An Alternative Method to Remove Duplicate Tuples Resulting from Operations in the Relational Algebra in a Cube-Connected Multicomputer System

NICHOLAS T. KARONIS*†

Preprint ANL/MCS-P604-0796

Abstract - The problem of performing database operations on parallel architectures has received much attention, both as applied and theoretical areas of research. Much of the attention has been focused on performing these operations on distributed-memory architectures, for example, a hypercube. Algorithms that perform, in particular, relational database operations on a hypercube typically exploit the hypercube's unique interconnectivity to not only process the relational operators efficiently but also perform dynamic load balancing. Certain relational operators (e.g., projection and union) can produce interim relations that contain duplicate tuples. As a result, an algorithm for a relational database system must address the issue of removing duplicate tuples from these interim relations. The algorithms accomplish this by compacting the relation into hypercubes of smaller and smaller dimensions. We present an alternative method for removing duplicate tuples from a relation that is distributed over a hypercube by using the embedded ring found in every hypercube. Through theoretical analysis of the algorithm and empirical observation, we demonstrate that using the ring to remove the duplicate tuples is significantly more efficient than using the hypercube.

Index Terms - Distributed algorithm, hypercube, relational algebra, relational database systems.

1 Introduction

Database processing has long been an area of intense theoretical and applied research. Much, if not all, of the recent work has focused not on creating new database models, but rather on improving

*Northern Illinois University, Computer Science Department, DeKalb, IL.

†This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

performance by using existing models. In the early days, special-purpose hardware was built to improve performance. These database machines were aimed at relieving I/O and other bottlenecks found in conventional machines [4, 23]. More recent efforts have focused on multiprocessor systems, for example, DIRECT [23], GAMMA [5], GRACE [13], MIRDM [17], NON-VON [10], SM3 [3], and TERADATA [24]. Many of these systems use parallel algorithms to enhance database processing.

One parallel architecture in particular, the hypercube, has been an architecture of choice for database processing. Several parallel computers based on a hypercube's connectivity currently exist, for example, Caltech's MARK III [16], the Floating Point Systems T-series machines [6], Intel iPSC/II [19], and the NCUBE/10 [9].

An algorithm was presented [1, 7, 2] to perform relational database operations in a distributed processing environment. It is described as a cube-connected multicomputer system whereby the processes are connected in a hypercube, they communicate via message passing, and the tuples of the relations are horizontally partitioned over the set of processes.

Some of the relational operators (e.g., projection and union) produce interim relations that potentially contain duplicate tuples that must be removed during a clean-up phase at the end of these operations. Accordingly, the algorithm describes how to remove duplicate tuples from these relations by using an operation called Relation Compaction (RC), which exploits the hypercube's unique topology.

We present an alternative algorithm for removing duplicate tuples. The algorithm we present uses the ring embedded in all hypercubes [14], thereby reducing the overall execution time. Additionally, the algorithm we present leaves the tuples more evenly distributed (balanced) over the processes. A balanced set of tuples is an *imperative* for proper load balancing [1, 7].

2 The Hypercube Remove-Duplicates Algorithm

Here we describe the algorithm presented in [1, 7, 2].

Figures 1a-e show an example 2-D hypercube with relation R having $4n$ tuples. We assume that this relation is the result of a recently performed projection and therefore might have duplicate

tuples that must be removed. For this example, we assume that there are no duplicate tuples. The tuples start evenly partitioned (balanced) over the set of processes.

Figure 1a: Remove local duplicates.

Figure 1b: Send n tuples.

Figure 1c: Remove from received.

Figure 1d: Send $2n$ tuples.

Figure 1e: Remove from received.

Step 1 of the algorithm requires each process to remove all duplicate tuples from its local set (Figure 1a). Processes 1 (01) and 3 (11) then send their tuples to processes 0 (00) and 2 (10), respectively (Figure 1b). Processes 0 and 2 remove those tuples *from the newly arrived set of tuples* that already exist in their local copies (Figure 1c). Process 2 sends its set of $2n$ tuples to process

0 (Figure 1d). Finally, process 0 removes those tuples from the newly arrived set of $2n$ tuples that already exist in its set of $2n$ tuples (Figure 1e). The entire relation R now resides entirely in process 0 with all the duplicate tuples removed. R must now be redistributed over the set of processes for proper load balancing.

In analyzing the hypercube remove-duplicates algorithm we estimate the computation time by counting the number of comparisons the algorithm requires to remove duplicates from a relation R. For this analysis we assume that R is initially balanced over the set of processes and that no duplicate tuples exist in R. We express the number of comparisons as a binary function $C_h(N, d)$ where N is the cardinality of the original relation R and d is the dimension of the hypercube.

During the first computation step of the algorithm (Figure 1a) each process removes all duplicate tuples from its local set. Recall that relation R, having N tuples, is evenly partitioned (balanced) over the set of 2^d processes, so each process has $\frac{N}{2^d}$ tuples. Therefore, the number of comparisons to perform this local check is

$$\begin{aligned} & \frac{(\frac{N}{2^d}-1)(\frac{N}{2^d})}{2} \\ = & \frac{(\frac{N-2^d}{2^d})(\frac{N}{2^d})}{2} \\ = & \frac{\frac{N(N-2^d)}{2^{2d}}}{2} \\ = & \frac{N(N-2^d)}{2^{2d+1}}. \end{aligned} \tag{1}$$

During the next computation step (Figure 1c) a set containing $\frac{N}{2^d}$ tuples is compared against another of equal size, requiring $(\frac{N}{2^d})^2$ comparisons. The final computation step (Figure 1e) compares a set containing $\frac{2N}{2^d}$ tuples against another of equal size, requiring $(\frac{2N}{2^d})^2$ comparisons. In general, for hypercubes of arbitrary dimension, this process continues until process 0 compares half of the relation against the other half, or a set containing $\frac{N}{2}$ tuples against another of equal size requiring $(\frac{N}{2})^2$ comparisons. Thus, in general, the number of comparisons required after each process removes duplicates from their local relations is

$$\begin{aligned} & (\frac{N}{2^d})^2 + (\frac{2N}{2^d})^2 + (\frac{4N}{2^d})^2 + (\frac{8N}{2^d})^2 + ... + (\frac{N}{2})^2 \\ = & (\frac{2^0N}{2^d})^2 + (\frac{2^1N}{2^d})^2 + (\frac{2^2N}{2^d})^2 + (\frac{2^3N}{2^d})^2 + ... + (\frac{2^{d-1}N}{2^d})^2 \end{aligned}$$

$$= \sum_{i=0}^{d-1} (\frac{2^i N}{2^d})^2$$

$$= (\frac{N}{2^d})^2 \sum_{i=0}^{d-1} 2^{2i}.$$

Consider the general form of the summation term in the above formula:

$$\begin{aligned} S_n &= \sum_{i=0}^{n} 2^{2i} \\ S_n + 2^{2(n+1)} &= 1 + \sum_{i=0}^{n} 2^{2(i+1)} \\ &= 1 + 2^2 \sum_{i=0}^{n} 2^{2i} \\ &= 1 + 4S_n. \end{aligned}$$

Thus,

$$S_n = \frac{2^{2(n+1)} - 1}{3}.$$

Solving this summation for a hypercube of dimension d, we have:

$$S_{d-1} = \frac{2^{2d} - 1}{3}.$$

Therefore the number of comparisons after the initial local check is

$$\begin{aligned} (\frac{N}{2^d})^2 \sum_{i=0}^{d-1} 2^{2i} &= (\frac{N}{2^d})^2 S_{d-1} \\ &= (\frac{N}{2^d})^2 (\frac{2^{2d} - 1}{3}). \end{aligned} \quad (2)$$

This gives us the function $C_h(N, d)$ as simply the sum of formulae (1) and (2). It is the number of comparisons needed for the hypercube remove-duplicates algorithm as a function of the cardinality of the relation (N)and the dimension of the hypercube (d):

$$C_h(N, d) = \frac{N(N - 2^d)}{2^{2d+1}} + (\frac{N}{2^d})^2 (\frac{2^{2d} - 1}{3}).$$

3 The Ring Remove-Duplicates Algorithm

Here we present an alternative algorithm to remove duplicates that views the hypercube as a ring.

Consider the same example with relation R having $4n$ tuples evenly partitioned over a 2-D hypercube set of processes. Each process has n tuples. Again, for this example, we assume that there are no duplicate tuples in R.

Step 1 of the algorithm requires each process to remove duplicate tuples from its local set (Figure 2a). Each process then sends a *copy* of its local set to its neighbor in the ring (Figure 2b). Those processes that receive a copy *that originated in a process with an id greater than itself* may remove duplicate tuples from the received copy (Figure 2c). In this example, processes 0 and 2 remove duplicate tuples from sets n_2 and n_3, respectively, while processes 1 and 3 are idle. Each process then sends its *mobile set* of tuples to its neighbor in the ring (Figure 2d). Processes 0 and 1 remove duplicates from their sets n_3 and n_2, respectively (Figure 2e). Again, their id's are less than the id's of the processes that *originated* their copy. Processes 2 and 3 are idle.

Figure 2a: Remove local duplicates.

Figure 2b: Send n tuples.

Figure 2c: Selected remove from received.

Figure 2d: Send n tuples.

Figure 2e: Selected remove from received.

This process is repeated until all the mobile sets simultaneously arrive at the node that is one short of their origins, and then the mobile sets are checked (by the appropriate nodes) for duplicates one final time. These mobile (possibly modified) sets are kept, and the original sets are discarded. The retained sets of tuples represent the original relation R with all duplicate tuples removed virtually balanced over the set of processes.

In analyzing the ring remove-duplicates algorithm we again estimate the computation time by counting the number of comparisons the algorithm requires to remove duplicates from a relation R. For this analysis we again assume that R is initially balanced over the set of processes and that no duplicate tuples exist in R. We express the number of comparisons as a binary function $C_r(N, d)$, where N is the cardinality of the original relation R and d is the dimension of the hypercube.

The first step of this algorithm also requires each process to remove duplicate tuples from its local set of $\frac{N}{2^d}$ tuples. As we demonstrated earlier in formula (1), the number of comparisons is

$$\frac{N(N-2^d)}{2^{2d+1}}.$$

During the remaining steps of the algorithm, selected processes compare sets containing $\frac{N}{2^d}$ tuples against other sets of equal size, requiring $(\frac{N}{2^d})^2$ comparisons each time. This procedure is done $2^d - 1$ times. The number of comparisons required for this portion of the algorithm is therefore

$$(2^d - 1)(\frac{N}{2^d})^2. \tag{3}$$

This gives us the function $C_r(N, d)$, which is simply the sum of formulae (1) and (3). It is the number of comparisons needed for the ring remove-duplicates algorithm as a function of the cardinality of the relation (N) and the dimension of the hypercube (d):

$$C_r(N, d) = \frac{N(N-2^d)}{2^{2d+1}} + (2^d - 1)(\frac{N}{2^d})^2.$$

4 Comparing the Two Algorithms

Having described the algorithms, we are now prepared to compare them. We choose to compare first their computations and then their communications.

4.1 Analysis of Computation

As a method to compare the two algorithms we consider the ratio of the two functions

$$\begin{aligned}
g(N,d) &= \frac{C_h(N,d)}{C_r(N,d)} \\
&= \frac{\frac{N(N-2^d)}{2^{2d+1}} + (\frac{N}{2^d})^2(\frac{2^{2d}-1}{3})}{\frac{N(N-2^d)}{2^{2d+1}} + (2^d-1)(\frac{N}{2^d})^2} \\
&= \frac{\frac{N(N-2^d)}{2^{2d+1}} + (\frac{N}{2^d})^2(\frac{2^{2d}-1}{3})}{\frac{N(N-2^d)}{2^{2d+1}} + (2^d-1)(\frac{N}{2^d})^2}(\frac{2^{2d}6}{2^{2d}6}) \\
&= \frac{3N(N-2^d) + 2N^2(2^{2d}-1)}{3N(N-2^d) + 6N^2(2^d-1)} \\
&= \frac{(3+2^{2d+1}-2)N + (-3)2^d}{(3+2^d6-6)N + (-3)2^d} \\
&= \frac{(2^{2d+1}+1)N - 2^d3}{(2^{d+1}3-3)N - 2^d3}.
\end{aligned}$$

We inspect g's behavior as N and d get large. First N,

$$\begin{aligned}
\lim_{N\to\infty} g(N,d) &= \lim_{N\to\infty} \frac{(2^{2d+1}+1)N - 2^d3}{(2^{d+1}3-3)N - 2^d3} \\
&= \frac{2^{2d+1}+1}{2^{d+1}3-3}.
\end{aligned}$$

Now d,

$$\lim_{d\to\infty} \frac{2^{2d+1}+1}{2^{d+1}3-3} = \frac{2^d}{3}.$$

Figure 3 depicts g's behavior.

Figure 3: Plot of $g(N,d) = \frac{C_h(N,d)}{C_r(N,d)}$.

As we increase N, the cardinality of the R, we observe that the ratio remains largely unchanged. However, we do note g's acute sensitivity ($\frac{2^d}{3}$) to increases in d, the dimension of the hypercube. We observe that the ring remove-duplicates algorithm performs significantly better (with respect to the number of comparisons) than the hypercube remove-duplicates algorithm. As our analysis shows and Figure 3 illustrates, the hypercube remove-duplicates algorithm requires roughly 300 times as many comparisons as the ring remove-duplicates algorithm on a 10-D hypercube.

The price the ring remove-duplicates algorithm must pay is in communication time. As we discover in the following sections, the benefit in the reduction of computation time far outweighs the increased cost of communication.

4.2 Analysis of Communications

In each step of both algorithms there is a computation part and a communication part. For each algorithm we derive a binary linear function

$$M(N, d) = at_0 + bt$$

that estimates the time it takes for all the algorithm's communications. In this function, N is the cardinality of the original relation R, d is the dimension of the hypercube, t_0 is the time it takes to initiate the transfer of a message, and t is the time it takes to transmit or receive a tuple. For simplicity we assume that all the data that must be transferred in any given step will always fit in a single message buffer. Again, for simplicity, we assume that there are no duplicate tuples in R.

4.2.1 Messages in the Hypercube Remove-Duplicates Algorithm

In the first communication of the hypercube algorithm, $\frac{N}{2^d}$ tuples are transferred; in the second, $\frac{2N}{2^d}$ tuples are transferred; and in the last $\frac{N}{2}$ tuples are transferred. Thus, for a hypercube of dimension d, we see

$$\begin{aligned} & \frac{N}{2^d} + \frac{2N}{2^d} + \ldots + \frac{N}{2} \\ = \quad & \frac{N}{2^d} + \frac{2N}{2^d} + \ldots + \frac{2^{d-1}N}{2^d} \end{aligned}$$

$$
\begin{aligned}
&= \sum_{i=0}^{d-1} \frac{2^i N}{2^d} \\
&= (\frac{N}{2^d}) \sum_{i=0}^{d-1} 2^i \\
&= (\frac{N}{2^d})(2^{(d-1)+1} - 1) \\
&= (\frac{N}{2^d})(2^d - 1) \\
&= N - \frac{N}{2^d}
\end{aligned}
$$

tuples being transferred over d messages. This gives us the communications time function for the hypercube algorithm

$$M_h(N, d) = dt_0 + (N - \frac{N}{2^d})t.$$

4.2.2 Messages in the Ring Remove-Duplicates Algorithm

The analysis for the ring is not as straightforward as for the hypercube. Message passing, in a worst case scenario, prohibits a single process from transmitting and receiving a message *at the same time*. In the hypercube remove-duplicates algorithm, a communicating process in a single communication step is called upon either to transmit *or* to receive a message, not both. That is not the case in ring algorithm. In each communication step each process must transmit *and* receive a set of tuples. For this reason each communication step has two phases.

In each of the phases $\frac{N}{2^d}$ tuples are transferred. Since there are two phases in each communication step, each step transfers $\frac{2N}{2^d}$ tuples over two messages. There are $2^d - 1$ such communication steps (recall that the mobile set ends up one node short of its origin). Thus, the total number of tuples transferred in this algorithm is

$$(2^d - 1)(\frac{2N}{2^d}) = 2N - \frac{2N}{2^d}$$

over $2(2^d - 1) = 2^{d+1} - 2$ messages. This gives us the communications time function for the ring algorithm

$$M_r(N, d) = (2^{d+1} - 2)t_0 + (2N - \frac{2N}{2^d})t.$$

4.2.3 Comparison of Communications

As we inspect M_h and M_r, we observe that the ring algorithm transmits roughly twice the number of tuples over an exponentially greater number of messages than does the hypercube algorithm. At first glance, this might appear to be troublesome. In practice, it is not.

Much of the increased communication time for the ring method can be attributed to the exponentially increasing, with respect to d, number of messages that must be sent, in other words, not an increase in the *amount of data*, but rather, an increase in the *number of messages*.

We believe that total increase in communication found in the ring method, both in its increase in the number of messages and its increase in the amount of data, is more than compensated for by its reduction in computation. We view the increase in the amount of data being sent/received as insignificant when compared with the reduction of computation, and although the increase in the number of messages being sent is significant, we do not believe it to be problematic because the time it takes to compare two tuples is much larger than the time it takes to *initiate* a communication. These assertions are empirically confirmed in the next section.

5 Experimentation

We implemented both algorithms and compared their execution times. This section first describes the environment in which we performed our experiments and then presents our results along with our analysis.

5.1 Description of Environment

In each experiement we constructed a relation with no duplicate tuples and measured the performance of each method as they attempted to remove duplicates. The relations were characterized by three parameters: their cardinality (number of tuples) $N > 0$, their key degree (number of attributes participating in the primary key) $p > 0$, and their non-key degree (number of attributes that do not participate in the primary key) $p' \geq 0$.

The attributes, all with integer domains, were named $A_1, A_2, \ldots, A_p, A_{p+1}, \ldots, A_{p+p'}$. The first p attributes composed the primary key while the remaining p' attributes composed the rest of the

tuple. The entire relation was initialized with 0's. Then, to assure that there were no duplicate tuples, each tuple t_i $(1 \leq i \leq N)$ was assigned the value i to attribute A_p. For example, Figure 4 is the relation characterized by $N = 4$, $p = 3$, and $p' = 2$.

R

A_1	A_2	A_3	A_4	A_5
0	0	1	0	0
0	0	2	0	0
0	0	3	0	0
0	0	4	0	0

Figure 4: Example relation $N = 4$, $p = 3$, $p' = 2$.

Two tuples are identical iff they have the same primary key values. We compare two tuples for equality by comparing each of the values in their primary keys, first in A_1, then A_2, and so on up to and including A_p. If, along the way, even one of the values of these attributes differs, the comparison is stopped and the two tuples are declared distinct.

Characterizing the relation in conjunction with testing for equality in this way provides us with an experimental environment where we can compare the two methods while tuning these three parameters. If we wish to increase the amount of time to compare two tuples, we simply increase p. If we wish to increase the amount of data being communicated without increasing the comparison time, we simply increase p' and/or N.

Both methods were implemented by using Argonne National Laboratory's implementation of MPI [8, 20, 21, 22]. They were executed on the 128-processor IBM POWERparallel System at Argonne National Laboratory running AIX version 3.2.4 and the AIX Parallel Environment [11], which includes IBM's message-passing library MPL. Argonne's implementation of MPI on their POWERparallel System calls MPL directly.

The machine was configured with 8 SP1 16-node frames where each node was capable of 125 MFlops and was equipped with 128 MBytes of memory and 1 GByte of disk space. The nodes communicated over a four-stage Omega switch rendering all nodes equidistant from each other.

The timing data was collected by using a timing library called UTP, which was written by Dave Kohr at Argonne National Laboratory's Mathematics and Computer Science Division. The version

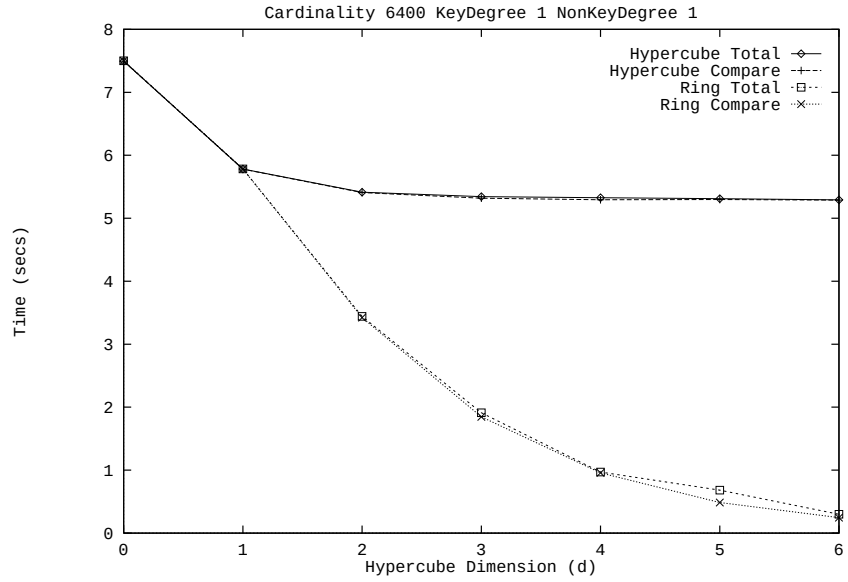

Figure 5: First experiment.

of UTP that runs on Argonne's POWERparallel System is written in IBM's RS/6000 assembler and accesses a continuously running hardware clock with microsecond resolution. Running programs on Argonne's POWERparallel System provides exclusive access to the nodes, so accumulating wall-clock time is not entirely inappropriate.

In both algorithms node 0 does as much or more work than all the other nodes. Therefore, in all our experiments the timing data was collected on node 0 only.

5.2 Experimental Data

For our first experiment we constructed a relation where $N = 6400$, $p = 1$, and $p' = 1$. We measured the performance of both methods on this relation on hypercubes of different dimensions ranging from 0 through 6. In measuring the performance we accumulated two different times for each method: total execution time and time for comparing tuples only. Figure 5 is a plot of all four times as a function of hypercube dimension (d). Again, all timing data was accumulated on node 0 only.

5.2.1 Analysis

The first thing we note about the graph is that the lines for execution time and comparison time for each method are virtually indistinguishable. This illustrates that the execution time is completely

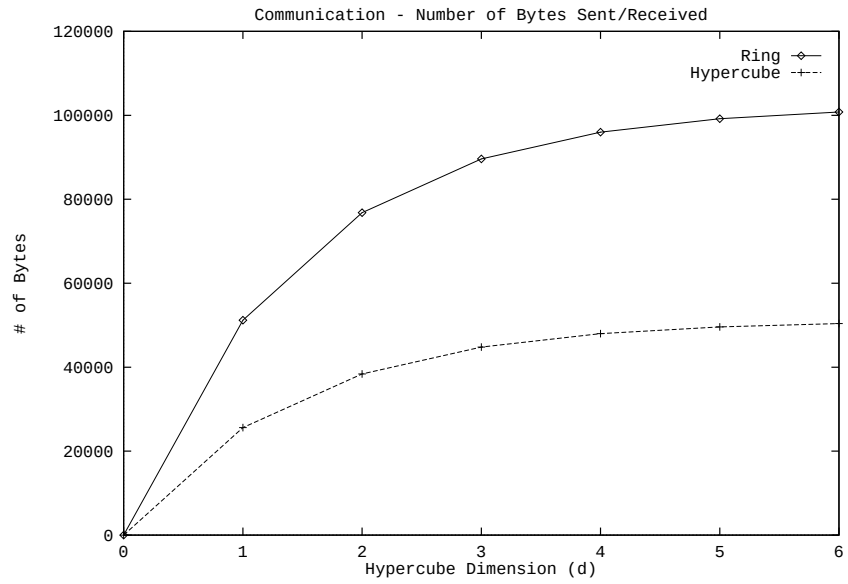

Figure 6: Communication - number of bytes sent/received.

dominated by the comparison time while the time for communication is relatively neglible.

We then note that the execution times for both methods are identical on 0-D and 1-D hypercubes. This result is not surprising, since there is no difference between the algorithms on hypercubes of these sizes. However, on 2-D hypercubes and above we see little to no improvement using the hypercube method while the ring method enjoys near-exponential improvement. That is, as we increase the number of processors in the hypercube, the hypercube method realizes virtually no speedup while the ring method realizes near-linear speedup.

5.2.2 Further Analysis

In an attempt to develop a better understanding of the data, we performed some further analysis. In addition to accumulating execution times on node 0, we also counted number of messages it sent/received, the number of bytes it sent/received, and the number of comparisons it made.

Figures 6 is a graph of number of bytes sent/received by node 0 vs. hypercube dimension. Both methods are plotted on the graph. Each grows at approximately the same rate; however, we observe that the ring method communicates approximately twice as much data as the hypercube method.

Figure 7 is a graph of the number of messages sent/received by node 0 vs. hypercube dimension.

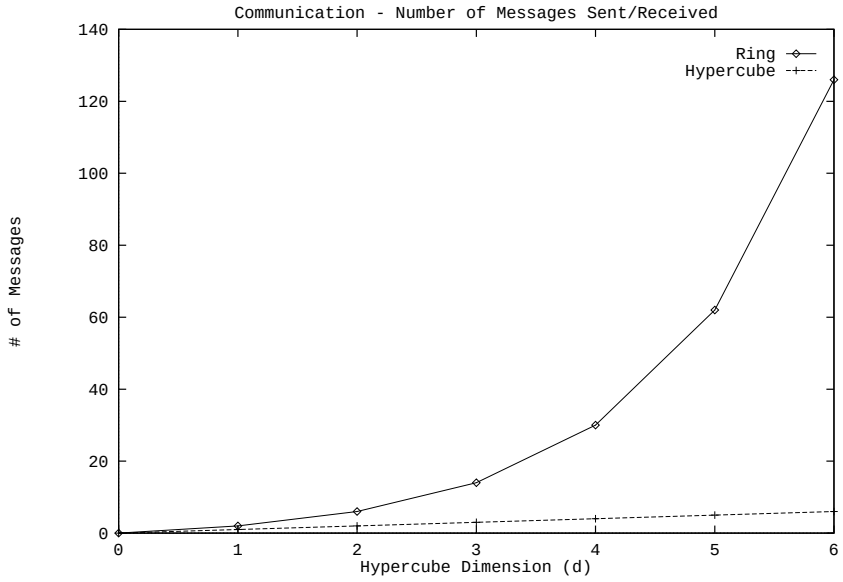

Figure 7: Communication - number of messages sent/received.

Both methods are plotted on the graph. Here we see a dramatic difference between the two methods. The number of messages sent by the hypercube method increases linearly as a function of d while the ring method increases exponentially.

Figure 8 is a graph of the number of comparisons made by node 0 vs. hypercube dimension. Both methods are plotted on the graph. The two methods are identical for 0-D and 1-D hypercubes. Again, this is expected, since there is no difference between the two methods (with respect to the number of comparisons) on 0-D and 1-D hypercubes, i.e., $C_h(N,0) = C_r(N,0)$ and $C_h(N,1) = C_r(N,1)$. However, for 2-D hypercubes and higher we observe a dramatic difference in the number of comparisons node 0 makes in each of the methods. The hypercube method experienced virtually no reductions in the number of comparisons while the ring method experienced a near-exponential reduction.

How do these numbers affect the overall execution time? We see that the dramatic increase in the number of messages and the not-so-dramatic increase in the amount of data, both serving to increase the communication time for the ring method, are more than compensated for by the dramatic reduction in the number of comparisons. This will always be the case in systems where the *overhead* associated with sending/receiving a message (t_0) is much smaller than the time it takes to

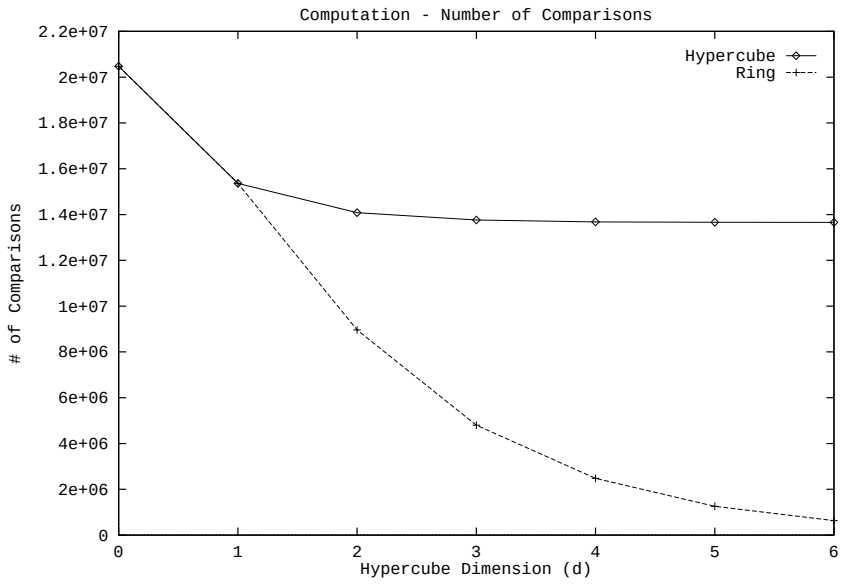

Figure 8: Computation - number of comparisons.

compare two tuples. Under these conditions, the total execution time is completely dominated by the comparison time, and hence, the time for communication becomes insignificant.

5.3 Further Experimentation

We conducted more experiments using different relations. In each case we kept the cardinality (N) the same as in the original experiment (6400 tuples) but varied both the degree of the primary key (p) and the degree of the non-key attributes (p'). We did so in order to study the effects of increasing both the time for comparing two tuples (increasing p) and increasing the volume of data sent/received (increasing p and/or p').

We conducted these experiments changing the values of both p and p' to 1, 5, and 10, independently. The graph for the relation characterized by $N = 6400$, $p = 1$, and p'=1 has already been presented in Figure 5. The graphs for the other 8 cases all have the same shape as the graph in Figure 5.

Figure 9 are graphs of the two extreme cases, ($p = 10$, $p' = 1$) and ($p = 1$, $p' = 10$). Each is a plot of time vs. hypercube dimension, and each plots the same four lines: the total execution and comparison time for both methods.

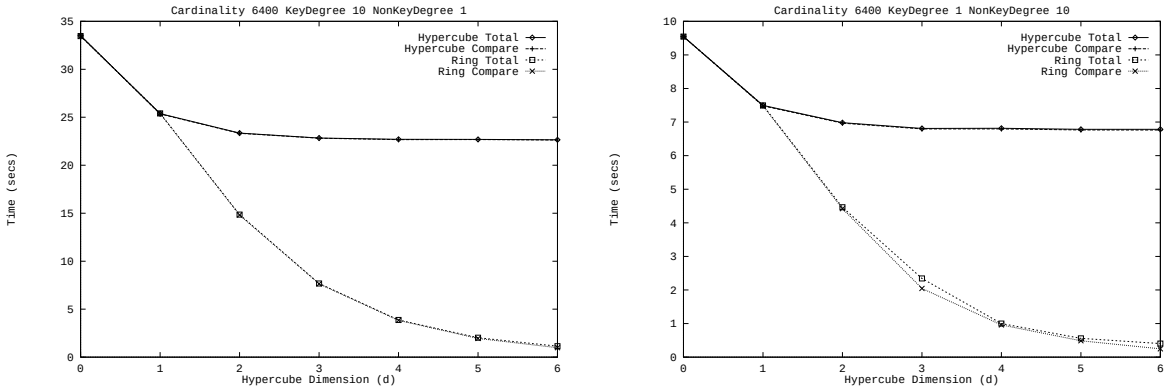

Figure 9: Further experimentation.

Conducting these experiments using relations of the same cardinality over the same range of hypercube dimensions results in the same number of comparisons and the same number of messages sent/received for all experiments. Changing p and p' in each experiment affects only the time to compare two tuples and the *variable* cost in time (t) spent sending/receiving data.

We first observe the same phenomenon that we did in Figure 5; that is, the total execution time and the comparison time for each method are virtually indistinguishable. We further observe that the general shape of the curves in both graphs are the same, which are, in turn, identical to those in Figure 5. This is true even in experiment $p = 1$ and $p' = 10$ where we deliberately attacked the ring method at its weakest point, communication time. In that experiment we attempted to change the relative positions of the lines by increasing communication time only, that is, increasing the amount of data communicated by an order of magnitude while keeping the comparison time the same. This empirically confirms the fact that the comparison time so overwhelmingly dominates the total execution time that changes in the characteristics of the relation that affects the communication time are lost.

6 Conclusion

Algorithms that perform relational database operations in a distributed processing environment, like a hypercube, must address the issue of efficiently removing the duplicate tuples that can result from certain relational operations (e.g., projection and union). One solution to this problem was

presented in [1, 7, 2], which compacts the relation into hypercubes of smaller and smaller dimensions.

We presented an alternate algorithm for removing duplicate tuples that used the embedded ring found in every hypercube. In comparing the two algorithms, we estimated their computation times by counting the number of comparisons required by each. We concluded that the ring remove duplicate algorithm was significantly better in that the computation time of the hypercube method required $(\frac{2^d}{3})$ times more comparisons than the ring method. We then asserted that, although the ring method required more communication time than the hypercube method, the overall execution time for the ring method would be less because the computation component of the methods would quickly dominate the communication component.

Finally, we performed experiments that empirically confirmed our theoretical analysis of the computation times as well as confirming our assertions regarding the computation component of the execution time dominating the communication time.

We have demonstrated, both in theory and in practice, that using the ring embedded within the hypercube is a superior technique in removing duplicate tuples from a relation in that (1) it requires fewer comparisons and hence significantly reduces the total execution time, and (2) once the duplicates are removed, the tuples are more likely to be evenly distributed, thus aiding in proper load balancing.

7 Acknowledgments

We thank Ophir Frieder for introducing us to his distributed relational database algorithm. We thank Gideon Frieder and Ophir for our many discussions that aided in this work. We thank Stephen Wright, Gail Pieper, and Linda Cirillo for their help in the preparation of this paper. We thank Dave Kohr for his superb job in implementing UTP and his patience with all our questions. We thank Bill Gropp and Ewing Lusk for their patience with our questions regarding MPI. We also thank the referees whose suggestions, particularly regarding the experimentation section, made this a better paper.

References

[1] C. K. Baru and O. Frieder, "Database operations in a Cube-connected multicomputer system," *IEEE Trans. on Computers*, vol. 38, no. 6, pp. 920–927, 1989.

[2] C. K. Baru and O. Frieder, "Implementing relational database operations in a cube-connected multicomputer," in *Proc. IEEE COMPDEC 3rd Int. Conf. Data Eng.*, Feb. 1987, Los Angeles, CA.

[3] C.K. Baru and S.Y.W. Su, "The architecture of SM3: A dynamically/partionable multicomputer with switchable memory," *IEEE Trans. Comput.*, vol. C-35, Sept. 1986.

[4] P. B. Berra and E. Oliver, "The role of associative array processors in data base machine architecture," *IEEE Computer*, vol. 12, Mar. 1979.

[5] D. J. DeWitt et al., "GAMMA - A high performance dataflow database machine," in *Proc. Int. Conf. Very Large Data Bases*, Aug. 1986, Kyoto, Japan.

[6] K. A. Frenkel, "Evaluating two massively parallel machines," *Commmun. ACM*, vol. 29, pp. 752–758, Aug. 1986.

[7] O. Frieder, "Database processing on a cube-connected multicomputer," Ph.D. Dissertation, Dep. EECS, University of Michigan, Ann Arbor, MI, 41809, Dec. 1987.

[8] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI*, MIT Press, Cambridge, MA., 1994.

[9] J. P. Hayes et al., "Architecture of a hypercube supercomputer," *IEEE Micro*, Aug. 1986.

[10] B. Hillyer, D. E. Shaw, and A. Nigam, "NON-VON's performance on certain database benchmarks," *IEEE Trans. Software Eng.*, vol. SE-12, Apr. 1986.

[11] "IBM AIX Parallel Environment–Parallel Programming Subroutine Reference (2.0)", International Business Machines Corporation, Kingston, NY., June, 1994.

[12] C. Jard, J. -F. Monin, and R. Groz, "Development of Veda, a prototyping tool for distributed algorithms," *IEEE Trans. on Software Eng.*, vol. 14, no. 3, pp. 339–352, Mar. 1988.

[13] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Architecture and performance of relational algebra machine GRACE," in *Proc. Int. Conf. Parallel Processing*, Aug. 1984.

[14] N. T. Karonis, "On the verification of complex protocols," Ph.D. Dissertation, Syracuse University, 1992.

[15] N. T. Karonis, "Timing parallel programs that use message passing," *J. Parallel Distribut. Comput.*, vol. 14, no. 1, pp. 29–36, Jan. 1992.

[16] J. C. Peterson, J. O. Tuazon, D. Lieberman, and M. Pniel, "The MARK III hypercube-ensemble concurrent computer," in *Proc. Int. Conf. Parallel Processing*, Aug. 1985.

[17] G. Z. Oadah and K. B. Irani, "A database machine for very large relational databases," *IEEE Trans. Comput.*, vol. C-34, Nov. 1985.

[18] S. Y. W. Su and C. K. Baru, "Dynamically partitionable multicomputers with switchable memory," *J. Parallel Distribut. Comput.*, vol. 1, no. 2, 1984.

[19] Intel iPSC Data Sheet, Order No. 280101-001, 1985.

[20] Message Passing Interface Forum, "MPI: A Message Passing Interface Forum," *Int. J. Supercomputer Apps.*, vol. 8, no. 3/4, pp. 165–414, 1994.

[21] Message Passing Interface Forum, "The MPI Message Passing Interface Standard", http://www.mcs.anl.gov/mpi/standard.html, May 1995.

[22] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, *MPI: The Complete Reference*, MIT Press, Cambridge, MA, 1995.

[23] Special issue on Database Machines, *IEEE Trans. Comput.*, vol. C-28, June 1979.

[24] TERADATA, DBC/1012 Data Base Computer Concepts and Facilities, Release 1.3, June 1985.