International Journal of Innovative Research in Computer Science and Engineering (IJIRCSE)
www.ioirp.com ISSN: 2394-6364, Volume – 1, Issue – 9. August 2015

Effective Nested Loop Reactive Join Algorithm for Multiple Relations on Input Data

S. Manikandan
Assistant Professor & Head
Department of Information Technology
EGSPEC, Nagai, Tamilnadu, India
(E-mail:manikandan@egspec.org)

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 DINER (Double Index NEsted-loops Reactive join), 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. Multi Active Relational join Algorithm (MARA) is a multiway join operator that inherits its principles from DINER. Our experiments using real and synthetic data sets demonstrate that TARA 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, MARA manages to produce a high percentage of early results, outperforming existing techniques for adaptive multiway join.

Index Terms: Query processing, DINER, MARA, Join.

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. 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. 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, online aggregation, and approximate query answering systems. Compared to traditional join algorithms

(be they sort-, hash-, or nested-loop-based), 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 [1]. 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 [2].

In this work, I propose two new adaptive join algorithms for output rate maximization in data processing over autonomous distributed sources. The first algorithm, double index nested-loops reactive join (diner) is applicable for two inputs; while multiple index nested-loops reactive join (miner) can be used for joining an arbitrary number of input sources. I 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 [3]. I 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.

Data mining consists of five major elements: Extract, transform, and load transaction data onto the data warehouse system.

- Store and manage the data in a multidimensional database system.
- Provide data access to business analysts and information technology professionals.
- Analyze the data by application software.
- Present the data in a useful format, such as a graph or table.

II. SYSTEM ANALYSIS

Adaptive query processing (AQP) is a promising approach to avoid the performance penalty caused by optimizer mistakes, unknown statistics, and changes in data and system conditions. With AQP, the optimization and execution stages of processing a query are interleaved, possibly multiple times over the running time of the query. AQP makes query processing more robust to optimizer mistakes, unknown statistics, and to changes in conditions over the running time of a query [4]. AQP is part of the general trend toward automatic computing, which aims to minimize human effort in running systems efficiently[3][4].

A. DINER (Double Index NEsted-loops Reactive join)

DINER algorithm for computing the join result of two finite relations RA and RB, which may be stored at potentially different sites and are streamed to my local system. Given the unpredictable behavior of the network delays and random temporary suspensions in data transmission may be experienced. Whenever an in-memory tuple helps produce a join result, its join bit is set to ***1.***

	Symbols	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 region 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 Algorithms

DINER algorithm for computing the join result of two finite relations RA and RB, which may be stored at potentially different sites and are streamed to my local system. Given the unpredictable behaviour of the network delays and random temporary suspensions in data transmission may be experienced. Whenever an in-memory tuple helps produce a join result, its join bit is set to 1. 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.[4]

overflow. Moreover, in the spirit of prior work, 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 my 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.[11]

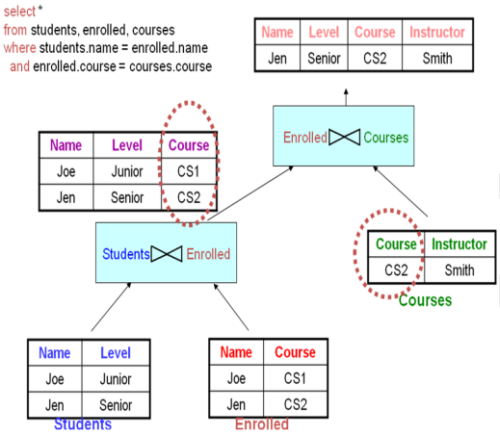

Figure. 1. Double Join: Structure

It is proposed Double Index NEsted- loops Reactive join (DINER) (Fig 1), a new adaptive 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. In this 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[7].

B. 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 ti is available, all matching tuples of the opposite relation that reside in main memory are located and used to generate result tuples as soon as the input data are available. When matching tuples are located, the join bits of those tuples are set, along with the join bit of the currently processed tuple (Line 9). Then, some statistics need to be updated (Line 11). This procedure will be described later in this section.

Algorithm 1. Arriving Phase

1: **while** R_A and R_B still have tuples **do**
2: **if** t_i 2 R_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* ≥ 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

When the MemThresh is exhausted, the flushing policy picks a Victim relation and memory-resident tuples from that relation are moved to disk in order to free memory space (Lines 13-15). The number of flushed tuples is chosen so as to fill a disk block. The flushing policy may also be invoked when new tuples arrive and need to be stored in the input buffer (Line 2). Since this part of memory is included in the budget (MemThresh) given to the DINER algorithm, I 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 Before Eventually, when both relations have been received in their entirety (Line 22), the Cleanup phase of the algorithm, discussed in previous chapters helps produce the remaining results.[11]

C. 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 $LastUpVal_i$ 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: $UpJoins_i$, $LwJoins_i$, $MdJoins_i \leftarrow 0$

D. 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 1 and 2. I 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 RA corresponds to block 1, the second to block 2, etc.). The notation used in the algorithm is available in Table 2.

Figure. 2 provide a helpful visualization of the progress of the algorithm. Its operation is based on the following points:

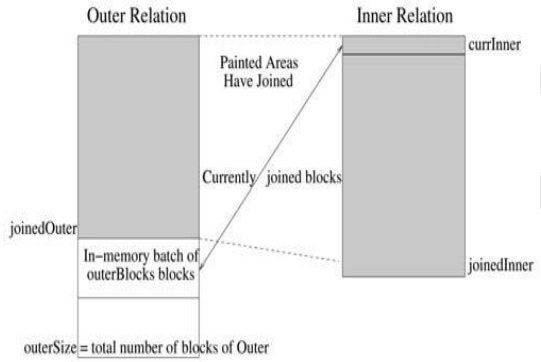

Figure. 2. Status of algorithm after swapping the roles of the two relations

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 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 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 Before 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 my novel adaptation of the traditional nested loops algorithm.

III. CHARACTERIZING AND EXPLOITING REFERENCE LOCALITY IN DATA STREAM APPLICATIONS

In this paper, we investigate a new approach to process queries in datastream applications. I show that reference locality characteristics of data streams could be exploited in the design of superior and flexible data stream query processing techniques. I identify two different causes of reference locality: popularity over long time scales and temporal correlation over shorter time scales An elegant mathematical model is shown to precisely quantify the degree of those sources of locality. Further, I analyze the impact of locality-awareness on achievable performance gains over traditional algorithms on applications such as MAX-subset approximate sliding window join and approximate count estimation. Ina comprehensive

experimental study, I compare several existing algorithms against my locality-aware algorithms over a number of real datasets. The results validate the usefulness and efficiency of my approach.

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 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. It also supports range queries; 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.

IV. CONCLUSION

In this work, I Introduce DINER, a new efficient 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 my technique. Through my experimental evaluation, I have demonstrated the advantages of this 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.

REFERENCES

[1]. Babu S and Bizarro P (2005) "Adaptive Query Processing in the Looking Glass," Proc. Conf. Innovative Data Systems Research (CIDR).
[2]. Li F, Chang C, Kollios G, and Bestavros A (2006) "Characterizing and Exploiting Reference Locality in Data Stream Applications," Proc. IEEE Int'l Conf. Data Eng. (ICDE),.
[3] Nandhini. R, Pavithra. P, Abinaya. P and S.Manikandan, "Information Technology Architectures for Grid Computing and Applications ", International Journal of Advanced Research Computer Engineering and Technology, ISSN:2278-1323, Vol.3, No.06, pp:2239-2242,June'2014.
[4] S. Manikandan and K. Manikanda Kumaran, "Identifying Semantic Identifying Semantic Relations Between Disease And Treatment Using Machine Learning Approach",International Journal of Engineering Research & Technology(IJERT), essn:2278-0181,Vol.2,June - 2013.
[5]. Dittrich J, Seeger B, and Taylor D (2002) "Progressive Merge Join: A Generic and Non-Blocking Sort-Based Join Algorithm," Proc. Int'l Conf. Very Large Data Bases (VLDB).
[6]. Haas P.J and Hellerstein J.M (1999) "Ripple Joins for Online Aggregation," Proc. ACM SIGMOD.
[7] Haripriya S, Indumathi , S.Manikandan, "Virtual Network Connection Using Mobile Phones", COMPUSOFT, An International Journal of Advanced Computer Technology, ISSN:2320-0790, Vol.03, Issue: 06, pp-980-984, June-2014.
[8]. Ives et al Z.G (1999) "An Adaptive Query Execution System for Data Integration," Proc. ACM SIGMOD.
[9]. Baskins Judy Arrays D (2004) http://judy.sourceforge.net.
[10]. Bornea M.A , Vassalos V, Kotidis Y, and Deligiannakis A, (2009) "Double Index NEsted-loops Reactive join for Result Rate Optimization," Proc. IEEE Int'l Conf. Data Eng. (ICDE).
[11]Manikandan.S, Manikanda Kumaran.K, Palanimurugan.S, Aravindan.S and Praveen Kumar.S, "Detecting and Preventing Distributed Denial of Service (DDOS) Attacks using BOTNET Monitoring System", International Journal of Engineering and Computer Science, ISSN:2319-7242, Vol.3,Issue.12,pp:9717-9720,December;2014.
[12]. Negri M. and Pelagatti G. (1985) "Join During Merge: An Improved Sort Based Algorithm," Information Processing Letters vol. 21, no. 1, pp. 11-16.
[13]. Suk-Ling Li, Kai-Chi Leug, L.M. Cheng, Chi-kwong Chan, performance Evalution of a Steganographic Method for Digital images Using Side Match, icicic 2006, IS16-004,Aug 2006.
[14]. J. Mielikainen, "LSB matching revisited," *IEEE Signal Process. Lett.*, vol. 13, no. 5, pp. 285–287, May 2006.
[15]. Westfeld and A. Pfitzmann, "Attacks on steganographic systems,"in *Proc. 3rd Int. Workshop on Information Hiding*, 1999, vol. 1768, pp.61–76.

Author Profile:

Manikandan.S is working as Assistant Professor and Head of Information Technology in EGS Pillay Engineering college, Nagapattinam. He completed M.E(CSE) with Honors and Distinction in Annamalai University in the year of 2012 and B.Tech (IT) with Distinction in EGS Pillay Engineering College in Nagapattinam in the year of 2010. Currently He is doing research in Pervasive Computing area his area of interest in Networking, Image Processing, Cloud Computing, Information Security.