Volume 2, Issue 4, April 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Optimization of Data Streaming Inputs Using Miner Algorithms

Ms. Pallavi Umap*
Student, Department of Information Technology,
Prof.Ram Meghe Institute of Technology Reasearch,
Anjangaon Bari Road, Badnera(Amravati), India.
E-mail Id: - ppt888@rediffmail.com

Dr.G.R.Bamnote
Professor & Head
Department of Computer Sci.& Engg,
Prof.Ram Meghe Institute of Technology & Research
Anjangaon Bari Road, Badnera (Amravati), India

Abstract **–It focuses on quality improvement on join. It takes shorter time for joining two or more fields and performance comparison between traditional and recent techniques. Also, evaluate the performance of DINER as well as MINER. 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. In traditional join techniques, 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 propose work, Evaluate the performance of Double Index Nested loops Reactive join (DINER), and Multiple Index Nested loop Reactive join(MINER) These both algorithms are a adaptive join algorithm for result rate maximization. DINER and MINER outperforms in comparison with the previous adaptive join algorithms in producing result tuples at a significantly higher rate, while making better use of the available memory.**

Keywords: **Query processing; join Streams; DINER and MINER.**

I. INTRODUCTION

Real systems rarely stored all their data in one large table. To do so would require maintaining several duplicate copies of the same values and could threaten the integrity of the data Instead, IT department everywhere almost always divide their data among several different tables. Because of this, a method is needed to simultaneously access two or more tables by using join operation. Here, main focus is to show how join operators work in databases

Joins are one of the basic constructions of SQL and Databases as they combine records from two or more databases tables into one row source, one set of rows with the same columns and these columns can originate from either of joined tables as well as be formed using expression or built in or user defined functions.

II. EXISTING WORK

There are three basic join algorithms: Hash based join, Sort Merge based join, Nested loop based join algorithm

A .Nested loop based join algorithm: In this, Nested loop join compares each row from one table (i.e. outer table) to each row from the other table (i.e. inner table) looking for rows that satisfy the join predicate. Inner join and outer join are the logical operations. The cost of this algorithm is proportional to the size of outer table multiplied by size of the inner table.

Pseudocode for algorithm:-

```
for each row R1 in the outer table
for each row R2 in the inner table
if R1 joins with R2
returns (R1,R2)
```

For Example, consider schema of two tables ‘Customers’and ‘Sales’.

Create First table ‘Customers’,

Create Table Customers(Cust_Id int, Cust_Name varchar(10))

Another table is ‘Sales’,

Create Table Sales (Cust_Id int,Item varchar(10))

Query is written as:-

Select * from Sales S inner join Customers C on S.Cust_Id = C.Cust_Id

***B.* Hash based join algorithm:** Hash joins [12] are used when the joining large tables. The optimizer uses smaller of the two tables to build a hash table in memory and the scans the large tables and compares the hash value (of rows from large table) with this hash table to find the joined rows.

The algorithm of hash join is divided in two parts:

1. Build: - In-memory hash table on smaller of the two tables.

2. Probe: - This hash table with hash value for each row second table

For Example:-

Consider schema of two tables Emp_Master and Emp_Info

Create Table Emp_Master (Id int, Name varchar (10), Designation varchar (10), Dept varchar (10))

Another table is,

Create Table Emp_Info (Id int, Dt_of_Joining Date Time)

Query is written as:-

Select Id int, Name varchar (10), Designation varchar (10), Dept varchar (10) From Emp_Master inner join Emp_Info on Emp_Master.Id = EmP_Info. Id Order by Emp_Info. Dt_of_Joining desc

***C.* Sort based join algorithm:** It is also known as sort merge join [3] algorithm. Sort merge join is used to join two independent data sources. They perform better than nested loop when the volume of data is big in tables but not as good as hash joins in general. They perform better than hash join when the join condition columns are already sorted or there is no sorting required.

Existing work on adaptive join algorithms can be classified in two groups:-hash based join and sort based join. Examples of hash based algorithms are XJoin, MJoin, Hash Merge join, and Progressive Merge join.

***D.* Double Pipelined Hash Join (DPHJ):** The double Pipelined Hash Join (DPHJ) [7] is another extension of the symmetric hash join algorithm. DPHJ has two stages. The first stage is similar to the in-memory join in the symmetric hash join and XJoin. In the second stage, pairs that are not joined together in the first phase are marked and are joined in disk. DPHJ is suitable for moderate size data, but does not scale well for large data sizes.

***E.* XJoin:** It is a non-blocking join operator, called XJoin[4] which has a small memory footprint, allowing many such operators to be active in parallel. XJoin is optimized to produce initial results quickly and can hide intermittent delays in data arrival by reactively scheduling background processing. We show that XJoin is an effective solution for providing fast query responses to user even in the presence of slow and bursty remote sources.

In previous work [9] of Xjoin, we identified three classes of delays that can affect the responsiveness of query processing: 1) initial delay [11], in which there is a longer than expected wait until the first tuple arrives from a remote source 2) slow delivery, in which data arrive at a fairly constant but slower than expected rate and 3) bursty arrival [10] ,in which data arrive in a fluctuating manner.

***F.* Hash Merge Join:_** HMJ [5] is a hybrid query processing algorithm combining ideas from XJoin and Progressive Merge Join. This introduces the hash-merge join algorithm (HMJ), for the join operator occasionally gets blocked. The HMJ algorithm has two phases: The hashing phase and the merging phase. The hashing phase employs an in-memory hash-based join algorithm that produces join results as quickly as data in data arrives. The merging phase is responsible for producing join results if the two sources are blocked.

***G.* MJoin:** The basic idea of the MJoin[8] algorithm is simple: generalize the symmetric binary hash join and the XJoin algorithms to work for more than two inputs. our primary goal is to maximize the output rate during the memory-to-memory phase of the MJoin. As with the binary XJoin, in MJoin, the disk to-memory phase is intended to allow the system to generate outputs while its inputs are blocked, while the disk to-disk phase is intended to generate any final answers after the inputs have terminated. Interestingly, for the MJoin, how we handle memory overflow determines the output rate of the memory-to-memory phase.

***H.* Progressive Merge Join:** PMJ [3] 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 arrives. The merging phase is responsible for producing join results if the two sources are blocked.

***I.* Rate based Progressive Join:** RPJ [6] 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.

In this work, we used RPJ (Rate-based Progressive Join), which continuously adapts its execution according to the data properties (e.g., their distribution, arrival pattern, etc.). RPJ utilizes a novel flushing algorithm which is optimal among all possible alternatives (based on the same statistics about data distributions, arrival patterns, etc.), and significantly enhances the efficiency of the memory-

memory stage. Furthermore, RPJ maximizes the output rate by invoking the memory-disk and disk-disk in a strategic order, i.e., the next stage selected for execution is the one expected to produce the highest output rate.

III PROPOSED WORK

***A.* DINER (Double Index Nested loop reactive join):** In proposed work, we will shown a new adaptive join algorithm for output rate maximization called Double Index Nested Loop Reactive Join (DINER)[2]. The important feature of this algorithm is, it is completely unblocking join techniques that support range join conditions. Range join queries are a very common class of joins in a variety of applications, traditional business data processing to financial analysis applications and spatial data processing. Progressive Merge Join [3] (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.

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. 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 . 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 that inner relation may change, as more blocks of that relation may be flushed to disk. To make it easier to keep track of joined blocks, we need to join each batch of OuterMem blocks of the outer relation with the same, fixed number of blocks of the inner relation – even if over time the total number of disk blocks of the inner relation increases. One of the key ideas of ReactiveNL is the following: at the first invocation of the algorithm, we record the number of blocks of the inner relation in JoinedInner. From then on, all successive batches of OuterMem blocks of the outer relation will only join with the first JoinedInner blocks of the inner relation, until all the available outer blocks are exhausted.

Point 3. When the outer relation is exhausted, there may be more than JoinedInner blocks of the inner relation on disk (those that arrived after the first round of the nested loop join, when DINER goes back to the Arriving phase). If that is the case, then these new blocks of the inner relation need to join with all the blocks of the outer relation. To achieve this with the minimum amount of book keeping, 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 thevalue 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.

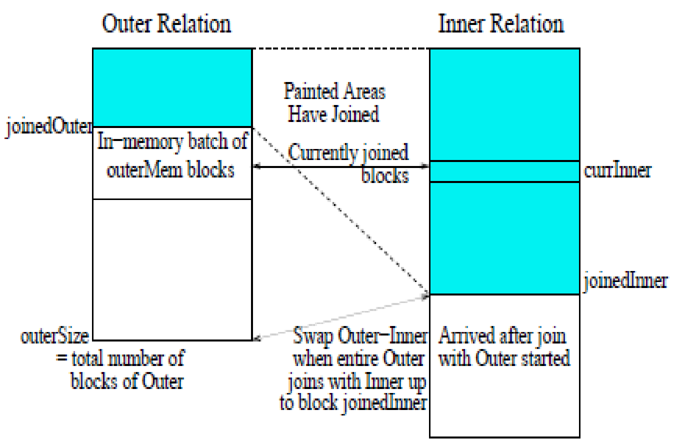
Outer Relation
Inner Relation
Painted Areas Have Joined
joinedOuter
In−memory batch of outerMem blocks
Currently joined blocks
currInner
joinedInner
outerSize = total number of blocks of Outer
Swap Outer−Inner when entire Outer joins with Inner up to block joinedInner
Arrived after join with Outer started

Fig. status of the algorithm during Reactive phase

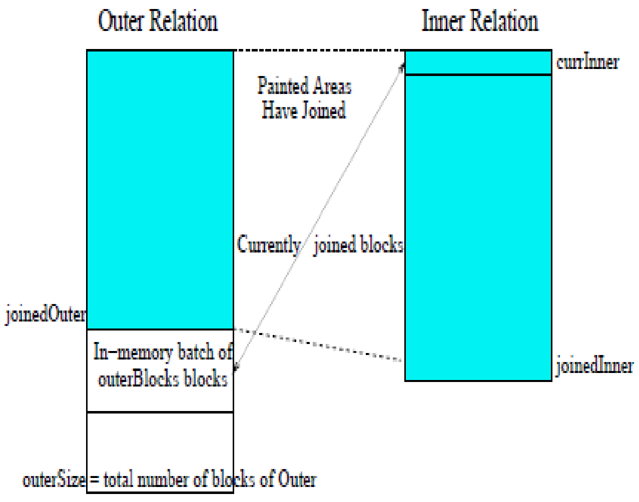
Outer Relation
Inner Relation
currInner
Painted Areas Have Joined
Currently joined blocks
joinedOuter
In−memory batch of outerBlocks blocks
joinedInner
outerSize = total number of blocks of Outer

Fig. status of the algorithm after swapping the roles of two relations

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, 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 the outer blocks with ids in the range

[JoinedOuter+1,JoinedOuter+OuterMem] and by joining them with inner block CurrInner.

Point 5. In earlier work, 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 join between tuples touter and tinner that have already joined during the Arriving phase. This is achieved by proper use of the ATS and DTS time stamps[4]. If the time intervals [touter.ATS, touter.DTS] and [tinner.ATS, tinner.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 ReactiveNL algorithm.

B. Difference between DINER and Existing algorithm:

1. DINER supports equi-joins and range queries.PMJ also supports range queries but it has some limitation due to its poor blocking behavior.

2. DINER will introduce flushing policy, is used to create and maintained three overlapping value regions.

3. DINER will introduce a more responsive reactive phase that allows the algorithm to quickly move into processing tuples when both data sources block**.**

4. In Leaner Algorithm, DINER improves its relative performance compared to the existing algorithm in terms of produced tuples during online phase in more constrained memory environment.

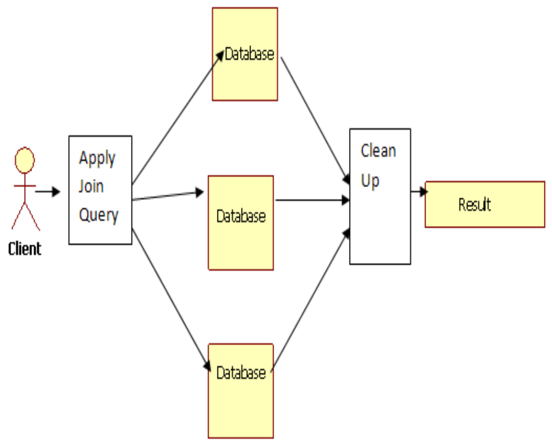

Fig. use case diagram for system

C. MINER (Multiple Index Nested loop reactive join):

MINER [1] extends DINER to multiway joins and it maintains all the distinctive and efficiency generating properties of DINER. MINER maximizes the output rate by: 1) adopting an efficient probing sequence for new incoming tuples which aims to reduce the processing overhead by interrupting index lookups early for those tuples that do not participate in the overall result; 2) applying an effective flushing policy that keeps in memory the tuples that produce results, in a manner similar to DINER; and 3) activating a Reactive phase when all inputs will 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, additional challenges in MINER namely: 1) updating and synchronizing the statistics for each join attribute during the online phase 2) more complicated book keeping in order being 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.

IV.CONCLUSION

In proposed work, the comparison between existing algorithm and DINER with their superiority will be shown. Its advantages are 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 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. It will 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 experimental evaluation, it will be demonstrate 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.

REFERENCES:

[1] Mihaela A.Bornea, Vasilis Vassalos, Yannis Kotidis, Antonios Deligiannakis: *Adaptive Join Operators for Result Rate Optimization on Streaming Inputs. IEEE Trans. Knowl. Data Eng. 22(8): 1110-1125 (2010)*

[2] M. A. Bornea, V. Vassalos, Y. Kotidis, and A. Deligiannakis. *DoubleIndex Nested-loop Reactive Join for Result Rate Optimization. In ICDEConf., 2009.*

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

[4] T.Urhan and M.J.Franklin.Xjoin*: A Relatively scheduled pipilined join operator.IEEE Data Eng.Bull,23920,2000*

[5] 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.nal conference on very large databases 2003.*

[6] 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.*

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

[8] S.D Viglas,J.F.Naughton and J.Burger.*Maximizing the output rate of multiway join queries over streaming information sources.In VLDB 2003: proceeding of the 29th international*

[9] L. Amsaleg, M. J. .Franklin, A. Tomasic, and T. Urhan. *Scrambling Query Plans to Cope With Unexpected Delays. PDIS Conf., Miami, USA, 1996*

[10] L. Amsaleg, M. J. .Franklin, and A. Tomasic. *Dynamic Query Operator Scheduling for Wide-Area Remote Access. Journal of Distributed and Parallel Databases, Vol. 6, No. 3, July 1998.*

[11] T. Urhan, M. J. .Franklin, and L. Amsaleg. *Cost Based Query Scrambling for Initial Delays. ACM SIGMOD Conf., Seattle, WA, 1998.*

[12] A. N. Wilschut and P. M. G. Apers. *Dataflow Query Execution in a Parallel Main-Memory Environment. In Proceedings of the First International Conference on Parallel and Distributed Information Systems, PDIS, pages 68–77,Miami, Florida, Dec. 1991.*

[13] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer. *On Producing Join Results Early. In Proceedings of the ACM Symposium on Principles of Database Systems, PODS, pages 134–142, San Diego, CA, June 2003.*

[14] P. J. Haas and J. M. Hellerstein. *Ripple Joins for Online Aggregation. In Proceedings of the ACM International Conference on Management of Data, SIGMOD, pages 287–298, Philadelphia, PA, June 1999.*