Volume 3, Issue 7, July 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Memory Flushing in Multijoin Query for Query Optimization

[1]Taruna Khemwani
Mtech(IT) Govt Engg College Ajmer(raj.), India

[2]Rakesh Rathi
Department of CS and IT, Govt Engg College Ajmer (raj.), India

[3]Vinesh Jain
Department of CS and IT, Govt Engg College Ajmer (raj.), India

Abstract: The enhancement of internet technology and advances in heterogeneous databases produced the requirement of database style querying over data from sources distributed around the world. When executing an online join query space is required in main memory to accommodate the data required to process the query. Memory of a specified amount is allotted to the query plan. When the main memory becomes full some data is required to move to disk to make room for new incoming tuples. Moving data from memory to disk is called Flushing. An optimal flushing strategy should be used to select the victim data so that the data which will least contribute in the future results would be flushed. All the previous algorithms are focused on flushing the memory when it becomes full but in this case some incoming data may be lost during the duration of flushing. In this paper we will propose a novel algorithm EarlyFlush to flush the memory before it becomes full to prevent the data loss and increase the throughput.

Keywords: Memory-Flushing, Database, Query processing, Join, Partition

I. Introduction

In online environment data comes from slow and busty networks .To produce result of join queries in online environment the tuples are received from the online sources and on the basis of join attribute att^{join} the result is produced. In online environment the production of result is required to be fast. Fast and efficient algorithms are required for the same. The data is received from continuous data sources in a memory of limited size. When this memory becomes full it is required to free some memory to receive new data or it will be lost otherwise. One solution to address the problem of memory shortage at run time is to move some data from memory to disk on the basis of some predefined criterion. Moving data from memory to disk is called memory Flushing. This solution have been discussed in [3],[4],[5],[6],[7],[9]. Various Flushing algorithms are used to select a victim and move it to the disk. These algorithms have two considerations (1) select the data (victim) to be flushed and (2) to decide how much memory is to be flushed. Most of the join algorithms requires that all input data is available beforehand. All these join algorithms are optimized to produce the entire query result. The shortcoming of these algorithms is that these are not appropriate for the applications and environments that require results as soon as possible. Such environments require a new non-blocking join algorithm. The symmetric hash join is the most widely used non-blocking join algorithm for producing early join results. But, it was designed for cases where all input data fits in memory. With the massive explosion of data sizes, several research attempts have aimed to extend the symmetric hash join to support disk-resident data. Such algorithms can be classified into three categories:

1. Algorithms that are based on hashing, which flushes in-memory hash buckets either in groups or individually.
2. Algorithms that are based on sorting. For these the in-memory data is sorted before its flushing to disk.
3. Algorithms that are based on Nested loop. In these a variant of the traditional nested-loop algorithm is employed.

Many other techniques have been suggested to enhance the working of these algorithms for other operators, e.g. MJoin[2] extends XJoin[3] for multi-way join operators while hash based joins have been extended for spatial join operators. But these join algorithms uses optimization techniques that considers only the problems of a single join operator, with no applicable extension for multi-join query plans. In terms of memory flushing algorithms, previous techniques can be classified to two categories:

1) Flushing a single hash bucket. Examples of this category include XJoin [3] that aims to flush the largest memory hash bucket and RPJ[6] that evicts a hash bucket based on an extensive probabilistic analysis of input rates.
2) Flushing a pair of corresponding buckets from both data sources. Examples of this category include hash-merge join [4] that keeps the memory balanced between the input sources and state spilling [5] that attempts to maximize the overall early result throughput of the query plan.

II. Related Work

In the category of non-blocking join algorithm symmetric hash join[16] is the most widely used algorithm for producing early join results. However, it takes into account only the cases where all input data fits in memory. As the data sizes are growing fast several algorithms have been proposed and implemented to extend the capability of symmetric hash join. These algorithms also support disk-resident data. Such algorithms can be classified into three categories:

1. Hash-based algorithms [3], [4],[6] that flush in-memory hash buckets to disk either individually or in groups
2. Sort-based algorithms [10] in which in-memory data is sorted before being flushed to disk, and
3. Nested- loop-based algorithms in which a variant of the traditional nested-loop algorithm is employed.

Also, several methods have been proposed to extend these algorithms for other operators, e.g .,MJoin [2] extends XJoin [3] for multi-way join operators while hash- based joins have been extended for spatial join operators[12].However ,these join algorithms employ optimization techniques that focus only on the case of a single join operator ,with no applicable extension for multi-join query plans.

In terms of memory flushing algorithms, previous techniques can be classified to two categories:

1) Flushing a single hash bucket. Examples of this category include XJoin [3] that aims to flush the largest memory hash bucket regardless fits input source and RPJ[6] that evicts a hash bucket based on an extensive probabilistic analysis of input rates.
2) Flushing a pair of corresponding buckets from both data sources. Examples of this category include hash-merge join[4] that keeps the memory balanced between the input source and state spilling[5] that attempts to maximize overall result throughput of the query plan.

Adaptive Global Flush method belongs to the latter category, as flushing a pair of buckets eliminates the need for extensive timestamp collection. Timestamps are necessary when flushing a single bucket to avoid duplicate join results.

III. **Overview**

Here we propose a new algorithm EarlyFlush() which will remove

1. The problem of data loss during the Flushing process and
2. Considers interdependency among operators to design a new flushing policy.

This algorithm keep a continuous count on memory usage or this purpose it uses a counter variable ‘Count’, Initially set to zero. Whenever a tuple is received at any input source the counter is increased by the size of the tuple. The algorithm continuously keep watching the value of ‘counter’. When the value of this counter reaches to some threshold value then the flush algorithm will be called to flush some memory(i.e.10% or 15%). We are calling the flush algorithm when memory consumption reaches upto the threshold value. it will omit the risk of loosing data during memory flush operation .

Our Flushing policy takes into account following properties to assign a weight to each partition group

1) Data arrival rate 2)Interdependency among operators 3)Participation in results

Some statistics are collected over the period of query runtime as described in table 1. We have collected two type of statistics

1. **Instantaneous** collected at a single instance during query run time.
2. **Periodic** collected over a small period of time.

The Flush(K) algorithm uses these statistics to assign a weight to eacs of statisticsh partition group. Whenever the flush algorithm is called it keep flushing the Partition group(s) with least score, to the disk until the memory of amount K get freed.

ARCHITECTURE ASSUMPTIONS. We assume the in-memory join employs the symmetric hash join algorithm to produce join results[15]. Figure2(a) gives the main idea of the symmetric hash join. Each input source(X and Y)maintains a hash table with hash function h and n buckets. Once a tuple r arrives from input X, its hash value h(r) is used to probe the hash table of Y and produce join results. Then, r is stored in the hash bucket $h(r_X)$ of source X. A similar scenario occurs when a tuple arrives at source Y.

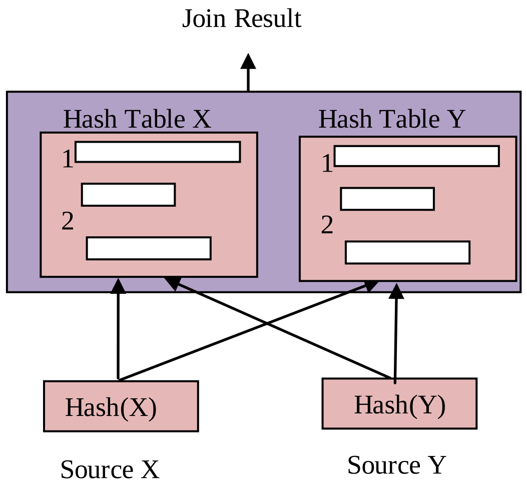

Fig. 1 Symmetric hash join

Table I Statistics

Type	**Statistics**	**Definition**
Instantaneous	P_Size_{Xi}	Size of hash partition i for input X
	G_Size_i	No. of tuples in partition group i
	$tSize_X$	Tuple size for input X
Periodic	In_{tot}	Total input count
	In_{Xi}	Input count at partition i for input X
	R_Imm_i	Local output of partition group i
	R_Final_i	Global output of partition group i

IV. **EarlyFlush Algorithm**

Algorithm1 EarlyFlush()

```
1. Function EarlyFlush(K)
/* Checking Memory Utilization*/
2. Int Count;
3. Whenever a tuple is received at a source S
4. Set Count=Count + $tSize_S$;
5.If(Count==$mem_{th}$) then
7. goto step 9
8. Else goto step 3
9. **For all** operator O in the query plan do
10.X←Source 1 of O Y←Source 2 of O  F_Size=0;
11.        **For all** partition groups i ∈ O do
                   /* New Data Approximation*/
12.                $N\_In_{Xi}$= M. $In_{Xi}/In_{tot}$;
13.                $N\_In_{Yi}$= M. $In_{Yi}/In_{tot}$;
                   /* Expected Immediate Results Approximation*/
14.                $E\_imm\_reslt_{Xj}$= ($P\_Size_{Xi}$. $N\_In_{Yi}$);
15.                $E\_Imm\_reslt_{Yj}$= ($P\_Size_{Yi}$. $N\_In_{Xi}$);
16.                $E\_Imm\_reslt_{NEWj}$= ($N\_In_{Xj}$.$N\_In_{Yi}$);
17.                $E\_Imm\_reslt_j$= $E\_Imm\_reslt_{Xj}$+ $E\_Imm\_reslt_{Yj}$+ $E\_Imm\_reslt_{NEWj}$
                   /* Expected Final Results Approximation*/
18.                $E\_Fin\_reslt_j$= $E\_Imm\_reslt_j(R\_Final_i/R\_Imm_i)$;
                   /* Assigning Weight To Partition Group*/
19.                G_Weight= $E\_fin\_reslt_j/G\_Size_i$
20.        **End For**
21. **End For**
    /* Partition Group Flushing*/
22. **While** F_Size<M
23.        G=partition group from input with minimum G_Weight
24.        Flush G to disk
25.        F_Size+=G_Size;
26. **End while**
```

The algorithm works according to following sequence of steps

1. **Checking Memory Utilization** the Earlyflush Algorithm continuously keeps on checking the memory utilization. For this purpose it uses a counter variable. Whenever a tuple is received from any input source, the counter is increased by the tuple size for that input. We continuously check the value of counter variable. When the counter reaches up to the memory threshold the algorithm switches to step 2.
2. **New Data Approximation** For each operator O we calculate the approximate number of incoming tuples in each partition group until next flushing takes place.
3. **Expected Immediate Result Approximation** For each operator O we calculate the number of immediate results produced by each partition group until next flushing takes place.
4. **Expected Final Result Approximation** this step estimates the contribution of each partition group in producing the final results from the join operations..
5. **Assigning Weight To Partition Group** now the algorithm assigns a weight to each partition group on the basis of its participation in the final results per tuple. The partition group which is participating in the final results more will

be assigned a better weight.

6. **Partition Group Flushing** this is the final step which actually flushes the partition group(s) to the disk with least weight until memory of amount 'M' get freed.

V. EXPERIMENTAL RESULTS

In our experiments we consider the query plan of fig 3 having three inputs X,Y and Z and two join operators O1 and O2. X Y and Z produces continuous data. The statistics are collected in every 5 seconds. Fig 2 shows the join results for different flush size

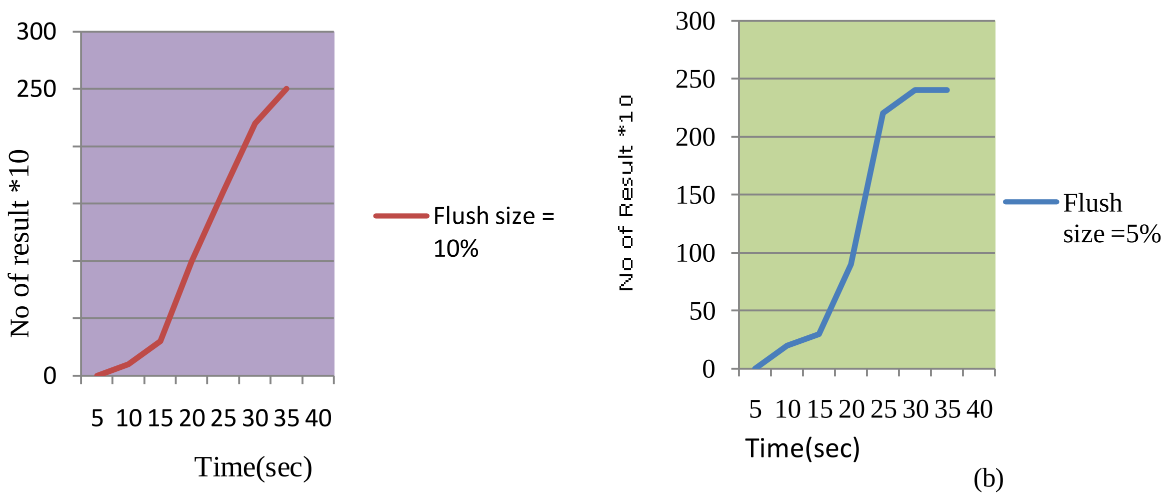

Fig 2 Join results with (a) flush size 10% of total memory (b) flush size 5% of total memory

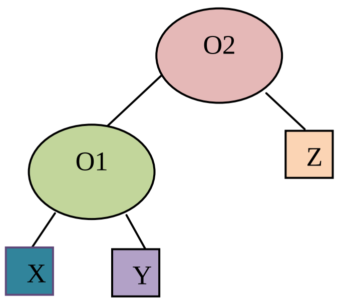

Fig 3 Query plan 1

VI Conclusion

This paper introduces a new algorithm 'Earlyflush' which removes the risk of loosing new incoming tuples in the duration of flushing in the processing of multijoin queries and considers interdependency among operators in the flushing policy. It would also increase the throughput by producing the result using these new tuples. Figure 2 shows the expected results of a query using EarlyFlush algorithm.

Acknowledgement

Authors are thankful to all who supported directly or indirectly.

References

[1] Abraham Silberschatz Yale University Henry F. Korth Lehigh University S. Sudarshan Indian Institute of Technology, Bombay TM "*Database System Concepts " Sixth Edition.*

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

[3] T. Urhan and M. J. Franklin, "*XJoin: A Reactively-Scheduled Pipelined Join Operator",*IEEE Data Engineering Bulletin,vol. 23, no. 2, pp. 27–33, 2000.

[4] 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, 2004.

[5] B. Liu, Y. Zhu, and E. A. Rundensteiner, "Run-time operator state spilling for memory intensive long-running queries," inSIGMOD, 2006.

[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 SIGMOD, 2005.

[7] G. Graefe, "*Query Evaluation Techniques for Large Databases*", ACM Computing Surveys, vol. 25, no. 2, pp. 73–170, 1993.

[8] J. J. Levandoski, M. E. Khalefa, and M. F. Mokbel, "PermJoin:An Efficient Algorithm for Producing Early Results in Multijoin Query Plans", in ICDE, 2008.

[9] Levandoski Et Al."*On Producing High And Early Result Throughput In Multi Join Query Plans"*, IEEE Transaction Of Knowledge And Data Engineering, Vol. 23, No. 12, December 2011

[10] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer, "*On producing join results early,*" in *PODS*, 2003.

[11] J. Kang, J. F. Naughton, and S. Viglas, "*Evaluating Window Joins over Unbounded Streams*," in *ICDE,* 2003.

[12] U. Srivastava and J. Widom, "Memory-Limited Execution of Windowed Stream Joins," in VLDB, 2004.

[13] J. Xie, J. Yang, and Y. Chen, "On Joining and Caching Stochastic Streams," in SIGMOD, 2005.

[14] Taruna Khemwani, Pranjal Bansal,Rakesh Rathi and Vinesh Jain, "*Review:Joining and Flushing techniques in join query environment,"* in ICRITO 2013

[15] A.N.WilschutandP.M.G.Apers,"DataflowQueryExecutioninaParallelMain-MemoryEnvironment,"in*PDIS*,1991.