PR-Join: A Non-Blocking Join Achieving Higher Early Result Rate with Statistical Guarantees

Shimin Chen
Intel Labs Pittsburgh
shimin.chen@intel.com

Phillip B. Gibbons
Intel Labs Pittsburgh
phillip.b.gibbons@intel.com

Suman Nath
Microsoft Research
sumann@microsoft.com

ABSTRACT

Online aggregation is a promising solution to achieving fast early responses for interactive ad-hoc queries that compute aggregates on a large amount of data. Essential to the success of online aggregation is a good non-blocking join algorithm that enables both (i) high early result rates with statistical guarantees and (ii) fast end-to-end query times. We analyze existing non-blocking join algorithms and find that they all provide sub-optimal early result rates, and those with fast end-to-end times achieve them only by further sacrificing their early result rates.

We propose a new non-blocking join algorithm, *Partitioned expanding Ripple Join* (PR-Join), which achieves considerably higher early result rates than previous non-blocking joins, while also delivering fast end-to-end query times. PR-Join performs separate, ripple-like join operations on individual hash partitions, where the width of a ripple expands multiplicatively over time. This contrasts with the non-partitioned, fixed-width ripples of Block Ripple Join. Assuming, as in previous non-blocking join studies, that the input relations are in random order, PR-Join ensures representative early results that are amenable to statistical guarantees. We show both analytically and with real-machine experiments that PR-Join achieves over an order of magnitude higher early result rates than previous non-blocking joins. We also discuss the benefits of using a flash-based SSD for temporary storage, showing that PR-Join can then achieve close to optimal end-to-end performance. Finally, we consider the joining of finite data streams that arrive over time, and find that PR-Join achieves similar or higher result rates than RPJ, the state-of-the-art algorithm specialized for that domain.

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems—*Query processing*; H.2.7 [**DATABASE MANAGEMENT**]: Database Administration—*Data warehouse and repository*

General Terms

Algorithms, Design, Performance, Theory

Keywords

PR-Join, Online Aggregation, Non-Blocking Join, Fast Early Result, Statistical Guarantee, Data Warehouse, Finite Data Stream

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

1. INTRODUCTION

Data warehousing and business intelligence (DWBI) is a fast growing multi-billion dollar market [7]. It is an increasingly high priority technology for enterprises because of the heightened need to make better, fact-based business decisions. DWBI queries are categorized into report generation and ad-hoc, interactive queries. Ad-hoc queries are important for detecting new trends and for designing new report generation queries for the new trends. While report generation queries are executed in a batch without user interaction, ad-hoc queries have two key differences: (i) fast response times are important for effective user interaction, and (ii) queries are composed on-the-fly, limiting the use of many pre-processing techniques (e.g., materialized views). Unfortunately, because the amount of digital information generated and managed by enterprises is increasing exponentially [13], achieving fast response times for ad-hoc, interactive queries is increasingly challenging.

Online aggregation [10, 11, 12, 14, 15] is a promising solution to this response time challenge for an important class of ad-hoc queries—computing aggregates (e.g., `COUNT`, `SUM`, and `AVERAGE`) on the output of select-project-join-groupby (SPJG) queries. In online aggregation, a DWBI system provides fast early estimates of the aggregates, together with accuracy metrics such as statistical confidence bounds, based on early representative results of the underlying SPJG query. For example, the system may display "average is 123.4 ± 5.6 with 95% confidence". The estimates and bounds are periodically updated as additional SPJG results are generated. A user can analyze the estimates and make early decisions about the query: (i) the query is not exactly what the user wants; (ii) early results of the query already satisfy the needs of the user; (iii) current estimates are not accurate enough; or (iv) the complete results of the query should be computed. In the first two cases, the user may stop the ongoing query early, while in the last case, the query is run to completion. Compared to traditional query processing, where users see only a progress bar during query execution, online aggregation not only facilitates user interactions but (to the extent that users stop their queries early) also saves time, computing resources, and energy. Thus, DWBI systems may support larger numbers of interactive users without increasing hardware resources or energy costs.

In this paper, we study non-blocking joins, which is essential to online aggregation. There are three requirements for a good non-blocking join algorithm that supports online aggregation:

1. Fast Early Results. Error bound formulae typically contain a $\frac{1}{\sqrt{N}}$ factor [10, 12], where N is the number of results that contribute to the estimate. This means that (i) the more results the join generates, the more accurate the estimate is (assuming the results are representative—see the next requirement), and (ii) to shrink the error bound at rate r, the join algorithm must generate join results at rate r^2. Moreover, orthogonal to the join algorithm design, several factors of the SPJG aggregate may have adverse impact on

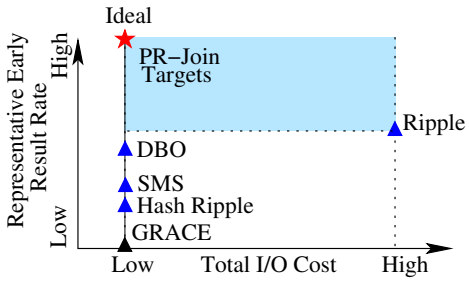


Figure 1: PR-Join targets $ResultRate$ higher than Ripple Join.

the accuracy convergence of estimates: high variance in the aggregated attributes, high selectivity of the selection criteria, high join selectivity, high group counts, and data skews across groups [15]. Therefore, providing a high early result rate is an important design goal of non-blocking join algorithms.

2. Representative Early Results. Following previous works [10, 14, 15, 20], we assume that the input relations to the join algorithm are in random order. Given this, the join algorithm must provide statistical guarantee: the generated early results must form a statistical sample of the complete set of results so that meaningful estimates and confidence bounds for the aggregates can be computed.

3. Good End-to-End Performance. As emphasized in [15], the time to complete the entire non-blocking join is important. Users may decide to obtain the complete results. Moreover, the convergence to accurate estimates may be slow because of the adverse factors discussed above. Therefore, the end-to-end performance of a non-blocking join algorithm should be comparable to that of the fastest blocking join.

In this paper, we focus on improvements with respect to the first requirement. Our goal is to design an algorithm that is capable of achieving result rates higher than previous algorithms while satisfying the second and the third requirements.

1.1 Existing Algorithms Cover Only Part of the Design Space

Figure 1 shows the design space of join algorithms. The X-axis is the end-to-end I/O cost, while the Y-axis is the result rate for generating representative early join results. We will define these terms formally in Section 2. As discussed above, an ideal solution would achieve both high representative early result rate and low total I/O costs. We depict the design points of existing join algorithms in the figure based on our analysis in Section 3.

Blocking join algorithms are used extensively in database systems. They are either hash-based or sort-based [8]. Hash-based algorithms are often variants of the GRACE hash join [4, 16], while sort-based algorithms are variants of the sort-merge join. We use GRACE hash join as a representative blocking join algorithm. As shown in Figure 1, it is optimized for low end-to-end I/O costs. However, GRACE join does not generate early join results when reading the input relations during the I/O partition phase (thus it is called a blocking algorithm).

Ripple Join is the first non-blocking join algorithm proposed for online aggregation [10]. When inputs can fit into memory, ripple join performs Symmetric Hash Join. It builds an in-memory hash table on every input relation. An incoming input record is inserted into the hash table of its relation, and is probed in the hash table of the other relation for matches. When inputs are larger than memory, they are spilled to temporary storage, and Block Ripple Join is employed, which involves scanning all spilled data for joining a block of new input records. As shown in Figure 1, Ripple Join incurs high I/O costs when inputs are larger than memory.

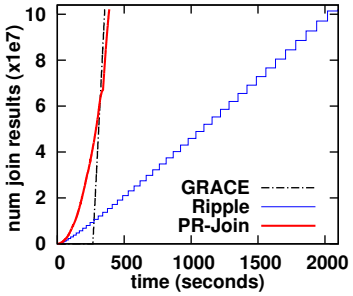


Figure 2: PR-Join versus Ripple Join and GRACE hash join, when joining two 10GB relations using an SSD for temporary storage. PR-Join provides significantly higher early result rate and good end-to-end performance.

Several non-blocking join algorithms, Hash Ripple Join [20], Sort-Merge-Shrink Join (SMS-Join) [15], and DBO [14], aim to avoid the high end-to-end I/O costs of Ripple Join while still producing representative early results. These algorithms avoid reading spilled data from temporary storage until all the inputs are processed. At this moment, they have essentially finished the partition phase of blocking join algorithms. Therefore, the join phase of the blocking algorithms can be applied to generate the remaining join results with minimal I/O costs. Unfortunately, because early results are generated without joining any spilled data, these algorithms sacrifice their early result rates, as depicted in Figure 1.

In summary, our analysis reveals that existing algorithms cover only part of the design space. In particular, the light-blue shaded region is completely empty. This paper targets design points in this empty region that are close to the ideal design point.

1.2 Our Solution: PR-Join

The behavior of a join algorithm can be visualized as covering the rectangular Cartesian product space, where the two input relations are pictured as two sides of the rectangle [10]. Then, the number of generated join results is roughly proportional to the covered area by the join algorithm. We observe that in a ripple-style join, as the width of a ripple step increases, the I/O cost of the ripple step increases linearly, while the newly covered area by the ripple step increases quadratically. Therefore, wider ripples lead to higher result rate. However, previous (Block) Ripple Join obtains only sub-optimal performance because (i) the width of a ripple in the (Block) Ripple Join is upper bounded by the available memory size, resulting in sub-optimal early result rate, and (ii) constant ripple width incurs quadratic number of total I/Os, resulting in poor end-to-end performance.

We propose a new non-blocking join algorithm, **Partitioned expanding Ripple Join (PR-Join)**. PR-Join breaks the "memory size barrier" by employing hash partitioning, and performing ripples on individual hash partitions. In this way, it can increase the ripple width greatly to achieve much higher result rate than prior algorithms. Moreover, PR-Join expands ripple width multiplicatively over time, thus achieving good end-to-end performance. We prove that PR-Join is capable of producing representative early results. We show both analytically and with real-machine experiments that PR-Join achieves higher early result rates and lower I/O costs than Ripple Join. We study configurations using only hard drives, as well as configurations using an **SSD** for temporary storage. An example of our findings is shown in Figure 2, which depicts cumulative join results as a function of time. Here, PR-Join returns results in the first 250 seconds at over an order of magnitude higher rate than Ripple Join, while nearly matching the end-to-end performance of GRACE. (Details are in Section 6.) Moreover, by varying

a parameter in PR-Join, we can obtain *a set of algorithms with different result rates and I/O costs* that all lie in the blue shaded region in the design space as shown in Figure 1.

Finally, several previous join algorithms (XJoin [26], Hash Merge Join [21], and RPJ [24]) aim to produce fast and early join results for **joining finite data streams** while disregarding statistical guarantees. RPJ has been shown to be the best among these algorithms [24]. Intuitively, statistical guarantees have negative impact on result rate. We extend PR-Join to this data streams setting and find in our experiments that PR-Join achieves comparable or higher result rates than RPJ while providing statistical guarantees.

1.3 Contributions

The paper makes the following main contributions. First, we formally define the result rate metric (Section 2) and compare existing join algorithms analytically, e.g., proving that Ripple Join has a higher early result rate than Hash Ripple Join, SMS-Join, and two-way DBO (Section 3). Second, we propose and analyze a non-blocking join algorithm, *Partitioned expanding Ripple Join (PR-Join)*, that achieves higher representative early result rates and lower I/O costs than Ripple Join (Section 4). Third, we describe a set of optimization techniques for implementing PR-Join, such as exploiting SSDs as temporary storage (Section 5). Fourth, we perform extensive real machine experimental evaluations (Section 6). Our experimental results show that PR-Join achieves over an order of magnitude higher early result rate than Ripple Join. With SSDs as temporary storage, PR-Join further achieves close to optimal end-to-end performance, nearly obtaining the ideal case in the design space. Finally, we evaluate PR-Join in the simulation framework as used in RPJ for joining finite data streams. We find that PR-Join achieves comparable or higher result rate than RPJ while providing statistical guarantees (Section 6).

2. DEFINITIONS AND TERMS

Our study considers equality joins of two finite relations, each of which is considerably larger than the available main memory. Moreover, as is typically true in realistic cases, the input relations are not so overwhelmingly large that they cannot be partitioned into memory-sized partitions in one pass (e.g., at most 1 PB relation for systems with 32GB main memory).

Cartesian Product Space. The behavior of a join algorithm can be illustrated in the Cartesian product space, where the two input relations are depicted as two sides of the rectangular space. A point in the space represents a pair of input tuples. Therefore, the space covers all the possible combinations of input tuple pairs. Of course, only a portion of the points are matching tuple pairs according to the join. The task of a join algorithm is to cover the entire Cartesian product space in order to find out all the matching tuple pairs.

Statistical Guarantee. Following previous works [10, 14, 15, 20], we assume that the input relations to the join algorithm are in random order. Under this condition, we say that a join algorithm provides *statistical guarantee* for online aggregation if early results generated by the join algorithm can be used to compute meaningful estimates and confidence bounds for aggregates (e.g., `COUNT`, `SUM`, `AVERAGE`, and `STD_DEV`). (Note that the random order assumption is used solely for statistical guarantees purposes, and in practice can be approximated, e.g., either because the physical layout is not correlated with the join aggregate or by reading segments of each relation in a random order [10].)

There are two ways to design join algorithms to provide statistical guarantee in the literature. The first way was proposed in the Ripple Join work [10]. Every algorithmic step in the join algorithm covers a (growing) **rectangular sub-space** in the Cartesian product space. Estimates of aggregates with confidence bounds can be computed if the inputs on the two sides of the rectangle are randomly selected from the two input relations [10]. Since we assume that the input relations are in random order, this condition is easily satisfied by simply scanning the two relations. SMS-Join proposed the second method for achieving the statistical guarantee [15]. Instead of growing a single rectangle, SMS-Join covers multiple independent rectangles in the Cartesian product space. For individual rectangles, the Ripple Join formulae are applied to compute estimates with confidence bounds. Then these independent estimates can be combined into a single estimate [15].

Performance Metrics. We define two performance metrics for the two performance requirements discussed in Section 1: (i) fast early results, and (ii) good end-to-end performance. In our analytical models, we focus on I/O performance but ignore in-memory CPU performance because I/O is often the performance bottleneck when input data are much larger than main memory.

To model end-to-end performance, we compute the total I/O time for performing a join algorithm from beginning to end:

$$TotalIO \stackrel{\text{def}}{=} \text{total I/O time from start to finish}$$

Result rate of an algorithmic step can be computed as the number of new results generated in the step divided by the I/O cost of the step.[1] The number of join results generated is proportional to the covered area in the Cartesian product space ($num\ results = covered\ area * selectivity$), where $selectivity$ is the number of join results per unit area in the Cartesian product space. Note that join algorithms may re-arrange the order to process input records through hash partitioning or sorting. After such re-arranging, some regions in the Cartesian product space have more join results than average, while other regions become empty. To account for this effect, we introduce a $density$ factor:

$$density = \frac{number\ of\ results\ in\ a\ region}{selectivity * area\ of\ the\ region}$$

$density = 1$ for rectangles with two sides representing random input records. Let us consider hash partitioning. Suppose input relations A and B are hash partitioned into n partitions. Let A_i and B_i be the corresponding partitions. Suppose the hash function is random. Then A_i is a random subset of A records, and $density_{A_i \times B} = 1$. B_i contains all matches for A_i, while $B_j (j \neq i)$ does not have any matching records. Since $A_i \times B_i$ has $\frac{1}{n}$ of the area of $A_i \times B$, we have $density_{A_i \times B_i} = n$, while $density_{A_i \times B_j} = 0 (j \neq i)$. We define result rate of an algorithmic step as follows:

$$ResultRate = selectivity \cdot \frac{\sum_i newly\ covered\ area_i * density_i}{IO\ time\ of\ the\ step}$$

Since $selectivity$ is a constant given the query and inputs, we often omit $selectivity$ in our analysis.

We are more interested in early $ResultRate$. We call an algorithmic step *early* if input data have not yet been fully read. $ResultRate^{early}$ is the $ResultRate$ in an early algorithmic step.

Terms Used in the Analysis. Table 1 lists the terms used in analyzing join algorithms. The two input relations are represented as A and B. Without loss of generality, we assume that relation A is the smaller one if they are not the same size.

To simplify our analytical models, we do not model selections or projections. Note that these operations can be included into the models by reducing the intermediate data sizes with certain factors.

[1] It would be more end-user relevant if we could use statistical confidence bounds in our analysis. However, confidence bounds are typically complex non-linear functions on the number of generated results. Because more results lead to better bounds, we use result rate in our analysis.

Table 1: Terms used in analyzing join algorithms.

A, B	Two input relations to the join algorithm
$\|A\|, \|B\|$	Relation size in pages. Assume $\|A\| \leq \|B\|$.
M	Memory size in pages that are allocated to the join
F	fudge factor for hash table in memory, e.g., 1.2
n	Number of hash partitions
A_i, B_i	the i-th partition of $A(B)$
A^m, B^m	in-memory portion of $A(B)$
A^{sp}, B^{sp}	portion of $A(B)$ that are spilled to temporary storage
A_i^m, B_i^m	in-memory portion of $A_i(B_i)$
A_i^{sp}, B_i^{sp}	portion of $A_i(B_i)$ that are spilled to temporary storage
a, b	$a = \frac{\|A\|M}{(\|A\|+\|B\|)F}$, $b = \frac{\|B\|M}{(\|A\|+\|B\|)F}$
T_{ri}	amortized time to **r**ead a page from **i**nput relations
T_{wt}	amortized time to **w**rite a page to **t**emporary storage
T_{rt}	amortized time to **r**ead a page from **t**emporary storage

Moreover, we assume that there are no data skews when deriving the analytical models. That is, $|A_i| = |A|/n$ and $|B_i| = |B|/n$. We will discuss data skew handling in Section 5.1.

For non-blocking joins, the two input relations are often read together to generate early join results. a (b) is the maximum in-memory portion A^m (B^m) before any data are spilled to temporary storage. Suppose the speeds to read the two relations are proportional to their sizes and hash tables are built on in-memory data, then we can compute $a = \frac{|A|M}{(|A|+|B|)F}$ and $b = \frac{|B|M}{(|A|+|B|)F}$, so that $a + b = M/F$.

We use three time terms, namely T_{ri}, T_{wt}, and T_{rt}, to account for the fact that (i) the storage device containing the input relations and the storage device for the temporary intermediate data may have different I/O performance; and (ii) read size for input storage, write size for temporary storage, and read size for temporary storage may be different, resulting in different per-page latencies. Note that we assume the output of the join is consumed by the aggregate computations in memory without I/Os. Therefore, we do not model the I/O cost of the join output.

3. COST ANALYSIS OF EXISTING JOIN ALGORITHMS

In this section, we analyze the early result rate and total I/O costs of representative blocking and non-blocking join algorithms.

3.1 Blocking Joins

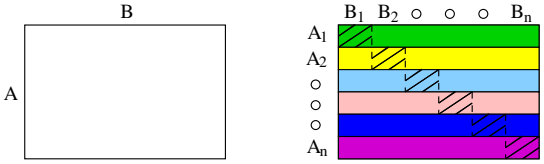
Blocking joins are extensively used in commercial database systems. Hash joins are often preferred if there are no indices on the join attribute and the input relations are not ordered according to the join attribute. We analyze GRACE hash join [16] because it defines the basic building blocks of all the hash join algorithms.

GRACE hash join consists of two phases:

(1) Partition Phase:
Hash Partition A into n partitions so that $\frac{|A|}{n} \leq \frac{M}{F}$;
Hash Partition B into n partitions using the same hash function;
(2) Join Phase:
For each pair of A_i and B_i do
Read A_i into memory and build hash table;
Scan B_i while probing A_i's hash table for matches;
/* Representative result reporting point */

Figure 3 illustrates the join behavior of the two phases in Cartesian product space.

The partition phase takes $(|A| + |B|)(T_{ri} + T_{wt})$ time, but does not generate any join results. Therefore, the Cartesian product space is still empty in Figure 3(a). The join phase takes $(|A| + |B|)T_{rt}$ time, while covering the entire $|A||B|$ area. As shown in Figure 3(b), the join phase processes the tuples one partition after another. Because of hash partitioning, join results are only found in the shaded areas, which have $density = n$ as discussed in Section 2. A loop iteration that joins A_i and B_i effectively covers an entire stripe of space, $A_i \times B$. (Because density=0 outside the shaded areas, the choice of $A_i \times B$ as the covered area is equivalent to other choices such as $A \times B_i$.) Therefore, the I/O cost of an iteration is $\frac{|A|+|B|}{n}T_{rt}$, and the newly covered areas multiplied by their densities is $\frac{|A||B|}{n}$.



(a) Partition phase (b) Join phase

Figure 3: GRACE hash join in Cartesian product space.

In summary, the performance metrics of GRACE hash joins can be computed as follows:

$$\begin{aligned} TotalIO &= (|A| + |B|)(T_{ri} + T_{wt} + T_{rt}) \\ ResultRate^{early}_{partition} &= 0 \\ ResultRate_{join} &= \frac{|A||B|}{|A|+|B|} \cdot \frac{1}{T_{rt}} \end{aligned} \quad (1)$$

Note that GRACE hash join is *blocking* because it generates join results only after reading all input data.

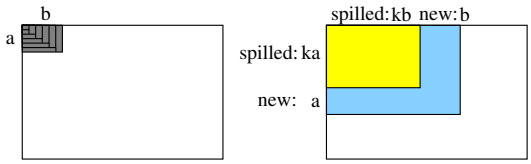
3.2 Non-Blocking Joins

In this paper, we consider non-blocking joins on finite input data that can eventually compute all matches. The existing non-blocking join algorithms can be categorized into two classes. The first class aims to generate early representative results for online aggregation. The algorithms are designed to provide statistical guarantee. This class includes Ripple Join [10], Hash Ripple Join [20], Sort-Merge-Shrink Join (SMS-Join) [15], DBO [14] and Turbo DBO [6]. The second class aims to generate fast early results, while ignoring the statistical properties of the results, including XJoin [26], Hash Merge Join [21], RPJ [24], and Early Hash Join [18]. Most of these algorithms target finite data stream processing. Since the second class does not provide statistical guarantee, we mainly focus on analyzing the join algorithms in the first class. However, we find in our experimental study in Section 6 that our proposed solution generates comparable or faster results than join algorithms designed for finite data streams.

3.2.1 Ripple Join

Ripple Join [10] consists of two phases: (i) Symmetric Hash Join when input data fit in memory; and (ii) Block Ripple Join when input data need to be spilled to temporary storage.

(1) Symmetric Hash Join when $|A^m| + |B^m| \leq \frac{M}{F}$:
For each incoming tuple t from A (B) do
Insert t into $A^m(B^m)$'s hash table;
Use t to probe $B^m(A^m)$'s hash table for matches;
/ Early representative result reporting point */*
(2) Block Ripple Join when input data are spilled:
While (inputs $\neq \emptyset$) do
Spill A^m and B^m to A^{sp} and B^{sp}; (†)
$A^m = \emptyset$; $B^m = \emptyset$;
While ($|A^m| + |B^m| < \frac{M}{F}$) do (‡)
Accumulate new incoming tuples in memory and build hash tables for new A^m and B^m;
Scan A^{sp} while probing B^m's hash table;
Scan B^{sp} while probing A^m's hash table;
Scan A^m in memory while probing B^m's hash table;
/ Early representative result reporting point */*



(a) Symmetric hash join when input data fit in memory (b) Block ripple join after data are spilled to storage

Figure 4: Ripple Join in Cartesian product space.

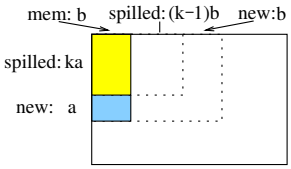
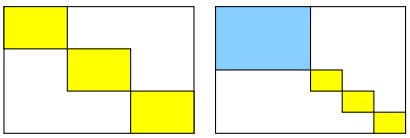


Figure 5: Hash Ripple Join when data are spilled.



(a) Run generation (b) Merge and shrink

Figure 6: SMS-Join in Cartesian product space.

Figure 4 illustrates the behavior of Ripple Join in the Cartesian product space. Representative results (with statistical guarantee) can be reported only at the end of each ripple after completing a rectangular subspace, as marked by the reporting point comment. In Symmetric Hash Join, the I/O cost is $(a+b)T_{ri}$, and the covered area is ab, where a (b) is defined in Table 1 to be the maximum $|A^m|$ ($|B^m|$) when memory is filled. In Block Ripple Join, a block of new input data are joined among themselves and with all the spilled data in every iteration. We choose (at step ‡) the largest possible block size, $\frac{M}{F}$, in order to maximize the early result rate. Therefore, the number of iterations is $\frac{|A|+|B|}{block\ size}-1 = \frac{(|A|+|B|)F}{M}-1$. In an iteration, step † spills $a+b = \frac{M}{F}$ data to temporary storage, taking $(a+b)T_{wt}$ time. In the k-th iteration, after step †, $|A^{sp}|+|B^{sp}| = k(a+b)$, as shown in Figure 4(b). Accumulating new incoming tuples take $(a+b)T_{ri}$ time. Scanning A^{sp} and B^{sp} take $k(a+b)T_{rt}$ time. Therefore, the I/O cost of the k-th iteration in the second stage is $(a+b)(T_{ri}+T_{wt}+kT_{rt})$. As shown in Figure 4(b), the covered area by the k-th iteration is the blue area, which is equal to $(2k+1)ab$.

The result rates of Symmetric Hash Join and the k-th iteration of Block Ripple Join can be computed as follows. ($\frac{ab}{a+b} = \frac{|A||B|M}{(|A|+|B|)^2F}$)

$$\begin{aligned} ResultRate^{early}_{symhash} &= \frac{ab}{(a+b)T_{ri}} = \frac{|A||B|M}{(|A|+|B|)^2F}\cdot\frac{1}{T_{ri}} \\ ResultRate^{early}_{blkrpl.kth} &= \frac{(2k+1)ab}{(a+b)(T_{ri}+T_{wt}+kT_{rt})} \\ &= \frac{|A||B|M}{(|A|+|B|)^2F}\cdot\frac{2k+1}{T_{ri}+T_{wt}+kT_{rt}} \end{aligned} \quad (2)$$

Note that if we choose a block size that is a factor of s smaller than $a+b$, then $ResultRate^{early}_{blkrpl.kth}$ would be roughly a factor of s smaller. Therefore, our choice maximizes the early result rate.

Let $l = \frac{(|A|+|B|)F}{M}$. There are $l-1$ iterations. The end-to-end performance of Ripple Join is computed as follows:

$$\begin{aligned} TotalIO &= (a+b)T_{ri} + \textstyle\sum_{k=1}^{l-1}(a+b)(T_{ri}+T_{wt}+kT_{rt}) \\ &= (a+b)[lT_{ri} + (l-1)T_{wt} + \tfrac{l(l-1)}{2}T_{rt}] \\ &\simeq (|A|+|B|)(T_{ri}+T_{wt}) + \tfrac{(|A|+|B|)^2F}{2M}T_{rt}, \text{if } l \gg 1 \end{aligned} \quad (3)$$

Compared to the performance of GRACE hash join in Equation 1, Ripple Join produces early results but incurs much worse end-to-end performance, performing a quadratic number of I/Os for reading from the temporary storage.

3.2.2 Hash Ripple Join

Hash Ripple Join [20] reduces the end-to-end I/O cost of Ripple Join by avoiding Block Ripple Join when data are spilled. The algorithm consists of three phases. The first phase of Hash Ripple Join is the same as Ripple Join. In the second phase, Hash Ripple Join avoids reading data from temporary storage. As illustrated in Figure 5, it only extends the covered area in one dimension, while spilling input data to temporary storage. In Figure 5, the algorithm is processing $(a+b)$ new input data after $k(a+b)$ data have been read. The I/O cost of processing $(a+b)$ new input data is $(a+b)(T_{ri}+T_{wt})$. The covered area is the blue area, which is equal to ab. Note that the newly spilled data of B are not used in the second phase. The third phase of Hash Ripple Join is similar to the join phase in GRACE hash join except that B^m is not used. Therefore, the I/O cost is $(|A|+|B|-b)T_{rt}$ and the covered area is $|A|(|B|-b)$.

(1) Symmetric hash join when $|A^m|+|B^m| \le \frac{M}{F}$
(2) When input data overflow the memory:
Write A^m to A^{sp} with partitioning, keep B^m in memory;
For each incoming tuple t do
If (t is from A)
Then probe B^m's hash table for matches;
Write t to A^{sp} with partitioning;
/ Early representative result reporting point */*
Else write t to B^{sp} with partitioning;
(3) When all input data are read:
Join each pair of spilled partitions as in GRACE hash join;

The performance metrics of Hash Ripple Join are computed as follows.

$$\begin{aligned} TotalIO &= (|A|+|B|)T_{ri} + (|A|+|B|-b)(T_{wt}+T_{rt}) \\ &\simeq (|A|+|B|)(T_{ri}+T_{wt}+T_{rt}), \text{ if } b \ll |B| \end{aligned} \quad (4)$$

$$\begin{aligned} ResultRate^{early}_{symhash} &= \frac{|A||B|M}{(|A|+|B|)^2F}\cdot\frac{1}{T_{ri}} \\ ResultRate^{early}_{hashrpl} &= \frac{ab}{(a+b)(T_{ri}+T_{wt})} \\ &= \frac{|A||B|M}{(|A|+|B|)^2F}\cdot\frac{1}{T_{ri}+T_{wt}} \\ ResultRate_{join} &\simeq \frac{|A||B|}{|A|+|B|}\cdot\frac{1}{T_{rt}}, \text{ if } b \ll |B| \end{aligned} \quad (5)$$

Comparing Equations 4 and 3, we see that Hash Ripple Join does not have the quadratic term. In fact, it does not incur more I/O overhead than GRACE hash join. However, comparing Equations 5 and 2, we see that the second phase of Hash Ripple Join has lower early result rate than Block Ripple Join because this phase covers only a small (blue) fraction of the possible area (i.e., the entire dotted region) in order to avoid I/O overhead.

3.2.3 Sort-Merge-Shrink Join (SMS-Join)

SMS-Join [15] is based on standard sort merge join. It consists of the following two phases:

(1) Sort Phase: generate sorted runs
While (input data $\neq \emptyset$) do
Read $|A^m| = aF$ ($|B^m| = bF$) data from A (B);
Sort and merge A^m and B^m to find matches;[2]
Spill A^m and B^m as new runs;
/ Early representative result reporting point */*
(2) Merge-Shrink Phase
Merge sorted runs, shrink estimates from every run;

The join behavior is shown in Figure 6. In the Sort Phase, the join results from each run consists of an independent sample (without replacement) of the complete join results. Therefore, the estimates from every run can be combined into a single estimate [15]. A run generation costs $(aF+bF)(T_{ri}+T_{wt})$ I/O time, while covering

[2]While the original SMS used symmetric hash join to find matches, for our purposes, using merging slightly improves the performance metrics.

Table 2: Comparing early result rates. ($c = \frac{|A||B|M}{(|A|+|B|)^2FT}$)

	$F = 1.2, T_{ri} = T_{wt} = T_{rt} = T$
Symmetric Hash	c (Note: must fit in memory)
Hash Ripple	$0.5c$
SMS	$0.6c$
Two-Way DBO	$1.2c$
Block Ripple	$\frac{2k+1}{k+2}c : c, 1.25c, 1.40c, 1.50c, 1.57c, \ldots$
PR-Join ($\gamma = 1$)	$c, 1.7c, 3.2c, 6.2c, 12.2c, \ldots$

$aF \cdot bF$ area. Note that the factor F is because hash tables are not needed. In the Merge-Shrink Phase, records from all sorted runs are merged. As shown in Figure 6(b), the blue area represents the area covered by records already used in the merge, while the yellow areas are the area covered by sorted runs with records not used in the merge yet. As the merge proceeds, the blue area grows while all the yellow areas shrink until the blue area covers the entire Cartesian product space. At any moment, independent estimates can be computed from the blue and all the yellow areas, which are then combined into a single estimate. The Merge-Shrink phase takes $(|A| + |B|)T_{rt}$ I/O time, while covering $|A||B| - \frac{(|A|+|B|)abF}{a+b}$ new area. The performance metrics can be computed as follows:

$$TotalIO = (|A| + |B|)(T_{ri} + T_{wt} + T_{rt}) \tag{6}$$

$$\begin{aligned} ResultRate^{early}_{sms.sort} &= \frac{abF^2}{(a+b)F(T_{ri}+T_{wt})} \\ &= \frac{|A||B|M}{(|A|+|B|)^2F} \cdot \frac{F}{T_{ri}+T_{wt}} \\ ResultRate_{sms.merge} &= \frac{|A||B| - \frac{|A||B|M}{(|A|+|B|)}}{(|A|+|B|)T_{rt}} \\ &\simeq \frac{|A||B|}{|A|+|B|} \cdot \frac{1}{T_{rt}}, \text{ if } |A| + |B| \gg M \end{aligned} \tag{7}$$

Like Hash Ripple Join, SMS-Join does not perform additional I/Os for reading temporary storage. However, the covered area in the Sort Phase is limited.

3.2.4 Two-Way DBO

DBO [14] and Turbo DBO [6] mainly focus on multi-way joins, which is beyond the scope of this paper. In this paper, we focus on two-way joins. Turbo DBO is not applicable to two-way joins because its central idea "partial match" exists only when joining three or more relations. Two-way DBO can be regarded as an improved SMS algorithm. We focus on its sort phase (a.k.a. scan phase) to compute the early result rate for two-way DBO:

Two-Way DBO Sort Phase: generate sorted runs
Read $|A^m| = aF$ data from A;
While (input data $\neq \emptyset$) do
  Read $|B^m| = bF$ data from B and find matches with $|A^m|$;
  Spill A^m as a new run A_i;
  Read $|A^m| = aF$ data from A and find matches with $|B^m|$;
  Spill B^m as a new run B_i;
  $i++$;

Compared to SMS-Join, Two-Way DBO alternates the spilling and reading between A and B. In this way, it not only computes the join results between A_i and B_i, but also computes the join results between B_i and A_{i+1}, effectively doubling the early result rate of SMS-Join:

$$ResultRate^{early}_{dbo.sort} = 2 \cdot ResultRate^{early}_{sms.sort} \tag{8}$$

3.2.5 Comparing Non-Blocking Joins

Table 2 compares the early result rates of the non-blocking join algorithms. We see that Hash Ripple, SMS, and two-way DBO obtain significantly lower early result rate than Block Ripple Join. We seek both a higher result rate and a lower total I/O cost than Block Ripple Join.

4. PR-JOIN DESIGN AND ANALYSIS

We propose *Partitioned expanding Ripple Join* (PR-Join) in this section. We discuss our design decisions in Section 4.1, then describe the algorithm in Section 4.2. In Section 4.3, we analyze the statistical properties of PR-Join. Finally, in Section 4.4, we develop analytical performance models of PR-Join and compare PR-Join with previous join algorithms analytically.

4.1 Design Decisions

How to Achieve Higher Result Rate? In the analysis of Block Ripple Join in Section 3.2, we observe that early result rate increases as the block size (i.e. ripple width). However, block size is limited by the memory size M. Can we do better?

Let us perform a thought experiment that *magically* removes this limitation. As shown in Figure 7(b), suppose the join algorithm has finished joining X and Y, which are now on temporary storage. In the current step, the algorithm obtains input X_{new} and Y_{new}, where $|X_{new}| + |Y_{new}|$ is larger than memory. Suppose the algorithm spills all X_{new} and Y_{new} to temporary storage, then reads X, X_{new}, Y, and Y_{new} from temporary storage while joining them for covering the entire colored area. The step takes $(|X_{new}| + |Y_{new}|)(T_{ri} + T_{wt} + T_{rt}) + (|X| + |Y|)T_{rt}$ I/O time, covering $(|X_{new}| \cdot |Y| + |X| \cdot |Y_{new}| + |X_{new}| \cdot |Y_{new}|)$ new area. Therefore, its result rate is as follows:

$$ResultRate^{early}_{magic} = \frac{|X_{new}| \cdot |Y| + |X| \cdot |Y_{new}| + |X_{new}| \cdot |Y_{new}|}{(|X_{new}|+|Y_{new}|)(T_{ri}+T_{wt}+T_{rt})+(|X|+|Y|)T_{rt}}$$

We introduce two parameters, μ and ν, given our assumption that the read speeds of the input relations are proportional to their sizes. Let

$$|X| = \mu b, |Y| = \mu a, |X_{new}| = \nu|X|, |Y_{new}| = \nu|Y|.$$

Then, we have the following:

$$\begin{aligned} ResultRate^{early}_{magic} &= \frac{ab}{a+b} \cdot \frac{\mu(\nu^2+2\nu)}{\nu(T_{ri}+T_{wt}+T_{rt})+T_{rt}} \\ &= \frac{|A||B|M}{(|A|+|B|)^2F} \cdot \frac{\mu(\nu^2+2\nu)}{\nu(T_{ri}+T_{wt}+T_{rt})+T_{rt}} \end{aligned} \tag{9}$$

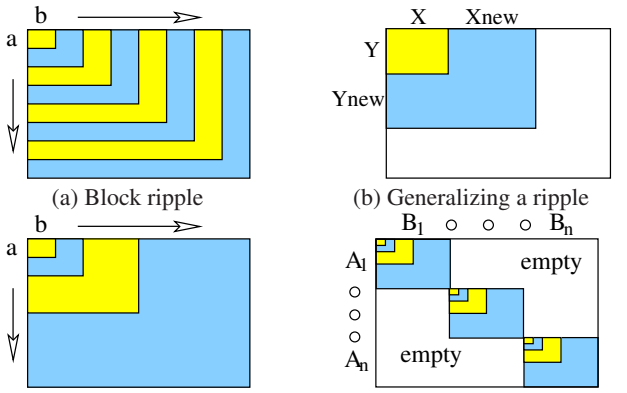
Note that this equation is an extension to the Block Ripple result rate. If we set $\mu = k$ and $\nu = \frac{1}{\mu}$, and we do not read X_{new} and Y_{new} from temporary storage, we can obtain the Block Ripple result rate in Equation 2.

Given the same condition as in Table 2, $ResultRate^{early}_{magic}$ is $\frac{\mu(\nu^2+2\nu)}{3\nu+1}c$. Note that μ is defined as the starting data size of the ripple (relative to memory size). Therefore, $ResultRate^{early}_{magic}$ increases with a factor of $\frac{\nu^2+2\nu}{3\nu+1}c$ as the data size grows. In Block Ripple Join, the choice of $\nu = \frac{1}{\mu}$ offsets this growth. To achieve high result rate, we choose a constant ν — the width of a ripple expands multiplicatively over time, as shown in Figure 7(c).

How to Make the Magical Algorithm Practical? We face two problems. First, how to realize the join of X, X_{new}, Y, and Y_{new} given limited memory size? Second, the I/O cost of a magic ripple step increases as the new data size (if ν is fixed). However, to maintain statistical guarantee, the algorithm can only compute the estimate of the desired aggregate when a magic ripple step completes. Therefore, the interval between two reporting points is longer, incurring slower responses to front-end users. The second problem is how to reduce the execution time of a magic ripple step.

We solve both problems by exploiting *hash partitioning*: performing hash partitioning for all spilled data, and more importantly, executing the *magic* algorithm for a partition if the partition satisfies a *join invocation condition* (e.g., $\nu \geq threshold$).

Figure 7(d) illustrates the basic idea. Conceptually, we view the inputs as re-arranged according to the n hash partitions. Join results



(a) Block ripple (b) Generalizing a ripple

(c) Width expands multiplicatively (d) Employing hash partitioning

Figure 7: PR-Join design decisions.

can only appear in the shaded regions. As input data come, all the partitions grow at similar speed. We perform expanding ripples on individual partitions. The first problem is solved because joining a partition requires only $\frac{1}{n}$ of memory space. The second problem is addressed because estimates can be updated after completing the magic ripple in *individual* partitions. Therefore, the report interval between two estimates can be reduced roughly by a factor of n.

We call this algorithm *Partitioned expanding Ripple Join* (PR-Join). Note that this algorithm contrasts with the non-partitioned, fixed-width ripples of Block Ripple Join. Although hash partitioning was extensively used in previous join algorithms, to our knowledge, our proposal is the first to exploit hash partitioning for faster representative early results.

4.2 PR-Join Algorithm

PR-Join algorithm consists of the following three phases.

(1) Symmetric Hash Join when $|A^m| + |B^m| \leq \frac{M}{F}$
(2) When input data overflow the memory:
Remove A^m's and B^m's in-memory hash tables for more space;
for $i = 1$ to n do
$oldA_i = |A_i^m|$; $oldB_i = |B_i^m|$; $newA_i = newB_i = 0$;
While (input data $\neq \emptyset$) do
Read record t;
$i = HashPartition(t)$;
If (t is from A)
Then $newA_i$++; Append t to A_i^m;
Spill oldest page of A_i^m to A_i^{sp} if no space;
Else $newB_i$++; Append t to B_i^m;
Spill oldest page of B_i^m to B_i^{sp} if no space;
If (JoinInvokeCond($oldA_i$, $oldB_i$, $newA_i$, $newB_i$)==True)
Then Join(i);
$oldA_i += newA_i$; $oldB_i += newB_i$;
$newA_i = newB_i = 0$;
(3) When input data are all read:
for $i = 1$ to n do Join(i);

Subroutine: Join(i):
While (free space < $|A_i^{sp}|F + |A_i^m|(F-1)$) do
Choose $j \neq i$ and spill one page of partition j;
Read A_i^{sp} into memory and build in-memory hash table for A_i;
Scan B_i^m and B_i^{sp} while probing the hash table for matches
and avoiding matches between previously joined records;
/ Early representative result reporting point */*

PR-Join performs symmetric hash join when input data fit in memory. When input data are larger than memory, the second phase of the algorithm hash partitions the input data. If a partition satisfies the join invocation condition, then the *Join* subroutine is invoked for generating join results among all the present records in the partition. We discuss the join invocation condition in Section 4.4. In the third phase when all input data are read, the *Join* subroutine is invoked for generating all remaining join results.

The algorithm maintains old and new record counts for every partition for two purposes. First, they are used to evaluate the join invocation condition. Second, the $Join$ subroutine ensures to generate join results *exactly once* by comparing record IDs against the counts to distinguish old records from new records.

4.3 Statistical Guarantee

We focus on sub-space $A_i \times B_i$ as shown in Figure 7(d). Given that the function $HashPartition()$ randomly maps records to partitions, the input records in A_i are randomly chosen from the records in A. Since the records in A are in random order, the records in A_i are also in random order. Similarly, the input records in B_i are in random order. Moreover, every invocation of $Join(i)$ covers a (growing) rectangle in $A_i \times B_i$. Therefore, Ripple Join formulae can be applied to $A_i \times B_i$ for computing the estimates and confidence bounds for aggregates on the join results of $A_i \bowtie B_i$.

The join results in $A_i \times B_i$ ($i = 1, ..., n$) are an independent subset of the complete set of the join results of $A \bowtie B$. SMS-Join [15] formulae can be applied to combine the estimates from individual partitions into a single estimate. Therefore, PR-Join provides statistical guarantee for online aggregation.

4.4 Cost Analysis

Analyzing a Join Step. In the second phase, reading input data is performed in the main code, while reading temporary storage is performed only in the $Join(i)$ subroutine. However, spilling data to temporary storage may happen in both the main code and the $Join(i)$ code. For simplicity of analysis, we define a conceptual join step for $Join(i)$ to include reading new input data of A_i and B_i, spilling them to temporary storage, and reading data in A_i and B_i from temporary storage for $Join(i)$ processing.

We can use the same notation as in Figure 7(b), while imagining the entire rectangle represents $A_i \times B_i$. Note that on average $\frac{a+b}{n}$ new data are still in memory. Therefore, the I/O cost of the join step is $(|X_{new}| + |Y_{new}|)(T_{ri} + T_{wt}) + (|X_{new}| + |Y_{new}| + |X| + |Y| - \frac{a+b}{n})T_{rt}$. The covered new area is $(|X_{new}| \cdot |Y| + |X| \cdot |Y_{new}| + |X_{new}| \cdot |Y_{new})|$. Note that the density of the area is n because of hash partitioning, as discussed in Section 2. Therefore, we have the following:

$$ResultRate_{pr}^{early} = \frac{n(|X_{new}|\cdot|Y|+|X|\cdot|Y_{new}|+|X_{new}|\cdot|Y_{new}|)}{(|X_{new}|+|Y_{new}|)(T_{ri}+T_{wt})+(|X_{new}|+|Y_{new}|+|X|+|Y|-\frac{a+b}{n})T_{rt}}$$

The size of the partition is $\frac{a+b}{n}$ at the end of Symmetric Hash Join. Let $|X| = \mu\frac{b}{n}$, $|Y| = \mu\frac{a}{n}$, $|X_{new}| = \nu|X|$, and $|Y_{new}| = \nu|Y|$.

$$ResultRate_{pr}^{early} = \frac{|A||B|M}{(|A|+|B|)^2 F} \cdot \frac{\mu(\nu^2+2\nu)}{\nu(T_{ri}+T_{wt}+T_{rt})+(1-\frac{1}{\mu})T_{rt}} \quad (10)$$

This is the same equation as Equation 9 except for the $\frac{1}{\mu}$ term representing the savings of keeping portions of partition i in memory.

Join Invocation Condition. We set ν to be a constant parameter γ. We would like to use the following join invocation condition: $\frac{new\ data\ size}{old\ data\ size} \geq \gamma$. It can be rewritten as $\frac{current\ data\ size}{old\ data\ size} \geq 1+\gamma$, where current data include both old and new data.

However, since every partition grows at similar speed, this join invocation condition is likely to become true for all partitions at almost the same time. The algorithm would alternate between two states: (i) input reading and spilling without generating any join

results; and (ii) invoking $Join$ subroutine for all partitions. This would result in a long time between two reporting points. To avoid this behavior, we stagger the invocations of the first $Join$ for different partitions so that all the join invocations of different partitions are distributed evenly.

Moreover, we introduce a stop constraint: when the algorithm almost completes reading the input data, we need to be cautious about invoking $Join$ subroutine because phase three may be too close and the current invocation could be wasteful.

The join invocation condition for j-th join step for partition i ($i = 0, 1, ...$) is as follows:

$$\begin{cases} \text{Return true:} & \frac{current\ data\ size}{old\ data\ size} \geq (1+\gamma)^{1+\frac{i}{n}}, j = 1 \\ \text{Return true:} & \frac{current\ data\ size}{old\ data\ size} \geq 1+\gamma, j > 1 \\ \text{Return false:} & \frac{(|A|+|B|)/n}{current\ data\ size} < 1+\gamma \ \text{(stop constraint)} \end{cases} \tag{11}$$

The first $Join$ is invoked for partition i when current data size is $\frac{a+b}{n}(1+\gamma)^{1+\frac{i}{n}}$. Subsequent $Join$ is invoked when the data size increases by a factor $(1+\gamma)$. Therefore, $Join$ is invoked for partition i when current data size is $\frac{a+b}{n}(1+\gamma)^{j+\frac{i}{n}} (j \geq 1)$.

We consider all the $Join$ invocations for all n partitions. The first $Join$ invocation occurs (for partition 0) when current data size is $(a+b)(1+\gamma)$. After that, there is a $Join$ invocation whenever current data size grows $(1+\gamma)^{\frac{1}{n}}$ times. Thus join invocations are indeed distributed evenly.

Result Rate Given the Join Invocation Condition. We compute μ and ν for the j-th join invocation of partition i:

$$\begin{cases} \text{When } j = 1: & \mu = 1, \nu = (1+\gamma)^{1+\frac{i}{n}} - 1 \\ \text{When } j > 1: & \mu = (1+\gamma)^{j-1+\frac{i}{n}}, \nu = \gamma \end{cases} \tag{12}$$

Note that Equation 10 increases as μ or ν increases. Since $\mu \geq (1+\gamma)^{j-1}$ and $\nu \geq \gamma$, we can compute the lower bound of the result rate. For example, if $\gamma = 1$, then the lower bound of result rate is $2^{j-1}\frac{3}{4-2^{1-j}}c$ given the same condition as in Table 2. By assigning j to be 1, 2, ..., we obtain c, $1.7c$, $3.2c$, $6.2c$, $12.2c$, ..., increasing exponentially. The maximum j is determined by input data size. For example, if the input data size is 32 times memory size, then $j_{max} = 5$ and $12.2c$ result rate can be achieved. Clearly, PR-Join achieves much higher early result rates, compared to those of Block Ripple Join in Table 2.

Total I/O Time. Note that input data are read once and at most spilled to temporary storage once. This part of the I/O cost is $(|A|+|B|)(T_{ri}+T_{wt})$. On the other hand, data on temporary storage may be read multiple times in the $Join$ invocations. We compute total number of I/O reads from temporary storage for partition i:

$$ReadTemp_i \leq \sum_{j=1}^{l} \frac{a+b}{n}(1+\gamma)^{j+\frac{i}{n}} + (|A_i| + |B_i|)$$

where l is the maximum number of join steps for partition i in phase two. Denote $q = 1+\gamma$, $p = \frac{a+b}{n}(1+\gamma)^{\frac{i}{n}}$, and $z = |A_i| + |B_i|$. Then the stop constraint requires $pq^{l+1} \leq z$. Therefore, we have

$$ReadTemp_i \leq p\sum_{j=1}^{l} q^j + z < p\frac{q^{l+1}}{q-1} + z \leq (\frac{1}{q-1}+1)z = (1+\frac{1}{\gamma})z$$

The total I/O cost can be computed as follows:

$$TotalIO \leq (|A|+|B|)(T_{ri} + T_{wt} + (1+\tfrac{1}{\gamma})T_{rt}) \tag{13}$$

We see that PR-Join achieves much better total I/O cost than Ripple Join: Compared to GRACE join, PR-Join incurs only an additional $\frac{1}{\gamma}T_{rt}$ cost per input page.

Impact of Different γ. As shown in Equation 13, larger γ leads to smaller total I/O costs. On the other hand, since current data size grows $(1+\gamma)^{\frac{1}{n}}$ between two join invocations, larger γ leads to sparser join invocations. Consider the case where two PR-Join with different γ's have both processed the same amount of input data. The partitions of the PR-Join with the larger γ must be joined longer time ago. Therefore, it must have generated fewer join results. Note that it may take the two joins different time to reach this point. Nevertheless, depending on the actual system configuration, larger γ tends to give lower result rate.

5. PR-JOIN OPTIMIZATIONS

In this section, we describe optimization techniques in PR-Join implementations. We not only discuss PR-Join implementation in DWBI systems (including the use of SSDs for temporary storage), but also describe how to adapt PR-Join for joining data streams.

5.1 PR-Join Optimizations in DWBI Systems

We implemented PR-Join to model the online aggregation in a DWBI environment. We describe several issues in the following.

Handling Data Skews. Data skews can result in skewed hash partition sizes, which leads to large variance of result rate and execution times for joining an individual partition. We avoid this problem by performing a skew remapping step in the PR-Join algorithm. The skew remapping step is performed at the point where the memory is half full during the Symmetric Hash Join phase. Since input records are in random order, the join keys in memory at this point form a representative sample of the entire set of input join keys. We build a histogram on the join keys to re-assign them evenly to hash partitions. At this moment, input data are all in memory, and half of the memory is still available. Therefore, we can re-partition the input in memory without incurring additional I/Os. After that, PR-Join uses the remapped hash function for partitioning input data.

Fine-Grain Memory Management. Given a temporary storage I/O size S, partition i is allocated at least $2S$ memory for A_i^m and B_i^m. Naively, one may organize this memory into two S-sized buffers for A_i^m and B_i^m, respectively. When a buffer is filled, it is spilled to the corresponding partition on temporary storage. However, it may happen that both buffers of a partition are just spilled. Therefore, in the worst case, the amount of memory-resident data per partition is close to 0. What we want is to keep as much data in memory as possible to reduce the number of I/O reads from the temporary storage for the join invocations. We achieve this goal by using fine-grain 4KB page sized memory management for every partition. A_i^m and B_i^m share all the 4KB buffers that consist of the $2S$ space in memory. Whenever $2S$ space is full, the larger of A_i^m and B_i^m must have at least S data. Therefore, S data is spilled from the larger of A_i^m and B_i^m. In this way, in the worst case, there are at least S memory-resident data per partition.

More Flexible Join Invocation Conditions. The behavior of PR-Join can be modified with different join invocation conditions. The join invocation condition with constant parameter γ is just one example. For example, we can design another condition to incorporate front-end user feedbacks. Users may want to see the estimates after certain percentage of inputs have been processed, while not caring about estimate updates before this point. Then we can use the join invocation condition to stop the join invocations until the point is reached.

I/O Parallelism. Our algorithm analysis in this paper assumes sequential I/Os. However, our implementations of all the join algorithms are capable of exploiting the I/O parallelism across multiple I/O devices and within a single device. For example, the input storage and the temporary storage may be separate devices. In such situations, our implementations exploit the parallelism by issuing asynchronous read/write I/O requests (e.g., through `libaio`).

5.2 Exploiting SSDs as Temporary Storage

Recent works have studied flash-based devices, such as Solid State Drives (SSDs), in various aspects of data management [1, 2, 3, 9, 17, 19, 22, 23, 25]. However, since SSDs are more expensive than HDDs in terms of $/GB, we expect that the majority of the data in a large DWBI system will still be stored in HDDs (at least in the near future), while SSDs can be employed judiciously for improving the performance of critical operations and data structures [2]. For example, SSDs are an attractive option for storing intermediate results during join processing.

Using an SSD for temporary storage has two main advantages over a hard drive. First, (current and future generations of) SSDs have higher I/O bandwidth than hard drives because SSD bandwidth is not limited by mechanical operations. Note that compared to the I/Os in GRACE join, the extra I/Os in non-blocking joins are all I/O reads from temporary storage. Faster temporary storage provides two benefits: (i) it reduces the absolute time of the extra I/O reads; (ii) it better overlaps the I/O reads from temporary storage with the reading from the (possibly slower) input storage.

Second, peak performance on SSDs can be achieved with much smaller sized I/Os. For example, one can achieve the peak performance of an Intel X25E SSD using 1MB sized reads, as well as using over eight concurrent 64KB sized reads (I/O parallelism). The I/O size determines the buffer size per partition in memory. Given a fixed memory size, $\frac{1}{k}$ of I/O size means k times maximum number of partitions, and thus k times maximum input data size can be handled with peak I/O performance. Moreover, the individual partition size reduces by a factor of $\frac{1}{k}$. The join invocation time for an individual partition reduces accordingly, leading to shorter intervals between join result generation and faster user response times.

5.3 PR-Join on Finite Data Streams

Like PR-Join, existing progressive join algorithms for finite data streams (e.g., RPJ [24], XJoin [26], Hash-Merge Join [21]) assume limited memory and hence migrate additional tuples to disk. The join execution switches between three stages. The *mm-stage* is active as long as the data transmission is not suspended and it joins tuples that are currently in-memory from both streams. The *md-stage* and the *dd-stage* are active when data transmission is suspended or completed. In the md-stage, in-memory tuples from one stream are joined with in-disk tuples from the other stream. Similarly, in the dd-stage, in-disk tuples from both streams are joined together.

Different progressive join algorithms differ in their flushing strategies and the order in which they invoke different join stages. These two design decisions are optimized with the general goal of producing early join results at a high rate. For example, RPJ dynamically chooses join stages and join buckets that maximize the output rate. It also flushes data from memory to disk intelligently—it prefers keeping tuples that are likely to join with future tuples in memory so that the default mm-stage can output at a high rate.

Unlike non-blocking joins designed for online aggregation, progressive joins do not provide statistical guarantee. For example, RPJ may choose to keep one particular bucket of tuples in memory (as that bucket is highly likely to join with future tuples) and flush others to disk; then the output of the default mm-stage will be statistically biased towards tuples falling into that particular bucket. Therefore, the generated results are not statistically representative of the entire join results.

Intuitively, statistical guarantees tend to have negative impacts on result rate because they require certain join behaviors. Interestingly, PR-Join, in addition to providing statistical guarantees, can achieve better early result rate than RPJ, the state-of-the-art progressive join algorithm. This is due to the following two reasons. First, PR-Join combines all mm-, md- and dd-stages for a bucket together in a single join invocation, incurring less overall I/O cost. For example, let m_i (d_i) denote the in-memory (in-disk) part of bucket i. Let (m_i, d_i) denote the join operation of m_i of the first stream and d_i of the second stream. (d_i, m_i) and (d_i, d_i) are defined similarly. Then, an instance of RPJ may decide to invoke different join stages in this order: $\ldots$, (m_1, d_1), (d_1, m_1), (d_5, d_5), $\ldots$, (d_1, d_1), $\ldots$, which results in reading d_1 of both streams twice for a single join operation on bucket 1. The last dd-stage in the above example needs to re-read d_1 of both streams from disk because they are evicted from memory to make space for (d_5, d_5) due to limited memory. In contrast, PR-Join always joins the entire bucket together as $(m_1 \cup d_1, m_1 \cup d_1)$, reading d_1 of each stream only once for joining bucket 1. Second, PR-Join uses hash partitioning and ensures that individual partitions fit into memory, which allows using efficient hash join on the partitions. In contrast, RPJ's buckets can be larger than the available memory and therefore they are joined using more expensive Progressive Sort Merge Join [5] during the dd-stage.

We have extended PR-Join for finite data streams. In such environments, input data may be stalled for a long time due to network congestion. Therefore, we modify the join invocation condition as defined in Section 4.4: We perform joins not only when the join invocation condition is true, but also when the system detects that the input is stalled (based on a timeout value).

6. EXPERIMENTAL EVALUATIONS

We evaluate PR-Join in two environments. We perform real machine experiments to model DWBI environments in Section 6.1. Then in Section 6.2, we study PR-Join on finite data streams using the simulation framework obtained from the authors of RPJ [24], the state-of-the-art algorithm specialized for this purpose.

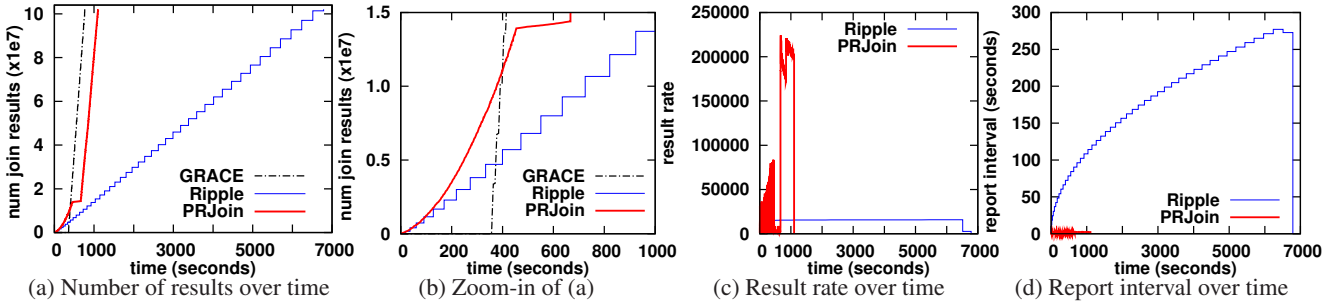
6.1 PR-Join in DWBI Environments: Real Machine Experiments

Machine Configuration. We perform all real machine experiments on a Dell Precision 690 workstation, which contains a quad-core Intel Xeon 5345 CPU (2.33GHz, 8MB L2 cache, 1333MHz FSB), 4GB DRAM, and four SATA drives. One drive stores system files and programs. We use the other three in our experiments: two 200GB 7200rpm Seagate Barracuda disks (77MB/s read/write bandwidth per disk) and one Intel 32GB X25-E SSD (250MB/s read and 170MB/s write bandwidth). The workstation runs Ubuntu 8.04.1 (Hardy) with Linux 2.6.24 kernel. All code is compiled with g++ 4.2.4 compiler with "-O2" optimization.

Join Implementations. We implemented GRACE, Ripple Join, and PR-Join as described in Section 3 and Section 4. As analyzed in Section 3, GRACE achieves low total I/O cost, while Ripple Join achieves the best representative early result rate among existing non-blocking joins. These two extreme cases mark the boundary of the empty region in the design space, which is PR-Join's target.

As discussed in Section 5.2, we believe that most data in DWBI systems will still be stored on hard drives (at least in the near future). Therefore, input relations are stored on one Barracuda disk in our experiments. On the other hand, we perform experiments using two types of temporary storage: the other Barracuda disk or the SSD. The join outputs are consumed in memory, modeling online aggregation. To avoid the caching effects of the operating system, our join implementations open raw devices with `O_DIRECT` flag. `libaio` asynchronous I/Os are used unless otherwise noted.

In our default configuration, we allocate 500MB memory space for the join operation. The two input relations are stored sequen-



(a) Number of results over time (b) Zoom-in of (a) (c) Result rate over time (d) Report interval over time

Figure 9: Disk as temporary storage. (10GB joins 10GB, $\gamma = 1$)

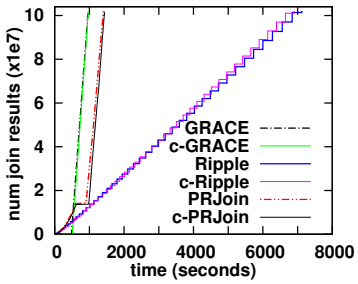


Figure 8: Verifying analytical models with experimental results. (10GB joins 10GB, temp disk, $\gamma = 1$)

tially on the input disk. Therefore, the join implementations perform 1MB sized I/O reads for the inputs. GRACE hash join use 1MB sized buffers for partitions, performing 1MB sized read and write I/Os to the temporary storage. Ripple Join performs sequential 1MB sized I/O reads and writes to the temporary storage (after data are spilled). Note that large I/O size achieves higher I/O bandwidth for both hard disks and SSDs. Therefore, our implementations maximize the performance of GRACE and Ripple Join.

PR-Join allocates $2S$ memory space for each pair of A_i^m and B_i^m, where S is the extent size for temporary storage. S is 1MB with disk as temporary storage, while S can be 4KB to 1MB when SSD is used as temporary storage. When the space is filled, S data from the larger of A_i^m and B_i^m will be spilled to temporary storage. We maintain 1MB sized spill buffer to accumulate spills from all partitions. When the spill buffer is full, we perform 1MB sized writes to the temporary storage. The temporary storage is read with S sized I/Os.

In most experiments, we join two 10GB input relations. A record is 100 bytes, consisting of a randomly generated 4-byte join key and a payload attribute. Every record in relation A matches one record in relation B. The record orders in the two relations are random. In addition, we also perform sensitivity analysis by varying the input relation size and the join key distribution.

Verifying Analytical Models. Figure 8 compares the computed performance using our analytical models (with "c-" prefix) and the experimental results. Since our model assumes synchronous I/Os, we replace `libaio` with `pread/pwrite` system calls for performing synchronous I/Os in this set of experiments. (All other experiments use the default `libaio` implementations.) In the computation, we use the following disk parameters measured by micro-benchmarks: (i) sequential 1MB read latency is 13.687ms; (ii) sequential 1MB write latency is 13.757ms; (iii) random 1MB read latency is 25.515ms. Fudge factor for the input is 1.29 in our implementation. As shown in Figure 8, we see that our analytical models match the real machine results very well. The predicted end-to-end time of Ripple Join is slightly smaller because our analytical model ignores CPU cost, which shows up slightly after a large number of Block Ripple iterations.

Disk as Temporary Storage. Figure 9 compares the performance of PR-Join with GRACE and Ripple Join when disk is used as temporary storage. Figure 9(a) and (b) show the cumulative join results over time, while Figure 9(c) and (d) compare result rates and report intervals of the algorithms, respectively. We see that GRACE does not generate any results until I/O partitioning phase completes. After that, it joins pairs of partitions and quickly generates join results, achieving low end-to-end time. Ripple and PR-Join report join results only at the reporting points, where statistical computations are meaningful. Ripple join suffers from poor end-to-end time. The increasing step sizes of Ripple curve is because the higher and higher cost of performing a Block Ripple iteration.

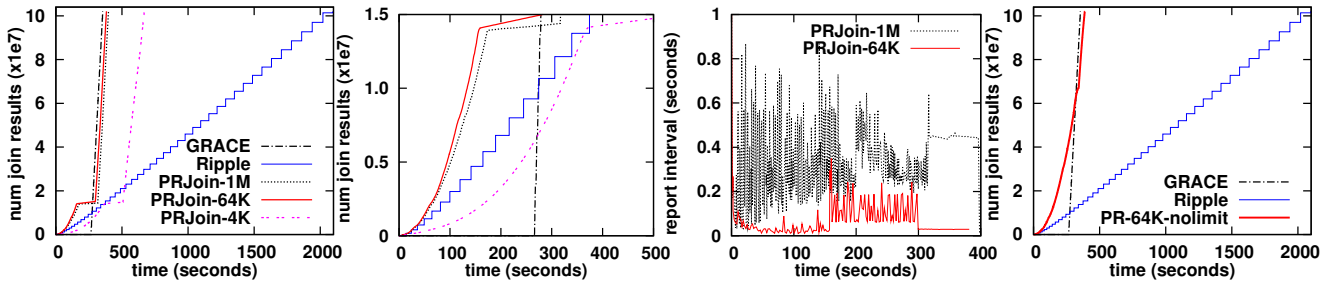
PR-Join achieves better end-to-end performance and better result rate than Ripple Join. The flat part of PR-Join is when it hits the stop constraint of the join invocation condition. The end-to-end time of PR-Join is 6.2x faster than Ripple Join, which is only 44% slower than GRACE. Moreover, as shown in Figure 9(c) we see that PR-Join achieves orders of magnitude higher result rate than Ripple Join. Furthermore, as shown in Figure 9(d), we see that PR-Join achieves orders of magnitude lower report intervals through partitioning, thus providing a smoother result generation curve for better front-end user experience.

SSD as Temporary Storage. Figure 10 compares the performance of PR-Join with GRACE and Ripple Join when SSD is used as temporary storage. As shown in Figure 10(a), comparing PR-Join and Ripple Join, we reach similar conclusions as in the case of disk as temporary storage: PR-Join achieves faster representative early results and shorter end-to-end time than Ripple Join. Compared to GRACE, the end-to-end performance of PR-Join is now very close to GRACE, because the SSD can process I/Os faster than the input disk, thus more effectively overlapping the I/Os of temporary storage with the input I/Os.

We vary the temporary storage extent size in Figure 10(a) and (b). We see that both 64KB and 1MB extent sizes achieve similar performance, while 4KB extent size performs significantly worse. This is because even with 32 parallel requests, 4KB reads can obtain only 60% of the SSD's peak read bandwidth. Figure 10(c) compares 64KB and 1MB extent size in terms of report intervals. We see that 64KB extent size achieves significantly shorter report intervals. As discussed in Section 5.2, smaller extent size results in larger number of partitions, thus processing time is shorter for joining individual partitions.

In Figure 10(d), we remove the stop constraint from the join invocation condition in PR-Join. The result is *almost an ideal PR-Join curve*: achieving smooth, fast representative early results, while obtaining close to optimal end-to-end performance. This is because SSD successfully overlaps the extra reads to temporary storage with input reading.

Given the above results, we use 64KB extent size and remove the stop constraint in PR-Join in the rest of the SSD experiments.



(a) Number of results over time (b) Zoom-in of (a) (c) Report interval comparison (d) Removing the stop constraint

Figure 10: SSD as temporary storage. (10GB joins 10GB, $\gamma = 1$)

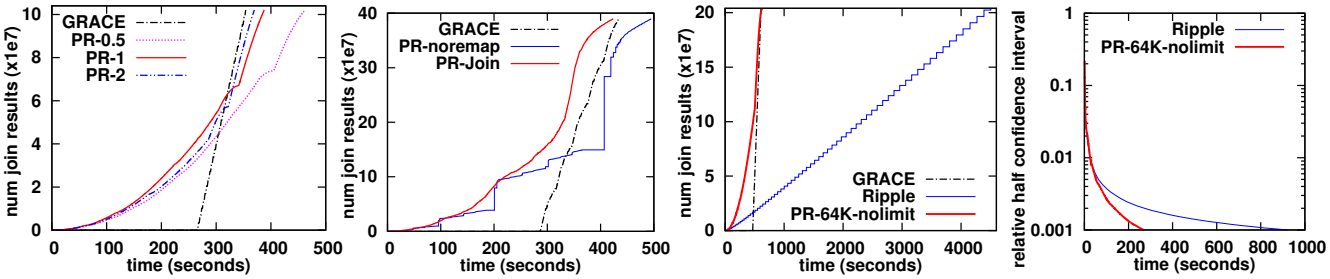


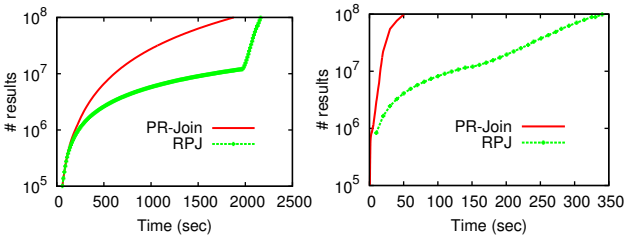
Figure 11: Varying γ. **Figure 12: Skewed join keys.** **Figure 13: 10G joins 20G.** **Figure 14: Confidence interval.**

Varying γ. Figure 11 varies γ for the PR-Join curve in Figure 10(d). By default, we use $\gamma = 1$ in the other experiments. Here, we compare $\gamma = 1$ with $\gamma = 2$ and $\gamma = 0.5$. As discussed in Section 4.4, increasing γ results in faster end-to-end performance but probably lower early result rate. This is clearly shown in the comparison of $\gamma = 1$ with $\gamma = 2$. However, the case of $\gamma = 0.5$ not only has longer end-to-end time, but also has lower early result rate. This is because when $\gamma = 0.5$, PR-Join performs dramatically more reads from temporary storage for performing the join invocations more frequently than $\gamma = 1$. It is impossible to overlap all the SSD I/Os with input reading, thus resulting poorer performance.

Handling Data Skews. Figure 12 shows the PR-Join performance with join key skews using otherwise the same setting as in Figure 10(d). For the particular hash function used in PR-Join, we generate a skewed key input according to Zipf distribution. As a result the largest partition contains more than 1GB of data each from A and B. There are also significant duplicates in the join keys. Therefore, the number of join results is about 4 times of the default configuration. In Figure 12, we compare PR-Join that detects data skews and remap data to hash buckets as described in Section 5.1, with GRACE and a variant of PR-Join that does not do skew remapping. The latter two joins are allowed to use more memory for joining partitions to avoid costly re-partitioning. In other words, we compare PR-Join against two enhanced baselines. From Figure 12, PR-join without skew remapping generates large step-shape curves. The reasons are two-fold: (i) it takes a long time for the smaller partitions to satisfy the join invocation condition; (ii) the larger partitions take longer time to do a join. In contrast, PR-Join achieves good end-to-end time and smooth early results.

Varying Relation Size. In the above experiments, we use equal-sized input relations, where the Cartesian product space is square. Figure 13 shows the results of joining 10GB A with 20GB B, where the Cartesian product space is rectangle. From Figure 13, we can reach similar conclusions as from Figure 10(d): PR-Join achieves almost ideal performance with fast, smooth representative early results and close to optimal end-to-end performance.

Impact on Confidence Intervals. Figure 14 compares the convergence of confidence intervals for PR-Join and Ripple Join. The experimental setting is the same as in Figure 10(d), where the query is "SELECT SUM(B.val) FROM A, B WHERE A.key=B.key". B.val is randomly generated according to the Pareto distribution with shape parameter 1.5. Pareto distribution has been observed in many social and scientific phenomena. For such a query, online aggregation reports the estimated sum to be in $[s-\epsilon, s+\epsilon]$ with 95% probability, where s is the estimated sum, and ϵ is the half confidence interval. The Y-axis in Figure 14 reports ϵ/s, the relative half confidence interval. As shown in Figure 14, to converge to a relative half confidence interval of 0.001, PR-Join is 3.4x faster than Ripple Join. This is because PR-Join can generate much higher representative early result rate.



(a) Slow streams (b) Fast streams

Figure 15: Comparison of PR-Join and RPJ.

6.2 PR-Join on Finite Data Streams: Simulation Study

To evaluate how PR-Join performs in a streaming scenario, we compare it with RPJ, the state-of-the-art progressive join algorithm. We use the same RPJ simulator, data set, and experimental settings used in [24]. The memory/page size is fixed to 1024 bytes. Each record has a length of 12 bytes. The available memory contains 1000 pages. Each stream contains 1 million tuples. The join attribute is an integer within the range [1,10000].

In the first set of experiments, we consider both streams to have the same data distribution (called the *harmony* scenario in [24]). We consider two scenarios. In the *slow streams* scenario, successive tuples in a stream have a fixed inter-arrival time of 1ms (called

the *reliable* scenario in [24]). In the *fast streams* scenario, one packet containing 100 tuples arrive every 1ms, and hence the average inter-arrival time of tuples is 0.01ms. Figure 15 shows the results of these two experiments. As shown, the output rate of PR-Join is significantly faster than RPJ for the both scenarios. This can be explained by the fact that with a very small inter-tuple arrival time, RPJ never invokes md- or dd-stages before streams terminate (at time 2000 and 20 in Figures 15(a) and 15(b), respectively), and hence its output rate is limited by the in-memory tuples. RPJ's output rate dramatically increases only after both the streams terminate, at which point RPJ starts using the tuples in disk for join execution. In contrast, PR-Join outputs results based on both in-memory and in-disk tuples from the very beginning, and hence yields a very high output rate from the very beginning. Even after the streams terminate, PR-Join requires less time to finish the entire join operation, as shown in Figure 15(b). This is because i) it reads in-disk part of each bucket exactly once (in contrast RPJ can read some buckets twice), and ii) it uses hash join for joining individual buckets (in contrast, RPJ uses more expensive Progressive Sort Merge Join [5]).

We have also considered a few additional experimental setups considered in [24]. We considered a setup where one stream is 5 times faster than the other, a setup where one stream has an opposite data distribution as the other (denoted as *reverse* in [24]), and a setup where inter-arrival time of successive tuples follow a Zipf distribution (denoted as *unreliable* in [24]). In the *unreliable* streams setup, RPJ and PR-Join show very similar output rates. Since unreliable streams occasionally suspend, the slow arrival rate becomes the bottleneck for join processing. Hence, RPJ can invoke md- and dd-stages to produce almost as many results as produced by PR-Join. In all other setups (with reliable streams), PR-Join shows a significantly higher output rate than RPJ. We omit the graphs here for lack of space (the graphs have similar trends as Figure 15(a)).

7. CONCLUSION

In this paper, we analyze existing non-blocking joins for online aggregation in DWBI environments, using a new metric, representative early result rate. We find that a large region of the design space is still empty. We propose a new non-blocking join algorithm, *Partitioned expanding Ripple Join* (PR-Join), targeting this empty region. Both analytical study and real-machine experiments show that PR-Join achieves an order of magnitude higher early result rates than previous non-blocking joins. Moreover, we exploit SSDs as temporary storage for join operations. Real machine experiments show that PR-Join achieves close to optimal end-to-end performance. As a result, we are able to obtain a nearly ideal design in the design space. Furthermore, we extend PR-Join for joining finite data streams. We find in our experiments that PR-Join achieves comparable or higher performance than the state-of-the-art algorithm specialized for this purpose.

In conclusion, PR-Join is a simple non-blocking join algorithm with good analytical properties that can efficiently handle two very different usage scenarios: online aggregation in DWBI systems and finite data stream processing.

8. REFERENCES

[1] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: An optimized index structure for flash devices. *PVLDB*, 2(1):361–372, 2009.

[2] M. Canim, B. Bhattacharjee, G. A. Mihaila, C. A. Lang, and K. Ross. An object placement advisor for db2 using solid state storage. *PVLDB*, 2(2):1318–1329, 2009.

[3] S. Chen. Flashlogging: exploiting flash devices for synchronous logging performance. In *SIGMOD*, 2009.

[4] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *SIGMOD*, 1984.

[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*, 2002.

[6] A. Dobra, C. Jermaine, F. Rusu, and F. Xu. Turbo-charging estimate convergence in dbo. *PVLDB*, 2(1):419–430, 2009.

[7] Gartner, Inc. Market share: Business intelligence platform software, worldwide, 2007. http://www.gartner.com/it/page.jsp?id=700410, 2008.

[8] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[9] G. Graefe. The five-minute rule twenty years later. In *DaMoN Workshop*, 2007.

[10] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, 1999.

[11] J. M. Hellerstein, R. Avnur, and V. Raman. Informix under control: Online query processing. *Data Min. Knowl. Discov.*, 4(4), 2000.

[12] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.

[13] IDC. The diverse and exploding digital universe. http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf, 2008.

[14] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the dbo engine. *ACM Trans. Database Syst.*, 33(4), 2008.

[15] C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, and A. Pol. The sort-merge-shrink join. *ACM Trans. Database Syst.*, 31(4):1382–1416, 2006.

[16] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Comput.*, 1(1):63–74, 1983.

[17] I. Koltsidas and S. Viglas. Flashing up the storage layer. In *VLDB*, 2008.

[18] R. Lawrence. Early hash join: A configurable algorithm for the efficient and early production of join results. In *VLDB*, 2005.

[19] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory ssd in enterprise database applications. In *SIGMOD*, 2008.

[20] G. Luo, C. J. Ellmann, P. J. Haas, and J. F. Naughton. A scalable hash ripple join algorithm. In *SIGMOD*, 2002.

[21] 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*, 2004.

[22] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. In *VLDB*, 2008.

[23] S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *ACM/IEEE IPSN*, 2007.

[24] 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*, 2005.

[25] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD*, 2009.

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