Adaptive Join Operators for Result Rate Optimization on Streaming Inputs

Mihaela A. Bornea, Vasilis Vassalos, Yannis Kotidis #1, Antonios Deligiannakis *2

# Department of Informatics, Department of Electronic and Computer Engineering
Athens U. of Econ and Business,Technical University of Crete

1{mihaela,vassalos,kotidis}@aueb.gr

2adeli@softnet.tuc.gr

Abstract—Adaptive join algorithms have recently attracted a lot of attention in emerging applications where data is 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 re-entrant 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 is not available. We then extend the applicability of the proposed technique for a more challenging setup: handling more than two inputs. MINER (Multiple Index NEste-loop Reactive join) is a multi-way 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 multi-way 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 is 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 towards 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 non-blocking 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, on-line aggregation and approximate query answering systems.

Compared to traditional join algorithms (be they sort-, hash- or nested loops-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

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.

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 multi-way join, most existing algorithms are limited to a two-way join. Devising an effective multi-way 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 non-overlapping 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 3-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 multi-way 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; (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 [4] (PMJ), 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 a 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 multi-way join algorithm that maximizes the output rate, designed for dealing with cases where data is 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 is 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 non-blocking join algorithms to be proposed. XJoin was inspired by Symmetric Hash Join (SHJ) [6], which represented the first step towards 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 extention 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 Progressive Merge Join (PMJ) [4].

Hash Merge Join (HMJ) [9], based on XJoin and PMJ, is a non-blocking 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_{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 timestamp-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 $R_1 \bowtie_{R_1.a=R_2.a} R_2 \bowtie_{R_2.a=R_3.a} R_3 \bowtie \ldots \bowtie_{R_{n-1}.a=R_n.a} R_n$. 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 can not be used directly for joins with multiple attributes or non-star joins. [15] describes a naive extention where only a flushing policy that randomly evicts the memory content can be applied.

Progressive Merge Join. PMJ is the adaptive non-blocking 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 arrives, 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 XJoin (and also DPHJ [7]) perform, 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 $p_i^{arr}[j]$, the probability of a new incoming tuple to belong to relation R_i and to be part of bucket j. Once all probabilities are computed, the flushing policy is applied. Let $p_i^{arr}[j]$ be the smallest probability. In this case, n_{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 *Double Index NEsted-loop Reactive join* (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.

	Symbol	Description ($i \in \{A, B\}$)
General	R_i	Input relation R_i
	t_i	Tuple belonging to relation R_i
	$Index_i$	Index for relation R_i
	$Disk_i$	Disk partition containing flushed tuples of relation R_i
	UsedMemory	Current amount of memory occupied by tuples, indexes and statistics
	MemThresh	Maximum amount of available memory to the algorithm
	WaitThresh	Maximum time to wait for new data before switching to Reactive phase
Statistics	$LastLwVal_i$ $LastUpVal_i$	Thresholds for values of join attribute in lower and upper regions of R_i
	$LwJoins_i$ $MdJoins_i$ $UpJoins_i$	Number of produced joins by tuples in the lower, middle and upper region, correspondingly, of R_i
	$LwTups_i$ $MdTups_i$ $UpTups_i$	Number of in-memory tuples in the lower, middle and upper region, correspondingly, of R_i
	$BfLw_i$, $BfMd_i$ $BfUp_i$	Benefit of in-memory tuples in the lower, middle and upper region, correspondingly, of R_i

TABLE 1
Symbols Used in Our Algorithm

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 $Index_i$ ($i \in \{A, B\}$) on the join attribute is maintained for the memory resident part of each input relation. Two index characteristics are important for our framework: (a) small memory footprint; and (b) 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, 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 indices 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 timestamp (ATS). Each tuple flushed to disk is further augmented with a departure timestamp (DTS). As explained later in this section, these timestamps are used in order to ensure that during the Reactive and

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 ATS
5:     $j = \{A, B\} - i$     {Refers to "opposite" relation}
6:     $matchSet$ = set of matching tuples (found using $Index_j$) 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:    $indexOverhead$ = Space required for indexing $t_i$ using $Index_i$
13:    while UsedMemory+$indexOverhead \geq$ MemThresh do
14:      Apply flushing policy     (see Algorithm 2)
15:    end while
16:    Index $t_i$ using $Index_i$
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
```

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 timestamp 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 $LastUpVal_i$ 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 $LastLwVal_i$ 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: $UpJoins_i$,$LwJoins_i$,$MdJoins_i \leftarrow 0$
```

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 is 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.

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.

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).[1]

Intuition. The algorithm should keep in main memory those tuples that are most likely to produce results by joining with

[1] 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.

Algorithm 3 UpdateStatistics

```
Require: $t_i$ ($i \in \{A, B\}$), $numJoins$
$j = \{A, B\} - i$    {Refers to the opposite relation}
$JoinAttr \leftarrow t_i.JoinAttr$
{Update statistics on corresponding region of relation $R_i$}
if $JoinAttr \geq LastUpVal_i$ then
    $UpJoins_i \leftarrow UpJoins_i + numJoins$
    $UpTups_i \leftarrow UpTups_i + 1$
else if $JoinAttr \leq LastLwVal_i$ then
    $LwJoins_i \leftarrow LwJoins_i + numJoins$
    $LwTups_i \leftarrow LwTups_i + 1$
else
    $MdJoins_i \leftarrow MdJoins_i + numJoins$
    $MdTups_i \leftarrow MdTups_i + 1$
end if
{Update statistics of opposite relation}
if $JoinAttr \geq LastUpVal_j$ then
    $UpJoins_j \leftarrow UpJoins_j + numJoins$
else if $JoinAttr \leq LastLwVal_j$ then
    $LwJoins_j \leftarrow LwJoins_j + numJoins$
else
    $MdJoins_j \leftarrow MdJoins_j + numJoins$
end if
```

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 is 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 vise 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: $LastLwVal_i$ and $LastUpVal_i$ 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 $LastLwVal_i$ ($LastUpVal_i$) 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).

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: $BfLw_i = \frac{LwJoins_i}{LwTuples_i}$. 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 towards 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 $LastLwVal_i$ (maximum $LastUpVal_i$) values such that the lower (upper) region contains enough tuples to fill a disk block. The new $LastLwVal_i$ and $LastUpVal_i$ 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

Symbol	Description
OuterSize	Size of outer relation in blocks
InnerSize	Size of inner relation in blocks
OuterMem	#fetched disk blocks from outer relation at each step
MaxOuterMem	Maximum allowed value of OuterMem
JoinedOuter	Last outer block that has joined with inner relation up to JoinedInner
JoinedInner	Last inner block that has joined with outer relation up to JoinedOuter
CurrInner	Inner block being joined with latest chunk of OuterMem outer-blocks
MaxNewArr	Maximum number of tuples accumulated in the input buffer before exiting Reactive Phase

TABLE 2
Notation Used in The ReactiveNL Algorithm

from each relation for the very first time. In this case, the $LastLwVal_i$ and $LastUpVal_i$ values have not been previously set. Thus, at the first time our algorithm (i) considers all the tuples of each relation to belong to its middle region and flushes from that area; and (ii) using the index, which can provide sorted access to the data, quickly identifies $LastLwVal_i$ and $LastUpVal_i$ 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 is extremely skewed, the middle region may get entirely squeezed. (As explained earlier in this section, the endpoints always contain at least enough tuples to fill a block.) In such rare cases we make sure that the $LastLwVal_i$ and $LastUpVal_i$ values never create overlapping intervals for the lower and upper regions (i.e., the invariant $LastLwVal_i < LastUpVal_i$ 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 loops-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 re-entering the Reactive phase.

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 if{Else the old parameter values are used}
 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= min(MaxOuterMem, OuterSize-JoinedOuter)
12:     end if{Else keep its previous value}
13:     Load OuterMem blocks from outer relation starting with block-id Joined-
        Outer+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
16:       CurrInner←CurrInner+1
17:       if Input buffer size greater than MaxNewArr then
18:         Break;
19:       end if
20:     end while
21:     if CurrInner>JoinedInner then
22:       {Mark completed join of [1..JoinedOuter] and [1..JoinedInner] blocks}
23:       JoinedOuter←JoinedOuter+OuterMem
24:       CurrInner←1
25:     end if
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
```

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. Figures 1 and 2 provide a helpful visualization of the progress of the algorithm. Its operation based on the following points.

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

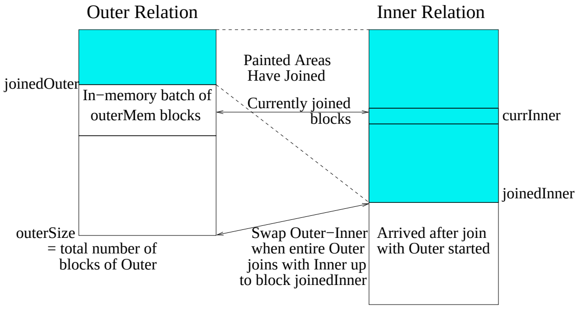
Outer Relation
Inner Relation
Painted Areas Have Joined
joinedOuter
In−memory batch of outerMem blocks
Currently joined blocks
currInner
joinedInner
outerSize = total number of blocks of Outer
Swap Outer−Inner when entire Outer joins with Inner up to block joinedInner
Arrived after join with Outer started

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

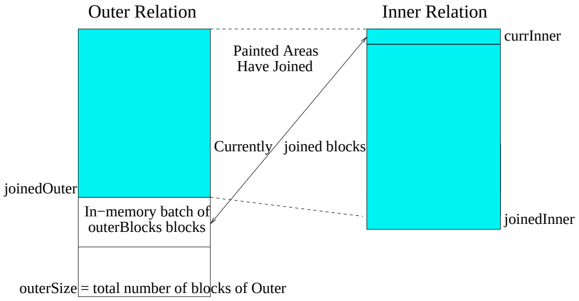
Outer Relation
Inner Relation
Painted Areas Have Joined
currInner
Currently joined blocks
joinedOuter
In−memory batch of outerBlocks blocks
joinedInner
outerSize = total number of blocks of Outer

Fig. 2. Status of algorithm after swapping the roles of the two relations.

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 re-entry. 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 timestamps. 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 starts 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 algorithm. In this case, one block of memory is kept for the inner relation and the rest of the blocks are allocated for the outer by properly setting the OuterMem parameter.

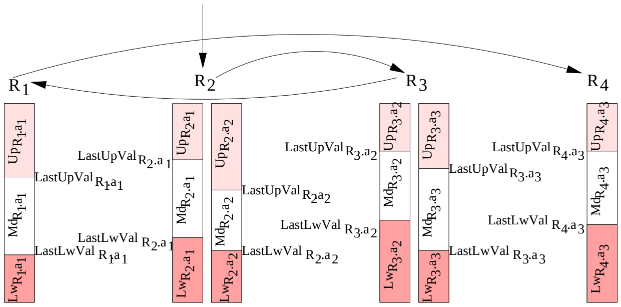

Fig. 3. Online Phase MINER

4 MINER Description

We now present an extension of the DINER algorithm, termed as *Multiple Index NEsted-loop Reactive* join (MINER), for joining multiple relations. For ease of presentation, we conduct our discussion for joins of the form $R_1 \bowtie_{R_1.a_1=R_2.a_1} R_2 \bowtie_{R_2.a_2=R_3.a_2} R_3 \bowtie \ldots \bowtie_{R_{n-1}.a_{n-1}=R_n.a_{n-1}} R_n$. 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. Figure 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 $\sigma_{R_i,R_{i+1}}$ 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 Figure 3 $\sigma_{R_2,R_3} \leq \sigma_{R_1,R_2} \leq \sigma_{R_3,R_4}$. 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 overall join results and the number of tuples in the area.[2] The benefit for each area as well as the statistics are maintained similarly to what happens in DINER.

Figure 3 depicts the corresponding conceptual areas and the maintained statistics for our sample 4-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 $Up_{R_2.a_1}$, $Md_{R_2.a_1}$, $Lw_{R_2.a_1}$, separated by the conceptual boundaries $LastUpVal_{R_2.a_1}$ and $LastLwVal_{R_2.a_1}$. The following statistics are kept for attribute $R_2.a_1$: $UpJoins_{R_2.a_1}$, $MdJoins_{R_2.a_1}$, $LwJoins_{R_2.a_1}$, 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.

$LocalJoins_{R_2.a_1}$ (resp. $LocalJoins_{R_2.a_2}$) represent the number of joins in which relations R_2 and R_1 (resp. R_2 and R_3) 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. $Tups_{R_i}$ 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 $\sigma_{R_i,R_{i+1}} = \frac{LocalJoins_{R_i.a_i}}{Tups_{R_i} * Tups_{R_{i+1}}}$

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 Figure 3, assume that $Up_{R_2.a_2}$ 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).

4.2 Extending the Reactive Phase

The Reactive phase is activated when all data sources experience delays.

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 of the inner

[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.

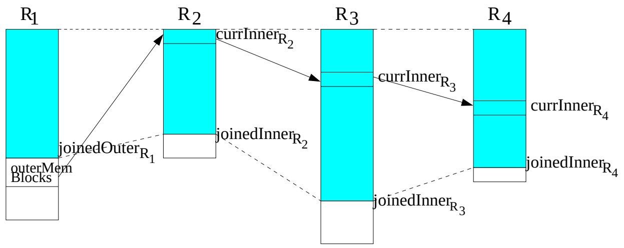

Fig. 4. Reactive Phase

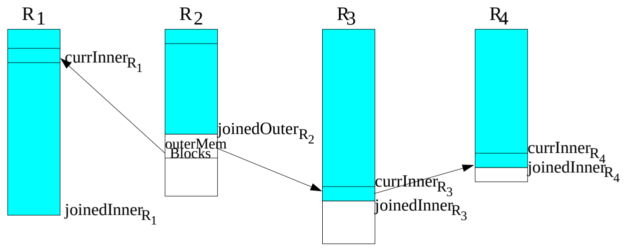

Fig. 5. Changing Outer-Inner Roles

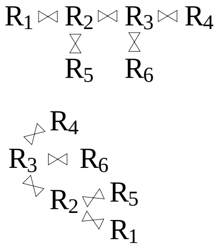

Fig. 6. Joins

relations, the JoinedInner statistic is initially set to the size of the relation.

Similarly to what happens in Algorithm 4, the MINER 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$_i$ 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$_i$ 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 multi-way joins considered, namely $R_1 \bowtie_{R_1.a_1=R_2.a_1} R_2 \bowtie_{R_2.a_2=R_3.a_2} R_3 \bowtie \ldots \bowtie_{R_{n-1}.a_{n-1}=R_n.a_{n-1}} R_n$, 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 join 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, assume the role of the outer relation. Figure 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_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 Figure 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. In the top of Figure 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 its 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 timestamp. 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 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 3 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 non stationary 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 our experimental section, it is a generally poor choice since its performance is limited by its blocking behavior.

- **An Efficient Multi-way 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% (when the allocated memory is 5% of the total input size) to more then 80% (when the allocated memory size is equal to 20% of the total input size).

Parameter Settings. The following settings are used during the experimental section: The tuple size is set to 200 bytes and the disk block size is set to 10KB. The inter-arrival delay (i.e., the delay between two consecutive incoming tuples), unless specified otherwise, is modelled using an exponential distribution with parameter $\lambda = 0.1$. The memory size allocated for all algorithms is 5% 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++. For RPJ, also implemented in C++, we used the code that the authors of [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 Figures 7, 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 Figure 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

[3] At http://cs-people.bu.edu/jching/icde06/ExStockData.tar

[4] At http://www-k12.atmos.washington.edu/k12/grayskies/nw_weather.html

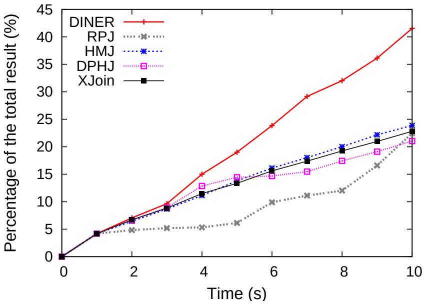

Fig. 7. CSCO data set

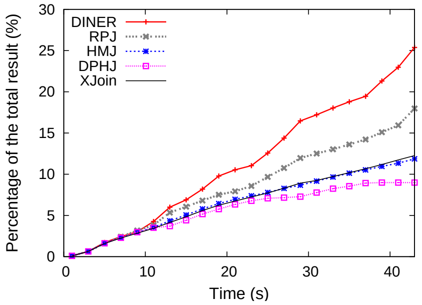

Fig. 8. Weather data set

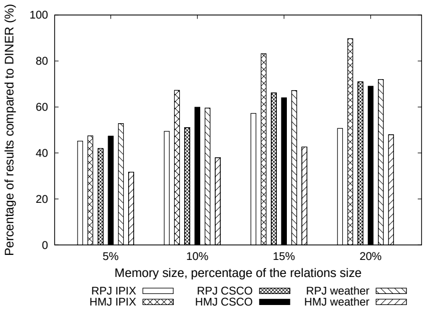

Fig. 9. Real data, varying memory

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 is used. In all runs, each relation contains 100,000 tuples. In Figure 10, the values of the join attribute follow a Zipf distribution in the 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 Figures 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 Figure 12 the first relation is the same as in Figure 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 value varies (Figure 11) and is largely attributed to the flushing policy that cleans up unproductive tuples between the mean values of the two distributions (middle region). In contrast, for the experiment in Figure 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 Figures 10, 11 and 12.

Range Predicates. In Figure 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 Figure 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% of the total input size. We note that RPJ and HMJ do not support such join predicates, since they are hash-based.

5.2 Non Stationary 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 Figure 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 $[1, 999]$ and skew 1. The data is 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 Figure 15, we divide the total relation transmission time in 10 periods of $10sec$ 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 $75 + i * shift$, where $shift$ is the value on the x axis. We can observe that the DINER algorithm produces up to 3 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 Figure 16, the streamed relations' join attributes follow a zipf distribution with a skew parameter of 1. The interarrival time is modelled 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 inter-tuple delays with $\lambda = 0.1$. After each fragment is transmitted, a delay equal to $x\%$ 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

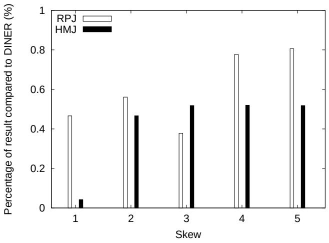

Fig. 10. Zipf distribution, memory 5% of input size

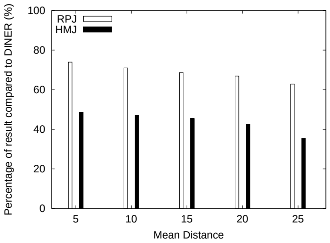

Fig. 11. Normal distribution, different mean

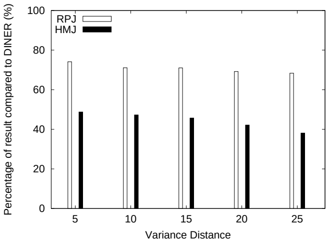

Fig. 12. Normal distribution, different variance

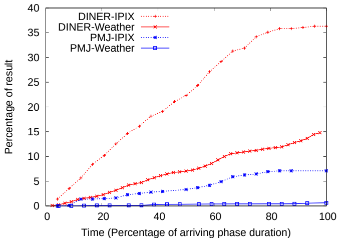

Fig. 13. Range Join Queries

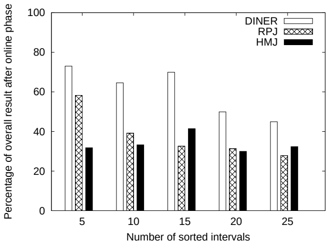

Fig. 14. Zipf distribution, partially sorted data

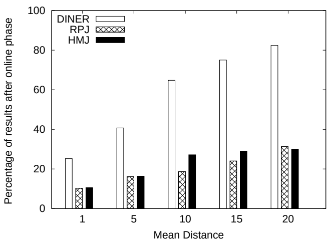

Fig. 15. Nonstationary normal, changing means

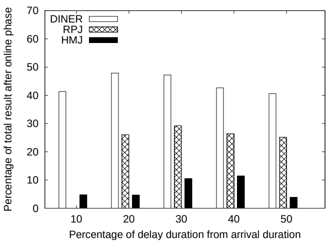

Fig. 16. Zipf distribution, network delays

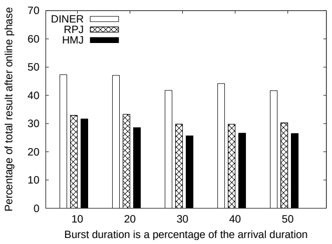

Fig. 17. CSCO stock data, network delays

	Inter-Arrival Delay (% of transmission time)				
Buffer Size	10 %	20 %	30 %	40 %	50 %
DINER Max	1030	1009	1008	1008	1004
DINER Avg	930	987	988	1001	1001
RPJ Max	**40000**	6548	6016	3422	2719
RPJ Avg	**40000**	3204	2880	1723	1377
HMJ Max	1443	423	2939	8269	7144
HMJ Avg	440	117	671	3356	2423

TABLE 3
Input buffer size during reactive phase

	Memory (% of input size)			
Data Set	5 %	10 %	15 %	20 %
CSCO	28 %	43 %	49 %	58 %
IPIX	38 %	59 %	72 %	82 %
Weather	10 %	17%	24 %	29 %

TABLE 4
Result obtained by DINER during online phase

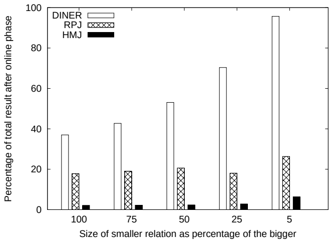

Fig. 18. Zipf Distribution, Different Relation Size

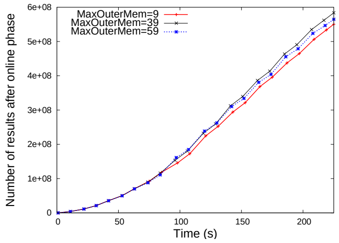

Fig. 19. Zipf distribution, varying MaxOuterMem

they all use the same triggering mechanism dictated by the value of parameter WaitThresh, which was set to 25msecs. For this experiments, 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 the authors of [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 inter-tuple delay is set as in the previous experiment. The results for the CSCO data are presented in Figure 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 Figure 16, RPJ fails due to memory overflow. You 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 non-stationarity in the tuple

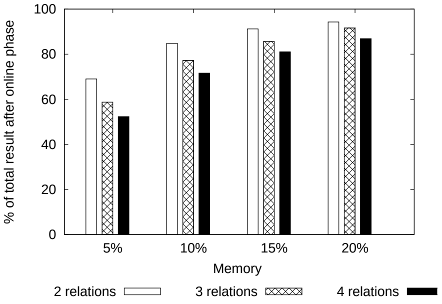

Fig. 20. Multiple Inputs, Online Phase

	Inter-Arrival Delay (% of transmission time)				
Buffer Size	10 %	20 %	30 %	40 %	50 %
2 Inputs Max	1003	1004	1003	1001	1002
2 Inputs Avg	945	918	885	842	827
3 Inputs Max	1003	1003	1003	1002	1003
3 Inputs Avg	1000	1000	1001	1000	1000
4 Inputs Max	1003	1003	1003	1002	1002
4 Inputs Avg	1001	1000	1000	1000	1000

Fig. 21. Input buffer for MINER during reactive phase

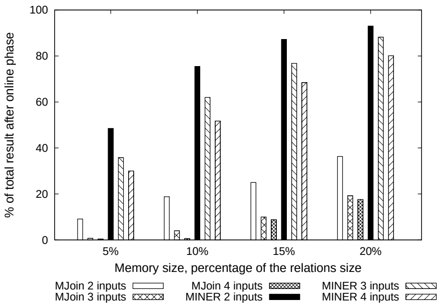

Fig. 22. SSTAR joins, MINER vs MJoin

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 Figure 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% 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% 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 Figure 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 is provided by more then two sources. We consider the following join: $R_1 \bowtie_{R_1.a_1=R_2.a_1} R_2 \bowtie_{R_2.a_2=R_3.a_2} R_3 \bowtie_{R_3.a_3=R_4.a_3} R_4$. 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_1) 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_2). 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 set up 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 multi-join operation remains the same, allowing us to draw conclusions on the performance of MINER.

Figure 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 indices. For a memory budget equal to 20% (i.e. $8.8MB$) of the input size and in a setup with four inputs, the memory occupied by the six indices is equal to $120KB$, which is less than 2% 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 Table 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 1000 tuples are gathered in the input buffer.

In Figure 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 datasets as in the previous experiment are used. The results in Figure 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 (i) 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 (ii) a novel re-entrant 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 book-keeping 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. We would like to thank the authors of [12] for making their code available to us. We would also like to thank the authors of [8] for providing access to the Stock data set.

References

[1] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *Proc. CIDR Conf.*, 2005.

[2] D. Baskins. Judy arrays. Available from http://judy.sourceforge.net, 2004.

[3] M. A. Bornea, V. Vassalos, Y. Kotidis, and A. Deligiannakis. Double Index Nested-loop Reactive Join for Result Rate Optimization. In *ICDE Conf.*, 2009.

[4] J. Dittrich, B. Seeger, and D. Taylor. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *Proceedings of VLDB*, 2002.

[5] P. J. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation. In *Proceedings of ACM SIGMOD*, 1999.

[6] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. In *PDIS*, 1991.

[7] Z. G. Ives, D. Florescu, and et al. An Adaptive Query Execution System for Data Integration. In *SIGMOD*, 1999.

[8] F. Li, C. Chang, G. Kollios, and A. Bestavros. Characterizing and Exploiting Reference Locality in Data Stream Applications. In *Proc. of ICDE*, 2006.

[9] M. F. Mokbel, M. Lu, and W. G. Aref. Hash-Merge Join: A Non-blocking Join Algorithm for Producing Fast and Early Join Results. In *ICDE Conf.*, 2004.

[10] M. Negri and G. Pelagatti. Join during merge: An Improved Sort Based Algorithm. *Inf. Process. Lett.*, 21(1), 1985.

[11] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, 2001.

[12] Y. Tao, M. L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis. RPJ: Producing Fast Join Results on Streams Through Rate-based Optimization. In *Proceedings of ACM SIGMOD Conference*, 2005.

[13] J. D. Ullman, H. Garcia-Molina, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.

[14] T. Urhan and M. J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Eng. Bull.*, 23(2), 2000.

[15] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 285–296. VLDB Endowment, 2003.

[16] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.

Mihaela A. Bornea is a fourth-year PhD student in the Department of Informatics at the Athens University of Economics and Business. She holds a B.Sc.and MSc degrees in Computer Science from the Politehnica University of Bucharest, Romania. In 2008 she was an intern at Microsoft Research, Cambridge, UK. Her research interests are in the areas of query processing, transaction processing in replicated databases and physical design tuning. She regularly serves as external reviewer for major international conferences related to data management.

Vasilis Vassalos is an Associate Professor in the Department of Informatics at the 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 2 US patents for work on information integration, and regularly serves on the program committees of the major conferences in databases.

Yannis Kotidis is an Assistant Professor in the Department of Informatics at the Athens University of Economics and Business. He holds a B.Sc. degree in Electrical Engineering and Computer Science from the National Technical University of Athens, an MSc and a Ph.D. in Computer Science from the University of Maryland (USA). 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 5 U.S patents. He has served on numerous organizing and program committees of international conferences related to data management.

Antonios Deligiannakis is an Assistant Professor of Computer Science at the Dept. of Electronic & Computer Engineering of the Technical Univ. of Crete, as of January 2008. He received the Diploma degree in Electrical and Computer Engineering from the National Technical Univ. of Athens in 1999, and the MSc and PhD degrees in Computer Science from the Univ. of Maryland in 2001 and 2005, respectively. He then performed his PostDoctoral research at the Dept. of Informatics and Telecommunications of the Univ. of Athens (2006-2007). Prof. Deligiannakis is also an adjunct researcher at the Digital Curation Unit (http://www.dcu.gr/) of the IMIS-Athena Research Centre.