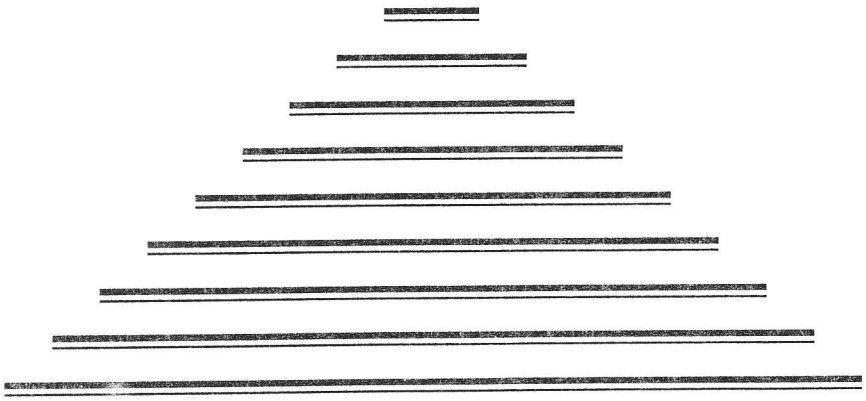
PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS

PDPTA'98



Volume IV

Editor:
H. R. Arabnia

Las Vegas, Nevada, USA
July 13 - 16, 1998
CSREA Press

Fragmented Join: A Pipelined Multidatabase Join Method *

Chung-Min Chen Naphtali Rishe
High Performance Database Research Center
School of Computer Science
Florida International University
Miami, FL, U.S.A.
e-mail: hpdrc@cs.fiu.edu, URL: http://hpdrc.cs.fiu.edu

Abstract

This paper describes a pipelined query processing technique for speeding up cross-database queries – queries that join tables from distributed, autonomous relational databases. This new join method, called "fragmented join", reduces query response and turnaround time by overlapping database access and communication at the application level. The fragmented join algorithm, along with a performance testbed, is implemented in a multidatabase environment. Preliminary results from experiments show that the fragmented join substantially reduces the turnaround time of large queries. In certain cases, the reduction could be as large as 40% of the turnaround time of a traditional non-pipelined approach.

Keywords: distributed databases, multidatabases, query processing

1 Introduction

The great success of relational database technology has spawned a database industry flourishing with many vendors of database management systems (DBMS). As a consequence, large enterprises and loosely-coupled business alliances are often confronted with an information system in which data are stored in more than one source. In many cases, these data sources are separate, independently operated relational databases that are inter-connected through a network. It is imperative for the system to provide users with the *global query* feature: the capability of drawing and integrating data from multiple data sources.

One common approach to supporting global queries is to configure the environment as a *federated database system* [1]. In this approach, an augmented DBMS called "federal DBMS" is adopted to handle global queries. The federal DBMS draws data from other DBMSs (through their native SQL interfaces) and performs the final integration, namely the *join* operation, at its own site. The federal DBMS, equipped with its own data storage system and query processor, may optimize the join operation internally as necessary. For example, it may pipeline the join with the incoming stream of a remote table that is imported across the network. The centralized processing nature of global queries, however, makes the federal DBMS a potential bottleneck under heavy query loads.

Alternatively, one may use the *multidatabase* approach [2], which relies completely on the SQL query facilities of the member DBMSs to process global queries. The common practice is to attach a lightweight "gateway" software to each member DBMS. This software translates data and SQL queries from a foreign format

*This research was supported in part by AFRL (F30602-98-C-0037), NASA (under grants NAGW-4080, NAG5-5095, and NRA-97-MTPE-05), NSF (CDA-9711582, IRI-9409661, and HRD-9707076), ARO (DAAHO4-96-1-0049 and DAAHO4-96-1-0278), DoI (CA-5280-4-9044), NATO (HTECH.LG 931449), Geonet Limited L.P., and State of Florida.

to one understood by the local host. From the view of the host DBMS, the gateway software is nothing but an application program that talks in SQL. To perform a cross-database join, one of the sites that host the operand tables is selected as the *join site*. All operand tables remote to the join site are then imported into the DBMS at the join site (subject to some a-prior *selections* or *projections*). Once this is done, the gateway software at the join site issues an SQL statement to the local DBMS, rendering the actual join operation among the tables.

In comparison to the federated approach, multidatabase systems achieve better load balance, because the global query load is distributed among the member DBMSs. Pipelined processing, however, is not applicable in the multidatabase approach because the handling of a global query does not go beyond the SQL level. This implies that all remote operand tables of a global join query must be *fully* imported into a temporary table at the join site *before* the final local SQL join query can be issued.

In this paper, we argue that the inapplicability of pipelined processing in multidatabase systems is the result of lacking appropriate support, rather than an inherent shortcoming. With appropriate support and optimization at the SQL level, we show that pipelined processing and, thus, substantial performance improvement can be achieved for global queries. We present a novel technique, called *fragmented joins*, that exploits application-level pipeline. The idea is to divide the remote table into a number of smaller *fragments* and issue an SQL join query as soon as a fragment arrives at the join site. A *double buffer* technique is devised to make possible the overlap between the execution of the SQL query at the join site and the transfer of the remote table. This technique also demands less storage space for temporary tables. Finally, for the fragmented join to work efficiently, the size of the fragments must be properly chosen. This is determined by a sophisticated algorithm with the goal of minimizing query turnaround time.

The rest of the paper is organized as follows: Section 2 overviews the traditional approach to supporting global queries in loosely-coupled multidatabases. Section 3 describes the fragmented join algorithm in details. The principles and some details of the fragment size determination algorithm is described in Section 4. The results of an empirical performance study are presented in Section 5. Section 6 concludes the paper and indicates possible future extensions.

2 Traditional Strategies for Cross-Database Joins

Throughout the paper, we consider a loosely-coupled multidatabase environment, in which a gateway software is used to reconcile cross-database join queries (i.e., no federal DBMS is used). First, we examine how cross-database joins are currently being handled by the state-of-the-art multidatabase systems. Consider a join query $R_1 \bowtie R_2$, where tables R_1 and R_2 reside, respectively, at sites DS_1 and DS_2. Without loss of generality, we assume R_1 to be the *outer* relation – the one to be sent across the network to the join site, and R_2 to be the *inner* relation – the one residing at the join site. To perform the join, R_1 is first extracted from DS_1 and imported into a temporary table, say T, in DS_2. Then an SQL query is issued against DS_2 to perform the join $T \bowtie R_2$. In this scenario, the SQL query can not be submitted until R_1 is completely imported into T. Apparently, the total turnaround time for the query is the time span of the two sequential steps: *table staging* (i.e., extracting R_1 and importing into T, denoted $T \leftarrow R_1$) and the *local join* (i.e., $T \bowtie R_2$). For historical reasons, we call the above sequential execution scheme the *join-whole* strategy [3], indicating that operand R_1 is transferred first, and then joined as a whole with R_2.

We argue that the join-whole strategy does not make the best utilization of the system resources. To see this, observe that (1) table staging incurs substantial communication and CPU overhead (due to network protocol stacks, data format conversion, and repetitive SQL "INSERT" commands needed to populate

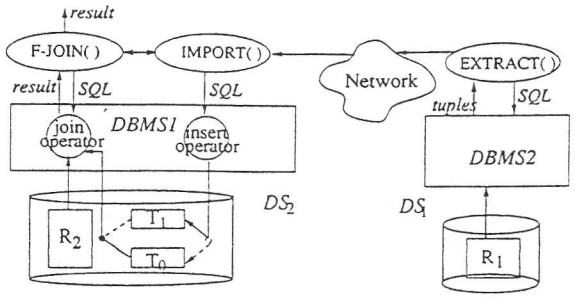


Figure 1: The execution of a fragmented join algorithm: fragments of R_1 are imported into buffers T_0 and T_1 alternately; the joins between R_2 and the buffers are executed as local SQL queries at the join site DS_2.

T), and (2) local join is normally I/O intensive, involving lots of disk operations. The result is that in either phase, either the disk drive (in the table staging phase) or the CPU (in the local join phase) is mostly idling and, thus, underutilized.

An immediate attempt one may think of to overlap both phases is to simulate the *ship-tuple* technique [3] – a tuple-level pipelined join processing technique that has been widely used in homogeneous distributed database systems. The difference now is that the pipeline has to be applied at the SQL level, no longer internally handled by the DBMS. It works as follows: As soon as a tuple of R_1, say t_i, arrives at the join site, an SQL query is issued to select from R_2 those tuples whose value on the join attribute matches that of t_i. A buffer is needed at the join site to hold the tuples extracted from R_1. Ship-tuple strategy provides a maximum degree of pipeline at the tuple granularity. Unfortunately, ship-tuple turns out to be a poor strategy because there is a substantial SQL overhead associated with each extracted outer tuple. As will be shown later, the accumulated SQL and database access overhead can easily offset the benefit of pipeline. In most cases, the accumulated overhead is prohibitively large.

3 The Fragmented Join Algorithm

The problem of the ship-tuple strategy is the wrong choice of the pipeline granularity – tuple, which causes the excessive SQL overhead. The fragmented join is devised to avoid such problem by using a granularity that is larger than tuples. The idea is to divide the outer relation, R_1, into a sequence of smaller, fixed-size tables called *fragments*. The original join then is carried out in terms of a sequence of joins between the fragments and the inner relation, R_2. The pipeline is realized at the granularity of fragments, rather than tuples. This reduces the number of SQL join queries executed at the join site and, thus, the total overhead.

Figure 1 shows the environment of the fragmented join. Subsequent fragments of R_1 are imported, alternately, into two temporary tables, T_0 and T_1, at the join site DS_2. When a buffer, say T_i ($i = 0$ or 1), is filled up with a fragment, an SQL query is issued against the DBMS at DS_2 to perform the join $T_i \bowtie R_2$. We call such a join an F-join. While T_i is engaged in the F-join, the other buffer T_{1-i} is used to import the next fragment from R_1. When both of the current F-join and import processes are finished, the two buffers switch roles (i.e. T_{1-i} becomes an operand of the next F-join and T_i becomes the placeholder for the next fragment to be imported). This procedure repeats until all fragments of R_1 are processed. The answer to the original query is simply the union of the results of all the F-joins.

Processes, `F-JOIN()` and `IMPORT()` are outlined in Figure 2. There are several issues worth mention. First, the processes must synchronize the access to the double buffers, ensuring that the faster process would not step into the buffer of the slower process that is still in progress. We have used the signal mechanism for this purpose in our implementation, though any mutual exclusion primitive can be used instead. The signal primitive `SendSig (x, y)` sends a signal `y` to process `x`; the primitive `WaitSig (x, y)` waits to receive a signal `y` from process `x`. Second, when an F-join

```
process IMPORT ( )  {
  const  m; /* fragment size */
  int  i = 0; /* buffer index initialization */

  while (more R1 tuples remain)
    Fill Ti with next m tuples from R1.
    SendSig (F-JOIN, buf_full);
    WaitSig (F-JOIN, buf_empty);
    i = 1 − i; /* switch buffer */
  end;
  SendSig (F-JOIN, kill); /* kill F-JOIN() */;
}

process F-JOIN ( )  {
  int  i = 1; /* buffer index initialization */

  while (true)
    SendSig (IMPORT, buf_empty);
    WaitSig (IMPORT, buf_full);
    i = 1 − i; /* switch buffer */
    Issue a local SQL query to perform Ti ⋈ R2;
    Upon receipt of the entire result, empty Ti;
  end
}
```

Figure 2: Outline of the two concurrent processes : `IMPORT()` imports the next fragment; `F-JOIN()` executes an local SQL join query

$T_i \bowtie R_2$ completes, the tuples in T_i must be discarded so that T_i can be used to import the next fragment of R_1. This can be done by issuing a "`DELETE * from` T_i" SQL command to delete all the tuples in T_i. Or one may simply drop the table and re-create a new one with the same name by issuing the following SQL commands in sequence: "`DROP TABLE` T_i", "`CREATE TABLE` T_i". We have chosen this approach as it appears to be more efficient than using the "DELETE *" command (Dropping and creating tables usually involve only system directory updates, while deleting all tuples of a table may require a visit to each tuple). The third issue, determination of a proper fragment size, is so important a factor on the performance that we discuss it separately in the next section.

4 Fragment Size Determination

The fragment size has a profound impact on the total query turnaround time. The smaller the fragment size, the greater degree of parallelism, but more SQL overhead would be incurred. On the other hard, a larger fragment size would incur less SQL overhead, but would result in a lower degree of parallelism. At one extreme, setting a fragment size of one tuple is equivalent to the ship-tuple strategy; at the other extreme, using a fragment the same size as the remote table degrades to the join-whole strategy. Finding the optimal fragment size is difficult due to the autonomy of the member database systems. Nonetheless, we have devised a constant-time heuristic algorithm that computes a "good" fragment size, based on a calibrated linear cost model. The complete algorithm is somehow tedious, due to several conditions needed to be considered. In the following we present only one case of the algorithm. Derivations for other cases can be reasoned in a similar manner [4].

Fix R_1 and R_2, and let variable x be the fragment size of R_1 (i.e. buffer T would contain x tuples). It is suggested in [5] that the cost of a join between two local tables, regardless of the join method being used, can be approximated by a generic expression, which is a linear function of the sizes of both tables. Accordingly, it was argued in [4] that the time it takes to evaluate the SQL query corresponding to $T \bowtie R_2$, where T contains x tuples, can be estimated as :

$$t_{F-join}(x) = a_0 + a_1 \cdot x,$$

in which a_0 and a_1 are some constants. Similarly, the time it takes to import a single fragment of size x into a buffer (temporary table) can be expressed as:

$$t_{imp}(x) = b_0 + b_1 \cdot x,$$

where b_0 and b_1 are some constants. Estimates of a_0, a_1, b_0, and b_1 can be obtained using a feedback technique similar to that proposed in [6], or using calibration methods such as those

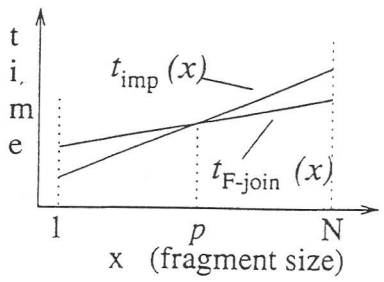


Figure 3: The case when $t_{F-join}(x)$ and $t_{imp}(x)$ intersects between $x = 1$ and $x = N$, where N is the cardinality of the outer relation (note $a_1 < b_1$ in this case.)

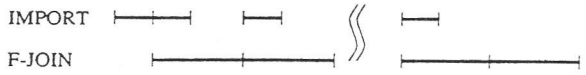


Figure 4: Progress overlap of processes F-JOIN and IMPORT along the timeline, when $t_{F-join}(x) > t_{imp}(x)$

proposed in [7, 8]. In the discussion that follows, we assume the estimates for these cost parameters have been obtained.

Figure 3 shows one possible case of the cost functions. In this case, $a_1 < b_1$ and the lines corresponding to $t_{F-join}(x)$ and $t_{imp}(x)$ cross over at $x = p$, where $1 \leq p \leq N$, N is the number of tuples in R_1. It is not hard to derive that

$$p = (a_0 - b_0)/(b_1 - a_1).$$

The algorithm finds the best fragment size by considering two disjoint ranges of x : $I_1 = [1, p]$ and $I_2 = [p, N]$. Here we restrict the discussion to the case of $x \in I_1$.

When $x \in I_1$, $t_{F-join}(x) > t_{imp}(x)$ holds. This means the import process (of the next fragment) always finishes before the F-join process (of the current fragment). Figure 4 overlaps the progress of both processes along the timeline. The discontinuity of the import process signifies its need to wait for the F-join process. The total turnaround time, therefore, is the time span between the beginning of the IMPORT process and the ending of the F-JOIN process. The turnaround time, as a function of the fragment size x, can be expressed

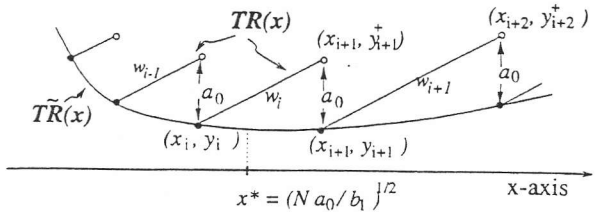


Figure 5: $TR(x)$: fragmented join turnaround time as a function of fragment size; $\tilde{TR}(x)$: an approximation to $TR(x)$

as

$$TR(x) = b_0 + b_1 x + \lceil \frac{N}{x} \rceil a_0 + a_1 N$$

Instead of finding the optimal x that minimizes $TR(x)$, which is extremely difficult, we used an approximate solution. Consider a simpler form of $TR(x)$ in which the ceiling is dropped:

$$\tilde{TR}(x) = b_0 + b_1 x + \frac{N a_0}{x} + a_1 N.$$

Figure 5 shows the curves of $TR(x)$ and $\tilde{TR}(x)$ under the real domain. It is not hard to see that $\tilde{TR}(x)$ is an "U-shape" function with a global minimum at

$$x^* = \sqrt{N a_0 / b_1},$$

which is obtained by solving the equation

$$\mathrm{d}\tilde{TR}(x)/\mathrm{d}x = 0.$$

$TR(x)$ is a piecewise linear function with each "piece" being a line segment of slope b_1. These segments are labeled from left to right as $w_0, w_1, \ldots, w_{N-1}$. The left/right endpoints of segment w_i are labeled as (x_i, y_i) and (x_{i+1}, y^+_{i+1}).

$\tilde{TR}(x)$ is a reasonable approximation to $TR(x)$ because $0 \leq TR(x) - \tilde{TR}(x) < a_0$, for $1 \leq x \leq N$, and a_0 is typically small. We can therefore seek an "good" fragment size as one that would minimize $\tilde{TR}(x)$. There are two immediate candidates :

$$\lfloor x^* \rfloor \text{ and } \lceil x^* \rceil.$$

Recall that x^* is the real domain solution for minimizing $\tilde{TR}(x)$. Now suppose that $x_i \leq$

$\lfloor x^* \rfloor < x_{i+1}$ for some i (i.e. $\lfloor x^* \rfloor$ is covered by segment w_i). An insight look reveals that $\lceil x_i \rceil$ is a better size than $\lfloor x^* \rfloor$ because $TR(\lceil x_i \rceil) \leq TR(\lfloor x^* \rfloor)$. $\lceil x_i \rceil$ can be computed as a function of x^*, $\rho(x^*)$, where $\rho(x)$ is defined as follows:

$$\rho(x) = \lceil N/\lceil N/\lfloor x \rfloor \rceil \rceil.$$

We then select between $\rho(x^*)$ and $\lceil x^* \rceil$ the one that yields the smaller value of $TR(x)$. We express the selection as $\phi(x^*)$, where

$$\phi(x) = \begin{cases} \rho(x) & \text{if } TR(\rho(x)) \leq TR(\lceil x \rceil) \\ \lceil x \rceil & \text{otherwise} \end{cases}$$

Note that when a tie occurs in the above expression, the smaller fragment size, $\rho(x^*)$, is favored.

Finally, it may happen that $\phi(x^*)$ falls outside the range of I_1. If $\phi(x^*) < 1$, we choose 1 as the fragment size; if $\phi(x^*) \geq p$, we choose $\rho(p)$ as the fragment size. This is based on the observation that $TR(x)$ is an "U-shape" curve in general. To summarize, the fragment size selected for the sub-range I_1, denoted $x^*_{I_1}$, is computed as:

$$x^*_{I_1} = \max\left(\min(\phi(\sqrt{Na_0/b_1}), \rho(\frac{a_0 - b_0}{b_1 - a_1})), 1\right).$$

A complete algorithm that includes all other cases not covered here can be found in [4].

5 Performance Evaluation

We have implemented the fragmented join algorithm in a multidatabase environment that contains two autonomous ORACLE 7 servers. One server runs on a Sun Ultra-2 machine; the other runs on a Sun Sparc 10 workstation. Both workstations reside within the premise of an Ethernet network. The programs were written in C, and used embedded SQL to interact with the ORACLE servers. The "cursor" mechanism is used to fetch the resultant tuples returned by the server. Communications between the gateway programs (`IMPORT()` at the join site and `EXTRACT()` at the outer relation site) are implemented using Unix socket APIs.

relation	cardinality	tuple size (bytes)
R_1	100 –16,000	40
R_2	12,000	150

Table 1: Data statistics of table R_1 and R_2

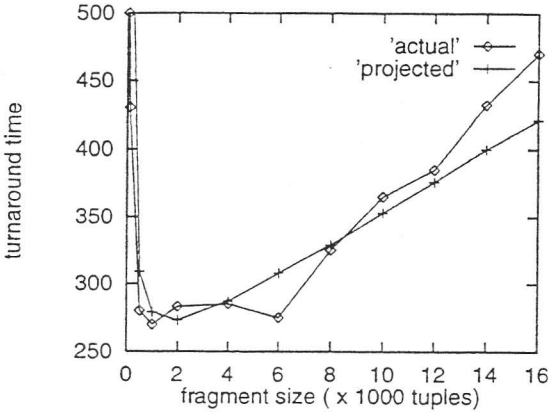


Figure 6: Comparison of actual and projected turnaround time as a function of fragment size.

Two synthetic tables were created and imported into separate servers. Table 1 shows the data statistics of the tables. We varied the cardinality (number of tuples) of R_1, ranging from 100 to 16,000 tuples, in certain experiment settings. The join selectivity between the tables was chosen so that the number of tuples in the result is about 30% of the cardinality of R_1. The values of the join attribute of R_1 is randomly distributed over all tuples of the table. In all queries, R_1 is the outer relation. The cost parameters for the F-join and import processes (see Section 4) were estimated a prior by running some sample queries.

5.1 Validation of the Cost Model

The accuracy of the cost model is essential to the effectiveness of the fragment size determination algorithm. To validate this, we set R_1 to 16,000 tuples, and tested with different fragment sizes. For each fragment size, we gauged the actual query turnaround time (measured in wall-clock time), and compared it with the projected turnaround time that is calculated using the cost functions. Figure 6 shows the result. The figure shows that the projected time is close to and exhibits the same

$\|R_1\|$	100	200	400	1000	2000	4000
frag. size	100	100	200	334	500	800

6000	8000	10000	12000	14000	16000
1000	1143	1250	1500	1556	1600

Table 2: Fragment sizes calculated for different outer table sizes

trend as the actual time when the fragment size varies. We applied the fragment size determination algorithm using the data and cost parameters, and obtained a fragment size of 1,600 tuples for R_1. In the figure, this size falls just around the corner where the actual turnaround time drops to the minimum. The sensitivity of the turnaround time with respect to the fragment size, as evidenced by the steep "U-shape" curves in the figure, has justified the use of the fine-tuned fragment size determination algorithm.

5.2 Comparison with join-whole and ship-tuple Strategies

In another experiment setting, we compared the performance among the fragmented join algorithm (FJ), the join-whole (JW) strategy, and the ship-tuple (ST) strategy. The size of the outer relation (R_1) was varied from 100 to 16,000 tuples. In the ST algorithm, a selection query was issued against R_2 for each tuple extracted from R_1. The selection is based on the value of the join attribute of the R_1-tuple. No temporary tables were created and no insertions were performed. For the FJ algorithm, Table 2 shows the fragment sizes computed by the fragment size determination algorithm, for different sizes of R_1.

Figure 7 compares the turnaround time among the three algorithms. The performance of ST is completely unacceptable. Such is true even for a small R_1 that contains as few as 1000 tuples. The fragmented join constantly outperforms JW, with an improvement of up to 40%. The performance improvement of FJ over JW increases as the size of R_1 grows. We found that this effect is in part attributed to the buffering mechanism of the DBMS at the

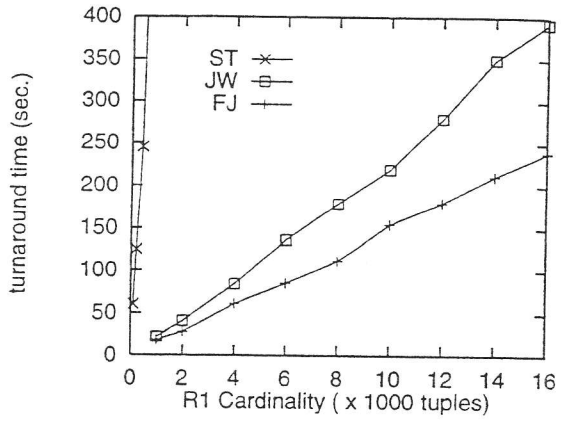


Figure 7: Turnaround time comparison among fragmented join (FJ), join-whole (JW) and ship-tuple (ST), for various outer relation sizes.

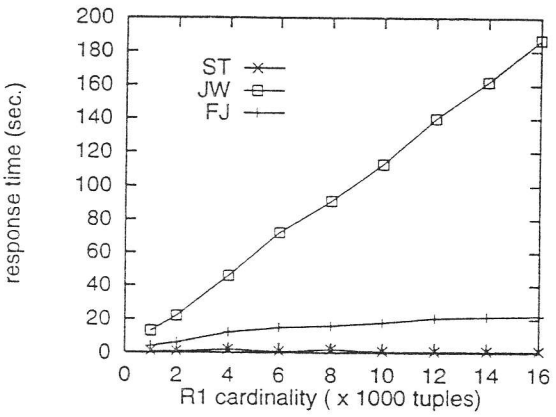


Figure 8: Response time comparison among fragmented join (FJ), join-whole (JW) and ship-tuple (ST), for various outer relation sizes.

join site. For smaller R_1, both strategies have a good chance of caching the table in the memory buffer. When R_1 is large, JW is unable to cache the entire table in memory. As a result, extra I/O operations are needed to write and read the table from the disk. The fragmented join does not have this problem because it decomposes R_1 into smaller fragments, each of which may still fit into the memory buffer.

Figure 8 compares the response time – the wall-clock elapsed time between the submission of the original query and the availability of the first result tuple. The fragmented join algorithm has a much lower and flatter response time curve than the join-whole strategy. This

is a direct consequence of using smaller fragments. The ship-tuple yields the least response time: the first result tuple would be produced shortly after the first tuple of R_1 has arrived at the join site. However, the prohibitively long delay of the turnaround time makes ship-tuple a forbidden choice. Finally, we note that the fragmented join algorithm consumes less storage space (twice the fragment size) than does the join-whole strategy (which requires a space for the entire outer table). This can be validated by comparing the size of R_1 and the corresponding fragment size, as shown in Table 2.

6 Conclusions

The traditional strategy to evaluate a cross-database join query in a loosely-coupled mulit-database system suffers a long delay due to the sequential execution of table staging and join operation. In this paper we have proposed a new technique called fragmented join, which reduces the delay by overlapping table staging with local join at the SQL level. A fragment size determination algorithm is also devised to avoid excessive SQL and database access overhead, which would otherwise offset the performance gain of the pipelined processing. Experimental results show that the fragmented join produces shorter query turnaround time and response time than the sequential execution, and requires much less storage space for temporary tables.

Presently, we are expanding the fragmented join technique to handle multi-way cross-database joins. An adaptive method to estimate the cost parameters on-the-fly is also under investigation. This eliminates the need of running sample queries to calibrate the cost parameters, and allows the fragment size to self-adjust based on the feedback of cost statistics. Finally, we are interested in applying the concept of fragmented joins to more complex queries (such as those containing nested queries, aggregate functions, or membership operators), and to other multidatabase join methods (such as those proposed in [9, 10]).

Acknowledgment

We thank Ruibiao Qiu for implementing an early version of the fragmented join programs and testbed.

References

[1] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3), 1990.

[2] A.R. Hurson, M.W. Bright, and S. Pakzad, editors. *Multidatabase systems : an advanced solution for global information sharing.* IEEE Press, 1994.

[3] L.F. Mackert and G.M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Procs. of the 12th Intl. Conf. on Very Large Data Bases*, 1986.

[4] C.-M. Chen, R. Qiu, and N. Rishe. A study on applying fragmented joins to multidatabase queries. Technical report, School of Computer Science, Florida International University, Miami FL, 1997.

[5] H. Boral R. Krishnamurthy and C. Zaniolo. Optimization of nonrecursive queries. In *Procs. of the 12th Intl. Conf. on VLDB*, pages 128–137, Kyoto, Japan, 1986.

[6] C.-M. Chen and N. Roussopoulos. Adaptive selectivity estimation using query feedback. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1994.

[7] W. Du, R. Krishnamurthy, and M.-C. Shan. Query optimization in heterogeneous DBMS. In *Procs. of the 18th VLDB Conference*, 1992.

[8] Q. Zhu and P.-A. Larson. Building regression cost models for multidatabase systems. In *Procs. of the 4th Intl. Conf. on Parallel and Distributed Information Systems*, 1996.

[9] U. Dayal. Query processing in multidatabase system. In Kim, Batory, and Reiner, editors, *Query Processing in Database Systems.* Springer-Verlag, 1985.

[10] A. Chen. Outerjoin optimization in multidatabase systems. In *Procs. of the Distributed and Parallel Database Systems*, Dublin, Ireland, 1990.