98-PQ

A Pipelined Query Processor for Distributed Multi-Databases *

Chung-Min Chen Ruibiao Qiu Naphtali Rishe
School of Computer Science
Florida International University
Miami, FL 33199

This paper describes a pipelined processing technique for queries involve data from distributed, autonomous relational databases. We highlight the basic idea, present a preliminary result, and sketches the planned work towards the development of a more complete multi-database query processor based on the proposed idea.

Accessing Distributed Autonomous Databases The advance of relational database technology in the past decade has made efficient storage and manipulation of massive data easier than ever. Accompanied with the wide adoption of relational database management systems (DBMS) is the diverse choices in commercial DBMS products. Due to certain technical or strategic considerations, a company may need to use several DBMSs from different vendors. These DBMSs are often installed on separate machines (servers) which are connected through a LAN. This results in a loosely-coupled multi-database environment: most of the time the DBMS servers operate on their own, but from time to time they have to cooperate with each other to handle queries that require data from more than one server (such queries are known as *global queries*). Efficient global query processing has been one of the key research areas in multi-databases.

Autonomous database servers in a multi-database environment can only communicate with each other through a high-level query interface: SQL (Structured Query Language), the standard query language for relational databases. The implication is that to evaluate a global query, the query must be first translated into a sequence of sub-queries in SQL format, which are then scheduled and submitted to respective database servers for execution. Consider for example a *join* query $R_1 \bowtie R_2$ where tables R_1 and R_2 reside at database servers DB_1 and DB_2 respectively. To perform the join, we must first retrieve R_1 from DB_1, send it to the site of DB_2, and import it into a temporary table, say T, in DB_2 (or the other way around by moving R_2). Since all operations must be carried out at the SQL level, the local join query $T \bowtie R_2$ can not be issued until R_1 is completely imported into T. Thus, the total turnaround time is the sum of two sequential steps: table staging [1] (denoted $T \leftarrow R_1$) and local join ($T \bowtie R_2$). The sequential delay could become intolerable when the join query involves large or many tables.

Pipelining Join Queries with Double Buffers We claim that a global join query can be sped up by overlapping the table staging and local join operations. This is based on the following observations: (1) table staging is usually communication and CPU intensive (due to the overhead of network protocol stacks, data format conversion, and the repetative SQL INSERT commands needed to populate the temporary table T with the records received from R_1), and (2) local join is disk IO intensive. Thus, in a sequential

*This research was supported in part by NASA (under grants NAGW-4080, NAG5-5095, and NRA-97-MTPE-05), NSF (CDA-9711582, IRI-9409661, and HRD-9707076), ARO (DAAH04-96-1-0049 and DAAH04-96-1-0278), DoI (CA-5280-4-9044), NATO (HTECH.LG 931449), and State of Florida.

[1] Table staging may be performed by overlapping communication and importing.

execution, the disk is partly idle during table staging, while the CPU is mostly idle during local join. If we manage to overlap these two steps, better resource utilization and thus shorter query turnaround time can be achieved. However, since all operations must be handled at the SQL level, applying pipeline at the granularity of tuples (records) requires the execution of an INSERT and a JOIN command at the DB_2 server for each tuple received from R_1. The excessive SQL statement processing and database access costs may well offset the time saved by pipelined processing.

To make pipeline processing beneficial, we have devised a pipelined algorithm called *fragmented join* which avoids excessive SQL and database access overhead. The idea is to divide R_1 into a number of smaller, fixed-size tables (called *fragments*), and pipeline data staging and local join at the granularity of fragments. To achieve parallelism, subsequent fragments are imported, in turn, into two temporary tables, T and T' (the double buffers). While a buffer is engaged in the local join operation with R_2, the other buffer can be used to import the next fragment from R_1. These two buffers exchange roles alternately for subsequent fragments. Thus, if R_1 is divided into n fragments as $R_1 = \cup_{i=1}^{n} R_{1,i}$, and at a certain point of time buffer T is holding fragment $R_{1,i}$, then the local join query $T \bowtie R_2$ and the staging of next fragment $T' \leftarrow R_{1,i+1}$ are performed simultaneously. The final result is simply the union of all the sub-results since $R_1 \bowtie R_2 = \cup_{i=1}^{n}(R_{1,i} \bowtie R_2)$.

The total turnaround time of a fragmented join depends on the fragment size. To determine a good fragment size, we have devised a constant-time heuristic algorithm that computes a fragment size based on certain table statistics (which are attainable from most database servers) and a calibrated linear cost model for both insert and join operations. We have implemented the fragmented join algorithm in a multi-database environment that contains two autonomous ORACLE 7 servers. A sample experimental result is shown in the table below which compares query turnaround time (in seconds) of the fragmented join and a sequential algorithm, at different sizes of R_1 (in unit of 1,000 tuples). Each R_1 tuple occupies 40 bytes. Table R_2 contains 12,000 tuples, with a tuple size of 150 bytes. Using a self-computed fragment size, the fragmented join algorithm is able to shorten the turnaround time of the sequential algorithm by 30-40% in most cases. The last row shows the fragment size (in unit of 1,000 tuples) calculated and used by the fragmented join algorithm.

R_1 size	0.8	1.6	2.4	3.2	4.0	4.8	5.6	6.4	7.2	8.0
fragmented	16	21	39	47	61	70	76	90	99	112
sequential	18	34	48	68	86	104	120	145	169	179
frag. size	0.27	0.53	0.60	0.64	0.80	0.96	0.93	1.07	1.02	1.14

Summary In this paper we have described the basic idea of the fragmented join algorithm which is aimed to reduce turnaround time for global queries in a distributed multi-database environment. In addition to shorter query turnaround time, the fragmented join has two other advantages. First, it takes much shorter time than the sequential strategy to produce the first tuple. The improvement is about $|R_1|/|R_{1,i}|$ times faster. Second, the fragmented join algorithm requires only a temporary disk space of twice the fragment size. This could be a significant save over the sequential strategy which requires a temporary disk space for the entire table R_1. Presently, we are undergoing the implementation of a first proof-of-concept multi-database prototype that incorporates the pipelined technique described above. To make the fragmented join algorithm practically useful, we are expanding the algorithm to handle join queries that involve more than two tables. This requires formation of a multi-level pipeline tree and determination of the fragment size at each join node. We are also exploring adaptive techniques that would adjust the fragment sizes dynamically during long query execution based on real-time feedback.