Output Rate Optimization on Streaming inputs by Adaptive join Operators

K.SRIDHAR¹, M.CHALAPATHI RAO², MD.SIRAJJUDDIN³

¹CSE, JNTU, PPcolony, Peddapalli, Karimnagar, INDIA
²³CSE, Vaageswari College of Engineering, Ramakrishna colony, Karimnagary, INDIA
¹sridharkontham@gmail.com, ²parichayam03@yahoo.com, ³siraj569@hotmail.com

ABSTRACT
Adaptive join operators algorithms have attracted a lot of attention in emerging applications where data are provided by autonomous data sources through nonhomogenous network environments. Their main advantage over join techniques is that they can start producing join results as soon as the first input tuples are available, thus, improving pipelining by smoothing join result production and by masking source or network delays. In this paper, we first propose Double Index nested-loops Reactive join (DINER), a new adaptive two-way join algorithm for result rate maximization. DINER combines two key elements: an intuitive flushing policy that aims to increase the productivity of in-memory tuples in producing results during the online phase of the join, and a novel reentrant join technique that allows the algorithm to rapidly switch between processing in-memory and disk-resident tuples, thus, better exploiting temporary delays when new data are not available. We then extend the applicability of the proposed technique for a more challenging setup: handling more than two inputs. Multiple Index nested-loop Reactive join (MINER) is a multiway join operator that inherits its principles from DINER. Our experiments using real and synthetic data sets demonstrate that DINER outperforms previous adaptive join algorithms in producing result tuples at a significantly higher rate, while making better use of the available memory. Our experiments also shows that in the presence of multiple inputs, MINER manages to produce a high percentage of early results, outperforming existing techniques for adaptive multiway join.

Keywords: QueryDINER, MINER, streams.

1. INTRODUCTION
MODERN information processing is moving into a realm where we often need to process data that are pushed or pulled from autonomous data sources through heterogeneous networks. Adaptive query processing has emerged as an answer to the problems that arise because of the fluidity and unpredictability of data arrivals in such environments [1]. An important line of research in adaptive query processing has been toward developing join algorithms that can produce tuples “online,” from streaming, partially available input relations, or while waiting for one or more inputs [4], [7], [9], [12], [14], [15]. Such nonblocking join behavior can improve pipelining by smoothing or “masking” varying data arrival rates and can generate join results with high rates, thus, improving performance in a variety of query processing scenarios in data integration, online aggregation, and approximate query answering systems. Compared to traditional join algorithms (be they sort-, hash-, or nested-loop-based [13]), adaptive joins are designed to deal with some additional challenges: The input relations they use are provided by external network sources. The implication is that one has little or no control over the order or rate of arrival of tuples. Since the data source reply speed, streaming rate and streaming order, as well as network traffic and congestion levels, are unpredictable, traditional join algorithms are often unsuitable or inefficient. For example, most traditional join algorithms cannot produce results until at least one of the relations is completely available. Waiting for one relation to arrive completely to produce results is often unacceptable. Moreover, and more importantly, in emerging data integration or online aggregation environments, a key performance metric is rapid availability of first results and a continuous rate of tuple production.

The contributions of our paper are:

. We introduce DINER a novel adaptive join algorithm that supports both equality and range join predicates. DINER builds on an intuitive flushing policy that aims at maximizing the productivity of tuples that are kept in memory.
. DINER is the first algorithm to address the need to quickly respond to bursts of arriving data during the Reactive phase. We propose a novel extension to nested loops join for processing disk-resident tuples when both sources block, while being able to swiftly respond to new data arrivals.
. We introduce MINER, a novel adaptive multiway join algorithm that maximizes the output rate, designed for dealing with cases where data are held by multiple remote sources.
. We provide a thorough discussion of existing algorithms, including identifying some
important limitations, such as increased memory consumption because of their inability to quickly switch to the Arriving phase and not being responsive enough when value distributions change. We provide an extensive experimental study of Diner including performance comparisons a sensitivity analysis. Our results demonstrate the superiority of Diner in a variety of realistic scenarios. During the online phase of the algorithm, Diner manages to produce up to three times more results compared to previous techniques. The performance gains of Diner are realized when using both real and synthetic data and are increased when fewer resources (memory) are given to the algorithm. We also evaluate the performance of MINER, and we show that it is still possible to obtain early a large percentage of results even in more elaborated setups where data are provided through multiple inputs. Our experimental study shows that the performance of MINER is 60 times higher compared to the existing 

**2. RELATED WORK**

Existing work on adaptive join techniques can be classified in two groups: hash based [6], [7], [9], [14], [12], [15] and sort based [4]. Examples of hash-based algorithms include DPHJ [7] and XJoin [14], the first of a new generation of adaptive nonblocking join algorithms to be proposed. XJoin was inspired by Symmetric Hash Join (SHJ) [6], which represented the first step toward avoiding the blocking behavior of the traditional hash-based algorithms. SHJ required both relations to fit in memory; however, XJoin removes this restriction. MJoin [15] algorithm appeared as an enhancement of XJoin in which its applicability is extended to scenarios where more than two inputs are present. The above-mentioned algorithms were proposed for data integration and online aggregation. Pipelined hash join [16], developed concurrently with SHJ, is also an extension of hash join and was proposed for pipelined query plans in parallel main memory environment. Algorithms based on sorting were generally blocking since the original sort merge join algorithm required an initial sorting on both relations before the results could be obtained. Although there were some improvements that attenuate the blocking effect [10], the first efficient non-blocking sort-based algorithm was PMJ [4].

Hash Join Merge Join (HJM) [9], based on XJoin and PMJ, is a nonblocking algorithm which tries to combine the best parts of its predecessors while avoiding their shortcomings. Finally, Rate-based Progressive Join (RPJ) [12] is an improved version of HJM that is the first algorithm to make decisions, e.g., about flushing to disk, based on the characteristics of the data. In what follows, we describe the main existing techniques for adaptive join. For all hash-based algorithms, we assume that each relation $R_i, i = \{A; B\}$ is organized in $n_{part}$ buckets. The presentation is roughly chronological.

**XJoin.** As with “traditional” hash-based algorithms, XJoin organizes each input relation in an equal number of memory and disk partitions or buckets, based on a hash function applied on the join attribute.

The XJoin algorithm operates in three phases. During the first, arriving, phase, which runs for as long as either of the data sources sends tuples, the algorithm joins the tuples from the memory partitions. Each incoming tuple is stored in its corresponding bucket and is joined with the matching tuples from the opposite relation. When memory gets exhausted, the partition with the greatest number of tuples is flushed to disk. The tuples belonging to the bucket with the same designation in the opposite relation remain on disk. When both data sources are blocked, the first phase pauses and the second, reactive, phase begins. The last, cleanup, phase starts when all tuples from both data sources have completely arrived. It joins the matching tuples that were missed during the previous two phases.

In XJoin, the reactive stage can run multiple times for the same partition. The algorithm applies a time-stamp-based duplicate avoidance strategy in order to detect already joined tuple pairs during subsequent executions. MJoin. MJoin [15] is a hash-based algorithm which extends XJoin and it was designed for dealing with star queries of the form $R_1 \bowtie R_2 \bowtie R_3 \bowtie \ldots \bowtie R_n$. The algorithm creates as many in-memory hash tables as inputs. A new tuple is inserted in the hash table associated with its source and it is used to probe the remaining hash tables. The probing sequence is based on the heuristic that the most selective joins should be evaluated first in order to minimize the size of intermediate results. When the memory gets exhausted, the coordinated flushing policy is applied. A hashed bucket is picked (as in case of XJoin) and its content is evicted from each input. MJoin cannot be used directly for joins with multiple attributes or nonstar joins. Viglas et al. [15] describes a naive extension where
only a flushing policy that randomly evicts the memory content can be applied.

Progressive Merge Join. PMJ is the adaptive nonblock- ing version of the sort merge join algorithm. It splits the memory into two partitions. As tuples arrive, they are inserted in their memory partition. When the memory gets full, the partitions are sorted on the join attribute and are joined using any memory join algorithm. Thus, output tuples are obtained each time the memory gets exhausted. Next, the partition pair (i.e., the bucket pairs that were simultaneously flushed each time the memory was full) is copied on disk. After the data from both sources completely arrive, the merging phase begins. The algorithm defines a parameter F, the maximal fan-in, which represents the maximum number of disk partitions that can be merged in a single “turn.” F=2 groups of sorted partition pairs are merged in the same fashion as in sort merge. In order to avoid duplicates in the merging phase, a tuple joins with the matching tuples of the opposite relation only if they belong to a different partition pair.

Hash Merge Join. HMJ [9] is a hybrid query processing algorithm combining ideas from XJoin and Progressive Merge Join. HMJ has two phases, the hashing and the merging phase. The hashing phase performs in the same way as XJoin (and also DPHJ [7]) performs, except that when memory gets full, a flushing policy decides which pair of corresponding buckets from the two relations is flushed on disk. The flushing policy uses a heuristic that again does not take into account data characteristics: it aims to free as much space as possible while keeping the memory balanced between both relations. Keeping the memory balanced helps to obtain a greater number of results during the hashing phase. Every time HMJ flushes the current contents of a pair of buckets, they are sorted and create a disk bucket “segment”; this way the first step in a subsequent sort merge phase is already performed. When both data sources get blocked or after complete data arrival, the merging phase kicks in. It essentially applies a sort-merge algorithm where the sorted sublists (the “segments”) are already created. The sort-merge algorithm is applied on each disk bucket pair and it is identical to the merging phase of PMJ.

Rate-based Progressive Join. RPJ [12] is the most recent and advanced adaptive join algorithm. It is the first algorithm that tries to understand and exploit the connec- tion between the memory content and the algorithm output rate. During the online phase, it performs as HMJ. When memory is full, it tries to estimate which tuples have the smallest chance to participate in joins. Its flushing policy is based on the estimation of \( p_{\text{arr}}^{i,j} \), the probability of a new incoming tuple \( t \) to belong to relation \( R_j \) and to be part of bucket \( j \). Once all probabilities are computed, the flushing policy is applied. Let

\[
p_{\text{arr}}^{i,j} \text{ be the smallest probability. In this case, } n_{\text{flush}} \text{ tuples belonging to bucket } j \text{ of the opposite relation are spilled. If the victim bucket does not contain enough tuples, the next smallest probability is chosen, etc. All the tuples that are flushed together from the same relation and from the same bucket are sorted and they form a sorted “segment” as in HMJ.}

In case both relations are temporarily blocked, RPJ begins its reactive phase, which “combines” the XJoin and HMJ reactive phases. The tuples from one of the disk buckets of either relation can join with the corresponding memory bucket of the opposite relation, as in case of XJoin, or two pairs of disk buckets can be brought in memory and joined as in case of HMJ (and PMJ). The algorithm chooses the task that has the highest output rate. During its cleanup phase, RPJ joins the disk buckets. The duplicate avoidance strategy is similar with the one applied by XJoin.

The following observation applies to the reactive phase algorithm run by both HMJ and RPJ. When a “disk to disk” process takes place, the algorithm checks for new incoming tuples after a set of F segments belonging to each bucket pair are merged, where F is the fan-in parameter of PMJ. The drawback is that after consecutive runs of this phase over the same pair of disk buckets, the progressively merged segments have larger and larger sizes and their merge takes longer; meanwhile, incoming tuples are ignored. So, the algorithm stays in its reactive phase longer, possibly generating additional join results, but at the price of requiring a much larger input buffer to store incoming tuples, which otherwise might have to be dropped, compromising the algorithm’s correctness.

3. DINNER DESCRIPTION

We now present our Diner algorithm for computing the join result of two finite relations \( R_A \) and \( R_B \), which may be stored at potentially different sites and are streamed to our local system. Given the unpredictable behavior of the network, delays and random temporary suspensions in data
transmission may be experienced. The goal of DINER is twofold. It first seeks to correctly produce the join result, by quickly processing arriving tuples, while avoiding operations that may jeopardize the correctness of the output because of memory overflow. Moreover, in the spirit of prior work [5], [9], [12] DINER seeks to increase the number of join tuples (or, equivalently, the rate of produced results) generated during the online phase of the join, i.e., during the (potentially long) time it takes for the input relations to be streamed to our system. To achieve these goals, DINER is highly adaptive to the (often changing) value distributions of the relations, as well as to potential network delays. Table 1 summarizes the main notation used in the presentation of the DINER algorithm. Additional definitions are presented in appropriate areas of the text.

3.1 Algorithm Overview
We now present our DINER algorithm for computing the join result of two finite relations $R_A$ and $R_B$, which may be stored at potentially different sites and are streamed to our local system. Given the unpredictable behavior of the network, delays and random temporary suspensions in data transmission may be experienced. The goal of DINER is twofold. It first seeks to correctly produce the join result, by quickly processing arriving tuples, while avoiding operations that may jeopardize the correctness of the output because of memory overflow. Moreover, in the spirit of prior work [5], [9], [12] DINER seeks to increase the number of join tuples (or, equivalently, the rate of produced results) generated during the online phase of the join, i.e., during the (potentially long) time it takes for the input relations to be streamed to our system. To achieve these goals, DINER is highly adaptive to the (often changing) value distributions of the relations, as well as to potential network delays. Table 1 summarizes the main notation used in the presentation of the DINER algorithm. Additional definitions are presented in appropriate areas of the text.

<table>
<thead>
<tr>
<th>Table 1 Symbols Used in Our Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Symbol</strong></td>
</tr>
<tr>
<td>$R_i$</td>
</tr>
<tr>
<td>$t_i$</td>
</tr>
<tr>
<td>$Index_i$</td>
</tr>
<tr>
<td>$Disk_i$</td>
</tr>
<tr>
<td>$UsedMemory$</td>
</tr>
<tr>
<td>$MemThresh$</td>
</tr>
<tr>
<td>$WaitThresh$</td>
</tr>
<tr>
<td>$LastLvVal_i$</td>
</tr>
<tr>
<td>$LwJoins_i, MdJoins_i, UpJoins_i$</td>
</tr>
<tr>
<td>$LwTups_i, MdTups_i, UpTups_i$</td>
</tr>
<tr>
<td>$BfLw_i, BfMd_i, BfUp_i$</td>
</tr>
</tbody>
</table>

A main memory B+-tree is a good option. In our implementation, we utilize the Judy [2] structure. Judy is a sparse dynamic array with built in sort, search, and sequential access capabilities. It can replace memory data structures such as arrays, hash-tables, and B+-trees, and an efficient implementation by Hewlett-Packard is widely available. The total amount of memory, denoted as $UsedMemory$, used by our algorithm is upper bounded by the value $MemThresh$. The value of $UsedMemory$ includes in its computation the memory occupied by the input buffer (stage where incoming tuples arrive), the in-memory tuples being processed by the algorithm, the Judy indexes, and any additional data structures utilized by DINER for maintaining statistics. Finally, each relation is associated with a disk partition, $Disk_A$ and $Disk_B$, which stores the tuples from the relation that do not fit in memory and have been flushed to disk. The memory and disk blocks used by the DINER algorithm are selected to be multiples of the operating system block size so that the disk
I/O operations can be performed efficiently. Format of stored tuples. Each tuple inserted in memory is augmented with an arriving time stamp (ATS). Each tuple flushed to disk is further augmented with a departure time stamp (DTS). As explained later in this section, these time stamps are used in order to ensure that during the Reactive and Cleanup phases every pair of tuples between the two relations will be joined exactly once, thus, ensuring the correctness of the produced result.

Each tuple residing in main memory is also augmented with a join bit, which is actually part of the index entry for the tuple. This bit is initially set to 0. Whenever an in-memory tuple helps produce a join result, its join bit is set to 1. As described in Section 3.3, the join bits are utilized by a process responsible for evicting tuples that do not produce joins from some areas of the memory. The latter process seeks to evict those tuples that have their join bit set to 0, while clearing the join bit of the tuples that it examines. Those tuples that have produced joins in the past will eventually have their join bit cleared and get evicted at a subsequent invocation of the process, if at some point they stop producing join results. Thus, the join bit serves as a 1-bit approximation to LRU, similarly to the clock algorithm for buffer page replacement [11]. Maintaining a counter (instead of a single join bit) for the number of joins that each tuple has produced would have been more accurate. However, such an approach would also consume more space and an experimental evaluation of both options did not show any significant benefit.

Phases of the algorithm. The operation of DINER is divided into three phases, termed in this paper as the Arriving, Reactive, and Cleanup phases. While each phase is discussed in more detail in what follows, we note here that the Arriving phase covers operations of the algorithm while tuples arrive from one or both sources, the Reactive phase is triggered when both relations block and, finally, the Cleanup phase finalizes the join operation when all data has arrived to our site.

3.2 Arriving Phase

Tuples arriving from each relation are initially stored in memory and processed as described in Algorithm 1. The Arriving phase of DINER runs as long as there are incoming tuples from at least one relation. When a new tuple \( t_i \) is available, all matching tuples of the opposite relation that reside in main memory are located and used to generate result tuples as soon as the input data are available. When matching tuples are located, the join bits of those tuples are set, along with the join bit of the currently processed tuple (Line 9). Then, some statistics need to be updated (Line 11). This procedure will be described later in this section.

Algorithm 1. Arriving Phase

1: while \( R_A \) and \( R_B \) still have tuples do
2: if \( t_i \) 2 \( R_j \) arrived (i 2 \( F_A \); Bg) then
3: Move \( t_i \) from input buffer to DINER process space.
4: Augment \( t_i \) with join bit and arrival timestamp ATS
5: \( j \approx \frac{1}{2} F_A ; B_g \)  [Refers to “opposite” relation]
6: Index\( j \) from opposite relation \( R_j \)
7: joinNNum \( \approx \) matchSet\( \frac{1}{2} \) set of matching tuples (found using
Index\( j \)) from opposite relation \( R_j \)
8: if joinNNum \( > 0 \) then
9: Set the join bits of \( t_i \) and of all tuples in matchSet
10: end if
11: UpdateStatistics\( (t_i, \text{numJoins}) \)
12: indexOverhead \( \approx \) Space required for indexing
\( t_i \) using Index\( j \)
13: while UsedMemory\( \approx \)indexOverhead MemThresh
\( \approx \) MemThreshold
14: Apply flushing policy
15: end while
16: Index \( t_i \) using Index\( j \)
17: Update UsedMemory
18: else if transmission of \( R_A \) and \( R_B \) is blocked more than WaitThresh then
19: Run Reactive Phase
20: end if
21: end while
22: Run Cleanup Phase
When the MemThresh is exhausted, the flushing policy picks a victim relation and memory-resident tuples from that relation are moved to disk in order to free memory space (Lines 13-15). The number of flushed tuples is chosen so as to fill a disk block. The flushing policy may also be invoked when new tuples arrive and need to be stored in the input buffer. Since this part of memory is included in the budget (MemThresh) given to the DINER algorithm, we may have to flush other in-memory tuples to open up some space for the new arrivals. This task is executed asynchronously by a server process that also takes care of the communication with the remote sources. Due to space limitations, it is omitted from presentation. If both relations block for more than WaitThresh msecs (Lines 18-20) and, thus, no join results can be produced, then the algorithm switches over to the Reactive phase, discussed in Section 3.4. Eventually, when both relations have been received in their entirety (Line 22), the Cleanup phase of the algorithm, discussed in Section 3.5, helps produce the remaining results.

3.3 Flushing Policy and Statistics Maintenance

An overview of the algorithm implementing the flushing policy of DINER is given in Algorithm 2. In what follows, we describe the main points of the flushing process.

Algorithm 2. Flushing Policy
1: Pick as victim the relation \( R_i \) (i.e., \( A; B_g \)) with the most in-memory tuples
2: \{Compute benefit of each region\}
3: \( Bf U_{p_i} \quad U_{pJoins_i} = U_{pTups_i} \)
4: \( Bf L_{w_i} \quad L_{wJoins_i} = L_{wTups_i} \)
5: \( Bf M_{d_i} \quad M_{dJoins_i} = M_{dTups_i} \)
6: \{Tups Per Block denotes the number of tuples required - to fill a disk block\}
7: \{Each flushed tuple is augmented with the departure time stamp DTS\}
8: if \( Bf U_{p_i} \) is the minimum benefit then
9: locate Tups Per Block tuples with the larger join attribute using \( Index_{x_i} \)
10: flush the block on Disk_{i}
11: update LastU_{pV_{al_i}} so that the upper region is
   \( \text{(about) a disk block} \)
12: else if \( Bf L_{w_i} \) is the minimum benefit then
13: locate Tups Per Block tuples with the smaller join attribute using \( Index_{x_i} \)
14: flush the block on Disk_{i}
15: update LastL_{wV_{al_i}} so that the lower region is
   \( \text{(about) a disk block} \)
16: else
17: Using the Clock algorithm, visit the tuples from the middle area, using \( Index_{x_i} \), until Tups Per Block tuples are evicted.
18: end if
19: Update U_{pTups_i}, L_{wTups_i}, M_{dTups_i}, when necessary
20: U_{pJoins_i}, L_{wJoins_i}, M_{dJoins_i} = 0

Selection of victim relation. As in prior work, we try to keep the memory balanced between the two relations. When the memory becomes full, the relation with the highest number of in-memory tuples is selected as the victim relation (Line 1).

Intuition. The algorithm should keep in main memory those tuples that are most likely to produce results by joining with subsequently arriving tuples from the other relation. Unlike prior work that tries to model the distribution of values of each relation [12], our premise is that real data are often too complicated and cannot easily be captured by a simple distribution, which we may need to predetermine and then adjust its parameters. We have, thus, decided to devise a technique that aims to maximize the number of joins produced by the in-memory tuples of both relations by adjusting the tuples that we flush from relation \( R_A \) (\( R_B \)), based on the range of values of the join attribute that were recently obtained by relation \( R_B \) (\( R_A \)), and the number of joins that these recent tuples produced.

For example, if the values of the join attribute in the incoming tuples from relation \( R_A \) tend to
be increasing and these new tuples generate a lot of joins with relation $R_B$, then this is an indication that we should try to flush the tuples from relation $R_B$ that have the smallest values on the join attribute (of course, if their values are also smaller than those of the tuples of $R_A$) and vice versa. The net effect of this intuitive policy is that memory will be predominately occupied by tuples from both relations whose range of values on the join attribute overlap. This will help increase the number of joins produced in the online phase of the algorithm.

Simply dividing the values into two regions and flushing tuples from the two endpoints does not always suffice, as it could allow tuples with values of the join attribute close to its median to remain in main memory for a long time, without taking into account their contribution in producing join results. (Recall that we always flush the most extreme value of the victim region.) Due to new incoming tuples, evicting always from the same region (lower or upper) does not guarantee inactive tuples with join values close to the median will ever be evicted. Maintaining a larger number of areas for each relation implies an overhead for extra statistics, which would also be less meaningful due to fewer data points per region.

In a preliminary version of this paper [3], we discuss in more detail and present experimental evaluation of the importance and benefit of the three regions. Conceptual tuple regions. The DINER algorithm accounts (and maintains statistics) for each relation for three conceptual regions of the in-memory tuples: the lower, the middle, and the upper region. These regions are determined based on the value of the join attribute for each tuple.

The three regions are separated by two conceptual boundaries: LastLwV $a_i$ and LastUpV $a_i$ for each relation $R_i$ that are dynamically adjusted during the operation of the DINER algorithm. In particular, all the tuples of $R_i$ in the lower (upper) region have values of the join attribute smaller (larger) or equal to the LastLwV $a_i$ (LastUpV $a_i$) threshold. All the remaining tuples are considered to belong in the middle in-memory region of their relation.

Maintained statistics. The DINER algorithm maintains simple statistics in the form of six counters (two counters for each conceptual region) for each relation. These statistics are updated during the Arriving phase as described in Algorithm 3. We denote by $UpJoins_i$, $LwJoins_i$, and $MdJoins_i$ the number of join results that tuples in the upper, lower, and middle region, respectively, of relation $R_i$ have helped produce. These statistics are reset every time we flush tuples of relation $R_i$ to disk (Algorithm 2, Line 20). Moreover, we denote by $UpTups_i$, $LwTups_i$, and $MdTups_i$ the number of in-memory tuples of $R_i$ that belong to the conceptual upper, lower, and middle region, respectively, of relation $R_i$. These numbers are updated when the boundaries between the conceptual regions change (Algorithm 2, Line 19).

Algorithm 3. UpdateStatistics

Require: $t_i$ ($i \leq FA;Bg$), $numJoins$

1: $j \not\in FA;Bg$  
2: $JoinAttr \not\in JoinAttr$
3: $\{Update statistics on corresponding region of relation $R_i$\}$
4: if $JoinAttr \not\in JoinAttr$
5: $UpJoins_i \not\in UpJoins_i \not\in numJoins$
6: $UpTups_i \not\in UpTups_i \not\in 1$
7: else if $JoinAttr \not\in JoinAttr$
8: $LwJoins_i \not\in LwJoins_i \not\in numJoins$
9: $LwTups_i \not\in LwTups_i \not\in 1$
10: else
11: $MdJoins_i \not\in MdJoins_i \not\in numJoins$
12: $MdTups_i \not\in MdTups_i \not\in 1$
13: end if
14: $\{Update statistics of opposite relation\}$
15: if $JoinAttr \not\in JoinAttr$
16: $UpJoins_j \not\in UpJoins_j \not\in numJoins$
17: else if $JoinAttr \not\in JoinAttr$
18: $LwJoins_j \not\in LwJoins_j \not\in numJoins$
19: else
20: MdJoin sj MdJoin sj ≧ numJoins
21: end if

Where to flush from. Once a victim relation $R_i$ has been chosen, the victim region is determined based on a benefit computation. We define the benefit $BF_{Lw_i}$ of the lower region of $R_i$ to be equal to: $BF_{Lw_i} = \frac{1}{4} \sum_{i} Lw_i$. The corresponding benefit $BF_{Md_i}$ and $BF_{Md_i}$ for the upper and middle regions of $R_i$ are defined in a completely analogous manner (Algorithm 2, Lines 2-5). Given these (per space) benefits, the Diner algorithm decides to flush tuples from the region exhibiting the smallest benefit. How to flush from each region, when the Diner algorithm determines that it should flush tuples from the lower (upper) region, it starts by first flushing the tuples in that region with the lowest (highest) values of the join attribute and continues toward higher (lower) values until a disk block has been filled (Algorithm 2, Lines 7-15). This process is expedited using the index. After the disk flush, the index is used to quickly identify the minimum LastLwV $a_i$ (maximum LastU $pV a_i$) values such that the lower (upper) region contains enough tuples to fill a disk block. The new LastLwV $a_i$ and LastU $pV a_i$ values will identify the boundaries of the three regions until the next flush operation.

When flushing from the middle region (Algorithm 2, Lines 16-18), we utilize a technique analogous to the Clock [11] page replacement algorithm. At the first invocation, the hand of the clock is set to a random tuple of the middle region. At subsequent invocations, the hand recalls its last position and continues from there in a round-robin fashion. The hand continuously visits tuples of the middle region and flushes those tuples that have their join bit set to 0, while resetting the join bit of the other visited tuples. The hand stops as soon as it has flushed enough tuples to fill a disk block. A special case occurs when the algorithm flushes data from each relation for the very first time. In this case, the LastLwV $a_i$ and LastU $pV a_i$ values have not been pre-viously set. Thus, at the first time our algorithm 1) considers all the tuples of each relation to belong to its middle region and flushes from that area; and 2) using the index, which can provide sorted access to the data, quickly identifies LastLwV $a_i$ and LastU $pV a_i$ values such that each of the lower and upper regions contain enough tuples to fill a disk block.

The expelled tuples from either relation are organized in sorted blocks by accessing the memory using the victim’s index. Thus, tuples that are flushed to disk in the same operation are sorted based on the join attribute. This is done in order to speed up the execution of the Reactive and Cleanup phases of the algorithm, discussed later in this section.

Implementation details. In situations where the data are extremely skewed, the middle region may get entirely squeezed. (As explained earlier in this section, the end-points always contain at least enough tuples to fill a block.) In such rare cases, we make sure that the LastLwV $a_i$ and LastU $pV a_i$ values never create overlapping intervals for the lower and upper regions (i.e., the invariant LastLwV $a_i$ < LastU $pV a_i$ always holds). In such extremely skewed data (i.e., all the data having the same key value), one region may end up containing almost all of the tuples, thus, raising the possibility that a flushed region may actually contain fewer tuples than the ones required to fill a disk block. In such situations, Diner continues applying its flushing policy to the region with the second (and possible even the third) highest (per space) benefit until enough tuples have been flushed to disk.

3.4 Reactive Phase
The Reactive phase join algorithm, termed ReactiveNL, is a nested-loop-based algorithm that runs whenever both relations are blocked. It performs joins between previously flushed data from both relations that are kept in the disk partitions Disk$_A$ and Disk$_B$, respectively. This allows Diner to make progress while no input is being delivered. The algorithm switches back to the Arriving phase as soon as enough, but not too many, input tuples have arrived, as is determined by the value of input parameter maxNewArr. The goal of ReactiveNL is to perform as many joins between flushed-to-disk blocks of the two relations as possible, while simplifying the bookkeeping that is necessary when exiting and reentering the Reactive phase.

Algorithm ReactiveNL is presented in Algorithm 4. We assume that each block of tuples flushed on disk is assigned an increasing block-id, for the corresponding relation (i.e., the first block of relation $R_A$ corresponds to block 1, the second to block 2, etc.). The notation used in the algorithm is available in Table 2. Figs. 1 and 2 provide a helpful visualization of the progress of the algorithm. Its operation is based on the following points:
Algorithm 4. ReactiveNL
Require: MaxOuterMem, MaxNewArr
1: if First Algorithm Invocation then
2: Set Outer      smallest(Disk_A, Disk_B)
3: Set Inner    largest(Disk_A, Disk_B)
4: Set JoinedInner to size of Inner Relation
5: Set JoinedOuter to 0 and CurrInner to 1
6: end
7: while unprocessed blocks exist do
8: while JoinedOuter < OuterSize do
9: [Load next chunk of outer blocks. CurrInner reveals whether some blocks from the outer
relation had started joining with the inner relation, but did not complete this task.]
10: if CurrInner > 1 then
11:      OuterMem  \frac{1}{3} \min(\text{MaxOuterMem},
                   \text{OuterSize-JoinedOuter})
12:      end
13:      if{Else keep its previous value}
14:      end if
15:      Load OuterMem blocks from outer relation starting with block-id JoinedOuter+1. If not
sufficient space
in memory, apply flushing policy to clear up enough space.
16:      while CurrInner     JoinedInner do
17:         Load CurrInner block of inner relation in memory and join with in-memory outer blocks based
on join attribute and non-overlapping ATS..DTS timestamps
18:         CurrInner    CurrInner+1
19:      end
20:      if Input buffer size greater than MaxNewArr then
21:         Break;
22:      end
23:    end while
24:    if CurrInner > JoinedInner then
25:       {Mark completed join of [1..JoinedOuter] and
[1..JoinedInner] blocks}
26:       JoinedOuter  JoinedOuter+\text{OuterMem}
27:       CurrInner    1
28:    end if
29:    if Input buffer size greater than MaxNewArr then
30:       {Switch back to Arriving Phase}
31:       exit
32:    end if
33:    end while
34: end while

Point 1. ReactiveNL initially selects one relation to behave as the outer relation of the nested loop
algorithm, while the other relation initially behaves as the inner relation (Lines 1-3). Notice that the “inner relation” (and the “outer”) for the purposes of ReactiveNL consists of the
blocks of the corresponding relation that currently reside on disk, because they were flushed during
the Arriving phase.

Point 2. ReactiveNL tries to join successive batches of OuterMem blocks of the outer relation with
all of the inner relation, until the outer relation is exhausted (Lines 13-20). The value of OuterMem
is determined based on the maximum number of blocks the algorithm can use (input parameter
MaxOuterMem) and the size of the outer relation. However, as DINNER enters and exits the
Reactive phase, the size of the inner relation may change, as more blocks of that relation may be
flushed to disk. To make it easier to keep track of joined blocks, we need to join each batch of
OuterMem blocks of the outer relation with the same, fixed number of blocks of the inner relation—
even if over time, the total number of disk blocks of the inner relation increases. One of the key ideas of ReactiveNL is the following: at the first invocation of the algorithm, we record the number of
blocks of the inner relation in JoinedInner (Line 4). From then on, all successive batches of
OuterMem blocks of the outer relation will only join with the first JoinedInner blocks of the inner relation, until all the available outer blocks are exhausted. Point 3. When the outer relation is exhausted, there may be more than JoinedInner blocks of the inner relation on disk (those that arrived after the first round of the nested loop join, when DINER goes back to the Arriving phase). If that is the case, then these new blocks of the inner relation need to join with all the blocks of the outer relation. To achieve this with the minimum amount of bookkeeping, it is easier to simply switch roles between relations, so that the inner relation (that currently has new, unprocessed disk blocks on disk) becomes the outer relation and vice versa (all the counters change roles also, hence JoinedInner takes the value of JoinedOuter, etc., while CurrInner is set to point to the first block of the new inner relation). Thus, an invariant of the algorithm is that the tuples in the first JoinedOuter blocks of the outer relation have joined with all the tuples in the first JoinedInner blocks of the inner relation.

Point 4. To ensure prompt response to incoming tuples, and to avoid overflowing the input buffer, after each block of the inner relation is joined with the in-memory OutMem blocks of the outer relation, ReactiveNL examines the input buffer and returns to the Arriving phase if more than MaxNewArr tuples have arrived. (We do not want to switch between operations for a single tuple, as this is costly.) The input buffer size is compared

26-28, and if the algorithm exits, the variables JoinedOuter, JoinedInner, and CurrInner keep the state of the algorithm for its next reentry. At the next invocation of the algorithm, the join continues by loading (Lines 9-13) the outer blocks with ids in the range [JoinedOuter+1, JoinedOuter+Outer-Mem] and by joining them with inner block CurrInner. Point 5. As was discussed earlier, the flushing policy of DINER spills on disk full blocks with their tuples sorted on the join attribute. The ReactiveNL algorithm takes advantage of this data property and speeds up processing by performing an in-memory sort merge join between the blocks. During this process, it is important that we do not generate duplicate joins between tuples touter and tinner that have already joined during the Arriving phase. This is achieved by proper use of the ATS and DTS time stamps. If the time intervals [touter:ATS, touter:DTSD and tinner:ATS, tinner:DTSD overlap, this means that the two tuples coexisted in memory during the Arriving phase and their join is already obtained. Thus, such blocks of tuples are ignored by the ReactiveNL algorithm.

Discussion. The Reactive phase is triggered when both data sources are blocked. Since network delays are unpredictable, it is important that the join algorithm is able to quickly switch back to the Arriving phase, once data start flowing in again. Otherwise, the algorithms risk overflowing the input buffer for stream arrival rates that they would support if they hadn’t entered the reactive phase. Previous adaptive algorithms [9], [12], which also include such a Reactive phase, have some conceptual limitations, dictated by a minimum amount of work that needs to be performed during the Reactive phase, that prevent them from promptly reacting to new arrivals. For example, as discussed in Section 2, during its Reactive phase, the RJ algorithm works on progressively larger partitions of data. Thus, a sudden burst of new tuples while the algorithm is on its Reactive phase quickly leads to large increases in input buffer size and buffer overflow, as shown in the experiments presented in Section 5. One can potentially modify the RJ algorithm so that it aborts the work performed during the Reactive phase or keeps enough state information so that it can later resume its operations in case the input buffer gets full, but both solutions have not been explored in the literature and further complicate the implementation of the algorithm. In comparison, keeping the state of the ReactiveNL algorithm only requires three variables, due to our novel adaptation of the traditional nested loops algorithm.

3.5 Cleanup Phase

The Cleanup phase starts once both relations have been received in their entirety. It continues the work performed during the Reactive phase by calling the ReactiveNL.

4 MINER DESCRIPTION

We now present an extension of the DINER algorithm, termed as MINER, for joining multiple relations. For ease of presentation, we conduct our discussion for joins of the form

R1 \( \bullet R_2 : a_1 \quad \bullet R_2 : a_2 \quad \bullet R_3 : a_2 \quad R_3 \quad \bullet R_n \quad 1 \quad a_n \quad 1 \quad R_n : a_n \quad 1 \quad R_n : a_1 \quad 1 \quad R_n : a_1 \quad 1 \quad R_n : a_1 \). 

We then present the required extensions for other categories of joins. Due to its inherent similarity with DINER, we focus our discussion to the adjustments, compared to DINER, required for the proper operation of MINER.

4.1 Extending the Online Phase

Processing the incoming tuples. Due to the existence of multiple join attributes per relation, MINER maintains for each joined relation a separate index on each join attribute. Fig. 3 depicts the
logical organization of in-memory tuples for a sample join of four joined relations $R_1, R_2, R_3,$ and $R_4$. When a new tuple arrives, it is stored and indexed in the memory space of its relation, based on the relation’s join attributes. For instance, pointers to a new tuple in stream $R_2$ are inserted in Index$R_2 : a_1$ and Index$R_2 : a_2$. This new tuple then needs to be joined with all the matching in-memory tuples belonging to all other relations participating in the join. An important factor that needs to be considered when executing a query over multiple inputs is the probing sequence of the relations. We address this problem by first estimating the join selectivity factor $R_1;R_1$ between each pair of relations $R_1; R_2$ and then forming the probing sequence by con-sidering pairs of increasing join selectivity factor. For example, assume that in Fig. 3 $R_2;R_3 R_1;R_2 R_3;R_4$. Then, an incoming tuple belonging to $R_2$ will first be joined with $R_3$, then with $R_1$ and, finally, with $R_4$. In order to estimate the join selectivity factors, we utilize simple statistics on the number of produced joins and on the relation cardinality. These statistics will be presented later in this section. Statistics maintenance. MINER maintains a separate index, and statistics regarding each such index, for every join attribute of a relation. Similarly to DINER, MINER maintains for each index three conceptual areas: Up, Md, and Lw. The three conceptual areas are separated by two conceptual boundaries: LastUpVal and LastLwVal. The statistics that are kept for each area are the number of

4.2 Extending the Reactive Phase

The Reactive phase is activated when all data sources experience delays and is illustrated in Figs. 4 and 5. General processing. During the reactive phase, MINER extends the nested loop join algorithm applied by DINER in the following manner: Only one of the joined relations is treated as the outer relation, while the remaining relations are treated as inner relations. Moreover, for each algorithm will load OuterSize blocks from the outer relation and start joining them with the blocks of the inner relations. From each inner relation, just one block will be loaded. For each inner relation $R_i$ we will not alter its CurrInner$R_i$ block until this block exhausts joining with all combinations of blocks (i.e., blocks up to JoinedInner of each relation) of the relations that lie to the right of $R_i$ in the current join sequence (i.e., those relations are considered as “more inner” in the join sequence). When it is time to advance the CurrInner$R_i$ block, the corresponding CurrInner counter of the relations that are “more inner” than $R_i$ are set to 1. Similarly, the next batch of OuterSize blocks from the outer relation will be loaded only when it has completely joined with all blocks (up to the CurrInner value) of the inner relations.

Given the above description of the Reactive phase of MINER at each point in time, a similar invariant holds, when considering the blocks that have completely joined: all the blocks until JoinedOuter have completely joined with the first JoinedInner of each inner relation $R_i$. Moreover, during the Reactive phase, the join is always performed between the OuterSize blocks of the outer relation, with the CurrInner block of each inner relation $R_i$.

The proposed extension still keeps the desirable prop-erty of the original DINER technique: the ability to efficiently hand over to the reactive phase when new input is available. The existence of new input is checked after each block of the “innermost” relation is joined. Moreover, the handover process has very small overload, as MINER only has to maintain the value of JoinedOuter for the outer relation, and the values of JoinedInner and CurrInner for each inner relation.

Changing the outer-inner roles. When the outer relation $R$ is exhausted, the outer-inner roles are changed. At this point, the outer relation becomes an inner relation in a join sequence, while any of the inner relations with unprocessed blocks can assume the role of the outer relation. For the types of multiway joins considered, namely

![Fig. 5. Changing outer-inner roles.](image-url)
currently joined outer blocks belong to \( R_2 \) in the figure), and one join sequence contains the join of relations \( R_2 \), \( R_3 \), and \( R_4 \), while the second join sequence contains the join of relations \( R_1 \) and \( R_2 \). Initially, the next batch of outer memory blocks is loaded in memory together with the first block of each inner relations, regardless of the join sequence. After the join between the loaded blocks is performed, the algorithm needs to pick the next inner block to be loaded from disk. In order to advance through blocks, a higher priority is given to one of the two join sequences, which simply means that blocks are fetched more frequently for the relations in that sequence. Inside the priority sequence, blocks belonging to the inner joined relations are fetched in the same order as described in the beginning of Section 4.2. When all the blocks of one join sequence are exhausted, the other join sequence proceeds by loading the next block. When the second join sequence also gets exhausted, then it is time for the next batch of blocks from the outer relation to be loaded in memory. In the example from Fig. 5, after relations \( R_3 \) and \( R_4 \) have been completely iterated for the current outer batch, a new block of relation \( R_1 \) is brought from disk and the iterations over \( R_3 \) and \( R_4 \) are performed again.

5 EXPERIMENTS

In this section, we present an extensive experimental study of our proposed algorithms. The objective of this study is twofold. We first evaluate the performance of DINER against the state of the art HJM [9] and RPJ [12] binary algorithms for a variety of both real-life and synthetic data sets. DPHJ and XJoin are both dominated in performance by RPJ and/or HJM, as showed in [9] and [12]. For brevity and clarity in the figures, we have decided to include their performance only in few of the figures presented in this section and omit them for the rest. We also investigate the impact that several parameters may have on the performance of the DINER algorithm, through a detailed sensitivity analysis. Moreover, we evaluate the performance of MINER when we vary the amount of memory allocated to the algorithm and the number of inputs. The main findings of our study include:

. A Faster Algorithm. DINER provides result tuples at a significantly higher rate, up to three times in some cases, than existing adaptive join algorithms during the online phase. This also leads to a faster computation of the overall join result when there are bursty tuple arrivals.

. A Leaner Algorithm. The DINER algorithm further improves its relative performance to the compared algorithms in terms of produced tuples during the online phase in more constrained memory environments. This is mainly attributed to our novel flushing policy.

. A More Adaptive Algorithm. The DINER algorithm has an even larger performance advantage over existing algorithms, when the values of the join attribute are streamed according to a nonstationary process. Moreover, it better adjusts its execution when there are unpredictable delays in tuple arrivals, to produce more result tuples during such delays.

. Suitable for Range Queries. The DINER algorithm can also be applied to joins involving range conditions for the join attribute. PMJ [4] also supports range queries but, as shown in Section 5, it is a generally poor choice since its performance is limited by its blocking behavior.

. An Efficient Multiway Join Operator. MINER retains the advantages of DINER when multiple inputs are considered. MINER provides tuples at a significantly higher rate compared to MJoin during the online phase. In the presence of four relations, which represents a challenging setup, the percentage of results obtained by MINER during the arriving phase varies from 55 percent (when the allocated memory is 5 percent of the total input size) to more than 80 percent (when the allocated memory size is equal to 20 percent of the total input size).

Parameter settings. The following settings are used during Section 5: The tuple size is set to 200 bytes and the disk block size is set to 10 KB. The interarrival delay (i.e., the delay between two consecutive incoming tuples), unless specified otherwise, is modeled using an exponential
distribution with parameter \(\lambda = 0.1\). The memory size allocated for all algorithms is 5 percent of the total input size, unless otherwise specified. All incoming unprocessed tuples are stored in the input buffer, whose size is accounted for in the total memory made available to the algorithms.

Real data sets. In our experimental evaluation, we utilized two real data sets. The Stock data set contains traces of stock sales and purchases over a period of one day. From this data set, we extracted the transactions relating to the IPIX and CSCO stocks.\(^3\) For each stock, we generated one relation based on the part of the stock traces involving buy orders, and a second relation based on the part of the stock traces that involve sells. The size of CSCO stream is 20,710 tuples while the size of IPIX stream is 45,568 and the tuples are equally split among the relations. The join attribute used was the price of the transaction. We also used a Weather data set containing meteorological measurements from two different weather stations.\(^4\) We populated two relations, each from the data provided by one station, and joined them on the value of the air temperature attribute. Each relation in the Weather data set contains 45,000 tuples.

Since the total completing time of the join for all algorithms is comparable, in most of the cases the experiments show the outcome for the online phase (i.e., until tuples are completely received from the data sources).

All the experiments were performed on a machine running Linux with an Intel processor clocked at 1.6 GHz and with 1 GB of memory. All algorithms were implemented in C \(\langle\rangle\). For RPJ, also implemented in C \(\langle\rangle\), we used the code that Tao et al. [12] have made available. The disk I/O operations for all algorithms are performed using system calls that disable the operating system buffering.

### 5.1 Overall Performance Comparison

In the first set of experiments, we demonstrate DINER’s superior performance over a variety of real and synthetic data sets in an environment without network congestion or unexpected source delays. In Figs. 7 and 8, we plot the cumulative number of tuples produced by the join algorithms over time, during the online phase for the CSCO stock and the Weather data sets. We observe that DINER has a much higher rate of tuples produced that all other competitors. For the stock data, while RPJ is not able to produce a lot of tuples initially, it manages to catch up with XJoin at the end. In Fig. 9, we compare DINER to RPJ and HMJ on the real data sets when we vary the amount of available memory as a percentage of the total input size. The y axis represents the tuples produced by RPJ and HMJ at the end of their online phase (i.e., until the two relations have arrived in full) as a percentage of the number of tuples produced by DINER over the same time. The DINER algorithm significantly outperforms RPJ and HMJ, producing up to 2.5 times more results than the competitive techniques.

### 5.2 Nonstationary Data Streaming Processes

The DINER algorithm adapts quickly to situations where the value distribution of the join attribute changes over.

### 5.3 Network/Source Delays

The settings explored so far involve environments where tuples arrive regularly, without unexpected delays. In this section, we demonstrate that DINER is more adaptive to unstable environments affecting the arriving behavior of the streamed relations. DINER uses the delays in data arrival, by joining tuples that were skipped due to flushes on disk, while being able to hand over to the Arriving phase quickly, when incoming tuples are accumulated in the input buffer.

In Fig. 16, the streamed relations’ join attributes follow a zipf distribution with a skew parameter of 1. The interarrival time is modeled in the following way: Each incoming relation contains 100,000 tuples and is divided in 10 “fragments.” Each fragment is streamed using the usual exponential distribution of intertuple delays with \(\lambda = 0.1\). After each fragment is transmitted, a delay equal to x percent of its transmission time is introduced (x is the parameter varied in the x axis of the figure). If the delays of both relations happen to overlap to some extent, all algorithms enter the Reactive phase, since they all use the same triggering mechanism dictated by the value of parameter WaitThresh, which was set to 25 msecs. For this experiment, DINER returns to the Arriving phase when 1,000 input tuples have accumulated in the input buffer (input parameter MaxNewArr in Algorithm 4). The value of the input parameter MaxOuterMem in the same algorithm was set to 19 blocks. The parameter F for RPJ discussed in Section 2 was set to 10, as in the code provided by Tao et al. [12] and for HMJ to 10, which had the best performance in our
experiments for HMI. We observe that DINER generates up to 1.83 times more results than RPI when RPI does not overflow and up to 8.65 times more results than HMI. We notice that for the smaller delay duration, the RPI algorithm overflows its input buffer and, thus, cannot complete the join operation. In Table 3, we repeat the experiment allowing the input buffer to grow beyond the memory limit during the Reactive phase and present the maximum size of the input buffer during the Reactive phase. We also experimented with the IPIX and CSCO stock data. The intertuple delay is set as in the previous experiment. The results for the CSCO data are presented in Fig. 17, which show the percentage of tuples produced by RPI and HMI compared to the tuples produced by DINER.

CONCLUSIONS
In this work, we introduce DINER, a new adaptive join algorithm for maximizing the output rate of tuples, when two relations are being streamed to and joined at a local site. The advantages of DINER stem from 1) its intuitive flushing policy that maximizes the overlap among the join attribute values between the two relations, while flushing to disk tuples that do not contribute to the result and 2) a novel reentrant algorithm for joining disk-resident tuples that were previously flushed to disk. Moreover, DINER can efficiently handle join predicates with range conditions, a feature unique to our technique. We also present a significant extension to our framework in order to handle multiple inputs. The resulting algorithm, MINER addresses additional challenges, such as determining the proper order in which to probe the in-memory tuples of the relations, and a more complicated bookkeeping process during the Reactive phase of the join. Through our experimental evaluation, we have demonstrated the advantages of both algorithms on a variety of real and synthetic data sets, their resilience in the presence of varied data and network characteristics and their robustness to parameter changes.

ACKNOWLEDGEMENTS
The authors would like to thank Tao et al. [12] for making their code available to them. They would also like to thank Li et al. [8] for providing access to the Stock data set. Mihaela A. Bornea was supported by the European Commission and the Greek State through the PENED 2003 programme.

REFERENCES

AUTHOR BIOGRAPHY
K.SRIDHAR. Received the B.tech. in Information Technology from JNTU Hyderabad, Andhra Pradesh, India.