Computing and Informatics, Vol. 24, 2005, 391–413

DISTRIBUTIVE JOIN STRATEGY BASED ON TUPLE INVERSION

Wang-Chan Wong

Computer Information Systems Department
California State University Dominguez Hills
Carson, CA 90747
e-mail: wcwong@csudh.edu

Lubomir F. Bic

Deptartment of Computer Science
University of California
Irvine, CA 92617
e-mail: bic@ics.uci.edu

Manuscript received 5 October 2005; revised 28 September 2005
Communicated by Atsushi Imiya

Abstract. In this paper, we propose a new direction for distributive join operations. We assume that there will be a scalable distributed computer system in which many computers (processors) are connected through a communication network that can be in a LAN or as part of the Internet with sufficient bandwidth. A relational database is then distributed across this network of processors. However, in our approach, the distribution of the database is very fine-grained and is based on the Distributed Hash Table (DHT) concept. A tuple of a table is assigned to a specific processor by using a fair hash function applied to its key value. For each joinable attribute, an inverted file list is further generated and distributed again based on the DHT. This pre-distribution is done when the tuple enters the system and therefore does not require any distribution of data tuples on the fly when the join is executed. When a join operation request is broadcast, each processor performs a local join and the results are sent back to a query processor which, in turn, merges the join results and returns them to the user. Note that the distribution of the DHT of the inverted file lists can be either preprocessed or distributed on the fly. If the lists are preprocessed and distributed, they have to be

maintained. We evaluate our approach by comparing it empirically to two other approaches: the naive join method and the fully distributed join method. The results show a significantly higher performance of our method for a wide range of possible parameters.

Keywords: Distributed hash table (DHT), distributed join, inverted file

1 INTRODUCTION

Processing a query often involves many join operations. The join is the most costly and time consuming operation in database query processing, and thus a good join strategy is essential to performance.

A large amount of work has been done to develop efficient algorithms to perform the join operations in the 1980s and '90s. The most common types of join strategies that have been studied are grace join, sort-merge join, nested loops join, and simple hash join [9, 10]. Most of these algorithms were developed and studied in a uniprocessor environment. With the constant decrease of hardware costs, multiprocessors or multi-computers become readily attainable. Finding efficient parallel join algorithms has become one of the most important research topics in data engineering. A significant number of parallel join algorithms have been proposed and studied [5, 20, 17, 16, 19, 14]. These algorithms are mostly the parallel versions of the traditional joins, such as sort-merge join, nested-loop join or a combination of different hashing techniques. All these studies have shown that hash based algorithms in most of cases outperform other join algorithms.

In most of these studies, a join operation between two relations R and S is implemented first by partitioning the relations and distributing them across a number of processors. The algorithms then adopt a certain traditional join algorithm (e.g. grace hash join) to perform the join in the local processors concurrently. The joined results are then merged to yield the final answers. Although the basic idea and implementation are quite simple, these approaches suffer two major drawbacks. First, given two joinable tables, the data distribution has to be sequential: each tuple is hashed on the joinable attribute and distributed to a processor. Local join operation cannot commence until the data distribution is finished. (At best, the join operation can be pipelined, but it still is not fully parallel; the size of data transfer is relatively large and frequent.) In addition to these disadvantages, once the join operation is completed, the distributed data files, join index files, etc. are discarded. The approach is therefore wasteful and becomes very expensive for repeated join operations.

In recent years, the problem of distributed query processing has been revisited in a different context: peer-to-peer (P2P) file-sharing systems. Distributed Hash Tables (DHTs) are used to simplify the construction of large-scale distributed systems without any centralized control or hierarchical organization. It is shown that

DHTs are very effective to locate a particular data item in a P2P network such as in PIERSearch [13] and Chord [21] and also in other studies [1, 3, 2, 18].

In this paper, we present our concept of a distributed inverted join, which preprocesses data distribution for joinable tables, allows fully parallel local join operations to be performed at each local site, and minimizes the number of messages and the size of the data records transferred. In our approach, data distribution is done when the table is defined and when tuples are being inserted. For instance, given two joinable tables, tuples of these two tables are first hashed according to their primary key values and distributed across the processors. When a join operation involving these keys is invoked (e.g. these keys are in the same domain and are joinable), it can be performed at the local processors in parallel.

For each possible joinable attribute (e.g. determined by examining their E-R diagrams, or from user requirements), their inverted files [12] are generated and distributed across the network. These distributed values may be retained permanently, which is in contrast to other studies where the joinable values are hashed and used on the fly, and are discarded afterward. Since all possible joinable values are distributed, they can be done prior to the actual join operation, instead of distributing the join values during execution of a join operation as in all other studies. Therefore, the join operations in our approach are more efficient. Our method is very appealing for the following reasons:

(1) the preprocessing of joinable values improves our join execution time dramatically;

(2) processors can carry out the join on the hashed join value in parallel, eliminating extensive data transfer among processors; hence, the traffic is minimal;

(3) with the assumption of a fair hashing function, the number of hashed tuples in each processor is relatively small. Therefore, the local join operation costs are also minimal.

The organization of this paper is as follows: in Section 2, the past related studies will be summarized. We present our concept of distributive joins based on tuple inversion in Section 3. Section 4 describes how update and deletion can be done using our approach. In Section 5, we analyze the performance of our approach and compare it to two other approaches. We conclude our paper in Section 6.

2 RELATED WORK

We summarize several major studies on parallel join in this section. Note that in these studies, the underlying computational model has not been changed. These algorithms are mostly the parallel versions of the traditional join algorithms, such as sort-merge join, nested-loop or a combination of traditional hashing technique with modifications to adopt them to a multiprocessor environment.

In [5], a multiprocessor version of grace-hash join, hybrid-hash join, and simple-hash join were studied in multiprocessor-shared-nothing environment. The archi-

tecture model in this study consists of a group of processors with or without disks, interconnected by an 80 Mbit/second token ring. Tuples in relations R and S are distributed horizontally across the disk drives and joins are performed in parallel at each processor. This study has shown that the performance of the multiprocessor simple hash join and multiprocessor sort merge were most affected due to primary memory limitations. The sort-merge performed best with 100 % of the relations in memory and the worst in limited memory situations.

The grace join has two stages for partitioning data. With decreasing primary keys, the number of buckets increases. The overhead due to overflow is small and the performance is relatively unaffected. The hybrid hash join has the best performance in limited memory. In [17], parallel versions of sort-merge join, hash-based sort-merge join, and multiprocessor hybrid-hash join are pipelined to achieve a more efficient performance. These algorithms are studied on an architecture that has a set of clusters linked by an intercluster bus or ring. Each cluster has a set of processors, a shared memory bank addressable by all the processors in the cluster, and a set of disk storage units and associated controllers. This study shows that the two hash based algorithms outperform the sort-merge algorithm if the output tuples are not required in sorted order. However, in the case when the source relations are already sorted, or the applications require the output tuples to be sorted on the join attributes, the sort-merge algorithm may be advantageous. The two hash-based algorithms have the best performance under a very large single cluster. When the number of clusters increases their performance worsens due to the communication among clusters.

In [14], the parallel version of the hash-based nested-loop join, simple hash join, and hybrid hash join are implemented in a shared memory environment. The multiprocessor architecture consists of a number of processors, where each processor shares the common memory with other processors through a global hash table. A locking mechanism is used to regulate any concurrent writes to this global hash table. The algorithms are studied under different amounts of available memory, different relation sizes, and different processors in the system. Among these three algorithms, the simple hash join algorithm performs worst in most cases. When the number of processors increases and the amount of memory available increases, its performance becomes comparable to the other algorithms. The hash based nested loops perform better when the size of both relations is similar. However, when the size difference between the two relations widens, the hybrid hash outperforms the other algorithms.

In recent years, the problem of distributed query processing has been revisited in a different context: peer-to-peer (P2P) file-sharing systems. In a P2P system, a program running at each node is equivalent in functionality and each node is a master and client at any given time. Existing P2P systems can be categorized into two types:

(1) *unstructured* P2P networks and

(2) *structured* P2P networks [18].

An unstructured P2P network connects millions of users dynamically in an ad hoc fashion and hence creates a giant federated file base in which data objects such as audio/music files, pictures, animations and videos, do not have global unique IDs and could have multiple copies co-existing in the same network. Queries are performed on a set of keywords. Gnutella [11], Kazaa [8] and Morpheus [15] are the most notable unstructured P2P systems. To do a search on the availability of a file, a query of certain keywords are processed and queries are carried out in a simple *flooding* fashion: a node sends the keywords to its P2P neighbors who forward the query recursively. The flooding mechanism is based on the so-called "time-to-live" algorithm that is limited to a finite number of hops. The flooding-based algorithm is not exhaustive. Therefore, these networks do not guarantee results and most often miss the results that lie beyond their time-to-live search space. On the other hand, a structured P2P network is characterized by the fact that each data item has a unique ID. With a unique global ID, a structured P2P network can be implemented efficiently using distributed hash tables (DHTs). DHTs are used to simplify the construction of large-scale distributed systems without any centralized control or hierarchical organization. There are several structured P2P networks under development, for example, Tapestry [22], Chord [21] and PIERSearch [13].

While structured P2P network based on DHTs are shown to be scalable in supporting file sharing operations, the data items in the search are mostly unstructured such as audio/music files, pictures, animations and videos. In this paper, we will demonstrate how DHTs can be applied in a structured and fixed network (instead of ad hoc, dynamic unstructured network) to improve the join operation performance of structured data objects, i.e. in a record-based relational database system.

3 OUR APPROACH

We assume that there are two joinable tables R and S in our study. We use a fine-grain distribution technique such as the DHT that distributes the original tuples (according to the key values of R and $\mathbf{S}$) and/or non-key values (the non-key values and the location of their tuples) across the array of processors. The distribution is done whenever the need to join is identified, instead of partitioning and distributing the data files right before the join operation is executed (as in most of the studies of join operation). We first describe the underlying architecture for our parallel join strategy and then present our algorithms along with examples in detail.

3.1 The Architecture of our Model

The architecture model used in our studies consists of a group of general-purpose processors with identical capabilities. These processors are linked through an interconnection network such as a Local Area Network or a P2P network. In this paper, the analysis is done based on a LAN connectivity. Each processor has its own local memory and local disk storage. These processors share nothing and communicate with one and other through messages across the network. Once the data

files are distributed across the network of processors, the distributed data files and all other index files are stored in the local disks. It is assumed that the bandwidth of the network is sufficient to handle the task at hand. We further assume that there is a central query processor that interfaces with the users, generates broadcast messages, and collects and assembles results for a finished query.

3.2 Our Algorithm

Our main concern is to achieve a better performance on distributive join operations. In our approach, horizontal partitioning of tables based on the primary key is done in the preprocessing phase. By examining the E-R diagram, or based on user requirements, joinable non-key attributes are identified. To support better performance, we build and distribute the inverted files of these non-key attributes. Note that generating and distributing the inverted files can be done either in the preprocessing phase or just in time for the join. If they are generated in the preprocessing phase, they require maintenance. The inverted file list contains only the non-key value with the *physical location* of the actual data records. When a join operation is invoked, no on-the-fly partitioning of relations and distributing of data records across the network are needed. Only the inverted files and/or local data files are used to implement the join. Our join strategy can be divided into three major phases: a preprocessing phase, a joining phase, and a merging phase. We describe the procedures for these three phases. We focus on the performance improvement on joins with two tables (e.g. tables R and S). Multi-way join chain will not be discussed in this paper.

(a) Preprocessing Phase

Consider the procedure in Figure 1. To distribute tuples of table R, we first apply a hash function, h_1, to the primary key of each tuple in R to determine the destination processor site that the tuple will be distributed to. After the distribution, at each site j, the tuples of R are stored locally in a file FR_j. Every existing table is distributed in the same fashion. To insert a record into table R, its primary key is hashed and distributed to the designated processor.

At each local processor j, a hash index file HR_j is created by hashing records from data file FR_j using h_2. This is illustrated in lines 2–3 of Figure 2. Another hash function, h_3, is applied to each of the joinable non-key attributes of the tuple. The idea is to generate the inverted file list tuple for each joinable non-key attribute. For example, the message $\langle \mathrm{insertInverted}, \mathrm{RNK}_i, j.a \rangle$ means that it is an insertInverted message for the i^{th} non-key value RNK_i of R; the corresponding primary key index is in the a^{th} record of the hash index file of R at site j (i.e. HR_j). Note that the a^{th} record number can be expressed as $\langle \mathrm{fid}, n \rangle$, where fid is the file ID of HR_j and n is the record number offset of the file. The site number j is basically the IP:port address of the processor. As such, the inverted file list tuple contains the physical address of the record in the hash index file of the primary key (i.e. *ip:port.fid.n*,

R is the table to be distributed across the network
P = number of processors;
FR_j = files of R at site j after distribution; where $0 < j \leq P-1$
h_1 = hashing primary key PK of a record to processor number between $\{0..P-1\}$

/* data distribution according to the hashed values of primary key of R */

1: for each tuple c in R do
2: begin
3: $j := h_1(c[\mathrm{PK}])$;
send message $\langle \mathrm{insertRecord}, c \text{ of } R \rangle$ to j;
/* at site j, upon receiving message $\langle \mathrm{insertRecord}, c \text{ of } R \rangle$, stored in the file FR_j */
4: $\mathrm{FR}_j := \mathrm{FR}_j \cup c$;
5: end;

Fig. 1. Distributing data records to processors

$\mathrm{FR}_j =$ data file of R at site j after distribution; where $0 < j \leq P-1$
$\mathrm{HR}_j =$ hash index file of FR_j at site j
$h_2 =$ hash (local) primary key PK of record of FR_j
$a = h_2(\mathrm{PK})$, the hashed record number of HR_j
$h_3 =$ hash (external) non-key NK of record of FR_j to distribute the inverted file list to the processor of $\{0..P-1\}$
$N =$ number of non-key joinable attributes of R
$\mathrm{RNK}_i =$ inverted file list of the i^{th} non-key attribute of R, where $0 < j \leq N-1$
$b = h_3(\mathrm{NK})$, the hashed processor number
message $= \langle \mathrm{insertInverted}, \mathrm{RNK}_i, j.a \rangle$, inverted file list tuple for the i^{th} non-key of R, where $0 < j \leq N-1$

for each tuple d in FR_j at site j do
insertRecord(d);

procedure insertRecord(d, FR_j)
1: begin // at each site j, create hash index file HR_j using h_2
2: $a = h_2(d[\mathrm{PK}])$; //hashed record number in HR_j
3: $\mathrm{HR}_j := \mathrm{HR}_j \cup d[\mathrm{PK}]$; // hash index file on PK of FR_j
4: for $i := 0$ to $N-1$ do // for each non-key joinable column of d
5: begin
6: message := $\langle \mathrm{insertInverted}, \mathrm{RNK}_i, j.a \rangle$; //inverted file list tuple
7: $b := h_3(d[\mathrm{RNK}_i])$;
8: send message to processor b;
9: end;

Fig. 2. Generating and distributing inverted file list

where *ip* is the IP address of the processor or device, *port* is the designated port for the operation, fid is the file ID of the relational table and n is the logical record number). Then external hash function h_3 is applied to the RNK_i and it returns the processor number, b. Then a message containing the inverted file list tuple is sent to processor b. This is illustrated in lines 4–9 in Figure 2.

$\text{FRNK}_{ij} =$ inverted file of the i^{th} Non-Key of R at site j;
$\text{HRNK}_{ij} =$ hash index file of FRNK_{ij} at site j;
At each site j, upon receiving the message $\langle \text{insertInverted}, \text{RNK}_i, j.a \rangle$ do:
(1) insert the tuple in FRNK_{ij}
(2) hash RNK_i into index file HRNK_{ij}

Fig. 3. Indexing the inverted file list

Lastly, upon receiving the inverted file list tuples, the non-key attributes are indexed and stored locally. This is shown in Figure 3. The same process is applied to all data tables.

At the end of the preprocessing phase, table R and its inverted file lists for each of its joinable non-key attributes are distributed across the network. Each processor j can potentially contain the following tables on R:

(1) FR_j, the data file of R;

(2) HR_j, the hash index file of FR_j containing tuples of $\langle$key value, address of record in $\text{FR}_j\rangle$;

(3) FRNK_{ij}, the file containing the inverted file list tuples of the i^{th} non-key attribute of R in the format $\langle$non-key value, physical address of record in some FR$\rangle$, and

(4) HRNK_{ij}, the hash index file of FRNK_{ij}.

Instead of containing tuples of $\langle$non-key value, record number of $\text{FRNK}_{ij}\rangle$, by resolving the indirection, HRNK_{ij} contains tuples of $\langle$non-key value, physical address of record in some FR$\rangle$. That is, with a non-key value, the physical address of the data record in the network is always known.

(b) Joining Phase

There are two cases for joining tables R and S. The first case is to join on the primary key (e.g., the keys of R and S are in the same domain and are joinable). The second case is when the join involves non-key attributes (primary key joins on non-key, non-key joins on non-key).

The simplest case is to join on primary keys of these two tables. In this case, a single message is broadcast to all the processors to do a local join on the keys using the hash index files of the keys (i.e., HR_j and HS_j, for the hash index files of tables R

and S at site j). Any traditional join algorithm, such as hash join, nested loop, etc., can be applied. If the tuples contain the same join values, they belong to the same buckets of HR_j and HS_j. Then the joined tuple $\langle$join value, addressR, addressS$\rangle$ is returned, where addressR and addressS are physical addresses of the data record of R and S in FR_j and FS_j.

When joining with non-key attributes (e.g., join the l attribute of R with the m attribute of S), each processor j retrieves the corresponding non-key index files for R and S (e.g. HRNK_{lj} and $HSNK_{mj}$). Join operations using key and non-key value are similar. The join, again, yields $\langle$join value, addressR, addressS$\rangle$. Figure 4 describes the procedure.

TR, TS are the hash index file types of either the key or a non-key;
msg := $\langle \mathrm{join}(R[\mathrm{PK}] \mid R[\mathrm{NK}_l]), (S[\mathrm{PK}] \mid S[\mathrm{NK}_m]) \rangle$;
// key or non-key l, m of R and S
broadcast msg to all processors;
at processor site j, do:
1: if (R[PK]) then TR := HR_j else TR := HRNK_{lj};
// use either the key or non-key hash index
2: if (S[PK]) then TS := HS_j else TS := HRNKS_{mj};
3: join TR and TS;
4: return joined tuples in the form of $\langle$join value, addressR, AddressS$\rangle$

Fig. 4. Join operation

(c) Merging Phase

The last phase of the join operation is for the query processor to merge results returned by each processor and assemble the final answers for the user. The actual data records indicated in addressR and addressS of the joined tuple are retrieved. Figure 5 describes the general procedure.

1: for each joined tuple $\langle$join-value, addressR, addressS$\rangle$
2: begin
3: retrieve tuple c at addressR;
retrieve tuple d at addressS;
4: result := result $\cup$ joined tuple of (c, d);
5: end

Fig. 5. Merging phase

Note that the retrieving and joining are done in a pipeline fashion to achieve higher parallelism.

4 DELETION AND MODIFICATION

In our method, the joinable non-key inverted file lists are distributed. If these inverted file lists are to be kept permanently, we need to maintain them when the database table is modified.

4.1 Deletion of a Tuple

To delete tuples from the database table, we first need to determine where the actual tuples are located. Messages are then sent to those sites. At each site, upon receiving the delete messages, we find out where the inverted file lists are distributed. Another set of messages is sent to delete records of the inverted file lists. The steps are shown in Figure 6.

$D =$ set of tuples to be deleted in table R
$j, k =$ sites (processors)
$N =$ number of non-key attributes of R
for each d in D do
    deleteRecord(d, R);
procedure deleteRecord(d, R)
    begin
        $j := h_1(d[\text{PK}])$; // to determine where the data record is distributed to
        send $\langle$*deleteRecord* $d[PK], R\rangle$ message to processor j;
    end;
At each site j
(1) Upon receiving $\langle$deleteRecord $d[\text{PK}], R\rangle$
    for each non-key value NK_i of d, $i = 0..N-1$ do
    begin
        $k := h_3(d[\text{NK}_i])$; //to determine where the inverted is distributed to
        send $\langle$deleteInverted $d[\text{NK}_i], R\rangle$ to k;
        delete d of FR_j and update HR_j;
    end;
(2) Upon receiving $\langle$deleteInverted $d[\text{NK}_i], R\rangle$, delete records in FNKR_i and its index file HNKR_i.

Fig. 6. Deletion procedure

4.2 Modification of a Tuple

An update can be in one of three forms:

(1) tuples identifiable by primary key, for instance, example (a) in Figure 7;

(2) tuples identifiable by a conditional expression over some attributes (key or non-key), for instance, examples (b) and (c); and finally,

(3) all tuples updated without any condition (example (d)).

(a) Update EMP
Set Salary = 1000
where EID = 100; //key = EID

(b) Update EMP
Set Salary = 1000
where Name = 'Smith'; //Name is non-key

(c) Update EMP
Set Salary = 1000
where Name = 'Smith'
And Dept = 'Sales';

(d) Update EMP
Set Salary = Salary + 100; //for all employee

Fig. 7. Update Examples in SQL

Updates in our approach are implemented with deletion and insertion. The basic idea is to identify all the affected data records, delete them, and insert the new data records with the updated values. Deletion of the data records triggers and propagates the deletion to all affected inverted file lists. Inserting a new data record with the updates generates the new inverted file lists. Figures 8 and 9 describe the update procedures. Further refinement without using the delete and insert approach is possible but will not be addressed in this paper.

5 EMPIRICAL STUDY

In this section, we compare our method to two other approaches. The first approach (Case 1) is the naive join method, in which the tables are not distributed. When a join is invoked, one of the tables is transported entirely to the node of the other table. Then a local join is performed. This approach sets the upper bound on how a join could behave in the worst-case scenario. The second approach (Case 2) is a fully distributed join strategy, in which the join tables R and S are not distributed to begin with. When the join operation is invoked, they are distributed across the entire network according to the join attribute values. The partitioned R and S at each local node are then joined. The results are assembled and returned. This is the state-of-the-art approach that many commercial database systems are using.

Our approach (Case 3), discussed in Section 3, is analyzed and compared to Cases 1 and 2. However, we *do not* implement the idea of pre-distribution of the possible inverted file list. In this case, only tables R and S are pre-distributed. The inverted file lists are to be generated and distributed on the fly. We demonstrate that even without the pre-distribution of the inverted file lists, our approach still outperforms Case 2.

msg: $\langle$update old$A$, new$A$, condition, $R\rangle$ where old$A$ and new$A$ refer to the old attribute value(s) and new attribute value(s) after the update and the *condition* is the update condition
D: set of affected data record addresses
I: set of updated data records to be inserted

At the query processor, upon receiving message $\langle$update old$A$, new$A$, condition, $R\rangle$
if condition is not null then
    begin
        D := eval(condition);
            //evaluate the conditions and return the set of affected data records
        I := replace(D, old$A$, new$A$); // generate the updated data records
        for each d in D, send message $\langle$deleteRecord$d$, $R\rangle$;
        for each i in I, send message $\langle$insertRecord$i$, $R\rangle$;
    end;
else // condition is null, for all data records in R, update old$A$ to new$A$
    begin
        refresh(old$A$, new$A$, R);
    end;
end if;

Fig. 8. Update procedures

5.1 Performance Analysis

For all the cases analyzed, we assume that there are two databases in a local area network connected by either a regular Ethernet or Fast Ethernet. We also assume that both databases (R and S) have the same size. Most of the constants and the parameters described in Figure 10 are based on common timing parameters adopted by different researchers and can be found in [17]. Based on this information, we define the cost functions of the above three cases.

Case 1: A Naive Join Approach
In the naive approach, tables are not distributed. When a join is invoked, one of the two joinable tables (e.g., the smaller one, say table S) will be transported to the site of the other table, followed by a local join. The time components can be described as follows

$$t_{ds} = \text{ps} \times (\text{T\_READ\_PAGE} + \text{T\_SEND\_PAGE}),$$

where t_{ds} is the time to read all pages of table S and sending them to the site of table R. Then, a hash join is invoked to join both tables and its time is computed

```
n = number of subconditions;
condition = subcondition_1 [and|or] subcondition_2 [and|or] ... subcondition_n
list_i = data records that satisfy subcondition_i;
D = Set of data record physical addresses
I = Set of updated data records
function eval(condition) return D
begin
    for each subconditioni in {0..n − 1} do
        retrieve data record addresses that satisfiy subcondition_i to list_i;
    resolving the conditional expression list_1 [and|or] list_2 [and|or] ... list_{n−1};
        /* e.g. intersection, union of the lists */
    return the result of the conditional expression evaluation;
end;
function replace(D, oldA, newA) return I
begin
    I is initially empty;
    for each d in D do
    begin
        retrieve data record t of d;
        replace oldA with newA in t;
        I := I ∪ t;
    end;
    return I;
end;
procedure refresh(oldA, newA, R)
begin
    deleteRecord(t, R);        //Figure 6
    replace oldA with newA in t;
    insertRecord(t, R);        //Figure 2
end;
```

Fig. 9. Update procedures

as

$$t_{hs} = \text{ps} \times \text{T\_READ\_PAGE} + \text{sizeof}(S) \times (\text{T\_HASH} + \text{T\_PUT\_HASH}),$$

where t_{hs} is the time to generate the hash table of S by summing up the time to scan the table, the time to hash the records, and the time to insert them into the hash table. Similarly, the time to generate the hash table for R is obtained by

$$t_{hr} = \text{pr} \times \text{T\_READ\_PAGE} + \text{sizeof}(R) \times (\text{T\_HASH} + \text{T\_PUT\_HASH}).$$

Constants

PAGE_SIZE = 32 Kbyte	– page size
T_READ_PAGE = 15	– time takes to read a page from disk to memory (ms)
T_HASH = 0.003	– time takes to compute the hash function (ms)
T_PUT_HASH = 0.01	– time takes to put hashed data in memory (ms)
T_COMPARE = 0.005	– time takes to compare data when hashing (ms)
T_SEND_PAGE	– time takes to send a page of data through network 25.6 ms if the network transmission rate is 10 Mbps 2.56 ms if the network transmission rate is 100 Mbps

Variables

pr	– number of pages of R
ps	– number of pages of S
ph_r	– number of pages of "hashed" R
ph_s	– number of pages of "hashed" S
t_{dr}	– time to distribute R
t_{ds}	– time to distribute S
t_{hr}	– time to create hash file of R
t_{hs}	– time to create hash file of S
t_{jhrs}	– time takes to join "hashed" R and "hashed" S
$t_{assemble}$	– time takes to assemble
t_{total}	– total time
nSites	– number of sites in the network
h_rec_per_page	– number of hash records that a page contains
selectivity	– the selectivity of the join

Variables for Case 3

inverted_rec_size	– inverted record size
FR_j, FS_j	– local tables R and S at site j
PFR_j, PFS_j	– number of pages of FR_j, FS_j
FRNK_{xj}, FSNK_{yj}	– inverted file list of the x^{th} and y^{th} non-key of R and S
PFRNK_{xj}, PFSNK_{yj}	– number of pages of FRNK_{xj}, FSNK_{yj}
HRNK_{xj}, HSNK_{yj}	– hash index files of FRNK_{xj}, FSNK_{yj}
PHRNK_{xj}	– number of pages of HRNK_{xj}
PHSNK_{yj}	– number of pages of HSNK_{yj}
t_{dr_j}, t_{ds_j}	– time to distribute inverted file list of R and S at site j
$t_{localJoin\_j}$	– time to do local join at site j
t_{merge}	– time to merge

Fig. 10. Empirical study constants and parameters

After the hash tables are created, a hash join is performed. The time to perform the hash join is

$$t_{jhrs} = (\text{ph}_r + \text{ph}_s) \times \text{T\_READ\_PAGE} + \text{ph}_r \times \text{h\_rec\_per\_page} \times \text{T\_COMPARE}.$$

The results of the local join is assembled and returned to the query processor. The time of assembling is

$$t_{assemble} = (\text{selectivity} \times \text{sizeof}(R) \times \text{sizeof}(S))/\text{PAGE\_SIZE} \times \text{T\_SEND\_PAGE}.$$

The total time to implement this approach is therefore:

$$t_{total} = t_{ds} + t_{hr} + t_{hs} + t_{jhrs} + t_{assemble} \tag{1}$$

Case 2: Fully Distributed Join at Execution
In this approach, tables R and S are partitioned and distributed across the network. At each site, a local hash join on the local R and S is performed. The results are assembled and sent back to the query processor. The total time of the join operation is calculated using the formula

$$t_{total} = \max(t_{dr}, t_{ds}) + \max(t_{hr}, t_{hs}) + t_{jhrs} + t_{assemble} \tag{2}$$

The equation takes the maximum of the times to distribute R and S, and the maximum of the times to hash both local R and S tables, plus the time to do a local join and the time to assemble and return the records.

The times to partition and distribute the tables are

$$t_{dr} = \text{pr} \times (\text{T\_READ\_PAGE} + \text{T\_SEND\_PAGE})$$

$$t_{ds} = \text{ps} \times (\text{T\_READ\_PAGE} + \text{T\_SEND\_PAGE}).$$

At each site, the times to create the local hash files for tables R and S are

$$t_{hr} = \text{ph}_r \times \text{T\_READ\_PAGE} + \text{h\_rec\_per\_page} \times (\text{T\_HASH} + \text{T\_PUT\_HASH})$$

$$t_{hs} = \text{ph}_s \times \text{T\_READ\_PAGE} + \text{h\_rec\_per\_page} \times (\text{T\_HASH} + \text{T\_PUT\_HASH}).$$

The time to do the local join at each site is

$$t_{jhrs} = \max(\text{ph}_r, \text{ph}_s) \times (\text{T\_READ\_PAGE} + \text{h\_rec\_per\_page} \times \text{T\_COMPARE}).$$

Finally, the average time it takes to assemble together the records is

$$t_{assemble} = ((\text{selectivity} \times \text{sizeof}(R) \times \text{sizeof}(S))/\text{PAGE\_SIZE})$$

$$\times \text{T\_SEND\_PAGE})/\text{nSites}.$$

Case 3: Distributed Join with Tuple Inversion
In our approach, we assume that the databases were already distributed over the local network. When a join query is posted, the inverted file list of the join attributes are generated and distributed. The time to process the join query has several components:

1. Create and distribute inverted file lists

   Upon receiving the join query, at each site, the following steps are carried out:

   (a) create the inverted files FRNK_{xj} and FSNK_{yj}, the inverted files based on FR_j and FS_j;
   (b) distribute FRNK_{xj} and FSNK_{yj};
   (c) at each site that receives FRNK_{xj} and FSNK_{yj}, create the hash index file on FRNK_{xj} and FSNK_{yj}, e.g., HRNK_{xj}, HSNK_{yj};

2. Process local joins on HRNK_{xj}, HSNK_{yj}, and
3. Merge the results.

The total time to process is

$$t_{total} = \text{the longest time in (1)} + \text{the longest time in (2)} + \text{merge}.$$

One of the improvements in the proposed method is that we do not send the actual records through the network to join. The inverted file list has the format of ⟨attribute value, physical address of the data record⟩. The physical address takes the form of *ip.fid.n*, where *ip* is the IP address of the site; fid is the file ID in the file system at this site, and n is the record number offset in fid. Since the inverted file list record is much smaller than the record itself in size, we can pack more information in each memory page, hence shortening the time it takes to send the data through the network.

The time to create and distribute the x^{th} inverted file list at site j is:

$$\begin{aligned} t_{dr\_j} = {} & \mathrm{PFR}_j \times (\mathrm{T\_READ\_PAGE} + \mathrm{T\_PUT\_PAGE}) && \text{// generate } \mathrm{FRNK}_{xj} \\ & + \mathrm{PFRNK}_{xj} \times \mathrm{T\_SEND\_PAGE} && \text{// send pages of } \mathrm{FRNK}_{xj} \\ & + \mathrm{PFRNK}_{xj} \times (\mathrm{T\_READ\_PAGE} + \mathrm{T\_PUT\_PAGE}) && \text{// create } \mathrm{HRNK}_{xj} \\ & + \mathrm{sizeof}(\mathrm{FRNK}_{xj}) \times (\mathrm{T\_HASH} + \mathrm{T\_PUT\_HASH}); \end{aligned}$$

The value of t_{ds_j} with the y^{th} inverted file list at site j is obtained in the same manner.

At each site j, the local join of the x^{th} and y^{th} inverted file list is denoted by the equation

$$t_{localJoin\_j} = \max(\mathrm{PHRNK}_{xj}, \mathrm{PHSNK}_{yj}) \times \mathrm{h\_rec\_per\_page} \times \mathrm{T\_COMPARE}.$$

After the distribution, hashing, and joining together of the inverted file lists, we can send the actual records, according to the inverted file lists, which contain the

physical addresses of the actual records, across the network to designated nodes for the merge. The average merge time of the records is calculated by

$$t_{merge} = ((\text{selectivity} \times (\text{sizeOf}(R) + \text{sizeOf}(S))/\text{PAGE\_SIZE})$$
$$\times(\text{T\_SEND\_PAGE} + t_{assemble})$$

where $t_{assemble}$ is the same as in Case 2.

Finally, the total time is the sum of all the equations from above:

$$t_{total} = \max(t_{dr\_j}, t_{ds\_j)} + \max(t_{\text{localJoin}\_j}) + t_{merge} \tag{3}$$

for all j where $0 \leq j \leq \text{nSites} - 1$.

5.2 Analytical Results

Based on the time functions (Equations (1), (2), and (3) discussed above, we carried out the empirical studies to examine the performance of the three approaches. To simplify the computations, we made the following assumptions:

1. both tables R and S are of the same size.
2. distributions of tables R, S and their inverted file lists to each site are uniform.

Therefore, all tables and hash index files at every site are of the same size. We further assumed that the record size is 256 bytes, bucket size is 128 bytes, and the hashed record size of the inverted file list is 68 bytes. With these assumptions, depending on the sizes of tables R and S, the sizes of local tables (FR_j, FS_j, PFR_j, PFS_j, FRNK_{xj}, FSNK_{yj}, etc., as shown in Figure 10 are determined.

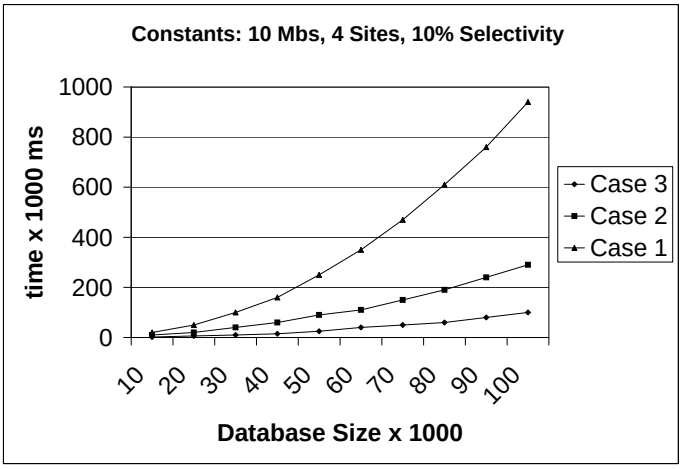

Fig. 11. Comparisons among Cases 1, 2, and 3

Study 1. We first compared all three cases in a network of 4 nodes with regular Ethernet (10 Mbs) and a selectivity of 10 %, by varying the table sizes. The results are depicted in Figure 11. It is obvious that the naive method of Case 1 performs worst since it has to transport the entire table from one node to the other. The number of nodes does not help improve the local join since both tables will be joined locally within one node. As for Case 2 and our method (Case 3), the gap between them widens as the table size increases. Case 1 serves as a base line to judge the preliminary performance of Case 2 and Case 3. Next we concentrate on comparing Case 2 and Case 3 alone, since both showed a significant improvement over Case 1.

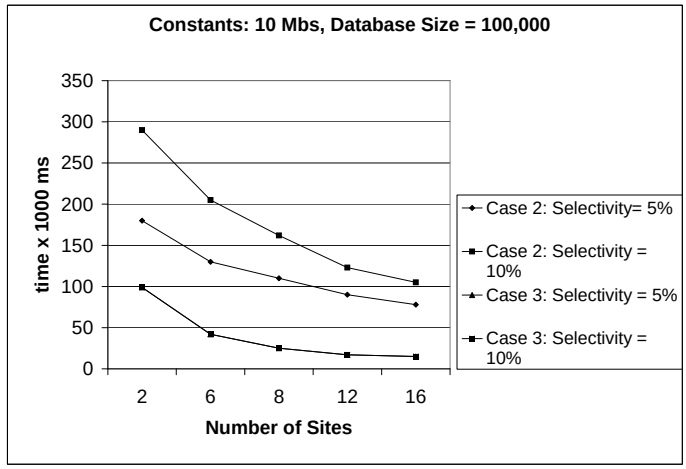

Fig. 12. Comparisons on Selectivity for Case 2 and Case 3

Study 2. The next empirical study was to compare Case 2 with our method. We wanted to see how the network distribution affects performance. In this study, we held both the network transmission rate of 10 Mbs (regular Ethernet) and the table size of 100 000 records constant. We varied the number of nodes and the selectivity between 5 % and 10 %. The results are plotted in Figure 12. There are four curves in the figure. However, both curves for case 3 are so close to each other that they cannot be distinguished. The results indicate that selectivity has more impact on Case 2 than on Case 3. Furthermore, it is obvious that both methods benefit from increased distribution (i.e., more nodes in the network that perform the local joins concurrently). However, the increase of distribution improves the overall performance much significantly in our method (Case 3) than in Case 2.

Study 3. In this study, we fixed the selectivity to 10 % and the network speed at 10 Mbs, which represents a typical database environment. We compared both cases by varying the number of nodes and between two different table sizes. The results are depicted in Figure 13. The results show that the table size and the

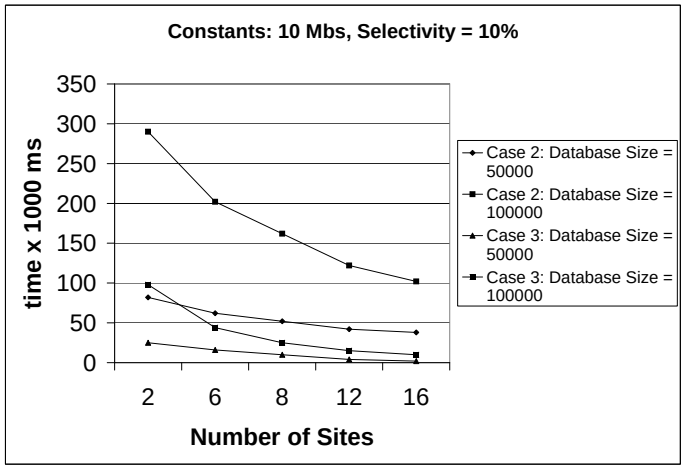

Fig. 13. Comparisons on Database Sizes for Case 2 and Case 3

degree of distribution (i.e. number of sites) are significant factors to distinguish our method from Case 2. As the table size grows, distribution has more impact on Case 2 than on our method.

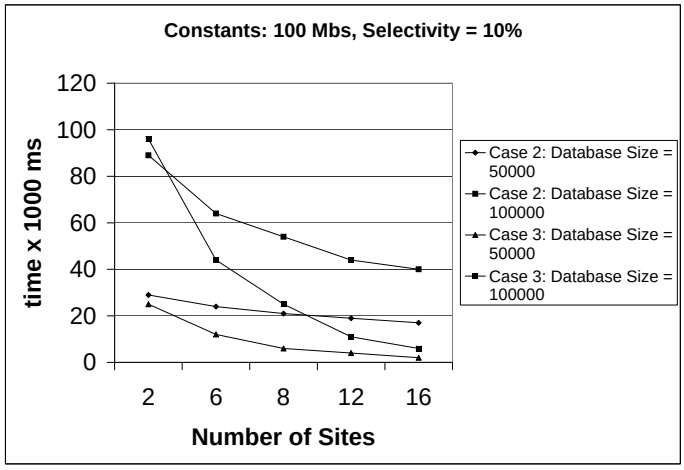

Fig. 14. Comparisons on Network Speed for Case 2 and Case 3

Study 4. Finally, we compared Case 2 and Case 3 to examine the effects of network speed. The study is the same as Study 3 except that the network speed was modified to 100 Mbs (as in Fast Ethernet). The results are shown in Figure 14. While both methods improve over Study 3 in terms of the total time required to perform the join, our method benefits more from the increase of distribution than Case 2. However, it also shows that if the degree of distri-

bution is relatively low (e.g., 3 sites), our method could perform worse than Case 2.

5.3 Discussions

The above empirical studies demonstrated that our method performs much better than both Cases 1 and 2. One reason is the apparent advantage of pre-distributing the tables before a join is invoked. The second reason is that our method distributes the tuple inversion list in a very fine-grained manner. It takes advantage of the distribution more than Case 2. The only situation where our method may not be desirable is when the degree of distribution is relatively low (e.g., 3 sites) and the network speed is high. In this case, our method suffers more overhead to send the inverted lists and retrieve the resulting tuples, and there is not enough distribution to benefit from.

6 CONCLUDING REMARKS

It is obvious that join performance can be improved if we distribute database tables as they are created and as records are entered from the applications. However, pre-distribution may have a problem because a join may not involve the attributes that a given pre-distribution is based on (e.g., non-key joins). In this paper, we presented a practical and effective strategy to fully utilize the availability of massive distributed database tables. Our method is based on tuple inversion, in which the inverted lists of the join attribute values are distributed to the nodes across the network. Local joins on these inverted lists are then performed. The results are then assembled and returned. We further identified the cost components of our method and carried out an empirical study to compare the performance of our method to two other methods. The empirical studies indicate that our method utilizes the increase of distribution more effectively than the other methods and, in general, our method outperforms them with a significant margin. Obviously, maintaining the distributed inverted file list adds overhead to the performance in general. However, the maintenance cost is subject to the update ratio of the application and is a tradeoff between the cost of join retrievals and database updates.

Our work is significantly different from a typical DHT-based file sharing P2P system in the following manner:

1. We are concerned mainly with structured, record-based relational database systems rather than unstructured file objects such as audio/music, pictures, animations and videos.
2. In our system, there exists only one single source of records instead of multiple occurrences of files as in a P2P system.
3. The underlying network in our approach is fixed instead of an ad hoc and dynamic structure as in P2P.

4. We apply DHT at two levels to improve the join performance. The first level DHT, similar to most P2P DHT implementations, is applied to distribute tuples of relational tables to the processors. At a second level, for each non-key attribute, a second DHT is applied to distribute the inverted list of the first DHT. Join operations on non-key attributes are done locally at each node on the non-key attribute of the inverted file list generated by the second level DHT without the need of distributing records throughout the network.

The motivation for this work is based on the fact that, as a rule of thumb, the selectivity of a typical join operation is around 10 % to 15 % on the average. It makes a lot of sense to distribute the inverted file list DHT instead of all the records.

With the rapid improvement of the Internet connectivity and bandwidth, we believe that our approach will work equally well in a P2P network. Assuming there are a large number of processors in a P2P network, the DHTs can be kept in main memory since each processor needs to maintain a relatively small portion of the DHTs. It has shown that a main memory database (MMDB) performs well since it reduces the disk I/Os [4, 20, 6, 7]. Of course, other issues, such as fault tolerance and recovery, will still need to be addressed.

REFERENCES

[1] Arcangeli, J. P.—Hameurlain, A.—Migeon, F.—Morvan, F.: An Adaptive Hash Algorithm using Mobile Agents. Proceedings Net. ObjectDays 2002, Erfurt, Germany, Oct. 8, 2002.

[2] Cates, J.: Robust and Efficient Data Management for a Distributed Hash Table, Master Thesis. Dept. of Electrical Engineering and Computer Science, MIT, June 2003.

[3] Considine, J.: Cluster-based Optimizations for Distributed Hash Tables. Technical Report 2002-031, Computer Science Dept, Boston University, November, 2002.

[4] Dewitt, D. J., et al.: Implementation Techniques for Main Memory Database System. Proceedings of SIGMOD 84, Boston, June 1984, 1–8.

[5] Dewitt, D. J.,—Gerber, R.: Multiprocessor Hashed-Based John Algorithms. Proceedings of VLDB 85, Stockholm, Aug. 1985, pp. 151–164.

[6] Garcia-Molina, H.—Lipton, R. J.—Valdes, J.: A Massive Memory Machine. IEEE Transactions on Computers, Vol. c-33, 1984, No. 5, pp. 391–399.

[7] Garcia-Molina, H.—Salem, K.: Main Memory Database Systems: An Overview. Trans. on Knowledge and Data Engineering, Vol. 4, 1992, No. 6, pp. 509–516.

[8] Kazaa: How Peer-to-Peer and Kazaa Media Desktop Work. `http://www.kazaa.com/us/help/guide-aboutp2p.htm`.

[9] Kim, W.: Global Optimization an SQL-Like Nested Query. ACM Trans. Database System. Vol. 7, No. 3, pp. 443–469.

[10] Kim, W.: Global Optimization of Relational Queries: A First Step. In Query Processing in Database Systems, W. Kim, D. Reiner, and D. Batory, Eds. Springer, New York.

[11] Klingberg, T.—Manfredi, R.: RFC-Gnutella. `http://rfc-gnutella.sourceforge.net`.

[12] Knuth, D.: The Art of Computer Programming. Vol. 3. Addison-Wesley, 1972.

[13] Loo, B. T.—Hellerstein, J.—Huebsch, R.—Shenker, S.—Stoica, I.:, Enhancing P2P File-Sharing with an Internet-Scale Query Processor. Proceedings of the 30th VLDB, Toronto, 2004.

[14] Lu, H.—Tan, K. L.—Shan, M. C.: Hashed-Based Join Algorithms for Multiprocessor Computers with Shared Memory. Proceedings of VLDB 90, Australia, Aug. 1990.

[15] Morpheus: `http://www.morpheus.com/index.html`.

[16] Qadah, G. Z.—Irani, K. B.: The Join Algorithms on a Shared-Memory Multiprocessor Database Machine. IEEE Trans. Software Engineering. Vol. 14, 1988, No. 11, pp. 1668–1683.

[17] Richardson, J. P.—Lu, H.—Mikkilineni, K.: Design and Evaluation of Parallel Pipelined Join Algorithms. Proceeding of SIGMOD 87, San Francisco, May 1987, pp. 399–409.

[18] Sarshar, N.—Boykin, P. O.—Roychowdhury, V.: Percolation Search in Power Law Networks: Making Unstructured Peer-to-Peer Networks Scalable. Proceedings of the 4th IEEE International Conference on Peer-to-Peer Computing, Use of Computers at the Edge of Networks, August 2004, Zurich, Switzerland.

[19] Schneider, D. A.—Dewitt, D. J.: A Performance Evaluation of Four parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. Proceedings SIGMOD '89, Portland, Oregon, June 1989, pp. 110–121.

[20] Shapiro, L. D.: Join Processing in Database Systems with Large Main Memories, ACM Trans. Database System., Vol. 11, 1986, No. 3, pp. 239–264.

[21] Stoica, I.—Morris, R.—Karger, D.—Kaashoek, M.—Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, SIGCOMM 01, San Diego, 2001.

[22] Zhao, B.—Kubiatowicz, J. D.—Joseph, A.: Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing, Tech Report CSD-01-1141, UC Berkeley, April, 2001. Conference, KR '89, Toronto, May 1989, pp. 276–288.

Wang-Chan Wong received his Ph. D. and M. Sc. in computer science, and M. Sc. in business administration from University of California at Irvine. He also received a B. B. A. (Hon) from the Chinese University of Hong Kong. He was a tenured full professor of computer science at California State University; he is currently with the Computer Information Systems of the same campus. He is the founder and president of KBQuest Group, a global IT service provider with offices in the US, Hong Kong and Shanghai, China. His research interests are in database systems, object technology, software engineering, performance evaluation, distributed computing systems, and eCommerce. Current research projects include eCommerce, Peer-to-Peer collaborative work, bio- medical informatics and knowledge management systems.

Lubomir F. Bic received his M. Sc. degree in computer science from the Technical University Darmstadt, Germany, in 1976 and his Ph. D. in information and computer science from the University of California, Irvine, in 1979. He is currently Professor and Co-Chair of the Computer Science Department at the University of California, Irvine. His primary research interests lie in the areas of parallel and distributed computing. Currently he is co-directing the Messengers Project, which explores the use of self-migrating threads to simplify the programming of computationally intensive applications and to improve their performance in distributed computer environments.