1981

VLSI Algorithms for Relational Database Operations

Ching C. Hsiao

Lawrence Snyder

Report Number:
81-375

TR 375

VLSI Algorithms for Relational Database Operations

(Extended Abstract)

Ching C. Hsiao and Lawrence Snyder

Department of Computer Sciences

Purdue University

West Lafayette, Indiana 47907

1. Introduction

Since the initial development expenses must be offset by volume production, VLSI implemented systems depend fundamentally on regularity and uniformity. Thus, for VLSI implementation of database operations, it is important to identify a nucleus of processing steps common to the many database operations.

In this paper we present a primitive operation for sorting, remove-duplicates, union, intersection, and subtraction (the latter three operations are slightly generalized to have multiset operands.) To evaluate the efficiency of this primitive, we develop a shallow complexity hierarchy among the operations (Figure 1). For join, we give an algorithm which uses our primitive to improve on known systolic methods [Kun80, Son80] in the average case. Therefore we have a uniform treatment of the database operations that facilitates VLSI implementation.

The key to unifying these database operations is to understand the role of remove-duplicates. We prove the reducibilities illustrated by the solid lines of

The research described herein is part of the Blue CHiP Project. Funding is provided in part by the Office of Naval Research under Contract N00014-80-K-0816 and Contract N00014-81-K-0360, Special Research Opportunities Program Task SRO-100.

Figure 1. In addition, we show the relationship given by the dashed edge of the figure (Theorem 1). This latter result states that the comparisons required for sorting constitute a subset of the comparisons needed to remove duplicates. In a sequential, comparison-based model this would suffice to demonstrate that duplicates removal is at least as hard as sorting. But in a parallel model where data communication is a significant consideration, producing the final order may be harder than amassing the comparison information. Nevertheless, this result suggests that sorting and duplicates removal have the same parallel complexity, collapsing the hierarchy*. Of course, being able to collapse the hierarchy implies that the primitive is optimal. If the hierarchy does not collapse, the primitive provides a solution for the lower elements of the hierarchy that is suboptimal by at most a factor of $\log n$.

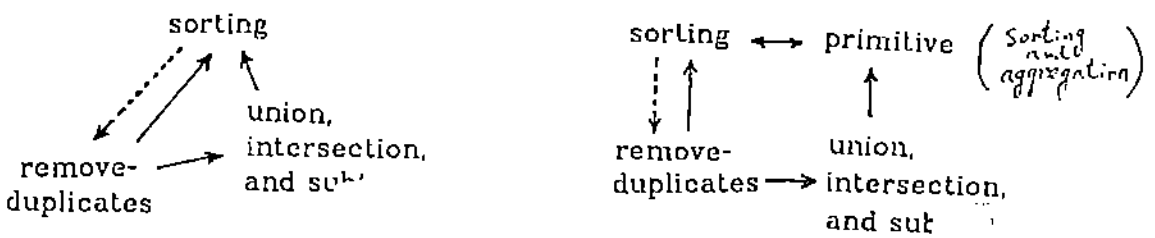

Figure 1. Complexity hierarchies. The arrow "→" denotes the notion "is reducible to" and the dashed arrow "⇢" suggests it.

Joining two relations each with size n can potentially run to the worst case of having n^2 result tuples. When using n processing elements (PE's), $O(n)$ time is the best one can expect. Any join algorithm of the "predetermined sequence of comparisons" type must handle the extreme case, and only halting mechanisms can avoid running in $O(n)$ time for the average cases. Our primitive plays a role here in making a halting mechanism work effectively.

* An unverified rumor of an $O(\log n)$ depth sorting network lends further credence to this conjecture.

2. A complexity hierarchy

Although a 2-fold comparison ($\geq$, $<$) is sufficient for sorting and other operations, the 3-fold comparison ($>$, $=$, $<$) is the comparison model used throughout this paper since it detects a duplicate right away. The 3-fold comparison can be done with two 2-fold comparisons; the 2-fold comparison can be done with a single 3-fold comparison. The time bounds (in terms of the comparison count) derived using the 3-fold model are thus also valid for the 2-fold model to within a constant factor (no larger than 2).

In this section we first consider the relationship between sorting and remove-duplicates. Union, intersection, and subtraction are then added as a family to the complexity hierarchy.

2.1. Elimination of duplicates

Let $X = \{ x_0, x_1, \ldots, x_{n-1} \}$ be a multiset consisting of n elements from a totally ordered set, and $Y = \{ y_0, y_1, \ldots, y_{m-1} \}$ be the reduced set after the elimination of duplicates in X. Define C as the minimum set of comparisons required for the elimination of duplicates. A *semi−digraph* which contains both directed and undirected edges can represent the set C:

$$x_i \longrightarrow x_j \quad \text{if } x_i > x_j, \text{ or}$$

$$x_i \longrightarrow x_j \quad \text{if } x_i = x_j.$$

The *semi−digraph* is composed of n nodes and no more than $n(n-1)/2$ edges. By removing the undirected edges and merging nodes, we can reduce the *semi−digraph* to a digraph of size m which represents the reduced ordering C_Y of elements in Y.

Lemma 1. The reduced ordering C_Y is total.

For any y_i and y_j in Y, $i \neq j$, there must exist a path between the two nodes, otherwise the relationship $y_i \neq y_j$ is not guaranteed. The unique

direction of the path further indicates whether $y_i > y_j$ or $y_i < y_j$. This implies that the rank of y_i is known and the ordering C_Y is total. ▪

Theorem 1. Duplicates removal requires a superset of comparisons needed for sorting.

By Lemma 1 the elimination of duplicates ends with a total ordering C_Y of elements in Y. One extreme case is that there are no duplicates at all. Elimination of duplicates must then have done the comparisons required for sorting. ▪

For sequential computation, the above theorem says that sorting is reducible to remove-duplicates. In a parallel model it says only that the logical ordering is obtainable; it does not guarantee the logical ordering coincide with any simple location indexing. There might exist a fast algorithm to remove duplicates for which the logical ordering is implicit and not immediately available.

Theorem 2. Duplicates removal is reducible to sorting.

A parallel marking algorithm can discover the duplicates in $\log n + 1$ comparison steps if the sequence is already sorted. The algorithm essentially performs a tree based scheme analogous to those of aggregation operations (max, min, sum, count, and carry propagation.) This suffices to prove the reducibility from duplicates removal to sorting even if the naive lower bound of $\Omega(\log n)$ is achieved for network sorting. ▪

2.2. Union, intersection, and subtraction

Any fast sorting procedure, as J. T. Schwartz [Sch80] remarked, is useful as the basis for a unified treatment of various important set-theoretic operations. Note that sets do not have any duplicates themselves. There is at most one duplicate for each x in the concatenation of two sets A and B. Having sorted the

concatenation A+B, one can easily detect all the duplicates with two more parallel comparison steps, i.e. comparisons between all the odd-even and even-odd pairs.

Lemma 2. [Sch80] The set operators "$\cup$", "$\cap$", and "-" are reducible to sorting.

algorithm(A $\cup$ B)	time
sort A+B	$S(n_1+n_2)$
mark duplicates	$O(1)$

algorithm(A $\cap$ B)	time
mark A; mark B	$O(1)$
sort A+B	$S(n_1+n_2)$
unmark duplicates	$O(1)$

algorithm(A - B)	time
mark B	$O(1)$
sort A+B	$S(n_1+n_2)$
mark x in A having a duplicate in B	$O(1)$

When considering the design of a unified basis for parallel database processing, one must consider how to handle multisets. Multisets are artifacts of operations such as projection and concatenation. Many query languages (SEQUEL, QUEL, and QBE [Ull80]) provide operators for working with multisets. The problem with providing union, intersection, and subtraction that take multisets as arguments and produce multisets as results is: for union, the operator is equivalent to concatenation ; for intersection and subtraction, there are several alternatives, none of which are compelling. Thus, we simply generalize union, intersection, and subtraction to allow multisets as operands and produce a set as result.

According to the generalized definition, the multiset operations can be equivalently done by remove-duplicates followed by the corresponding set operations (rd is the sort form of remove-duplicates):

$$union(A,B) \equiv rd(A) \cup rd(B) \equiv rd(A+B),$$
$$intersection(A,B) \equiv rd(A) \cap rd(B),$$
$$subtraction(A,B) \equiv rd(A) - rd(B).$$

Since both remove-duplicates and the set operations are reducible to sorting,

the three operations are therefore reducible to sorting. On the other hand, elimination of duplicates can be implemented by any fast algorithm for union, intersection and subtraction. Specifically, $rd(A) = union(A,\varphi)$, $rd(A) = intersection(A,A)$, or $rd(A) = subtraction(A,\varphi)$. This implies that remove-duplicates is reducible to these operations.

Theorem 3. Union, intersection, and subtraction are reducible to sorting and reducible from remove-duplicates. ▪

3. A primitive operation

In developing the complexity hierarchy, we learned that a primitive for implementing the many database operations should be able to perform both sorting and aggregation (marking) functions. Batcher's bitonic merge sort [Bat68] is the best scheme known for sorting*. Furthermore, the bitonic sort itself has the potential of performing the aggregation function due to its merge-oriented characteristic. We therefore consider a primitive based on extending the bitonic sorting scheme to remove duplicates (rd-sort).

By using a more sophisticated comparison function, the bitonic sorting scheme achieves both sorting and marking functions. The comparison function preserves the ordering among distinct elements. It marks off a duplicate whenever one is detected. It also enforces an ordering between x and its marked duplicate x^+ such that x^+ is a little larger than x but never larger than y for any $y > x$. The marking process is idempotent, i.e. $x^{++} \equiv x^+$, and the ordering among x^+'s is arbitrary.

Definition (rd-comparison) The comparison function used with the bitonic sort

* The bitonic sort requires $O(\log^2 n)$ time using the perfect shuffle [Sto71] or the cube-connected cycles [Pre81] interconnection, $O(\sqrt{n})$ time using mesh interconnection [Tho77, Nas79], or $O(\sqrt{n} / w)$ time using the configurability of the CHiP computers [Hsi81] (where w is the width of corridors.)

for remove-duplicates is defined as:

(x, y) --> $(\max(x, y), \min(x, y))$, when $x \neq y$;
(x, x) --> (x^+, x);
(x^+, x) or (x, x^+) --> (x^+, x);
(x^+, x^+) --> (x^+, x^+).

For any comparison-based method which merges two ordered lists, every pair of neighboring elements in the result list must have been compared directly, unless both elements are from the same list. The bitonic sort starts by merging ordered lists of 1,2,4,... elements. The rd-comparison guarantees that two ordered lists with all their duplicates marked off must be merged into a list of this type. These observations lead us to the conclusion that the bitonic sorting scheme together with the rd-comparison can sort a sequence and mark off all duplicates.

Since the rd-sort serves as a primitive for both sorting and remove-duplicates, the primitive should also work for the three multiset operations (Theorem 3). It is trivial to implement the union operation by the rd-sort. The following algorithms show that the primitive also solves intersection and subtraction.

Subtraction

1. Mark each x in B as x^- which is a little smaller than x in A.
2. Perform rd-sort to mark off duplictes in A as x^+ and also sort the sequence A÷B.
3. The sorted sequence must appear as $\cdots x^+..x^+xx^-..x^- \cdots$. As in Lemma 2, two more steps are for the marking (x, x^-) --> (x^+, x^-).
4. The result is the set of all unmarked elements.

Intersection

1. Mark each x in A as x^+; mark each x in B as x^-.
2. Perform rd-sort to mark off duplicates in A as x^{++} and those in B as x^{--}.
3. The sorted sequence must appear as $\cdots x^{++}..x^{++}x^+x^-x^{--}..x^{--} \cdots$. Again two more steps are for the unmarking (x^+, x^-) --> (x, x^-).

4. The result is the set of all the unmarked elements.

4. The join operation

Let n_r be the number of result tuples after joining two relations A and B. Assume that both relations are of the same size, say n, for simplicity. The figure n_r, denoting the minimum totality of comparisons required for join, may become as large as n^2. Kung's [Kun80] and Song's [Son80] linear time algorithms are optimal only in the sense of handling the worst case.

Highly parallel algorithms usually adopt a predetermined sequence of computation since there has not been a single technique to support dynamically determining the computation pattern. Any join algorithm of this type must take care of the worst case and so only a halting mechanism can avoid running all the cases in linear time.

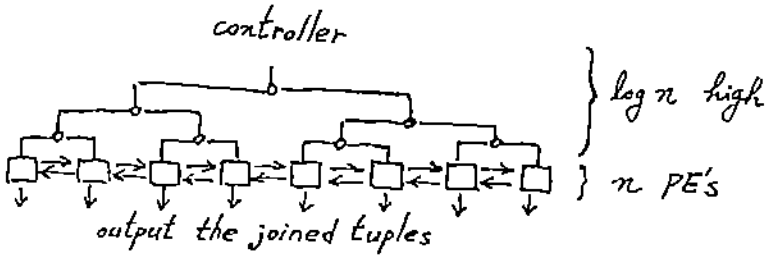

Figure 2. A system for the join operation.

A system for the join operation is depicted in Figure 2. The tree interconnection provides control links for the halting mechanism. The linear interconnection provides data links for moving tuples back and forth. An "easy catch" method requires that the concatenation of A and the marked version B^+ is sorted. The B-tuples can then move in one direction to catch their joinable A-tuples easily. Let l be the longest distance between any B-tuple and its joinable A-tuple. The time complexity of this method is thus $S(n) + O(l)$. Notice that this improved performance is achieved by applying the two primitive functions: sorting and aggregation. The rd-sort (with or without its marking function)

comprises the first phase. In the second phase, most of the interconnection network can be dedicated to an aggregation function that detects the earliest time to stop the join processing.

The join system can be implemented with shuffle-exchange interconnection. The exchange edges provide communication channels for moving B-tuples, while the shuffle edges can simulate the tree interconnection in $\log n$ steps (Figure 3). The total time is $O(\log^2 n) + O(l)$. The CHiP computer [Sny81] is also a good candidate for implementing the join system. It provides the flexibility that interconnections can be dynamically reconfigured for the presorting phase and the joining phase. Figure 4 shows the two co-existing interconnections in the joining phase on a CHiP computer (switches are of one cross-over capability.) If the mesh interconnection is used in the presorting phase then the total time is $O(\sqrt{n}) + O(l)$.

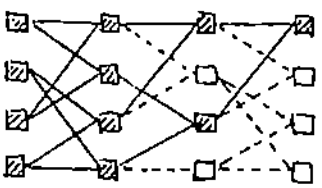

Figure 3. Tree interconnection simulated by $\log n$ steps of shuffle.

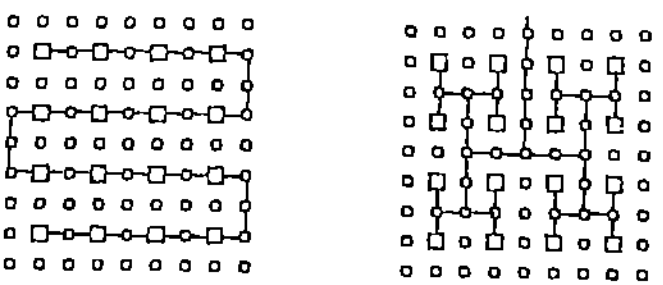

Figure 4. Configuration on a CHiP computer for the join system.

5. Discussion

An efficient implementation of the rd-comparison uses one extra bit appended to each data element as the least significant bit. The extra bits are

initially cleared. The rd-comparison works simply to set one l.s.b. to 1 whenever two elements are compared to be equal. The increased overhead in the comparison function is thus very limited since comparison is easily done from high to low order bits. Notice that the rd-sort performs both sorting and marking functions. The rd-sort for intersection and subtraction requires that the marking function work separately on two multisets. Therefore, in addition to the extra bit, one more bit is needed to distinguish the multisets.

The result sequence could be sparse due to the marked-off duplicates. The marked-off duplicates can be filtered out while outputing the sequence. Alternatively, in some applications one might want to compress the sequence internally so that the marked duplicates are squeezed out. A trivial solution is to run the bitonic sort again using another comparison function which treats the marked duplicates as $+\infty$. Schwartz presented an ingenious method to separate and pack marked data on the Ultracomputer in $O(\log n)$ time [Sch80]. If the shuffle-exchange interconnection is available the compression job is best done by his pack algorithm.

The performance of the join system is proportionally better as the number of result tuples decreases. Assume that the average size of the result relation after the join operation is smaller than $O(n)$. It is then safe to say that the value for l is no larger than $O(\sqrt{n})$. That is to say both implementations of the join system require no more than $O(\sqrt{n})$ time.

Acknowledgements

It is a pleasure to thank Dennis Gannon and Jeremy Epstein for many useful conversations. Special thanks to Janice Cuny for her careful reading of this extended abstract and suggestions.

References

[Bat68] K. E. Batcher, " Sorting networks and their applications," AFIPS, 1968, 307-313.

[Hsi81] Ching C. Hsiao and Lawrence Snyder, " Etudes of Sorting and Selections," Purdue University, in preparation.

[Kun80] H. T. Kung and Philip L. Lehman, " Systolic (VLSI) Array for Relational Database Operations," ACM SIGMOD, International conf. 1980.

[Nas79] David Nassimi and Sartaj Sahni, " Bitonic Sort on a Mesh-Connected Parallel Computer," IEEE Tran. on Computers, vol. c-27, no. 1, Jan. 1979.

[Pre81] Franco P. Preparata and Jean Vuillemin, " The Cube-Connected Cycles: A Versatile Network for Parallel Computation," CACM, vol. 24, no. 5, May 1981, 300-309.

[Sch80] J. T. Schwarz, " Ultracomputer," ACM Tran. on Programming Languages and Systems, vol. 2, no. 4, Oct. 1980, 484-521.

[Sny81] Lawrence Snyder, " Introduction to the Configurable, Highly Parallel Computer," TR-351, Purdue University, 1981.

[Son80] S. W. Song, " A Highly Concurrent Tree Machine for Database Applications," IEEE Conf. on Parallel Processing, 1980.

[Sto71] Harold S. Stone, " Parallel Processing with the Perfect Shuffle," IEEE Tran. on Computers, vol. c-20, no. 2, 1971.

[Tho77] C. D. Thompson and H. T. Kung, " Sorting on a Mesh-connected Parallel Computer," CACM, vol. 20, no. 4, 1977.

[Ull80] J. D. Ullman, "Principle of Database Systems," Computer Science Press, 1980, Chapter 4.