Adaptive Join Operators for Result Rate Optimization on Streaming Inputs

Mihaela A. Bornea, Vasilis Vassalos, Yannis Kotidis, and Antonios Deligiannakis

Abstract—Adaptive join algorithms have recently attracted a lot of attention in emerging applications where data are provided by autonomous data sources through heterogeneous network environments. Their main advantage over traditional 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.

Index Terms—Query processing, join, DINER, 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.

Adaptive join algorithms were created in order to lift the limitations of traditional join algorithms in such environments. By being able to produce results whenever input tuples become available, adaptive join algorithms overcome situations like initial delay, slow data delivery, or bursty arrival, which can affect the efficiency of the join. All existing algorithms work in three stages. During the Arriving phase, a newly arrived tuple is stored in memory and it is matched against memory-resident tuples belonging to the other relations participating in the join. Since the allocated memory for the join operation is limited and often much smaller than the volume of the incoming data, this results in tuple migration to disk. The decision on what to flush to disk influences significantly the number of results produced during the Arriving phase. The Arriving phase is suspended when all data sources are temporarily blocked and a reactive phase kicks in and starts joining part of the tuples that have been flushed to disk. An important desideratum of this phase is the prompt handover to the Arriving phase as soon as any of the data sources restarts sending tuples. Each algorithm has a handover delay which depends on the minimum unit of work that needs to be completed before switching phases. This delay has not received attention in the past, but we show that it can easily lead to input buffer overflow, lost tuples, and hence incorrect results. When all sources complete the data
transmission, a Cleanup phase is activated and the tuples that were not joined in the previous phases (due to flushing of tuples to disk) are brought from disk and joined.

Even if the overall strategy has been proposed for a multiway join, most existing algorithms are limited to a two-way join. Devising an effective multiway adaptive join operator is a challenge in which little progress has been made [15].

In this paper, we propose two new adaptive join algorithms for output rate maximization in data processing over autonomous distributed sources. The first algorithm, Double Index NEsted-loop Reactive join (DINER) is applicable for two inputs, while Multiple Index NEsted-loop Reactive join (MINER) can be used for joining an arbitrary number of input sources.

DINER follows the same overall pattern of execution discussed earlier but incorporates a series of novel techniques and ideas that make it faster, leaner (in memory use), and more adaptive than its predecessors. More specifically, the key differences between DINER and existing algorithms are 1) an intuitive flushing policy for the Arriving phase that aims at maximizing the amount of overlap of the join attribute values between memory-resident tuples of the two relations and 2) a lightweight Reactive phase that allows the algorithm to quickly move into processing tuples that were flushed to disk when both data sources block. The key idea of our flushing policy is to create and adaptively maintain three nonoverlapping value regions that partition the join attribute domain, measure the “join benefit” of each region at every flushing decision point, and flush tuples from the region that doesn’t produce many join results in a way that permits easy maintenance of the three-way partition of the values. As will be explained, proper management of the three partitions allows us to increase the number of tuples with matching values on their join attribute from the two relations, thus, maximizing the output rate. When tuples are flushed to disk they are organized into sorted blocks using an efficient index structure, maintained separately for each relation (thus, the part “Double Index” in DINER). This optimization results in faster processing of these tuples during the Reactive and Cleanup phases. The Reactive phase of DINER employs a symmetric nested loop join process, combined with novel bookkeeping that allows the algorithm to react to the unpredictability of the data sources. The fusion of the two techniques allows DINER to make much more efficient use of available main memory. We demonstrate in our experiments that DINER has a higher rate of join result production and is much more adaptive to changes in the environment, including changes in the value distributions of the streamed tuples and in their arrival rates.

MINER extends DINER to multiway joins and it maintains all the distinctive and efficiency generating properties of DINER. MINER maximizes the output rate by: 1) adopting an efficient probing sequence for new incoming tuples which aims to reduce the processing overhead by interrupting index lookups early for those tuples that do not participate in the overall result; 2) applying an effective flushing policy that keeps in memory the tuples that produce results, in a manner similar to DINER; and 3) activating a Reactive phase when all inputs are blocked, which joins on-disk tuples while keeping the result correct and being able to promptly hand over in the presence of new input. Compared to DINER, MINER faces additional challenges namely: 1) updating and synchronizing the statistics for each join attribute during the online phase, and 2) more complicated bookkeeping in order to be able to guarantee correctness and prompt handover during reactive phase. We are able to generalize the reactive phase of DINER for multiple inputs using a novel scheme that dynamically changes the roles between relations.

An important feature of both DINER and MINER is that they are the first adaptive, completely unblocking join techniques that support range join conditions. Range join queries are a very common class of joins in a variety of applications, from traditional business data processing to financial analysis applications and spatial data processing. Progressive Merge Join (PMJ) [4], one of the early adaptive algorithms, also supports range conditions, but its blocking behavior makes it a poor solution given the requirements of current data integration scenarios. Our experiments show that PMJ is outperformed by DINER.

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 to existing adaptive join algorithms and 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 MJoin algorithm when a four-way star join is executed in a constrained memory environment.

The rest of the paper is organized as follows: Section 2 presents prior work in the area. In Section 3, we introduce the DINER algorithm and discuss in detail its operations.
MINER is analyzed in Section 4. Section 5 contains our experiments, and Section 6 contains concluding remarks.

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 nonblocking sort-based algorithm was PMJ [4].

Hash Merge Join (HMJ) [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 HMJ 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 $n\text{part}$ 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 $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$. 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 (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 nonblocking 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 connection 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 $f^{(a)}_j$, the probability of a new incoming tuple to belong to relation $R_j$ and to be part of bucket $j$. Once all probabilities are computed, the flushing
policy is applied. Let \( p^{\text{out}}_j \) be the smallest probability. In this case, \( n_{\text{flush}} \) tuples belonging to bucket \( j \) 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 Diner 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

**Algorithm internals and data structures.** Incoming tuples from both relations share the available memory. A separate index \( \text{Index}_i (i \in \{A,B\}) \) on the join attribute is maintained for the memory resident part of each input relation. Two index characteristics are important for our framework: 1) small memory footprint; and 2) the ability to have sorted access based on the index keys. 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 \( \text{MemThresh} \). The value of \( \text{UsedMemory} \) 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, \( \text{Disk}_A \) and \( \text{Disk}_B \), 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 arrives, 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. 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 $t_i \in R_i$ arrived ($i \in \{A, B\}$) then
3. Move $t_i$ from input buffer to Diner process space.
4. Augment $t_i$ with join bit and arrival timestamp
5. $j = \{A, B\} - i$  
6. $matchSet = \text{set of matching tuples (found using } Index_j \text{)}$ from opposite relation $R_j$
7. $joinNum = |matchSet|$ (number of produced joins)
8. if $joinNum > 0$ then
9. Set the join bits of $t_i$ and of all tuples in $matchSet$
10. end if
11. UpdateStatistics($t_i, numJoins$)
12. $indexOverhead = \text{Space required for indexing}$
13. $t_i$ using $Index_i$
14. while $\text{UsedMemory} + indexOverhead \geq MemThresh$ do
15. Apply flushing policy  
16. end while
17. UpdateUsedMemory
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 (discussed in Section 3.3) 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 (Line 2). 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 $R_i$ ($i \in \{A, B\}$) with the most in-memory tuples
2. [Compute benefit of each region]
3. $BfUp_i = UpJoins_i / UpTups_i$
4. $BfLw_i = LwJoins_i / LwTups_i$
5. $BfMd_i = MdJoins_i / MdTups_i$
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 $BfUp_i$ is the minimum benefit then
9. locate $Tups\_Per\_Block$ tuples with the larger join attribute using $Index_i$
10. flush the block on $Disk_i$
11. update $LastUpVal$, so that the upper region is (about) a disk block
12. else if $BfLw_i$ is the minimum benefit then
13. locate $Tups\_Per\_Block$ tuples with the smaller join attribute using $Index_i$
14. flush the block on $Disk_i$
15. update $LastLwVal$, so that the lower region is (about) a disk block
16. else
17. Using the Clock algorithm, visit the tuples from the middle area, using $Index_i$, until $Tups\_Per\_Block$ tuples are evicted.
18. end if
19. Update $UpTups_i, LwTups_i, MdTups_i$ when necessary
20. $UpJoins_i, LwJoins_i, MdJoins_i \leftarrow 0$

**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).3

**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.

---

3. After a relation has completely arrived, maintaining tuples from the opposite relation does not incur any benefit at all. At that point it would be useful to flush all tuples of the relation that is still being streamed and load more tuples from the relation that has ended.
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: \(LastLwVal\) and \(LastUpVal\), 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 \(LastLwVal\) (\(LastUpVal\)) 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**

```plaintext
Require: \(t_i (i \in \{A, B\}), numJoins\)
1. \(j = \{A, B\} - i\) \{Refers to the opposite relation\}
2. JoinAttr \leftarrow t_i.JoinAttr
3. \{Update statistics on corresponding region of relation \(R_i\)\}
4. if JoinAttr \(\geq LastUpVal\), then
5. \(UpJoins_i, UpJoins_i + numJoins\)
6. \(UpTups_i, UpTups_i + 1\)
7. else if JoinAttr \(\leq LastLwVal\), then
8. \(LwJoins_i, LwJoins_i + numJoins\)
9. \(LwTups_i, LwTups_i + 1\)
10. else
11. \(MdJoins_i, MdJoins_i + numJoins\)
12. \(MdTups_i, MdTups_i + 1\)
13. end if
14. \{Update statistics of opposite relation\}
15. if JoinAttr \(\geq LastUpVal\), then
16. \(UpJoins_i, UpJoins_i + numJoins\)
17. \(UpTups_i, UpTups_i + 1\)
18. else if JoinAttr \(\leq LastLwVal\), then
19. \(LwJoins_j, LwJoins_j + numJoins\)
20. \(LwTups_j, LwTups_j + 1\)
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 \(BfLw_i\), of the lower region of \(R_i\) to be equal to: \(BfLw_i = \frac{LwJoins_i}{LwTups_i}\). The corresponding benefit \(BfUp_i\) and \(BfMd_i\) 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 \(LastLwVal\) (maximum \(LastUpVal\)) values such that the lower (upper) region contains enough tuples to fill a disk block. The new \(LastLwVal\) and \(LastUpVal\) 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 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 LastLwVal and LastUpVal values have not been previously 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 LastLwVal and LastUpVal 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 LastLwVal and LastUpVal values never create overlapping intervals for the lower and upper regions (i.e., the invariant LastLwVal < LastUpVal 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 DiskA and DiskB, 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(DiskA, DiskB)
3: Set Inner ← largest(DiskA, DiskB)
4: Set JoinedInner to size of Inner Relation
5: Set JoinedOuter to 0 and CurrInner to 1
6: end if {Else the old parameter values are used}
7: while unprocessed blocks exist do
8: while JoinedOuter < CurrInner 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 = min(MaxOuterMem, OuterSize-JoinedOuter)
12: end if {Else keep its previous value}
13: 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.
14: while CurrInner <= JoinedInner do
15: 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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OuterSize</td>
<td>Size of outer relation in blocks</td>
</tr>
<tr>
<td>InnerSize</td>
<td>Size of inner relation in blocks</td>
</tr>
<tr>
<td>OuterMem</td>
<td>Fetched disk blocks from outer relation at each step</td>
</tr>
<tr>
<td>JoinedOuter</td>
<td>Last outer block that has joined with inner relation up to JoinedInner</td>
</tr>
<tr>
<td>JoinedInner</td>
<td>Last inner block that has joined with outer relation up to JoinedOuter</td>
</tr>
<tr>
<td>CurrInner</td>
<td>Inner block being joined with latest chunk of OuterMem outer blocks</td>
</tr>
<tr>
<td>MaxNewArr</td>
<td>Maximum number of tuples accumulated in the input buffer before exiting Reactive Phase</td>
</tr>
</tbody>
</table>

Fig. 1. Status of the algorithm during the reactive phase.

Fig. 2. Status of algorithm after swapping the roles of the two relations.
16: 
17: CurrInner ← CurrInner + 1
18: if Input buffer size greater than MaxNewArr then
19: Break;
20: end if
21: end while
22: if CurrInner > JoinedInner then
23: [Mark completed join of [1..JoinedOuter] and [1..JoinedInner] blocks]
24: JoinedOuter ← JoinedOuter + OuterMem
25: CurrInner ← 1
26: if Input buffer size greater than MaxNewArr then
27: [Switch back to Arriving Phase]
28: exit
29: end if
30: if JoinedOuter == OuterSize then
31: Change roles between inner and outer
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 DINER 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 (Lines 30-32), 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 OuterMem 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 at Lines 17-19 and 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+OuterMem] 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 \( t_{outer} \) and \( t_{inner} \) 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 \([t_{outer,ATS}, t_{outer,DTS}]\) and \([t_{inner,ATS}, t_{inner,DTS}]\) overlap, this means that the two tuples coexisted in memory during the Arriving phase and their join is already obtained. Thus, such pairs of tuples are ignored by the ReactiveNL algorithm (Line 15).

**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 RPJ 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 RPJ 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
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 join of the form

$$R_1 \bowtie_{R_1.a_1 = R_2.a_1} R_2 \bowtie_{R_2.a_2 = R_3.a_2} R_3 \bowtie \cdots \bowtie_{R_{i-1}.a_{i-1} = R_i.a_i} R_i.$$ 

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 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 $\sigma_{R_i.R_{i+1}}$ between each pair of relations $R_i, R_{i+1}$ and then forming the probing sequence by considering pairs of increasing join selectivity factor. For example, assume that in Fig. 3 $\sigma_{R_i.R_i} \leq \sigma_{R_{i+1}.R_i} \leq \sigma_{R_{i+1}.R_{i+1}}$. Then, an incoming tuple belonging to $R_3$ will first be joined with $R_1$, then with $R_2$, 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 overall join results and the number of tuples in the area. The benefit for each area as well as the statistics are maintained similarly to what happens in DINER.

Fig. 3 depicts the corresponding conceptual areas and the maintained statistics for our sample four-way join. Let us focus on the second relation $R_2$. This relation participates in joins involving two of its attributes, namely attributes $a_1$ and $a_2$. For join attribute $a_1$, MINER maintains the three areas $Up_{R_2.a_1}, Md_{R_2.a_1}, and Lw_{R_2.a_1}$, separated by the conceptual boundaries $LastUpVal_{R_2.a_1}$ and $LastLwVal_{R_2.a_1}$. The following statistics are kept for attribute $R_2.a_1$: $UpJoins_{R_2.a_1}, MdJoins_{R_2.a_1}, and LwJoins_{R_2.a_1}$, representing the number of overall joins produced by matches in areas $Up_{R_2.a_1}, Md_{R_2.a_1}, and Lw_{R_2.a_1}$. In addition, $UpTups_{R_2.a_1}, MdTups_{R_2.a_1}, and LwTups_{R_2.a_1}$ represent the number of tuples belonging in the conceptual areas $Up_{R_2.a_1}, Md_{R_2.a_1}, and Lw_{R_2.a_1}$. The conceptual areas and the statistics for attribute $R_2.a_2$ are maintained in a similar manner.

$LocalJoins_{R_i.a_i}$ (respectively, $LocalJoins_{R_i.a_i}$) represent the number of joins in which relations $R_2$ and $R_1$ (respectively, $R_2$ and $R_i$) have participated, regardless of whether these joins led to an overall result or not. These values are used in order to estimate the join selectivity factors and are zeroed after every flush operation. $Tups_{R_i.a_i}$ denotes the number of in-memory tuples of $R_i$. The join selectivity factor between relations $R_i$ and $R_{i+1}$ is estimated as $\sigma_{R_i.R_{i+1}} = \frac{LocalJoins_{R_i.a_i}}{Tups_{R_i.a_i}}$.

Flushing policy. The flushing policy is applied when the memory is exhausted. MINER flushes tuples from the relation with the largest number of in-memory tuples. From the selected relation, tuples are flushed from the area with the smallest benefit. An important observation is that, due to the multiple join attributes per relation, the flushed tuples may also be part of a conceptual area that is maintained for the same relation on a different join attribute. Thus, during the flushing process, the index for the additional join attribute is modified as well as the tuple counters for the affected conceptual regions.

In our sample join scenario depicted in Fig. 3, assume that $Up_{R_4.a_5}$ is selected as the victim area from which tuples will be flushed. For each tuple that is flushed from this area, the value of attribute $R_2.a_1$ is extracted and the corresponding entry is removed from $Index_{R_2.a_1}$. Next, the value of attribute $R_2.a_1$ is compared to $LastUpVal_{R_2.a_1}$ and $LastLwVal_{R_2.a_1}$ in order to determine its region and then update the cardinality statistics for attribute $R_2.a_1$ (possibly modifying the region boundaries). 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 of the

2. The counter of join results refers to the number of produced (final), and not partial, join results that tuples in each region helped produce. Additional counters related to partial join results are also maintained to estimate join selectivity factors.
sequences are created: the first join sequence computes the 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 the inner relations. (i.e., those relations are considered as “more inner” in the join sequence). When it is time to advance the CurrInner 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, blocks 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 property 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

$$R_1 \times_1 R_2 \times_2 R_3 \times_3 R_4$$

when relation $R_i$ becomes the outer relation, two join sequences are created: the first join sequence computes the join of $R_i$ with all relations $R_j$, where $j > i$, while the second sequence computes the join of $R_i$ with all relations $R_j$, where $j < i$. Just one join sequence will exist only when either the first or the last relation, in the original joined order, assumes the role of the outer relation. Fig. 5 depicts an example where the second relation $R_2$ has assumed the role of the outer relation (please note that the batch of currently joined outer blocks belong to $R_2$ in the figure), and one join sequence contains the join of relations $R_3$, $R_4$, and $R_5$, 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.

Even though our discussion was limited to chain joins, MINER can also be extended to snowflake joins, where each relation may participate in joins with more than two join attributes. At the top of Fig. 6, we illustrate such an example, while at the bottom of the same figure we depict the join sequences when $R_3$ has assumed the role of the outer relation. The experimental evaluation of this extension is a topic of future work.

**Redundant tuple detection.** During the Reactive Phase, it is possible that output tuples are generated multiple times. MINER uses the same mechanism as DINER for duplicate avoidance, and augments each tuple with an arrival and a departure time stamp. During the reactive phase, the algorithm determines that a matching tuple combination has already produced a result during the
online phase if the arrival, departure intervals of all tuples in the combination overlap.

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 HMJ [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 HMJ, as shown 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. 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. 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++. For RPJ, also implemented in C++, 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. The benefits of DINER are more significant when the size of the available memory given to the join algorithms is reduced.

In the next set of experiments, we evaluate the performance of the algorithms when synthetic data are used. In all runs, each relation contains 100,000 tuples. In Fig. 10, the values of the join attribute follow a Zipf distribution in the range $\frac{y}{C_21} = 0.1$. 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.

interval $[1, 999]$ and the skew parameter is varied from 1 to 5, with the leftmost value (i.e., value 1) exhibiting the highest probability of occurrence in each case. We observe that the benefits of DINER are even larger than in the real data.

In Figs. 11 and 12, the join attribute follows the normal distribution. In the first figure, the join attribute of the first relation has a normal distribution with mean 500 and variance 50, while the join attribute of the second relation is normally distributed, with variance 50 and mean equal to $500 + x$. Thus, by increasing the value of $x$ (x-axis) we increase the degree of separation between the two distributions. For the experiments in Fig. 12, the first relation is the same as in Fig. 11, while the mean of the second relation is 500 but, this time, the variance is modeled as $50 + x$, where $x$ is the value on the x-axis. We observe that the larger the divergence between the two distributions, the larger the benefits of DINER when compared to RPJ and HMJ. The increase in the benefit is more evident when the mean values of the two distributions (middle region). In contrast, for the experiment in Fig. 12, the distributions have the same mean value and most of the flushed tuples come from the upper and lower regions of DINER.

The above experiments demonstrate that DINER adapts better to different distributions of the join attribute, as shown in Figs. 10, 11, and 12.

Range predicates. In Fig. 13, we present experimental results for range joins on the Stock and Weather data sets. The streams for the two relations are formed as described earlier, with the exception of the join conditions: A pair of tuples from the Stock data set joins if their price difference is smaller than 5 cents. For the Weather data set, each relation contains environmental information from a specific location and two tuples join if their temperature difference is smaller than 5 degrees. The y axis of Fig. 13 shows the fraction of the overall result that was obtained during the online phase of DINER and PMJ. The memory size is set at 5 percent of the total input size. We note that RPJ and HMJ do not support such join predicates, since they are hash based.

5.2 Nonstationary Data Streaming Processes

The DINER algorithm adapts quickly to situations where the value distribution of the join attribute changes over
time. We present two experiments that demonstrate DINER’s superior performance in such situations. In Fig. 14, the input relations are streamed partially sorted as follows: The frequency of the join attribute is determined by a zipf distribution with values between $\frac{1}{999}$ and skew 1. The data are sorted and divided in a number of “intervals,” shown on the x axis. The ordering of the intervals is permuted randomly and the resulting relation is streamed. The benefits of DINER are significant in all cases.

In the second experiment, shown in Fig. 15, we divide the total relation transmission time in 10 periods of 10 sec each. In each transmission period, the join attributes of both relations are normally distributed, with variance equal to 15, but the mean is shifted upwards from period to period. In particular, the mean in period $i$ is $75 + i \times \text{shift}$, where $\text{shift}$ is the value on the x axis. We can observe that the DINER algorithm produces up to three times more results during the online phase than RPJ and HMJ, with its relative performance difference increasing as the distribution changes more quickly.

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 HMJ. We observe that DINER generates up to 1.83 times more results than RPJ when RPJ does not overflow and up to 8.65 times more results than HMJ.

We notice that for the smaller delay duration, the RPJ 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 RPJ and HMJ compared to the tuples produced by DINER.
at the end of the online phase. The graph for the IPIX data is qualitatively similar and is omitted due to lack of space.

From the above experiments, the following conclusions can be drawn:

- DINER achieves a faster output rate in an environment of changing data rates and delays.
- DINER uses much less memory, as depicted in Table 3. The reason for that is that it has a smaller "work unit" (a single inner relation block) and is able to switch over to the Arriving phase with less delay, without letting the input buffer become full.
- Finally, notice that in the first experiment of Fig. 16, RPJ fails due to memory overflow. We can see in Table 3 that the maximum input buffer size RPJ needs in this case is 40,000, while the available memory is 10,000 tuples. For less available memory, HMJ fails also—the relevant experiment is omitted due to lack of space.

5.4 Sensitivity Analysis

In the previous sections we have already discussed its performance in the presence of smaller or larger network/source delays, and in the presence of nonstationarity in the tuple streaming process. Here, we discuss DINER's performance when the available memory varies and also when the skew of the distribution varies.

Table 4 refers to the experiments for real data sets shown in Fig. 9 and shows the percentage of the total join result obtained by DINER during the online phase. As we can see, if enough memory is made available (e.g., 20 percent of the total input size), a very large percentage of the join tuples can be produced online, but DINER manages to do a significant amount of work even with memory as little as 5 percent of the input size, which, for very large relation sizes, is more realistic.

We also evaluate the DINER algorithm when one relation has a smaller size than the second. In Fig. 18, the value of the join attribute is zipf distributed in both relations. While one relation has a constant number of tuples (100,000), the other relation’s size varies according to the values on the x-axis. The memory size is set to 10,000 tuples and the interarrival rate is exponentially distributed with $\lambda = 0.1$. DINER is a nested-loop-based algorithm and it shows superior performance as one relation gets smaller since it is able to keep a higher percentage of the smaller relation in memory.

We finally evaluated the sensitivity of the DINER algorithm to the selected value of the MaxOuterMem parameter, which controls the amount of memory (from the total budget) used during the Reactive phase. We varied the value of MaxOuterMem parameter from 9 to 59 blocks. The experiment, shown in Fig. 19, showed that the performance of DINER is very marginally affected by the amount of memory utilized during the Reactive phase.

5.5 Multiple Inputs

During the previous sections, we have analyzed the performance of our technique in the presence of two inputs. Next, we evaluate our extended technique when the streamed data are provided by more than two sources. We consider the following join:

$$R_1 \bowtie_{R_1.a_1 = R_2.a_2} R_2 \bowtie_{R_2.a_2 = R_3.a_2} R_3 \bowtie_{R_3.a_3 = R_4.a_3} R_4.$$ 

When two inputs ($R_2$, $R_3$) are considered, the join attributes in each relation have a zipf distribution in $[0, 9999]$ and skew 1. When the third input ($R_4$) is added, it participates in the join with one join attribute, which is unique in the domain $[0, 9999]$ (i.e., a key), such that a foreign key-primary key join is performed with the first of the two initial relations ($R_1$). The foreign key values from relation $R_2$ follow the zipf distribution with skew 1. The fourth input ($R_4$) is generated with the same logic and it performs a foreign key-primary key join with the second initial relation ($R_3$). This particular setup has a few advantages. First, it contains both a many-to-many join (between the two initial inputs) and foreign key-primary
key joins (between the initial relations and the additional ones). Second, when additional inputs are added (in particular, when we move from two to three and from three to four inputs) the selectivity of the multijoin operation remains the same, allowing us to draw conclusions on the performance of MINER.

Fig. 20 presents the percentage of the total result obtained by MINER during the online phase, in the presence of multiple inputs while we vary the available memory. We observe only a modest decrease in the number of results obtained as we increase the number of inputs. Moreover, in this experiment, we measured memory overhead of MINER due to indexes. For a memory budget equal to 20 percent (i.e., 8.8 MB) of the input size and in a setup with four inputs, the memory occupied by the six indexes is equal to 120 KB, which is less than 2 percent of the total allocated memory.

In the presence of network/source delays, MINER still produces a large number of results by the end of the arrival phase, while being able to handover to the online phase when new input is available. This is reflected in Fig. 21, where the size of the input buffer is presented. During the reactive phase, the delays were introduced as in the corresponding DINER experiments, and handover is triggered when 1,000 tuples are gathered in the input buffer.

In Fig. 22, we present a comparison between MINER and MJoin for the joins of the form $R_1 \bowtie_{R_1.a=R_2.a} R_2 \bowtie_{R_2.a=R_3.a} R_3 \bowtie_{R_3.a=R_4.a} R_4$ that can be handled by MJoin. We call such a join Simple Star Join (SSTAR). The same data sets as in the previous experiment are used. The results in Fig. 22 show the benefits of the data-aware flushing policy of MINER and indicate that MJoin is practically not very suitable for our motivating applications.

**6 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.

**Acknowledgments**

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. Vasilis Vassalos was supported by the European Commission through a Marie Curie Outgoing International Fellowship and by the PENED 2003 programme of research support.

**References**


Vasilis Vassalos is an associate professor in the Department of Informatics at Athens University of Economics and Business. Between 1999 and 2003, he was an assistant professor of information technology at the NYU Stern School of Business. He has been working on the challenges of online processing of data from autonomous sources since 1996. He has published more than 35 research papers in international conferences and journals, holds two US patents for work on information integration, and regularly serves on the program committees of the major conferences in databases.

Yannis Kotidis received the BSc degree in electrical engineering and computer science from the National Technical University of Athens, the MSc and PhD degrees in computer science from the University of Maryland. He is an assistant professor in the Department of Informatics at the Athens University of Economics and Business. Between 2000 and 2006, he was a senior technical specialist at the Database Research Department of AT&T Labs-Research in Florham Park, New Jersey. His main research areas include large-scale data management systems, data warehousing, p2p, and sensor networks. He has published more than 60 articles in international conferences and journals and holds five US patents. He has served on numerous organizing and program committees of international conferences related to data management.

Antonios Deligiannakis received the diploma degree in electrical and computer engineering from the National Technical University of Athens in 1999, and the MSc and PhD degrees in computer science from the University of Maryland in 2001 and 2005, respectively. He is an assistant professor of computer science in the Department of Electronic and Computer Engineering at the Technical University of Crete, as of January 2008. He then performed his post-doctoral research in the Department of Informatics and Telecommunications at the University of Athens (2006-2007). He is also an adjunct researcher at the Digital Curation Unit (http://www.dcu.gr/) of the IMIS-Athena Research Centre.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.