk, respectively (similarly for R_2) . The join execution switches among three stages. The *mm-stage* is active as long as data transmission is not suspended. Assume, without loss of generality, that a record t_1 arrives at R_1. The algorithm searches R_2^{mem} (i.e., the memory portion of R_2), and reports all the results that involve t_1 and tuples in R_2^{mem}. Then, t_1 is inserted into R_1^{mem}. If the memory is not exceeded, the phase terminates; otherwise, a set of records are selected from R_1^{mem}, R_2^{mem}, and *flushed* to R_1^{disk}, R_2^{disk} respectively. The mm-stage switches to the *md-stage* when (i) both relations are "blocked" (i.e., data transmission is currently suspended), or (ii) the entire R_1 and R_2 have been received. In this phase, the algorithm joins the tuples in R_1^{mem} with those of R_2^{disk}, and R_1^{disk} with R_2^{mem}. When the md-stage completes, if the data transmission has not resumed, the algorithm starts the *dd-stage*, which joins R_1^{disk} with R_2^{disk}.

The efficiency of a progressive join algorithm is determined by its "flushing" strategy, i.e., in case the memory

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.
SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA.
Copyright 2005 ACM 1-59593-060-4/05/06 $5.00.

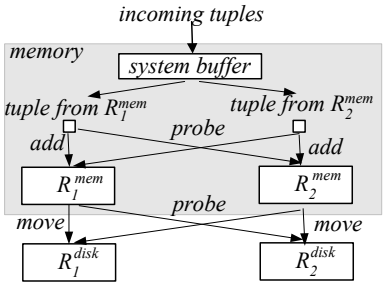


Figure 1: Architecture of progressive join

is full, which data should be moved (flushed) to the disk in order to accommodate the subsequent traffic. The existing flushing methods, however, are not adequate because they are based on several heuristics that do not have solid theoretical foundation. In particular, they do not take into account the tuple arrival rates (i.e., the number of incoming records during a timestamp) of the input streams, which significantly affect the performance of general stream algorithms [16].

In this paper, we develop RPJ (Rate-based Progressive Join), which continuously adapts its execution according to the data properties (e.g., their distribution, arrival pattern, etc.). RPJ utilizes a novel flushing algorithm which is optimal among all possible alternatives (based on the same statistics about data distributions, arrival patterns, etc.), and significantly enhances the efficiency of the mm-stage. Furthermore, RPJ maximizes the output rate by invoking the md- and dd-stages in a strategic order, i.e., the next stage selected for execution is the one expected to produce the highest output rate. Extensive experiments show that RPJ delivers results significantly faster than the previous methods.

The rest of the paper is organized as follows. Section 2 surveys the previous work that is directly related to ours. Section 3 formally defines the problem and presents an overview of the proposed solutions. Sections 4 and 5 explain the details of RPJ, focusing on its operations in memory and disk, respectively. Section 6 contains an extensive experimental evaluation to demonstrate the efficiency of RPJ, and Section 7 concludes the paper with directions for future work.

2. RELATED WORK

The existing join algorithms on streams can be classified into two categories. The first one [4, 7, 10, 13] considers that the participating relations R_1 and R_2 are "unbounded" (i.e., each contains an infinite number of tuples). The objective is to report all pairs of records $(t_1, t_2) \in R_1 \times R_2$ (i) that qualify a join predicate, and (ii) their arrival times differ by less than W timestamps (i.e., the so-called "window-join"). Here, we focus on the second category, where the goal is to provide progressive results for joining two *finite* relations ("progressiveness" is not defined in window-joins). In the sequel, we review the two existing techniques for solving this problem: XJoin in Section 2.1 and HMJ in Section 2.2.

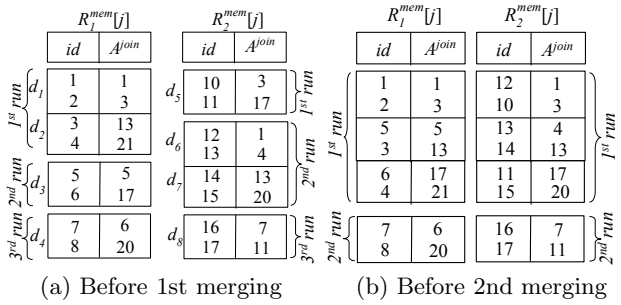
2.1 XJoin

XJoin [15] can be regarded as a variation of "symmetric hash join" [9]. Records in R_i^{mem}, R_i^{disk} $(1 \le i \le 2)$ are organized in hash tables, using the *same* hash function $\mathcal{H}$ on the join attribute A^{join}. Let $\mathcal{H}(t)$ be the partition assigned to tuple t and n_{part} be the number of hash partitions produced by $\mathcal{H}$ (i.e., $1 \le \mathcal{H}(t) \le n_{part}$). $R_i^{mem}[j]$ ($R_i^{disk}[j]$) represents the j-th partition $(1 \le j \le n_{part})$ of R_i^{mem} (R_i^{disk}). In the mm-stage, given an arriving tuple t_1 from R_1 (the case that the tuple belongs to R_2 is symmetric), XJoin joins it with $R_2^{mem}[\mathcal{H}(t_1)]$, and then inserts it into $R_1^{mem}[\mathcal{H}(t_1)]$. If the memory is exceeded, XJoin flushes a memory partition using the *flush largest* policy. Specifically, it identifies the $R_i^{mem}[j]$ (among all $1 \le i \le 2$ and $1 \le j \le n_{part}$) with the highest number of records, and appends all the data of $R_i^{mem}[j]$ to the end of $R_i^{disk}[j]$. In the md-stage, the largest memory partition (e.g., $R_1^{mem}[j]$) joins with the corresponding disk partition ($R_2^{disk}[j]$) of the other relation. XJoin assumes that the transmission of R_1 or R_2 will resume before the md-stage terminates. Hence, the dd-stage is invoked only after the entire R_1, R_2 have been received. The stage simply joins $R_1^{disk}[j]$ with $R_2^{disk}[j]$ for each $1 \le j \le n_{part}$, using the traditional hash-join algorithm.

The md- and dd-stages may generate duplicate results. Consider a tuple t_1 (t_2) from R_1 (R_2) that arrives at time 0 (5), and pair (t_1, t_2) qualifies the join predicate (i.e., $t_1.A^{join} = t_2.A^{join}$), indicating $\mathcal{H}(t_1) = \mathcal{H}(t_2)$. Since t_1 is in memory when t_2 arrives, the result pair (t_1, t_2) is reported at time 5 (in the mm-stage). Assume that, at time 10, the memory becomes full, and t_1 (along with other data in $R_1^{mem}[\mathcal{H}(t_1)]$) is flushed to the disk (while t_2 still remains in memory). At time 15, the transmission of R_1, R_2 is blocked, and the memory partition $R_2^{mem}[\mathcal{H}(t_2)]$ is joined with the disk partition $R_1^{disk}[\mathcal{H}(t_1)]$ (in the md-stage), which discovers the join result (t_1, t_2) for the second time. Continuing the example, assume that at time 20, the memory is full again and $R_2^{mem}[\mathcal{H}(t_2)]$ is flushed. Then, in the dd-stage, (t_1, t_2) will be produced for the third time (when joining $R_1^{disk}[\mathcal{H}(t_1)]$ and $R_2^{disk}[\mathcal{H}(t_2)]$).

To avoid duplicate reporting, XJoin associates each tuple t with an interval $[t.ATS, t.DTS]$, where $t.ATS$ ($t.DTS$) denotes the time that t arrives at the system (is flushed to the disk). If t has not been flushed, $t.DTS$ is set to ∞, indicating the current time. Clearly, $[t.ATS, t.DTS]$ corresponds to the period when t stays in memory (e.g., in the previous example, $t_1.ATS = 0$, $t_1.DTS = 10$, $t_2.ATS = 5$, $t_2.DTS = 20$). Furthermore, for each $R_i^{mem}[j]$ $(1 \le i \le 2, 1 \le j \le n_{part})$, XJoin remembers *all* the timestamps that this partition was used in the md-stage (to join with the corresponding disk partition of the other relation). Specifically, assume that $R_i^{mem}[j]$ is used totally $cnt_i^{md}[j]$ times; then XJoin stores an array $T_{i[j]}^{md}[k]$ $(1 \le k \le cnt_i^{md}[j])$, where $T_{i[j]}^{md}[k]$ is the time that $R_i^{mem}[j]$ was deployed (in the md-stage) for the k-th time. In the previous example, $cnt_1^{md}[\mathcal{H}(t_1)] = 0$ (i.e., $R_1^{mem}[\mathcal{H}(t_1)]$ was never involved in any md-stage), while $cnt_2^{md}[\mathcal{H}(t_2)] = 1$ and $T_{2[\mathcal{H}(t_2)]}^{md}[1] = 15$ (i.e., $R_2^{mem}[\mathcal{H}(t_2)]$ was utilized to join with $R_1^{disk}[\mathcal{H}(t_1)]$ at time 15).

Whenever a pair of qualifying tuples (t_1, t_2) is found in the md-/dd-stage, XJoin reports it if and only if *none* of the following conditions holds: (i) $[t_1.ATS, t_1.DTS]$ intersects with $[t_2.ATS, t_2.DTS]$, which shows that (t_1, t_2) was reported in the mm-stage, (ii) there exists a k $(1 \le k \le cnt_i^{md}[\mathcal{H}(t_1)])$ such that $T_{1[\mathcal{H}(t_1)]}^{md}[k] \in [t_1.ATS, t_1.DTS]$ and $t_2.DTS < T_{1[\mathcal{H}(t_1)]}^{md}[k]$, implying that (t_1, t_2) was produced in the md-stage that joined $R_1^{mem}[\mathcal{H}(t_1)]$ with $R_2^{disk}[\mathcal{H}(t_2)]$ at time $T_{1[\mathcal{H}(t_1)]}^{md}[k]$, and (iii) a condition similar to (ii) but reversing the roles of t_1 and t_2. In [17], the XJoin is extended



(a) Before 1st merging (b) Before 2nd merging

Figure 2: Illustration of PSMJ

to joining multiple (more than 2) relations.

2.2 Hash Merge Join (HMJ)

HMJ [12] differs from XJoin in the following ways. First, it uses more complex policies to select the memory partitions for flushing. Second, it does not involve md-stages; instead, whenever data transmission is blocked, the algorithm invokes the dd-stage directly. Third, its dd-stage is performed using the *Progressive Sort Merge Join* (PSMJ) algorithm [5]. In the sequel, we explain these differences in detail.

HMJ adopts "concurrent flushing": whenever a partition from one relation (e.g., $R_1^{mem}[j]$ for some $1 \leq j \leq n_{part}$) is flushed, the corresponding partition $R_2^{mem}[j]$ from the other relation is *also* evicted. Before flushing a partition, all its records are sorted on the join attribute A^{join}. Four flushing policies are empirically compared in HMJ. The first one, called *flush all*, simply evicts all the memory partitions, i.e., the memory becomes empty afterwards. The second one is the *flush largest* used in XJoin. The third policy is *flush smallest* which is the opposite of *flush largest*, i.e., the memory partition $R_i^{mem}[j]$ (for all $1 \leq i \leq 2, 1 \leq j \leq n_{part}$) with the minimum number of records is expunged. The experiments of [12] indicate that the best policy is an *adaptive* one that aims at balancing the sizes of memory allocated to R_1^{mem} and R_2^{mem}, respectively. Towards this, it flushes the pair of $R_1^{mem}[j]$ and $R_2^{mem}[j]$ such that (i) their size difference is small, and (ii) the sum of their sizes is large. The motivation of *adaptive* is that the algorithm should maintain sufficient tuples in memory for *both* relations to increase the probability that an arriving tuple t can produce join results, no matter which relation t belongs to.

We illustrate the dd-stage of HMJ using Figure 2a, which shows two disk partitions $R_1^{disk}[j]$ and $R_2^{disk}[j]$ corresponding to the same hash value j ($1 \leq j \leq n_{part}$). Each partition has 3 *runs*, where the k-th ($1 \leq k \leq 3$) run of $R_1^{disk}[j]$ and $R_2^{disk}[j]$ contain the data in $R_1^{mem}[j]$ and $R_2^{mem}[j]$ when they were flushed concurrently for the k-th time. Tuples in each run are sorted on A^{join}, but such relative order does *not* exist for records in different runs. To join $R_1^{disk}[j]$ with $R_2^{disk}[j]$, the dd-stage applies PSMJ which sorts both $R_1^{disk}[j]$ and $R_2^{disk}[j]$ *simultaneously*, as opposed to traditional SMJ (sort-merge join) that sorts the join relations *separately*. PSMJ continuously outputs results soon after the sorting starts.

Assume that 6 memory pages are available. In each *merging operation*, PSMJ assigns 3 pages to $R_1^{disk}[j], R_2^{disk}[j]$ respectively, and combines 2 runs[1] of each partition into a *merged run* (the merging for the two partitions is carried out simultaneously). Specifically, PSMJ first loads the starting pages d_1, d_3 (d_5, d_6) of the initial two runs of $R_1^{disk}[j]$ ($R_2^{disk}[j]$) into memory. Then, it identifies the record with the smallest value on A^{join} (among all the tuples in these 4 pages). In Figure 2a, two tuples (with ids) 1, 12 have the smallest A^{join} (=1). Therefore, they are reported as a join result, and then written to the merged runs of $R_1^{disk}[j]$, $R_2^{disk}[j]$ respectively.

Next, PSMJ obtains the records (2, 10) with the smallest A^{join} (=3) among the remaining data in d_1, d_3, d_5, d_6. Tuple 2 (10), which comes from the first run of $R_1^{disk}[j]$ ($R_2^{disk}[j]$), is written to the merged run of this partition. However, unlike the record pair (1, 12) processed earlier, (2, 10) is *not* produced as a join result. Recall that the first runs of $R_1^{disk}[j]$ and $R_2^{disk}[j]$ were flushed concurrently, which implies that records 2, 10 appeared in memory at the same time. Hence, they must have been reported in the mm-stage. In general, HMJ avoids duplicate results as follows: given tuple t_1 (t_2) from the k_1- (k_2-) th run of $R_1^{disk}[j]$ ($R_2^{disk}[j]$), if $k_1 = k_2$, then pair (t_1, t_2) is not output even if it satisfies the join predicate.

Continuing the example, now the smallest A^{join} is given by a single record 13, which is simply written to the merged run of $R_2^{disk}[j]$. After this, all the data in d_6 have been processed, and PSMJ reads into memory the next page d_7 of the second run in $R_2^{disk}[j]$ (d_7 is placed in the same memory page where d_6 was stored). Then, the algorithm repeats the above steps until all the records in the first two runs of $R_1^{disk}[j]$ and $R_2^{disk}[j]$ have been examined. The resulting merged runs are illustrated as the first runs in Figure 2b. So far two join results have been reported: {(1, 10), (6, 11)}.

Having finished the first two runs, PSMJ proceeds to merge the next two runs in $R_1^{disk}[j]$ and $R_2^{disk}[j]$. In Figure 2a, only one run is left in each partition, and thus the first *merging pass* is completed. Then, PSMJ starts the second merging pass on the (merged) runs (in Figure 2b) obtained from the previous pass. As before, two runs of each relation are merged at a time (hence, after another merging, the entire $R_1^{disk}[j]$ and $R_2^{disk}[j]$ are completely sorted). The second merging pass is performed in the same manner as the first one (including the mechanism for duplicate avoidance). Finally, we mention that the dd-stage of HMJ joins $R_1^{disk}[j]$ with $R_2^{disk}[j]$ for all $1 \leq j \leq n_{part}$ before switching back to the mm-stage.

3. PROBLEM DEFINITION AND SOLUTION OVERVIEW

Assume that R_1, R_2 are two finite relations with a common discrete attribute A^{join}, which are stored separately at two remote sites. We aim at answering join $R_1 \bowtie R_2$ with condition $R_1.A^{join} = R_2.A^{join}$ under the following settings: (i) data of R_1 and R_2 are transmitted to the local system in the form of continuous streams through a network, and (ii) the network may incur temporary suspensions in transmission. We do not assume any pre-processing on R_1 or R_2 so that tuples of each relation are received in *random order*. The proposed algorithm RPJ (Rate-based Progressive Join) is also based on the hashing methodology. For each relation R_i (i = 1 or 2), RPJ stores the tuples that have already

[1] At most 2 runs can be merged at a time because, as with SMJ, one memory page per relation is reserved as the output buffer for writing the merged run to disk.

Algorithm *mm-stage*

1. if the whole R_1 and R_2 have been received
2. invoke *clean-up*
3. if transmission of R_1 and R_2 is blocked
4. invoke *reactive-stage*
5. if the arriving tuple $t \in R_1$
6. scan $R_2^{mem}[\mathcal{H}(t)]$ to report $t \bowtie R_2^{mem}[\mathcal{H}(t)]$ //$\mathcal{H}$ is the hash function
7. insert t to $R_1^{mem}[\mathcal{H}(t)]$

/* the case $t \in R_2$ is omitted as it is similar */

8. if $|R_1^{mem}| + |R_2^{mem}| > M_{size}$ // $|R_i^{mem}|$ and M_{size} are the sizes of R_i^{mem} and memory, respectively
9. invoke *optimal-flush*
10. go to line 1

End *mm-stage*

Algorithm *reactive-stage*

1. $maxOR = 0, best_{task} = best_{part} = \emptyset$
2. for $j = 1$ to n_{part} // n_{part} is the total number of partitions
3. estimate the output rates $OR_{12}^{md}, OR_{21}^{md}, OR^{dd}$ of $R_1^{mem}[j] \bowtie R_2^{disk}[j], R_2^{mem}[j] \bowtie R_1^{disk}[j]$, $R_1^{disk}[j] \bowtie R_2^{disk}[j]$, respectively
4. if $OR_{12}^{md} > maxOR$
5. $maxOR = OR_{12}^{md}, best_{task} = md_{12}, best_{part} = j$
6. if $OR_{21}^{md} > maxOR$
7. $maxOR = OR_{21}^{md}, best_{task} = md_{21}, best_{part} = j$
8. if $OR^{dd} > maxOR$
9. $maxOR = OR^{dd}, best_{task} = dd, best_{part} = j$
10. if $best_{task} = md_{i_1 i_2}$
11. perform md-task $R_{i_1}^{mem}[best_{part}] \bowtie R_{i_2}^{disk}[best_{part}]$
12. else
13. perform dd-task $R_1^{disk}[best_{part}] \bowtie R_2^{disk}[best_{part}]$
14. if data transmission has not resumed
15. go to line 1

End *reactive-stage*

Algorithm *clean-up*

1. for $j = 1$ to n_{part}
2. flush $R_1^{mem}[j]$, $R_2^{mem}[j]$ to $R_1^{disk}[j]$, $R_2^{disk}[j]$, respectively
3. for $j = 1$ to n_{part}
4. perform dd-task $R_1^{disk}[j] \bowtie R_2^{disk}[j]$

Figure 3: High-level algorithms of RPJ

arrived in two separate parts R_i^{mem} and R_i^{disk}, which reside in memory and disk, respectively. Data in R_i^{mem} and R_i^{disk} are organized into n_{part} partitions, according to a hash function $\mathcal{H}$ that generates an integer in the range of $[1, n_{part}]$. Following the terminology in Section 2, we denote the j-th ($1 \le j \le n_{part}$) partition of R_i^{mem} as $R_i^{mem}[j]$, and similarly, $R_i^{disk}[j]$ for the j-th partition of R_i^{disk}.

RPJ is motivated by the following observations on the performance of XJoin and HMJ: first, the efficiency of the mm-stage depends strongly on the content of R_1^{mem} and R_2^{mem}. Intuitively, the memory should maintain only tuples that are likely to join with subsequent arrivals. *The join probability, therefore, should be taken into account when designing flushing policies.*

Second, in general, the md-stage is expected to produce faster results than the dd-stage because: (i) one of the two joining partitions is memory-resident, while the dd-stage must retrieve both partitions from the disk, and (ii) the md-stage performs mostly sequential (I/O) accesses, while the dd-stage requires a large number of random accesses. *Hence, the md-stage should not be discarded* (as opposed to the design of HMJ).

Third, under certain circumstances, *joining two disk partitions may actually produce faster results than the md-stage.* Consider, for example, that, for any $j \in [0, n_{part}]$, $R_1^{mem}[j]$ and $R_2^{mem}[j]$ have very few records, while the sizes of $R_1^{disk}[1]$ and $R_2^{disk}[1]$ are large. As a result, joining $R_1^{mem}[j]$ with $R_2^{disk}[j]$ (or $R_1^{disk}[j]$ with $R_2^{mem}[j]$) is expected to produce much fewer results than $R_1^{disk}[j] \bowtie R_2^{disk}[j]$. Therefore, the output rate of $R_1^{disk}[j] \bowtie R_2^{disk}[j]$, calculated as the ratio between the number of reported results and the execution time, may be considerably higher.

Symbol	Description
n_{part}	the number of hash partitions
s	the number of A^{join} values in a partition
$R_i^{mem}[j]/R_i^{disk}[j]$	the j-th memory/disk partition in R_i
$n_i^{mem}[j]/n_i^{disk}[j]$	the number of tuples in $R_i^{mem}[j]/R_i^{disk}[j]$
$n_i^{total}[j]$	the number of tuples in $R_i^{mem}[j] \cup R_i^{disk}[j]$
$p_i^{arr}[j]$	the arrival probability at $R_i^{mem}[j]$
$t.ATS/t.DTS$	the arrival/flushing time of tuple t
$T_1^{md}[i]$	the timestamp when the md-task between $R_1^{mem}[1]$ and $R_2^{disk}[1]$ was performed for the i-th time in history

Table 1: Frequently used symbols

The design of RPJ incorporates the above observations through a two-phase architecture. A mm-stage joins an incoming tuple with the memory-resident data in the other relation. RPJ deploys a new flushing algorithm to maximize the expected number of join results using the current arrival pattern of R_1, R_2 and the data distribution on A^{join}. The second phase of RPJ is a *reactive stage* that *mixes* the traditional md- and dd-stages. There are totally $2n_{part}$ possible "md-tasks" ($R_1^{mem}[j] \bowtie R_2^{disk}[j]$, $R_1^{disk}[j] \bowtie R_2^{mem}[j]$ for all $1 \le j \le n_{part}$) and n_{part} "dd-tasks" ($R_1^{disk}[j] \bowtie R_2^{disk}[j]$, for $1 \le j \le n_{part}$). Each iteration performs the task with the highest *expected output rate* defined as Er/Et, where Er equals the (expected) number of results that can be reported from the task, and Et the task execution time. It is important to note that the conventional md- and dd-stages in XJoin and HMJ are essentially restricted versions of the reactive stages of RPJ, where all the iterations perform md- and dd-tasks respectively. Therefore, RPJ can be regarded as an optimized combination of these algorithms based on a sophisticated probabilistic analysis.

After all the tuples have arrived, RPJ starts the "clean-up" process, which simply performs a full join over the received relations, returning the results that have not been output before. For this purpose, all the memory data are first flushed to their corresponding disk partitions, and then $R_1^{disk}[j]$ is joined with $R_2^{disk}[j]$ for each $1 \le j \le n_{part}$. $R_1^{disk}[j] \bowtie R_2^{disk}[j]$ can be computed using the dd-task algorithm with, however, one difference. Specifically, in a "normal" dd-task (during data transmission), only a small amount of memory can be utilized for performing the join (most memory is occupied by R_1^{mem} and R_2^{mem}), while in a "clean-up" dd-task, almost the entire memory is available (i.e., R_1^{mem} and R_2^{mem} are no longer retained).

Figure 3 summarizes the high-level functionality of RPJ. Next, we explain the details of RPJ, starting with the mm-stage in Section 4, and then clarifying the reactive stage in Section 5. Table 1 shows the most frequently used symbols.

4. THE MM-STAGE OF RPJ

In Section 4.1, we present an optimal flushing policy deployed by RPJ to maximize the output rate of the mm-stage.

Then, Section 4.2 clarifies the maintenance of the necessary statistics for applying this policy. Finally, Section 4.3 provides a detailed explanation on the superiority of our strategy over the existing ones.

4.1 Optimal Flush

When the total size of R_1^{mem} and R_2^{mem} equals the capacity of the allocated memory, a flushing algorithm moves some records from R_1^{mem} and R_2^{mem} to the disk, such that the data that remain in memory are expected to produce the largest number of join results with the subsequent arriving tuples until the next memory overflow. In order to analyze the best flushing strategy, we first represent the expected number of join results to be obtained until the next flushing, as a function of the content of R_1^{mem} and R_2^{mem}. Then, the targeted strategy can be naturally obtained by maximizing this function.

Theoretical Foundation. Consider every value v in the domain of the join attribute A^{join}. Let $n_i^{mem}(v)$ $(1 \leq i \leq 2)$ be the number of tuples with values v that are still in R_i^{mem} after flushing. Denote $p_i^{arr}(v)$ as the probability that the next incoming tuple belongs to R_i and has value v (on A^{join}). Assume that, starting from now, the system will receive $n_i^{arr}(v)$ records in R_i with values v before the memory is exceeded again. Then, the number n^{rslt} of join results reported in the mm-stage before the next flushing equals:

$$n^{rslt} = \sum_{\forall v \in A^{join}} (n_1^{mem}(v) \cdot n_2^{arr}(v) + n_1^{arr}(v) \cdot n_2^{mem}(v) + n_1^{arr}(v) \cdot n_2^{arr}(v)) \quad (1)$$

The above equation is due to the fact that a join result (t_1, t_2) can be produced in one of the following ways: (i) t_1 is currently in R_1^{mem} (a remaining record after the previous flushing) while t_2 is to be received later, contributing $n_1^{mem}(v) \cdot n_2^{arr}(v)$ results in Equation 1; (ii) t_1 will arrive subsequently and t_2 exists in R_2^{mem}, contributing $n_1^{arr}(v) \cdot n_2^{mem}(v)$; (iii) neither t_1 nor t_2 is in the memory, that is, both of them will be received before the next overflow occurs, contributing $n_1^{arr}(v) \cdot n_2^{arr}(v)$.

Let n_{flush}, a system parameter, be the number of tuples to be flushed to disk to solve a memory overflow. It is also the number of arrivals that will be received from now until the memory becomes full next time. Recall that there is probability $p_i^{arr}(v)$ for each of these records to have value v and at the same time belong to relation R_i. Thus, the expected value of $n_i^{arr}(v)$, the number of to-be-obtained R_i-tuples having value v before the next flushing, equals $n_{flush} \cdot p_i^{arr}(v)$. Replacing $n_i^{arr}(v)$ with this representation, Equation 1 becomes:

$$n^{rslt} = n_{flush} \cdot \sum_{\forall v \in A^{join}} (n_1^{mem}(v) \cdot p_2^{arr}(v) + p_1^{arr}(v) \cdot n_2^{mem}(v) + n_{flush} \cdot p_1^{arr}(v) \cdot p_2^{arr}(v)) \quad (2)$$

In Equation 2, $p_i^{arr}(v)$ (for all $v \in A^{join}$) is decided by the data arrival pattern (i.e., *independent* of the records in R_1^{mem}, R_2^{mem}), and cannot be controlled by the flushing algorithm. Therefore, to maximize n^{rslt}, an optimal algorithm should decide the values of $n_1^{mem}(v)$ and $n_2^{mem}(v)$ (for all $v \in A^{join}$) that maximize:

$$\eta = \sum_{\forall v \in A^{join}} (n_1^{mem}(v) \cdot p_2^{arr}(v) + p_1^{arr}(v) \cdot n_2^{mem}(v)) \quad (3)$$

subject to the constraint $\sum_{\forall v \in A^{join}} (n_1^{mem}(v) + n_2^{mem}(v)) = M - n_{flush}$, where M is the memory capacity (i.e., the sum of the sizes of R_1^{mem} and R_2^{mem} before flushing).

To illustrate the idea of optimal flushing, let us consider a special case, where we want to keep only one tuple after flushing, namely, $n_{flush} = M - 1$. Assume, without loss of generality, that the best record to retain is from relation R_1, and its value on A_{join} equals v'. In other words, $n_1^{mem}(v) = 0$ and $n_2^{mem}(v) = 0$ for all v except $n_1^{mem}(v') = 1$. Then, η in Equation 3 is equivalent to $p_2^{arr}(v')$. Since this is the largest possible value of η, $p_2^{arr}(v')$ is the maximum among the arrival probabilities $p_1^{arr}(v)$ and $p_2^{arr}(v)$ for all v. That is, the optimal flushing algorithm should identify the largest arrival probability $p_2^{arr}(v')$, and then keep a tuple with value v' from relation R_1.

Extending the idea to the general case (evicting any number n_{flush} of tuples), the best flushing strategy that maximizes η works as follows. We first identify the smallest arrival probability $p_i^{arr}(v)$ among all possible i (1 or 2) and v. If the smallest probability is $p_1^{arr}(v_1)$, then we evict n_{flush} records from R_2^{mem} with values v_1 — note that the eviction is applied on the stream *opposite* to the relation where $p_1^{arr}(v_1)$ belongs. If R_2^{mem} does not contain enough such tuples (i.e., less than n_{flush}), we apply the above process repeatedly. Specifically, n_{flush} is decreased by the number of records in R_2^{mem} with values v after migrating them to the disk. Then, the next smallest $p_i^{arr}(v)$ (other than $p_1^{arr}(v_1)$) is identified, and a set of tuples with values v but from the relation different than stream R_i (that $p_i^{arr}(v)$ belongs to) are expunged. If more tuples need to be removed, this process is repeated again.

We illustrate the strategy using a concrete example. Assume that the domain of A_{join} includes only two values 1 and 2. The arrival probabilities for R_1 are $p_1^{arr}(1) = 35\%$ (i.e., with 35% chance the next incoming record belongs to relation R_1 and has value 1), $p_1^{arr}(2) = 25\%$, and the corresponding values for R_2 are $p_2^{arr}(1) = 10\%$ and $p_2^{arr}(1) = 30\%$. When the memory overflows, R_1^{mem} and R_2^{mem} have 20 tuple respectively, half of which have values 1 and the other half have values 2. Suppose that our target is to evict totally $n_{flush} = 15$ records from memory. Towards this, we identify the smallest arrival probability $p_2^{arr}(1) = 10\%$, and hence, migrate the 10 tuples in R_1^{mem} with values 1 to the disk. After this, n_{flush} is reduced to 5. Since the next smallest probability is $p_1^{arr}(2) = 25\%$, we remove 5 records in R_2^{mem} with value 2, and complete the flushing, after which $n_1^{mem}(1) = 0$ (no tuple remaining in R_1^{mem} has value 1), $n_1^{mem}(2) = 10$, $n_2^{mem}(1) = 10$, and $n_2^{mem}(2) = 5$.

Practical Consideration. Maintaining $n_i^{mem}(v), p_i^{arr}(v)$ for all $1 \leq i \leq 2$ and $v \in A^{join}$ is impractical if the domain of A^{join} is large. Our goal is to limit the amount of statistics to be proportional to n_{part} (the number of different hash values). For this purpose, instead of $n_i^{mem}(v)$, our system records the number $n_i^{mem}[j]$ of records in $R_i^{mem}[j]$ (i.e., the j-th partition of R_i^{mem}). Similarly, instead of $p_i^{arr}(v)$, we store the probability $p_i^{arr}[j]$ that the next arriving record falls in $R_i^{mem}[j]$. Since statistics are maintained only at the "partition" level, we infer $n_i^{mem}(v)$ and $p_i^{arr}(v)$ for individ-

Algorithm ***optimal-flush***
/* flush n_{part} tuples from R_1^{mem}, R_2^{mem}; when the algorithm starts, $n_i^{mem}[j]$ ($i = 1$ or 2, and $1 \le j \le n_{part}$) equals the number of records in partition $R_i^{mem}[j]$ */

```
L = the list of all p_i^arr[j] in ascending order
    1 ≤ i ≤ 2, 1 ≤ j ≤ n_part
while (n_part > 0) //need to flush more
   get the next smallest p_i^arr[j] from L
   if i = 1
      if n_2^mem[j] ≥ n_flush
         sort the first n_flush tuples in R_2^mem[j] and append
         them to the end of R_2^disk[j]
         n_2^mem[j] = n_2^mem[j] − n_flush; n_flush = 0
      else
         append everything in R_2^mem[j] to the end of R_2^disk[j]
         n_flush = n_flush − n_2^mem[j]; n_2^mem[j] = 0;
/* similarly for the case i = 2 */
```

End ***optimal-flush***

Figure 4: Optimal flush

ual values v using the local uniformity assumption. Let s be the number of distinct values in one partition, namely, $s = |A^{join}|/n_{part}$, where $|A^{join}|$ is the total number of distinct values in A^{join}. Then, for each value v in $R_i^{mem}[j]$, $n_i^{mem}(v) \approx n_i^{mem}[j]/s$, and $p_i^{arr}(v) \approx p_i^{arr}[j]/s$. Accordingly, Equation 2 can be re-written into:

$$n^{rslt} = \frac{n_{flush}}{s} \cdot \sum_{j=1}^{n_{part}} (n_1^{mem}[j] \cdot p_2^{arr}[j] + p_1^{arr}[j] \cdot n_2^{mem}[j] + n_{flush} \cdot p_1^{arr}[j] \cdot p_2^{arr}[j]) \quad (4)$$

The results derived earlier for Equation 2 also hold for Equation 4, except that here the operations are performed on partitions, instead of individual values. Figure 4 demonstrates the pseudo-code for the *optimal flush* in RPJ. The algorithm first obtains the smallest $p_i^{arr}[j]$ among all $1 \le i \le 2$ and $1 \le j \le n_{part}$. Given such a $p_i^{arr}[j]$, it flushes records in the j-th partition of the *opposite* relation. If the number of records from the partition is smaller than n_{flush} (i.e., the target number of tuples to evict), the algorithm selects the next smallest $p_i^{arr}[j]$, and flushes another partition. We close this section by claiming the optimality of *optimal flush*.

LEMMA 1. *Given the same statistics about the arrival probabilities and the number of records in each relation with partition values on column A_{join},* optimal flush *achieves the largest n^{rslt} among all the alternative flushing strategies.*

4.2 Statistics Maintenance

Next, we explain how to dynamically maintain $n_i^{mem}[j]$ and $p_i^{arr}[j]$ for each partition $R_i^{mem}[j]$. Updating $n_i^{mem}[j]$ is trivial: we simply increase (decrease) it by 1 whenever a tuple arrives at (is flushed from) $R_i^{mem}[j]$. To compute $p_i^{arr}[j]$, we consider its equivalent form $P(R_i) \cdot P(j|R_i)$, where $P(R_i)$ denotes the probability that the next arriving tuple t belongs to R_i, and $P(j|R_i)$ gives the conditional probability that t falls in the j-th partition, *knowing that* $t \in R_i$. In the sequel, we first elaborate $P(j|R_i)$ and then discuss the maintenance of $P(R_i)$.

Since tuples of each relation arrive in random order, the data obtained earlier can be used to infer the distribution of the tuples to be retrieved subsequently. For this purpose, we store the number $n_i^{total}[j]$ of records that have *ever been* received in the j-th partition $R_i[j]$ of R_i since the *beginning* of the join operation (note that $R_i[j]$ includes both $R_i^{mem}[j]$ and $R_i^{disk}[j]$). Then, $P(j|R_i)$ corresponds to the percentage of $n_i^{total}[j]$ over the total number of R_i tuples currently in the system, or formally:

$$P(j|R_i) = n_i^{total}[j] / \sum_{j=1}^{n_{part}} n_i^{total}[j] \quad (5)$$

Tuples that have already been received for each relation R_i constitute a random sample set of R_i, whose size increases with time. Since a larger sample set reflects the overall data distribution more accurately, the precision of Equation 5 continuously improves with time.

The probability $P(R_i)$, on the other hand, is not related to the data that have arrived, but instead depends on the relative speeds of the networks delivering R_1 and R_2 respectively. Furthermore, unlike $P(j|R_i)$ which tends to stabilize as time progresses (converging to the final percentage of $R_i[j]$ in the entire R_i), $P(R_i)$ may vary with time considerably. For example, assume that currently R_1 tuples arrive faster than R_2, implying $P(R_1) > P(R_2)$ (i.e., the next record is more likely to be from R_1). At some later time when the network of R_1 incurs congestion, $P(R_1)$ may become smaller than $P(R_2)$.

We estimate $P(R_1)$ and $P(R_2)$ by maintaining a value n_i^{rcnt} for each relation R_i as follows. The initial n_i^{rcnt} is set to the number of arriving R_i records in the first time unit $[0, 1]$. During each subsequent unit $[t, t+1]$ ($t \ge 1$ is an integer), the system counts the amount $\alpha_i(t)$ of incoming R_i records. At time $t+1$, n_i^{rcnt} is updated to $\lambda \cdot n_i^{rcnt} + (1-\lambda) \cdot \alpha_i(t)$, where λ is a constant in [0, 1].

The value of n_i^{rcnt} reflects the volume of "recent" arrivals in R_i. To see this, observe that the influence of $\alpha_i(t)$ (the number of incoming records during a *particular* interval $[t, t+1]$) on the current n_i^{rcnt} decays exponentially with time. For example, $\alpha_i(0)$ is exactly n_i^{rcnt} at time 1, but contributes (to n_i^{rcnt}) by only $\lambda \cdot \alpha_i(0)$ at time 2. In general, the contribution of $\alpha_i(0)$ at time t equals $\lambda^{t-1} \cdot \alpha_i(0)$, which eventually becomes negligible (for large t). The constant λ controls the "rate" of decay – a low (high) λ quickly (slowly) reduces the effect of historical $\alpha_i(t)$ to the current n_i^{rcnt}.

Therefore, using 4 values, namely, n_1^{rcnt}, n_2^{rcnt}, and $\alpha_1(t)$, $\alpha_2(t)$ for the latest time interval $[t, t+1]$ (the historical $\alpha_i(t)$ need not be retained), we collect sufficient information for estimating $P(R_i)$: Out of the "recent" $n_1^{rcnt} + n_2^{rcnt}$ tuples, n_i^{rcnt} come from R_i ($1 \le i \le 2$). Hence, $P(R_i)$ is approximately $n_i^{rcnt}/(n_1^{rcnt} + n_2^{rcnt})$. Combining with Equation 5, we can represent $p_i^{arr}[j]$ using Equation 6 (recall that $p_i^{arr}[j] = P(R_i) \cdot P(j|R_i)$).

$$p_i^{arr}[j] = \frac{n_i^{total}[j]}{\sum_{j=1}^{n_{part}} n_i^{total}[j]} \cdot \frac{n_i^{rcnt}}{n_1^{rcnt} + n_2^{rcnt}} \quad (6)$$

If relation R_i has been completely received (indicated by a special "end-of-stream" symbol), $p_i^{arr}[j]$ is set to 0 for all $1 \le j \le n_{part}$.

4.3 Comparison of Flushing Policies

In this section, we analytically compare *optimal flush* with the alternative flushing policies (i.e., *flush all*, *flush smallest*, *flush largest*, *adaptive*, reviewed in Section 2) using Equation 4. Towards this, we discuss two "extreme" arrival patterns. The first one, referred to as *harmony*, is such that, if the arrival probability $p_1^{arr}[j]$ is high (low), then $p_2^{arr}[j]$ (for

the same partition in R_2) is also high (low). The other one, *reverse*, is the opposite of *harmony*: if $p_1^{arr}[j]$ is high (low), then $p_2^{arr}[j]$ is low (high). Note that other arrival patterns are between *harmony* and *reverse*, in which case the relative performance of each method can be inferred accordingly. To facilitate discussion, we consider the networks for R_1, R_2 are equally fast (i.e., $P(R_1) = P(R_2)$), unless specifically stated.

Flush All. This is a radical policy that actually *minimizes* the efficiency of the mm-stage (observe that n^{rslt} in Equation 4 obtains its minimum when $n_1^{mem} = n_2^{mem} = 0$). Hence, we do not discuss it further.

Flush Smallest. The strategy (very probably) performs well for *harmony*. To explain this, consider the *first* flushing after the join starts. Since $P(R_1) = P(R_2)$, n_1^{rcnt} is equivalent to n_2^{rcnt} in Equation 6, meaning that the same number of tuples have been received from each stream. Furthermore, no flushing has happened before, so $n_i^{mem}[j] = n_i^{total}[j]$ for all partitions ($1 \le i \le 2$, $1 \le j \le n_{part}$), where $n_i^{mem}[j]$ is the number of records in $R_i^{mem}[j]$, and $n_i^{total}[j]$ denotes the size of both $R_i^{mem}[j]$ and $R_i^{disk}[j]$ (which is empty). As a result, Equation 6 can be simplified to $p_i^{arr}[j] = \frac{1}{2} n_i^{mem}[j] / \sum_{j=1}^{n_{part}} n_i^{mem}[j]$. Since the denominator is the same for all $p_i^{arr}[j]$, the arrival probability $p_i^{arr}[j]$ is proportional to $n_i^{mem}[j]$.

Assume that the victim partition of the first flushing is $R_1^{mem}[1]$ (the first partition of R_1^{mem}), i.e., it receives the smallest number $n_1^{mem}[1]$ of records among all partitions in R_1^{mem} and R_2^{mem}. Hence, $p_1^{arr}[1]$ is the smallest among all the arrival probabilities $p_i^{arr}[j]$. Due to the property of *harmony*, $p_2^{arr}[1]$ is also small (but is *not* the smallest). Hence, decreasing $n_1^{mem}[j']$ is not expected to lower n^{rslt}, quantified in Equation 4, considerably, although the best strategy should decrease $n_2^{mem}[1]$, as in *optimal flush*.

On the other hand, *flush smallest* is inefficient for *reverse*. To illustrate this, assume again $p_1^{arr}[1]$ is the smallest so that $n_1^{mem}[1]$ will decrease after flushing. Since $p_2^{arr}[1]$ is expected to be large (by the definition of *reverse*), reducing $n_1^{mem}[1]$ brings down the value of n^{rslt} significantly according to Equation 4, resulting in much lower output rate than *optimal flush*.

Flush Largest. Since *flush largest* is the opposite of *flush smallest*, it is expected to perform well for *reverse*, but poorly for *harmony*, which can be verified in a way similar to the above analysis.

Adaptive. Recall that *adaptive* aims at balancing the numbers of records in R_1^{mem}, R_2^{mem}. It is indeed the best choice *if* (i) tuples of both relations arrive equally fast, and (ii) no further statistics are available about the arrival status in individual partitions. In this case, *adaptive* is equivalent to a special version of *optimal flush*, where each relation contains a single partition (i.e., $n_{part} = 1$ and $p_1^{arr}[1] = p_2^{arr}[1]$ in equaiton 4). Thus, it is expected to perform reasonably well for both *harmony* and *reverse*.

The rationale of *adaptive*, however, is not correct if the arrival rates of R_1 and R_2 are different. For example, if data of R_1 are faster, then (by Equation 4) more tuples from R_2 should be maintained in memory to maximize the join probability (for incoming records), as is captured by *optimal flush*. Finally, *adaptive* does not take advantage of the additional statistics about individual partitions, which are utilized by *optimal flush* to maximize the output rate.

5. THE REACTIVE STAGE

The reactive stage is invoked when the transmission from both streams is being suspended. The execution of this stage is divided into multiple iterations. Specifically, each iteration selects the most "beneficial" task, from the possible $2n_{part}$ md-tasks and n_{part} dd-tasks, that is expected to offer the fastest output rate. In Sections 5.1 and 5.2, we first explain the details of the md- and dd-tasks, respectively. Then, Section 5.3 proposes an efficient way to accurately estimate the output rate of each possible task.

5.1 Performing an Md-task

Recall that an md-task performs a join between a chosen memory partition (how to make this choice is the topic of Section 5.3) with the corresponding disk partition of the other relation. Without loss of generality, in the sequel, we assume that the selected partition is the first one in R_1^{mem}, i.e., $R_1^{mem}[1]$, which is to be joined with $R_2^{disk}[1]$ in the md-task.

To perform the task, we first sort all the data in $R_1^{mem}[1]$ in ascending order of their A^{join} values (if they are not already sorted). Then, we simply scan each page of $R_2^{disk}[1]$, and for each record t_2 encountered, a binary search is invoked to identify the tuples t_1 in $R_1^{mem}[1]$ with the same (A^{join}) values as t_2. Although such pairs of records (t_1, t_2) satisfy the join predicate, only a subset of them are output — those that have not been reported before. There are two possible cases where this could happen: (t_1, t_2) was already identified in the mm-stage, or in a previous md-task involving $R_1^{mem}[1]$ and $R_2^{disk}[1]$.

To avoid a duplicate result (t_1, t_2) that has been produced before in the mm-stage, we associate each tuple t with an interval $[t.ATS, t.DTS]$ where, as in XJoin, $t.ATS$ is the time that t is received, and $t.DTS$ the time it is flushed (if t is still in memory, $t.DTS$ equals ∞). Hence, $[t.ATS, t.DTS]$ denotes the period during which t stays in memory. Then, (t_1, t_2) is reported in the mm-stage if and only if the intervals of the two tuples, $[t_1.ATS, t_1.DTS]$ and $[t_2.ATS, t_2.DTS]$, intersect. Hence, a pair (t_1, t_2) encountered in the md-stage is ignored if the two intervals overlap.

Next we elaborate how to avoid a duplicate (t_1, t_2) already generated in a previous md-task between $R_1^{mem}[1]$ and $R_2^{disk}[1]$. Note that this is possible only if t_1 and t_2 already appeared in $R_1^{mem}[1]$ and $R_2^{disk}[1]$ respectively, when the md-task was performed. Denote T as the execution time of this md-task. It follows that, (i) T is larger than $t_1.ATS$, the arriving timestamp of t_1 (note that $t_1.DTS = \infty$ because t_1 is still in memory), and (ii) T is larger than $t_2.DTS$ since t_2 started being in $R_2^{disk}[1]$ only after $t_2.DTS$.

Motivated by this, we adopt a mechanism similar to, but simpler than, that of XJoin. In particular, RPJ records all the timestamps in the history when $R_1^{mem}[1]$ was deployed in an md-task to join with $R_2^{disk}[1]$. Specifically, assume that $R_1^{mem}[1]$ was used in an md-task c times; then the time of the i-th ($1 \le i \le c$) execution is recorded in the i-th element $T_1^{md}[i]$ of array T_1^{md} (subscript 1 indicates the memory partition in the md-task comes from relation R_1). For each (t_1, t_2) encountered, we check whether this result was produced in the last md-task between $R_1^{mem}[1]$ and $R_2^{disk}[1]$, that is, whether $T_1^{md}[c] > t_1.ATS$ and $T_1^{md}[c] > t_2.DTS$. If yes, (t_1, t_2) is ignored. Although the values $T_1^{md}[1]$, $T_1^{md}[2]$,

Algorithm *md-task*
/* without loss of generality, we assume that the md-task is between the first partitions of R_1^{mem} and R_2^{disk}, denoted as $R_1^{mem}[1]$ and $R_2^{disk}[1]$, respectively; also assume that these partitions have been involved in md-tasks c times before, at timestamps $T_1^{md}[1]$, $T_1^{md}[2]$, ..., $T_1^{md}[c]$, respectively */

1. sort the records in $R_1^{mem}[1]$ by their A_{join} values
2. sequentially scan the tuples in $R_2^{disk}[1]$
3. for each tuple t_2 in $R_2^{disk}[1]$
4. perform a binary search on the data of $R_1^{mem}[1]$ to identify the tuples t_1 with the same A_{join} value as t_2
5. for each such tuple t_1
6. if $[t_1.ATS, t_1.DTS]$ intersects $[t_2.ATS, t_2.DTS]$
7. continue to the next tuple at line 5
8. if $T_1^{md}[c] > t_1.ATS$ and $T_1^{md}[c] > t_2.DTS$
9. continue at line 5
10. report (t_1, t_2) as a join result

End *md-task*

Figure 5: The algorithm of an md-task

..., $T_1^{md}[c-1]$ are not used in the above procedures, they are needed in dd-tasks (as will be clarified shortly), and hence, must be stored. Figure 5 formally presents the pseudo-code of an md-task.

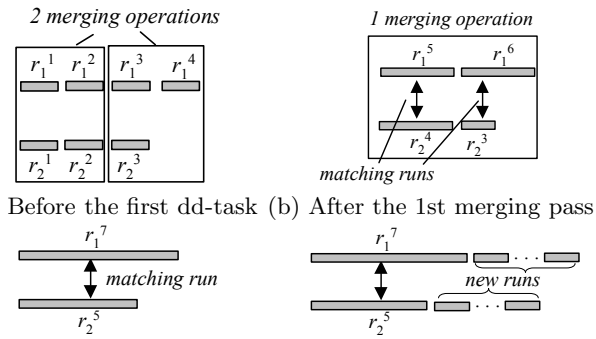
5.2 Performing a Dd-task

In this section, we elaborate the details of a dd-task, using as an example the join between the first disk partition $R_1^{disk}[1]$ of R_1^{disk} and that $R_2^{disk}[1]$ of R_2^{disk} (the discussion about joining other partitions is similar). As with HMJ, a dd-task applies the PSMJ algorithm (reviewed in Section 2.2), but eliminates duplicate results in a different way. The elimination approach of HMJ is not applicable to RPJ, because it is limited to the HMJ's concurrent flushing policy.

Since the execution of PSMJ in a dd-task is the same as that in HMJ, in the sequel we focus on duplicate elimination. Assume that PSMJ encounters a pair of records (t_1, t_2) where t_1 (t_2) is from R_1^{disk} (R_2^{disk}), and the two tuples satisfy the join condition. There are three possible cases for (t_1, t_2) to have been identified before. Next we discuss each of them in turn, together with the corresponding mechanism for avoiding duplicate reporting.

The first possibility is that (t_1, t_2) was reported in the mm-stage. As explained in the previous section, this happens if and only if t_1 appeared in $R_1^{mem}[1]$ at some timestamp when t_1 was also present in $R_2^{mem}[1]$. This can be detected by checking whether their memory-alive intervals, $[t_1.ATS, t_1.DTS]$ and $[t_2.ATS, t_2.DTS]$, intersect. If yes, (t_1, t_2) is ignored in the current dd-task.

The second scenario is that, (t_1, t_2) was already reported in a previous md-task, which can be a task joining $R_1^{mem}[1]$ with $R_2^{disk}[1]$, or $R_1^{disk}[1]$ with $R_2^{mem}[1]$. Due to symmetry, we discuss the case involving $R_1^{mem}[1]$ and $R_2^{disk}[1]$. Then, at the time T of this md-execution, t_1 was still in memory, but t_2 was already flushed to the disk, or equivalently, T falls in the interval $[t_1.ATS, t_1.DTS]$, but is larger than $t_2.DTS$. Assume that in history there have been c md-tasks involving $R_1^{mem}[1]$ and $R_2^{disk}[1]$, and their happening timestamps are recorded in array $T_1^{md}[1]$, ..., $T_1^{md}[c]$ (as mentioned in the previous section). From these timestamps, we collect the ones that are within $[t_1.ATS, t_1.DTS]$, and check if any of them is larger than $t_2.DTS$. If yes, the result (t_1, t_2) is ignored.



(a) Before the first dd-task (b) After the 1st merging pass

(c) After the 2nd merging pass (d) Before the 2nd dd-task

Figure 6: Duplicate avoidance in dd-tasks

As the third case, (t_1, t_2) can also have been produced in one of the previous dd-tasks (for $R_1^{disk}[1]$ and $R_2^{disk}[1]$). To explain how duplicates of this type are avoided, we need to clarify the concept of *matching runs* of PSMJ. Figure 6a shows the situation before the *first* dd-task between $R_1^{disk}[1]$ and $R_2^{disk}[1]$, which contain 4 and 3 runs, respectively. Note that, unlike HMJ, the number of runs in the two disk partitions may be different.

Assume that in each *merging operation* PSMJ combines (at most) 2 runs of *each* partition into a merged run. Figure 6b shows the situation after the first merging pass, which involves two *merging operations*. The first operation combines r_1^1, r_1^2 of R_1 into run r_1^5, and at the same time r_2^1 and r_2^2 of R_2 into r_2^4. The second operation combines only r_1^3, r_1^4 of R_1 into r_1^6 (no combination is necessary for R_2 because there is only one run r_2^3 left). In the second merging pass, (Figure 6c), runs r_1^5 and r_1^6 (r_2^4 and r_2^3) are merged into r_1^7 (r_2^4). Figure 6d shows the situation before the second dd-task involving these two disk partitions, where some new runs have been flushed to each partition.

We say two runs in partitions $R_1^{disk}[j], R_2^{disk}[j]$ *match* each other if they are produced from the same *merging operation*. For example, recall that the first merging pass in Figure 6b involves two operations, and accordingly, run r_1^5 matches r_2^4, and r_1^6 matches r_2^3. Similarly, the second merging pass involves a single merge, creating matching runs r_1^7 and r_2^5. These are the only matching pairs in all examples of Figure 6.

To eliminate duplicates produced in previous dd-tasks (on $R_1^{disk}[j], R_2^{disk}[j]$), we do not report (t_1, t_2) if the two tuples are obtained from matching runs. For example, if t_1 (t_2) comes from r_1^5 (r_2^4), then (t_1, t_2) must have already been reported in the merging operation that combined r_1^1 and r_1^2 (r_2^1 and r_2^2) into r_1^5 (r_2^4). Similarly, if t_1 (t_2) comes from r_1^7 (r_2^5), then (t_1, t_2) has already been checked when r_1^5, r_1^6 and r_2^4, r_2^3 were combined. Figure 7 explains the pseudo-code for performing a dd-task.

5.3 Task Output Rate Estimation

As mentioned earlier, the output rate of an (md- or dd-) task is Er/Et, where Er is the number of *new* results to be produced, and Et the task execution time. The reactive stage of RPJ depends on accurate estimation of the output rate of individual tasks, so that each iteration can select the one that promises to produce the fastest results. Predicting the output rate involves the estimation of both Er and Et. We start with the analysis of Et since it is relatively simple.

Algorithm *dd-task*
/* assume that the dd-task is between the first partitions $R_1^{disk}[1]$ and $R_2^{disk}[1]$ of R_1^{disk} and R_2^{disk}, respectively; the md-tasks involving $R_1^{mem}[1]$ and $R_2^{disk}[1]$ have been performed c_1 times before, at timestamps $T_1^{md}[1]$, ..., $T_1^{md}[c_1]$, respectively; similarly the md-tasks involving $R_1^{disk}[1]$ and $R_2^{mem}[1]$ have been performed c_2 times at timestamps $T_2^{md}[1]$, ..., $T_2^{md}[c_2]$ */

1. use PSMJ to join $R_1^{disk}[1]$ and $R_2^{disk}[1]$
2. for each join result (t_1, t_2)
3. if $[t_1.ATS, t_1.DTS]$ intersects $[t_2.ATS, t_2.DTS]$
4. continue at line 2
5. if t_1, t_2 come from matching runs then continue at line 2
6. for $i = 1$ to c_1
7. if $T_1^{md}[i] \in [t_1.ATS, t_1.DTS]$ and $T_1^{md}[c_1] > t_2.DTS$
8. continue at line 2
9. for $i = 1$ to c_2
10. if $T_2^{md}[i] \in [t_2.ATS, t_2.DTS]$ and $T_2^{md}[i] > t_1.DTS$
11. continue at line 2
12. report (t_1, t_2) as a join result

End *dd-task*

Figure 7: The algorithm of an dd-task

Estimating Et. For an md-task, e.g., joining $R_1^{mem}[1]$ with $R_2^{disk}[1]$), Et is dominated by the cost of scanning $R_2^{disk}[1]$ — the time accessing the memory-resident $R_1^{mem}[j]$ is negligible. Records in each run of $R_2^{disk}[1]$ are stored in sequential pages (as in Figure 6, a run contains data flushed together or produced from the same merging operation in PSMJ). Hence, if $R_2^{disk}[1]$ has x runs and occupies y pages, scanning it requires x random and $y - x$ sequential accesses. Let c_{ran} (c_{seq}) represent the cost of one random (sequential) access; then $Et = x \cdot c_{ran} + (y - x) \cdot c_{seq}$.

Deriving Et for a dd-task, e.g., $R_1^{disk}[1] \bowtie R_2^{disk}[1]$, is reduced to the cost analysis of PSMJ [6], which gives the following formula: $Et = 2c_{ran} \cdot (y_1 + y_2) \cdot ps$, where y_1 (y_2) is the number of disk pages in $R_1^{disk}[1]$ ($R_2^{disk}[1]$), ps the number of merging passes given by $\max(\lceil log_f y_1 \rceil, \lceil log_f y_2 \rceil)$, and f the number of runs in a partition that can be combined in each merging.

Main Idea of Estimating Er. An obvious attempt to predict Er would be to use join selectivity estimation techniques. For example, the Er of a dd-task $R_1^{disk}[1] \bowtie R_2^{disk}[1]$ could be computed using the cardinalities $n_1^{disk}[1]$, $n_2^{disk}[1]$ of $R_1^{disk}[1]$, $R_2^{disk}[1]$ respectively, and the number s of values covered by each partition [1]. Unfortunately, this method does *not* give the accurate Er since it cannot distinguish the results that have been reported earlier.

Motivated by this, instead of computing Er, our system *incrementally maintains* it as tuples are received and flushed to the disk. Specifically, for each memory partition, e.g., $R_1^{mem}[1]$, we keep a value Er^{mem} which equals the expected number of new results if an md-task is performed *now* using $R_1^{mem}[1]$ to join with the first disk partition of R_2. Similarly, for each disk partition, e.g., $R_1^{disk}[1]$, we maintain a number Er^{disk} that gives the (new) result size if a dd-task $R_1^{disk}[1] \bowtie R_2^{disk}[1]$ is invoked at the current time. In the sequel, we first discuss the maintenance of Er^{mem} and then clarify Er^{disk}.

Maintaining $\mathbf{Er^{mem}}$. Recall that Er^{mem} quantifies the number of join results not previously reported if an md-task between $R_1^{mem}[1]$ and $R_2^{disk}[1]$ is performed immediately (the computation of Er^{mem} for other md-tasks is similar). Obviously, once such an md-task is performed, Er^{mem} should be reset to 0 — if the same md-task is performed right after the previous one (without receiving any incoming tuple), no new result is expected. In general, since Er^{mem} depends on the tuples of $R_1^{mem}[1]$ and $R_2^{disk}[1]$, its value can be affected only when the content of $R_1^{mem}[1]$ or $R_2^{disk}[1]$ changes, for which there are three cases:

- case 1: A new tuple arrives at $R_1^{mem}[1]$;
- case 2: A tuple in $R_1^{mem}[1]$ is flushed;
- case 3: A tuple in $R_2^{mem}[1]$ is flushed.

In the sequel, we discuss the updates to Er^{mem} for each scenario. For case 1, let t be the incoming tuple. Evidently, compared to the situation before receiving t, joining $R_1^{mem}[1]$ and $R_2^{disk}[1]$ now is expected to produce additional results — those that are produced by t. Hence, we should increase Er^{mem} by the number of join results involving t, which equals the cardinality of tuples in $R_2^{disk}[1]$ that have the same value on A_{join} as t. Recall that a partition includes tuples with s different A_{join}-values, where s equals $|A_{join}|$ / n_{part}, $|A_{join}|$ is the total number of distinct values in the domain of A_{join}, and n_{part} is the number of different values produced by the hash function $\mathcal{H}$ used to hash tuples into the corresponding partitions. As a result, t is expected to satisfy the join condition with $n_2^{disk}[1]/s$ records in $R_2^{disk}[1]$, where $n_2^{disk}[1]$ is the cardinality of $R_2^{disk}[1]$. In other words, for case 1, Er^{mem} should be increased by $n_2^{disk}[1]/s$.

For case 2, let t be the record flushed from $R_1^{mem}[1]$ to $R_1^{disk}[1]$. At the first glance, it appears that Er^{mem} should be reduced by $n_2^{disk}[1]/s$ — since $R_1^{mem}[1]$ does not contain t any more, joining it with $R_2^{disk}[1]$ would lose all the results that could be produced by t. This, however, is true only if no md-task involving $R_1^{mem}[1]$ and $R_2^{disk}[1]$ has been performed since t arrived. In fact, as indicated in the following lemma, if such an md-task exists, the value of Er^{mem} before expunging t already excludes the results produced by t, and hence, no change to Er^{mem} is necessary after flushing t.

LEMMA 2. *Assume that t is a tuple being flushed to $R_1^{disk}[1]$. If an md-task between $R_1^{mem}[1]$ and $R_2^{disk}[1]$ has been executed since the arrival of t, all the results involving t and the data in $R_2^{disk}[1]$ must have been reported before.*

PROOF. Let S_{bfr} (S_{aft}) be the set of tuples in $R_2^{disk}[j]$ that were flushed *before* (*after*) t arrived. Apparently, the union of S_{bfr} and S_{aft} constitutes the entire $R_2^{disk}[j]$. All results produced by t and the data of S_{bfr} have been reported in the md-task between $R_1^{mem}[j]$ and $R_2^{disk}[j]$ after the arrival of t_1. Furthermore, every tuple in S_{aft} stayed *simultaneously with* t in memory for some time. Hence, all the results generated by t and the records in S_{aft} must have been reported in the mm-stage, thus completing the proof. □

Checking if an md-task has been executed since the arrival of t is easy — recall that the timestamps of each md-execution involving $R_1^{mem}[1]$ and $R_2^{disk}[1]$ are recorded in an array T_1^{md}, where $T_1^{md}[j]$ equals the timestamp of performing the j-th md-task in history. Hence, it suffices to examine whether there exists an element $T_1^{md}[j]$ of T_1^{md} that is larger than the arrival time $t.ATS$ of t.

Finally for case 3, no modification for Er^{mem} is needed in any case. Let t be the tuple that is being flushed to $R_2^{disk}[1]$. Obviously, t and the tuples in $R_1^{mem}[1]$ existed in

the memory at the same time (right before the eviction of t), and hence, the join results produced by them must have been reported before in the mm-stage. Therefore, the current Er^{mem} already excludes these results (remember that Er^{mem} only counts the results that have not been reported before).

Maintaining $\mathbf{Er^{disk}}$. Er^{disk} corresponds to the size of the new results in a dd-task joining $R_1^{disk}[1]$ with $R_2^{disk}[1]$, i.e., the first partitions of R_1^{disk} and R_2^{disk} (the maintenance of Er^{disk} for other dd-tasks follows the same idea). Clearly, similar to resetting Er^{mem} described earlier, Er^{disk} is reset to 0 every time a dd-task $R_1^{disk}[1] \bowtie R_2^{disk}[1]$ is executed. Other than this, potential updates to Er^{disk} are necessary only when a tuple t is flushed to $R_1^{disk}[1]$ or $R_2^{disk}[1]$. Due to symmetry, it suffices to discuss the case where t is flushed from $R_1^{mem}[1]$ to $R_1^{disk}[1]$.

No change to Er^{disk} is necessary if an md-task joining $R_1^{mem}[1]$ with $R_2^{disk}[1]$ has been performed since the arrival of t. In fact, by Lemma 2, the results involving t and the data in $R_2^{disk}[1]$ must have already been produced before, and hence, were not counted in Er^{disk} before t was expunged.

It remains to clarify the scenario where no such md-task was performed since t arrived. For this purpose, we divide the tuples in $R_2^{disk}[1]$ into two parts: the set S_{bfr} of tuples that were flushed (to $R_2^{disk}[1]$) before the arrival of t, and the set S_{aft} of records flushed after. After incorporating t into $R_1^{disk}[1]$, the dd-task joining $R_1^{disk}[1]$ with $R_2^{disk}[1]$ will produce additional join results, corresponding to those produced by t and records in S_{bfr}. The results generated by t and the data in S_{aft} must have been produced in the mm-stage before (see the proof of Lemma 2), and hence, have been excluded from Er^{disk}. Tuple t is expected to have the same A_{join} values with around $|S_{bfr}|/s$ records in S_{bfr}, where s is the number of distinct values in the domain of A_{join} hashed into a single partition (as mentioned before, $s = |A_{join}|/n_{part}$). Therefore, Er^{disk} should be increased by $|S_{bfr}|/s$.

Obtaining $|S_{bfr}|$ is straightforward. Assume that the content of $R_2^{mem}[1]$ has been flushed to $R_2^{disk}[1]$ totally c times in history. We maintain two arrays T^{flush} and n^{flush} both with size c, such that $T^{flush}[j]$ $(1 \leq j \leq c)$ equals the time of the j-th flushing, and $n^{flush}[j]$ is the number of records written to the disk in this flushing. As a result, the size $|S_{bfr}|$ of S_{bfr} equals the sum of the numbers migrated to disk during all flushings that happened before time $t.ATS$, i.e., the arrival time of t.

6. EXPERIMENTS

In this section, we empirically demonstrate the efficiency of RPJ by comparing it against XJoin and HMJ. All the experiments are performed using a 2.4Ghz CPU. The memory/disk page size is fixed to 1024 bytes. Each record has a length of 10 bytes. The available memory contains 1000 pages (i.e., it is large enough to accommodate roughly 100k records). The datasets are generated as follows. The domain of the join attribute A^{join} consists of integers in the range [1, 10000]. All algorithms adopt a hash function ($\mathcal{H}(t) = t.A^{join}$ *modulo* 20) that produces $n_{part} = 20$ partitions (each covering 500 A^{join} values). The first record is received at time 0, and two successive tuples are separated by an interval with length l_{intv}^{arr}, which may vary with time. Each record t belongs to R_1 (R_2) with probability $P(R_1)$ ($P(R_2)$), which is fixed during the entire execution. After deciding the relation (e.g., R_1) of t, the partition to which t belongs is selected according to probabilities $P(j|R_1)$ $(1 \leq j \leq n_{part})$ (i.e., $R_1^{mem}[j]$ is selected with probability $P(j|R_1)$). Finally, $t.A^{join}$ is set to a random value covered by the chosen partition. The total number of tuples is fixed to 2 million, i.e., the expected cardinality of R_1 (R_2) equals $2 \cdot P(R_1)$ $(2 \cdot P(R_2))$ million. We examine alternative algorithms with respect to the following stream characteristics.

- *Network reliability.* A *reliable network* never incurs suspensions of data transmission. For achieving this, we set l_{intv}^{arr} to a fixed value 10^{-3} (seconds). To simulate an *unreliable network*, we generate l_{intv}^{arr} in the range $[10^{-3}, 0.03]$ (seconds) according to the Zipfian distribution (skewed towards 10^{-3}), i.e., the longest transmission delay is 0.03 seconds. The algorithm handles a suspension after waiting for 0.025 seconds.

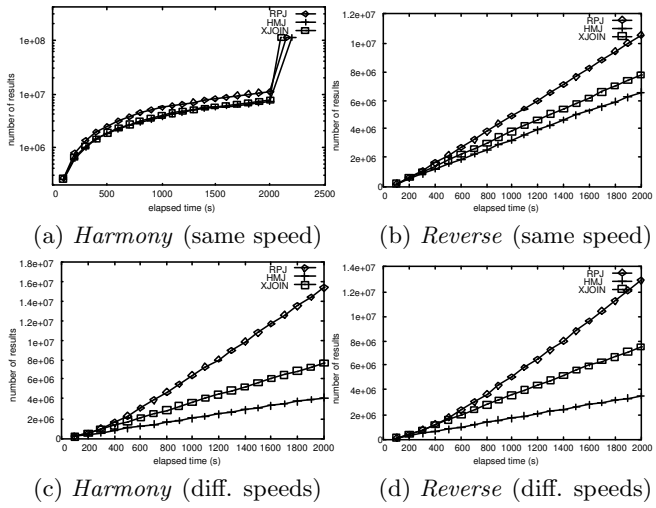
- *Arrival distribution.* We create arrival patterns *harmony* and *reverse* (discussed in Section 4.3) by first fixing $P(j|R_1)$ $(1 \leq j \leq 20)$ for R_1 and then adjusting $P(j|R_2)$ accordingly. In particular, $P(j|R_1)$ equals $1\% + (8/19)\% \times (j-1)$ (i.e., $P(20|R_1) = 9\%$ is the largest and $P(1|R_1) = 1\%$ is the smallest). For *harmony*, $P(j|R_2) = P(j|R_1)$ for all $(1 \leq j \leq n_{part})$, while for *reverse*, $P(j|R_2) = P(20-j|R_1)$.

- *Relative speed.* To create transmissions of R_1 and R_2 with different speeds, we vary the ratio between $P(R_1)$ and $P(R_2)$ (keeping $P(R_1) + P(R_2) = 1$). Following the settings in [12], we show the results with ratios 1 (i.e., the two streams are equally fast), and 5 (i.e., R_1 is 5 times faster).

We measure the quality of a method in terms of: (i) the "progressiveness", i.e., how fast the algorithm can deliver join results, and (ii) efficiency, i.e., what is the "amortized processing cost" for each result produced. In particular, the cost shown in the sequel involves both CPU and I/O time. We implement the following optimization (proposed in [12]) to reduce the CPU overhead. Each memory partition $R_i^{mem}[j]$ $(1 \leq i \leq 2, 1 \leq j \leq n_{part})$ is organized using another hash-table with 5 (sub-)partitions. Given an incoming tuple $t \in R_1$, for instance, approximately 1/5 of $R_2^{mem}[\mathcal{H}(t)]$ is inspected to find records of R_2^{mem} that can join with t. In the sequel, we present the results in two parts: Section 6.1 focuses on reliable networks, and Section 6.2 on unreliable transmission.

6.1 Reliable Networks

The first experiment uses the *harmony* dataset where both streams are equally fast (the last record arrives at the 2000-th timestamp). Figure 8a plots the number of reported tuples as a function of the elapsed time for RPJ, HMJ, and XJoin. RPJ produces the largest number of results during data transmission. All algorithms demonstrate similar performance in the "clean-up" stage (after timestamp 2000). This is expected because, the clean-up stage is in fact a join between two relations that have been completely received, for which the join algorithms deployed by the three methods have comparable performance.



(a) *Harmony* (same speed) (b) *Reverse* (same speed)

(c) *Harmony* (diff. speeds) (d) *Reverse* (diff. speeds)

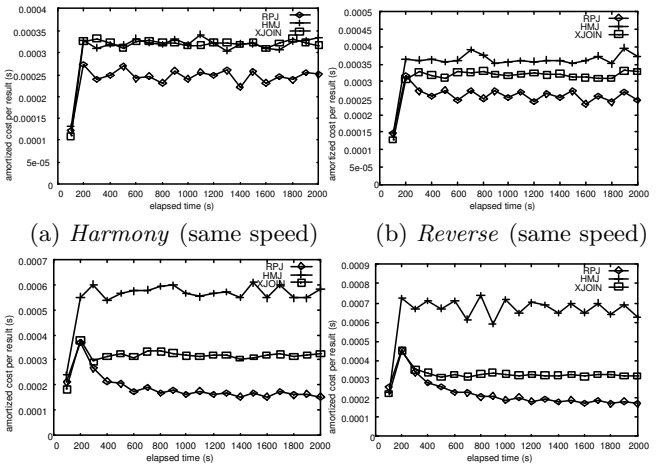
Figure 8: Progressiveness comparison (Reliable Networks)

Figure 8b illustrates the progressiveness of alternative methods for *reverse* till the end of data transmission. Since the performance of alternative methods in the clean-up state is similar to that shown in Figure 8a, the results for this stage are omitted to enhance the clarity of the diagram. It is worth mentioning that, since data delivery is never blocked, the mm-stage is the only stage in the entire execution of each approach (i.e., md- and dd-stages never occur). Hence, the efficiency of RPJ essentially demonstrates the superiority of *optimal flush*.

Figures 8c and 8d present the results of similar experiments for the case where R_1 is transmitted 5 times faster than R_2. Note that the numbers of join results differ from those in Figures 8a and 8b due to the difference in the cardinalities of the participating relations. RPJ again outperforms its competitors in all cases. Figure 9 shows the amortized cost (per result) as time evolves for the previous experiments. RPJ has the lowest overhead at all times because its result size is always larger than those of the competitors. The cost surge of each method at the initial join phase is caused by the flushing operation for handling the first memory overflow. Specifically, the processing cost involves only CPU time prior to the overflow, while the flushing performs considerable disk accesses, thus significantly increasing the average processing time. As time progresses, the increase in the result size compensates the flushing overhead, thus gradually stabilizing the amortized cost.

6.2 Unreliable Networks

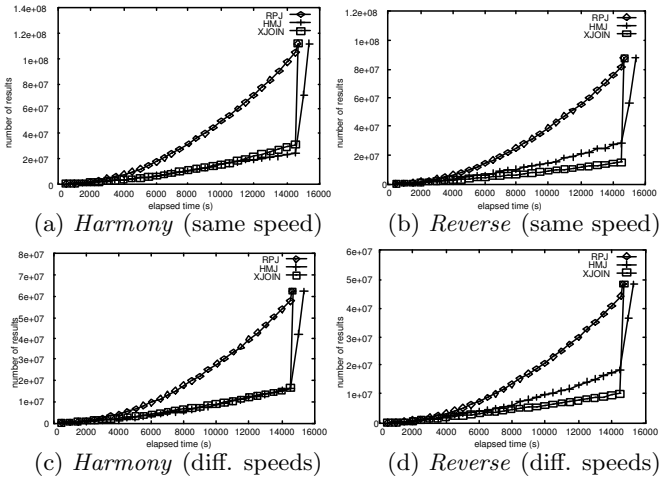
Having evaluated RPJ in the absence of transmission suspensions, we proceed to examine its performance for unreliable networks. Figure 10 shows the progressiveness during the entire join, for different arrival distributions and relative speeds of R_1 and R_2. The performance gain of RPJ over XJoin and HMJ is more significant than the reliable case, confirming that RPJ is superior not only in its flushing policy, but also in its reactive processing (at transmission delays). In particular, notice that before the clean-up stage (starting at around the 14000-th timestamp), RPJ has reported most of the results, while the other methods can produce only about 20% of the final output.



(a) *Harmony* (same speed) (b) *Reverse* (same speed)

(c) *Harmony* (diff. speeds) (d) *Reverse* (diff. speeds)

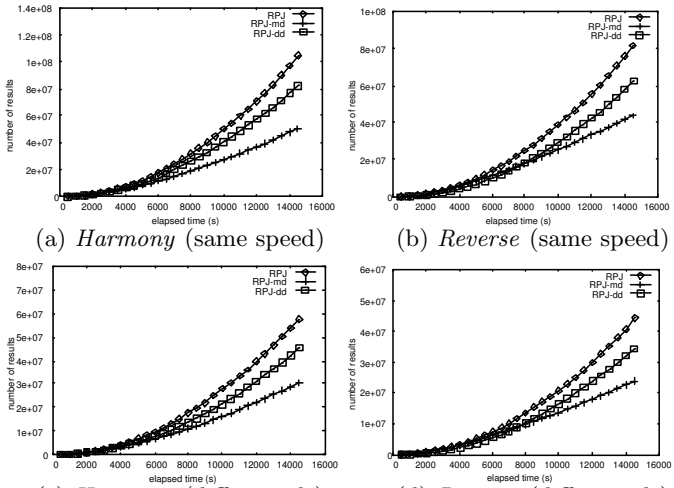
Figure 9: Efficiency comparison (Reliable Networks)



(a) *Harmony* (same speed) (b) *Reverse* (same speed)

(c) *Harmony* (diff. speeds) (d) *Reverse* (diff. speeds)

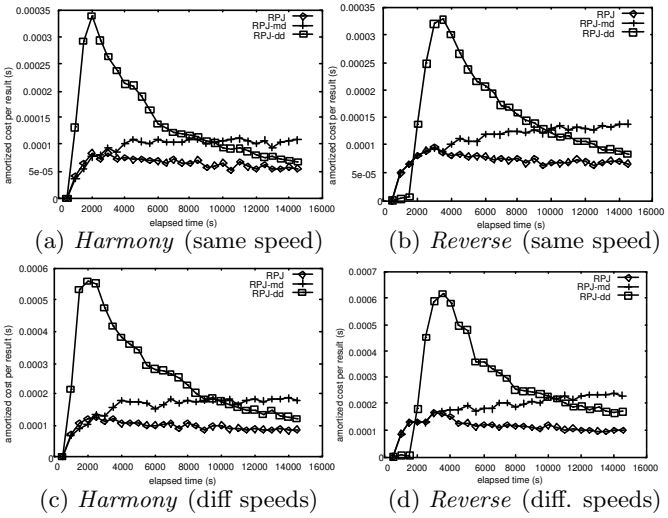
Figure 10: Progressiveness comparison (Unreliable Networks)



(a) *Harmony* (same speed) (b) *Reverse* (same speed)

(c) *Harmony* (diff. speeds) (d) *Reverse* (diff. speeds)

Figure 11: Progressiveness comparison of RPJ variants (Unreliable Networks)



(a) *Harmony* (same speed) (b) *Reverse* (same speed)

(c) *Harmony* (diff speeds) (d) *Reverse* (diff. speeds)

Figure 12: Efficiency comparison of RPJ variants (Unreliable Networks)

To further study the reactive characteristics of RPJ, we implement two interesting variations: RPJ-md (RPJ-dd), whose re-active stage chooses only md- (dd-) tasks. Particularly RPJ-md (RPJ-dd) can be regarded as an optimized version of XJoin (HMJ), since XJoin (HMJ) performs only md- (dd-) stages during the suspensions of data transmission. Figure 11 compares the progressiveness of RPJ, RPJ-md, and RPJ-dd using the same data as in Figure 10 (omitting the clean-up stage). Observe that RPJ-dd has a faster output rate than RPJ-md, which confirms the phenomenon in Figure 10 that HMJ has better progressiveness than XJoin. As expected, RPJ yields the largest output size by effectively mixing md- and dd-tasks.

Finally, Figure 12 plots the amortized costs of RPJ variations as a function of time. The overhead of RPJ-dd surges to a high value soon after the join starts. This is because, as shown in Figure 11, at the early stage of the join all algorithms output approximately the same number of results while RPJ-dd performs considerable random accesses (md-tasks involve mostly sequential accesses). As time progresses, however, the cost of RPJ-dd decreases, and eventually becomes lower than that of RPJ-md. These observations indicate that at the beginning of the join it is advantageous to invoke md-tasks (at transmission suspensions), while as more data are flushed, dd-tasks become more beneficial. RPJ combines the advantages of RPJ-dd and RPJ-md, and indeed has the smallest amortized cost.

7. CONCLUSIONS

This paper proposes RPJ, a novel algorithm for progressively joining stream relations. Unlike the previous heuristic approaches, RPJ is based on a probabilistic study of the problem characteristics that maximizes the output rate. We empirically verify that RPJ delivers results significantly faster than its competitors and incurs lower processing overhead. This work also initiates several directions for future work. For example, the existing algorithms focus on joins with equality conditions, while it would be interesting to investigate their extensions to range predicates (e.g., $R_1 \bowtie_{|R_1.A^{join} - R_2.A^{join}| < \epsilon} R_2$, where ϵ is a constant). Another challenging problem is the progressive join processing between multi-dimensional relations [11]. For instance, let R_1 (R_2) be a 2D dataset containing the location of hotels (restaurants). A "spatial distance join" would return all pairs of hotels t_1 and restaurants t_2 such that t_1 and t_2 are within 1 kilometers. Further, in practice, the volume of arriving tuples may exceed the computation capacity of the system, such that some data must be discarded. In this case, a load shedding technique should minimize the number of join outputs missed [3].

ACKNOWLEDGEMENTS

This work was fully supported by 3 grants from the Research Grants Council of Hong Kong SAR, China: CityU 1163/04E, HKU 7380/02E, and HKUST 6178/04E. We would like to thank the anonymous reviewers for their insightful comments.

8. REFERENCES

[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, pages 275–286, 1999.

[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] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *ICDE*, pages 350–361, 2004.

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

[5] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *VLDB*, pages 299–310, 2002.

[6] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer. On producing join results early. In *PODS*, pages 134–142, 2003.

[7] L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, 2003.

[8] S. Guha, C. Kim, and K. Shim. Xwave: Approximate extended wavelets for streaming data. In *VLDB*, pages 288–299, 2004.

[9] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in xprs. *Distributed and Parallel Databases*, 1(1):9–32, 1993.

[10] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.

[11] G. Luo, J. F. Naughton, and C. Ellmann. A non-blocking parallel spatial join algorithm. In *ICDE*, pages 697–705, 2002.

[12] 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.

[13] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *VLDB*, pages 324–335, 2004.

[14] N. Tatbul, U. Cetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.

[15] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *TKDE*, 23(2):27–33, 2000.

[16] S. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, pages 37–48, 2002.

[17] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2002.