http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 5 (1): 159-165, 2006
ISSN 1812-5638

Non-blocking Disk-tape Join Algorithm for Data on Tertiary Storage Systems

Baoliang Liu and Jianzhong Li
School of Computer Science and Technology, Harbin Institute of Technology, China

Abstract: The Non-blocking Disk-tape join (NDT) presented in this study was the first disk-tape join algorithm designed with the goal of producing join results as early as possible. It has three phases: the hashing phase, the merging phase and the probing phase. Join results can be produced in each phase. Tuples of disk resident relation and tape resident relation was read simultaneously into memory and be joined in the hashing phase. The merging phase joins those tuples that flushed onto disk during the hashing phase. After the first two phases, disk resident relation has been partitioned and then joined with remaining tape resident relation in the probing phase. Experimental results showed that NDT can produce join results much earlier than the-state-of-art CDT-GH and the performance of NDT was about the same with that of CDT-GH.

Key words: Non-blocking disk tape, algorithm, tertiary storage system, CDT-NB, CDT-GH, NDT

INTRODUCTION

New applications that need to manage massive data have emerged in business environments as well as in scientific areas. For example, the data warehouse system of China Mobile at Heilongjiang province has to deal with the data amount of several terabytes (TB) a year. All the historical phone call records have to be saved in case of further data analysis requirements, which has to resort to tertiary storage like tape libraries to accommodate such a vast amount of data. In scientific area the data amount generated by sensors are even larger. For example, the data amount returned by the Earth Observation System[1] one day was more than 1TB and it was required that the data should be saved for at least 15 years! In order to make full use of these gold mine, tools of data analysis and data mining need to be developed. In these applications, it is often required that a tertiary resident relation to be joined with a disk resident relation. For example, in data warehouse system, fact table consists of many detailed records whose data amount is too large to fit in disk, but the dimension table, whose data amount isn't too large, can be stored on disk. Such join operations will take a long time to complete. It is desirable to deliver at least a few result tuples as fast as possible. The reason may be that all results need further processing, which should start as early as possible. This behavior is highly desired for operations such as online aggregation, or when a quick visual inspection of first results is desired, in order to decide whether the join operation should continue or be aborted. The purpose of this research was to study disk-tape join method that can produce join results as early and fast as possible.

Many researchers have been done to study disk-tape join methods[2-4]. The joins presented in these works could be classified into two kinds: nested loop joins and hash-based joins. The emphases of these works were to parallelize the disk I/O and tape I/O to reduce the total I/O cost which determined the join performance. The first work that studied disk-tape join methods is presented by Myllymaki and Livny[2], its main purpose was to study the resource (main memory, disk) requirements of different disk-tape join methods. In Myllymaki and Livny[3] the methods of migrating disk join into tertiary join were studied. And the performances of different tertiary join were compared. It was showed that CDT-GH[3] is the candidate to join disk resident relation with tape resident relation. So we only compare NDT with CDT-GH in our experiments. Kraiss *et al.*[4] proposed a detailed cost model for nested block join under resource competition and he also studied how to improve the performance of nested block join methods. All the join algorithms presented above are all blocking algorithms that users need to wait quite a long time to see the first result. Because in these algorithms disk resident relation has to be loaded into memory, hashed and written back onto disk (this process is also called partitioning phase in some papers). No join results can be delivered until partitioning phase is over.

This study presents NDT which is a non-blocking join algorithm that join results can be produced as well as partitioning the disk resident relation. NDT is designed with two goals in mind: (1) Join result should be produced as early as possible. (2) The join performance shouldn't deteriorate much than the state-of-the-art disk tape join algorithms like CDT-GH. The execution of NDT can be

Corresponding Author: Baoliang Liu, School of Computer Science and Technology, Harbin Institute of Technology, China
Tel: 86 21 50270900/215

divided into three phases: the hashing phase, the merging phase and the probing phase. The hashing phase is similar with the partitioning phase of CDT-GH but differ in that tuples of both R and S are read into in-memory hash buckets simultaneously based on their hash values. Then tuples in the corresponding buckets are joined together and the early join results are output. Once the memory gets filled, some bucket pair of relation R and S are chosen to be flushed onto disk. After relation R is finished hashing, the merging phase takes control and tuples that are previously flushed to disk are joined together. The probing phase is similar with that of CDT-GH since relation R has been partitioned. Chunks of relation S are loaded into remaining disk buffer space, hashed and joined with the corresponding partition of R. The probing phase is iterated until relation S is exhausted.

RELATED WORK

Disk-tape Join methods: Disk-tape joins can be classified into two kinds: nested loop joins[2,3] and hash-based joins[2,3]. The two kinds of disk-tape joins are described below:

Disk-tape Nested Loop join (DT-NB)[3] is simple but not very efficient. It is the first join that studies the join between disk resident relation and tape resident relation. It reads a chunk of S into buffers and joins it with R. For each chunk of relation S, R needs to be scanned once. CDT-NB[3] is the concurrent version of DT-NB, in which double buffering technology[5] is used to parallelize disk I/O and tape I/O. When chunk of S is read from tape into one buffer, chunk of S in another buffer which was read in the previous cycle is simultaneously joined with R. Then the two buffers are switched their roles. Memory or disk can both be used as buffers. Technologies are proposed that one physical buffer is used to implement two logical buffers[5]. According to the buffers used, CDT-NB has the memory buffer version (CDT-NB/MB) and the disk buffer version (CDT-NB/DB).

Disk-tape Hybrid Hash join (DT-HH)[2] is a modification of classic Hybrid Hash Join[6]. It operates exactly as the classic one except that Phase I where both relations are hash partitioned on disk is modified to read relation S from tapes. Disk-tape Grace Hash join (DT-GH)[3] is similar with Grace Hash join [6] except that S are read from tapes. CDT-GH[3] is the concurrent version of DT-GH. CDT-GH is the candidate to join disk resident relation and tape resident relation[3]. So we only compare NDT with CDT-GH in the present experiments. We briefly describe the working procedure of CDT-GH below:

- In Step 1, relation R is partitioned and it is required that each partition can be loaded into memory in full.
- Step 2 is iterated until S is exhausted. In each iteration i ($i \geq 0$), $|S_i| = |D| - |R|$ blocks of S data are read from tape, hashed and written into disk partitions (S is partitioned by the same method with partitioning R). A join process then reads each partition of R into memory and joins it with the corresponding partition of S. Simultaneously a hash process reads data from S and produces the hash partitions needed in the next iteration.

Non-blocking Join methods: The blocking behavior has been noticed by the database research community recently and several disk non-blocking join methods have been proposed. Examples are Symmetric Hash Join (SHJ)[7], XJoin[8], Hash Merge Join (HMJ) [9] and Progressive Merge Join (PMJ)[10].

SHJ is the first non-blocking join method. In SHJ, two hash tables are built simultaneously in memory. When a tuple arrives, it firstly inserted into its corresponding hash table and then probes against the other. SHJ requires that both relations can fit in memory at the same time. XJoin is multi-thread extension of SHJ. In XJoin, when memory gets filled, the largest bucket of the two hash table is flushed onto disk. The flushed buckets are then joined in the last phase of XJoin. HMJ consists of hashing phase and merging phase. In the hashing phase, tuples of the two data resources are joined in memory which is similar with SHJ. Once the memory becomes full, a bucket pair with the same hash value is flushed onto disk. In the merging phase, buckets that previously flushed are joined. The hashing phase and the merging phase can execute alternatively. The merging phase is only executed when the two data sources get blocked or the hashing phase is over. PMJ is the non-blocking version of traditional sort-merge join. One chunk of R and one chunk of S are read simultaneously into memory, sorted and merged. Then they are written back onto disk. Finally the sorted chunks on disk are joined. Care must be taken that no duplicated results are produced.

COST MODEL

In this study we employed a transfer-only I/O cost model[11,12], that is to say, I/O cost is the main factor that determines the performance of join algorithms and the CPU processing time is ignored. In tertiary storage systems, we consider not only disk I/O but also tape I/O. Note that disk I/O may be overlapped with tape I/O because the data loading from disk and tape drives can be parallelized.

We assume that all disk accesses are multi-page I/O requests. The cost of a disk access is therefore derived by the amount of the data transferred. The seek cost and rotational latency are ignored. As shown by Hagman[12],

Table 1: Cost model parameters

Parameter	Description
R	Disk resident relation
S	Tape resident relation
|R|	Number of blocks in relation R
T	First chunk of relation S
$S_i(i \geq 1)$	Chunks of relation S except the first chunk
X_D	Data transfer rate of disk
X_T	Data transfer rate of tape

disk seeks and rotational latency play a relatively minor role compared to the transfer cost when disk requests are at least moderately large. NDT scans relation S sequentially and doesn't randomly access data until it is migrated onto disk. So the tape positioning time is ignored too. Typically the analyses of join algorithms[11,12] do not take into account the output cost of join results because they are deemed the same for all join algorithms. It is also ignored in this study. Table 1 shows the cost model parameters.

NON-BLOCKING DISK-TAPE JOIN

The basic idea behind NDT is to read tape resident relation S as well as we partition R and join them as early as possible. NDT has three phases: the hashing phase, the merging phase and the probing phase. The hashing phase is first executed and after the hashing phase is finished, the merging phase and the probing phase are executed concurrently to parallelize disk I/O and tape I/O. Join results can be produced in each phase. To describe the algorithms clearly, we refer to the first chunk of S that is loaded onto disk from tape as T and the following chunk of S as $S_i(i \geq 1)$. We describe each phase in detail below.

HASHING PHASE

In the hashing phase, tuples of R and T are read from disk and tape simultaneously, hashed into their in-memory hash table, respectively. The two hash tables are built with the same hash function, hence they have the same number of buckets, denoted by R_i and T_i (i = 1, …, N), respectively. Tuples in R_i and T_i have the same hash value. Once a new tuple t arrives, it is first hashed to its destination buckets, take R_i for example. Then the corresponding bucket of T_i is probed. A join result will be output if there is a match. If the memory becomes full, a bucket pair (R_i, T_i) will be chosen according to some flush policy. The tuples in the two buckets are sorted and then be written onto disk. In each hash table, the tuples in the flushed bucket is only part of all the tuples with the same hash value. The flushed bucket is called bucket run and all the tuples of R (similar for S) with the same hash value are called a partition of R (S) in the following to distinguish them clearly.

There are four flush policies proposed in[9]. The flush policy has much influence with the efficiency of HMJ, but it has little influence to NDT because the cost of the probing phase determines the overall join performance in NDT. In this experiment we choose Flush Largest Policy[9] which means that the largest bucket pair in memory are chosen to be flushed when memory get filled.

The hashing phase is ended when R is partitioned. All the tuples in memory are sorted and written into the last bucket runs of the corresponding partitions. Note in the hashing phase the hash function is chosen so that each partition of relation R can be loaded into memory in full, because these partitions are needed to join with remaining chunks of S in the probing phase.

The details of hashing phase are described below:

```
Procedure HashingPhase
Input: disk resident relation R
       first chunk of tape resident relation S, denoted by T
Output: part of the join results
        Sorted bucket mns of R and T
Begin
1.   WHILE not end R DO
2.        read tuple from R and T concurrently
3.        IF there is no enough memory to accommodate tuple h THEN
4.             The flushing policy chooses a bucket pair (Rk, Tk).
5.             Rk and Tk are sorted respectively.
6.             Flush buckets Rk and Tk onto disk.
7.        Dispatch h to its destination buckets Rh (or Th).
8.        Join h with all tuples in corresponding bucket Th (or Rh).
9.   Flush the memory tuples to the last corresponding bucket runs
End
```

The cost of hashing phase is obviously $(2|R|+|T|)/X_D$, because in the hashing phase, relation R need to be read into memory once, hashed and then written back onto disk. T needs to be written onto disk from memory. Note that the size of T is chosen so that the cost of reading T from tape into memory is overlapped with reading R from disk.

MERGING PHASE

The merging phase deals with those bucket runs that are previously flushed onto disk during the hashing phase. For each hash bucket with hash value h, there are m_h bucket runs for relation R_h and T_h.

In the merging phase, we allocate one block of memory for buffer space for each bucket run of partition R_h and T_h. Then we merge all the runs of R_h and concurrently merge all the runs of T_h. As tuples of R_h and T_h are generated in sorted order by these merges, they can be checked for a match. When a tuple from R_h matches one from T_h and the two tuples didn't be flushed onto disk together, output the pair. This procedure is iterated for all the partitions of R and T.

The details of merging phase are described below:

```
Procedure MergingPhase
Input: Sorted bucket runs of R and T
Output: Part of join results
Begin
1.   FOR i = 1 to N DO
2.        tr = first tuple of Multi-way merge bucket mns of Ri.
3.        ts = first tuple of Multi-way merge bucket runs of Ti.
4.        WHILE not end R AND not end T DO
5.             IF tr match ts AND they are not be flushed together
6.             THEN output the join result
7.             ELSE IF tr < ts THEN
8.                  tr = next tuple of Multi-way merge bucket runs of Ri.
9.             ELSE ts = next tuple of Multi-way merge bucket runs of Ti.
End
```

The cost of merging phase is obviously $(|R|+|T|)/X_D$.

PROBING PHASE

The probing phase is iterated until S is exhausted. In each iteration I($i \geq 1$), $|S_i| = |D| - |R|$ blocks of S data are read from tape, hashed and written into partitions which are on the disk. A join process then reads each partition of R into memory and joins it with the corresponding partition of S. Note that a hash process can simultaneously read more data from S and produce the hash partitions needed in the next iteration.

The details of probing phase are described below:

```
Procedure ProbingPhase
Input: Partitioned R
     Remaining tape relation S
Output: Join results
Begin
1.   k = 0, i = 1
2.   copy Si from tape and hash it onto disk buffer Ik
3.   WHILE not empty Ik DO
4.        start copying Si+1 from tape and hash it into disk buffer I1-k
5.        WHILE not end of R DO
6.             Load partition Rj into memory
7.             Load partition Sij from disk into memory and compute Sij⋈Rj
8.             j = j+1
9.        wait until Si+1 copied to I1-k
10.  i = i+1, k = 1-k
End
```

Same physical disk space is used to implement two logical buffers[5]. The cost of reading remaining S from tape onto disk is $(|S| - |T|)/X_T$ and the data need to be read into memory from disk again, the cost of which is $(|S|-|T|)/X_T$. The cost of reading partitioned R from tape into memory is $|R|/X_D$. And the partitioned R needs to be read into memory for $\lceil |S - T| / |D - R| \rceil$ times. Partitioned R and remaining S can be read simultaneously, so the cost of probing phase is:

$$\max \{|S-T|/X_T, \lceil |S-T| / |D - R| \rceil \cdot |R| / X_D + 2|S - T| / X_D \}$$

CORRECTNESS OF NDT

Here, we show that NDT is correct in that all the join result can be produced by NDT and no duplicate result can be output either.

Theorem 1: Let R be a disk resident relation and S be a tape resident relation, then NDT returns all the results of $R \bowtie S$.

Proof: Assume that $r \in R$ and $s \in S$ and (r, s) satisfies the join condition but they are not outputted as join result. There two cases we shall consider: (1) the case when $s \in T$. (2) The case when $s \in S_i (i \geq i)$.

Case 1: Since r and s satisfy the join condition, they must be dispatched by the hash function to the corresponding partitions R_k and T_k. Assume $r \in R_{k_i}$ and $s \in T_{k_j}$, R_{k_i} and T_{k_j} are the bucket runs of partition R_k and T_k. If i equals to j, tuple r and s are flushed together. If r is first read into R_{k_i}, s will probe all the tuples in R_{k_i} when s arrives. Tuple r must be joined with s and (r, s) be output. If i doesn't equal to j, tuple r and s are not flushed together. Since r matches s, they satisfy the join condition in the algorithm of merging phase and (r, s) must be output as join result.

Case 2: tuple s must be hashed into some disk partition of S_{i_k} by the hash process. Then the join process reads the corresponding partition of R_k into memory, all the tuples in S_{i_k} are joined with all tuples of R_k. Since r must be in R_k, (r, s) can be output.

From the discussion above, we conclude that the assumption that (r, s) is not output by NDT is not possible. Thus, NDT produces all output results.

Theorem 2: Let R be a disk resident relation and S be a tape resident relation, then NDT returns all the results of $R \bowtie S$ exactly once.

Proof: Assume that $r \in R$ and $s \in S$ and (r, s) satisfies that join conditions. We assume that (r, s) are reported by NDT more than once. Note that the chunks of S are the partitions of tuples of S. There is no tuple s that belongs to S_i and S_j ($i \neq j$) at the same time. So we have:

$$R \bowtie S = \bigcup_{S_i \subset S} (R \bowtie S_i) \; (i \geq 0, S_0 = T)$$

So we can prove the theorem by proving that NDT returns all tuples of $R \bowtie S_i$ ($i \geq 0$) exactly once. There two cases we shall consider: (1) the case when $s \in T$. (2) The case when $s \in S_i$ ($i \geq 0$).

Case 1: (r, s) could be produced in the merging phase, the hashing phase or both. It is obviously that both hashing

phase and merging phase can't output (r, s) more than once. Let's first assume that (r, s) is produced in the hashing phase more than once. Then if r is first inserted into the hash table, it is only joined with s when s arrives and there is no other chance to join r and s again. So the assumption that (r, s) is produced more than once in the hashing phase is not correct. (r, s) can not be outputted more than once in the merging phase too, because r and s are generated in sorted order and r is checked for a match with s only once. (r, s) can not be produced both in the hashing phase and in the merging phase. If (r, s) is produced in the hashing phase, r and s must be flushed onto disk together which prevent r from joining with s in the merging phase. With the same reason, if (r, s) is generated in the merging phase, (r, s) can not be generated in the hashing phase.

Case 2: tuple s must be hashed into some disk partition of S_{i_k} by the hash process. Then the join process reads the corresponding partition of R_k into memory, each tuple in S_{i_k} is joined with the tuples of R_k. So (r, s) is only produced exactly once in this case.

From the discussions above we can conclude that the assumption that (r, s) be output more than once is not possible. Thus, NDT produces each join result exactly once.

RESULTS

We implemented NDT on a 550 MHz Pentium system running Linux 7.2. The computer has 128 Mbytes of main memory, 20 Gbytes of disk space and an INITIO SCSI-2 bus, which connect to a tape library (Exabyte 220L). The tape library has two Eliant 820 drives. The cartridge capacity is about 10 GB without compression.

We only compare NDT with CDT-GH because CDT-GH is the candidate to join a disk resident relation with a tape resident relation. Since our purpose is to compare the performances of NDT and CDT-GH under the same condition, we reduce the size of main memory and disk proportionally and at the same time we reduce the size of dataset. The dataset was generated synthetically. The tuple size of relation R is set to 1024 bytes long and the tuple size of relation S is fixed at 9 times of that of relation R. The number of tuples in both relations was the same in all the experiments. The dataset size is increased by increasing the tuple number of both R and S at the same time. And the join keys are uniformly distributed in a range of 1,000,000,000 values.

Total performance: We first tested the efficiency of the NDT and compare it with that of CDT-GH. In the experiment, dataset size varied from 2 to 10 Gbytes. The memory size and disk size was fix at 20 and 400 Mbytes, respectively. The total execution time of NDT is very

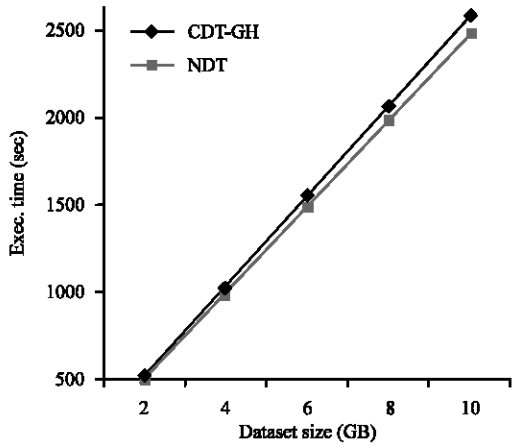

Fig. 1: Total performance related to the dataset size

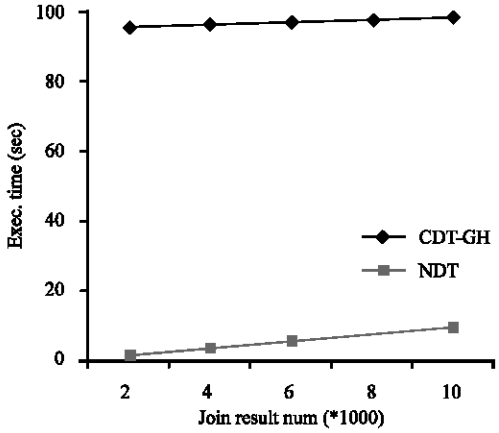

Fig. 2: Performance of producing the first k results

similar with that of CDT-GH and the slopes of the two lines are very similar too, which indicate that the NDT algorithm is as efficient as CDT-GH to deal with massive data and scales well with the increase of dataset size (Fig. 1).

Performance of producing first k results: In the second experiment, we compared the efficiencies of producing first k results of NDT and CDT-GH. We use the 2 Gbytes dataset, in which relation R is 200 Mbytes. Memory size and disk size is fixed at 20 and 400 Mbytes, respectively. The results showed that NDT took much less time to produce the first 10000 join result than CDT-GH (Fig. 2). It took only about 10 in NDT to produce the first 10000 join results while it took about 92 in CDT-GH to produce those results. The first tuple is seen at almost the beginning of NDT, but it took about 72 to produce the first result in CDT-GH. We can see from the two experiments that NDT is as efficient as CDT-GH and can produce join result much earlier that CDT-GH.

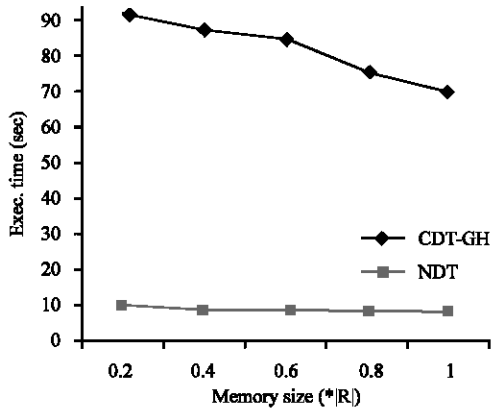

Fig. 3: Effectiveness of memory size to the performance of producing the first k results

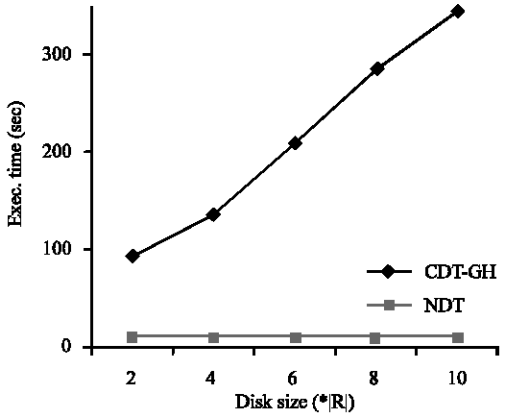

Fig. 4: The effectiveness of disk size to the performance of producing the first k results

Performance of producing the first k results related to memory size: In the third experiment, we fixed the dataset size at 2 and varied the memory size from 20 to 100 Mbytes to study how memory size affects the performances of producing the first k results. The execution time of CDT-GH and NDT both decreased as we increase the memory size (Fig. 3). Although the execution time of CDT-GH decreased much faster than that of NDT, the execution time of CDT-GH is still much larger than that of NDT, because in CDT-GH join result can only be produced until relation is partitioned. In NDT, these result are produced in the first two phase, so they are produced much earlier than in CDT-GH.

Performance of producing the first k results related to disk size: In the last experiment, we tested the performance of producing the first k result related to the disk buffer size. 2 Gbytes dataset was used and the memory size is set to 20 Mbytes. The disk size varied from 400 Mbytes to 2 Gbytes. The execution time of producing the execution time of CDT-GH increased as the disk size is increased while the execution time of NDT remains the same (Fig. 4). The reason is in CDT-GH the first chunk of relation S needs to be hashed onto disk first and the larger the disk size the longer it will take to hash the first chunk since the chunk size increases with the disk size. While in NDT, the first chunk of S is read concurrently with partitioning relation R and many the join results were produced in the first two phase.

CONCLUSIONS

In this study, Non-blocking Disk-tape (NDT) join is proposed which join disk resident relation with tape resident relation. Such join operation is often required by modern applications that need to manage massive data. Examples are data analysis tools and data mining tools. This kind of join operation often needs a long time to complete, so it is favourable to produce the join result as early as possible. The execution of NDT can be divided into three phases: the hashing phase, the merging phase and the probing phase. In the hashing phase, tuples of R and S are read into in-memory hash buckets simultaneously based on their hash values. Then tuples in the corresponding buckets are joined. Once the memory gets filled, a bucket pair is chosen to be flushed onto disk. After relation R is finished hashing. The merging phase takes control and tuples that are previously flushed to disk are joined together. In the probing phase, S are hashed onto disk in chunks and joined with the hashed version of R. The probing phase is iterated until S is exhausted. Experimental results show that NDT can produce join results much earlier than the-state-of-art CDT-GH and the performance of NDT is about the same with that of CDT-GH.

ACKNOWLEDGMENTS

Supported by National Natural Science Foundation of China under Grant No. 60273082; the National High-Tech Research and Development Plan of China under Grant No. 2001AA41541; the National Grand Fundamental Research 973 Program of China under Grant No. G1999032704.

REFERENCES

1. Frew, J. and J. Dozier, 1997. Data management for earth system science. ACM SIGMOD Rec., 26: 27-31.
2. Myllymaki, J. and M. Livny, 1996. Disk-tape joins: Synchronizing disk and tape access. Proc. ACM SIGMETRICS, Lausanne, Switzerland, pp: 279-290.
3. Myllymaki, J. and M. Livny, 1997. Relational joins for data on tertiary storage. Intl. Conf. Data Eng., Birmingham, UK, pp: 159-168.

4. Kraiss, A., P. Muth and M. Gillmann, 1999. Tape-disk strategies under disk contention. Intl. Conf. Data Eng., Birmingham, Australia, pp: 552-559.
5. Myllymaki, J. and M. Livny, 1996. Efficient buffering for concurrent disk and tape I/O. Proc. Performance'96, Lausanne, Switzerland, pp: 453-472.
6. Shapiro, L., 1986. Join Processing in database systems with large main memories. ACM Trans. Database Sys., 11: 239-264.
7. Wilschut, A. and P. Apers, 1999. Pipelining in query execution. Conference on Databases, Parallel Architecture and their Applications, Miami, USA, pp: 68-77.
8. Urhan, T. and M.J. Franklin, 2000. XJoin: A reactively-scheduled pipelined join operator. Data Eng. Bull., 23: 27-33.
9. Mokbel, M. and M. Lu and W. Aref, 2004. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. Intl. Conf. Data Eng., pp: 251-263.
10. Dittrich, J.P., B. Seeger, D.Taylor and P.Widmayer, 2002. Progressive merge join: A generic and non-blocking sort-based join algorithm. Proc. Conf. Very Large Databases, Hong Kong, China, pp: 299-310.
11. Hass, L.M., M.J. Carey and M. Livny, 1993. Seeking the truth about ad hoc join costs. Tech. Rep., 1148, Univ. Wisconsin Madison, pp: 241-256.
12. Hagmann, R.B., 1986. An observation on database buffering performance metrics. Proc. Conf. Very Large Databases, Kyoto, Japan, pp: 289-293.