1985

Order Statistics and Other Complex Data Operations On Tree-Structured Dictionary Machines

Michael T. Goodrich

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Report Number:
85-532

ORDER STATISTICS AND OTHER DATA OPERATIONS ON TREE-STRUCTURED DICTIONARY MACHINES

Michael T. Goodrich
Mikhail J. Atallah

CSD-TR-532
August 1985

ORDER STATISTICS AND OTHER DATA OPERATIONS ON TREE-STRUCTURED DICTIONARY MACHINES

Michael T. Goodrich
Mikhail J. Atallah

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

August 1985

CSD-TR-532

Abstract

We show how to extend previously proposed tree-structured dictionary machines so they can perform order statistics and other data operations. In particular, we consider how to perform multi-selections, sorting, and operations based on the ranks of data items, such as **Extract**(j), which simultaneously selects and deletes the j-th smallest data item. All these data operations can be performed without ever interrupting the pipelining of responses coming from the machine. Moreover, the extension allows for graceful recovery from redundant operations, and it preserves the order of the responses.
Keywords. Dictionary machine, order statistics, pipe-lining, multiple-response queries.

1 Introduction

An application of VLSI which seems to offer many rewards is that of manufacturing powerful, special-purpose computing devices which allow general-purpose (host) computers to off-load some of their computational burden. Specifically, one such application is the design of dictionary machines (sometimes referred to as database machines) [1,2,3,4,6,7,8,9]. A dictionary machine is a computing device which is to be used to allow a host computer to off-load certain operations for maintaining and querying data files.

For a dictionary machine to be attractive for use with a general-purpose computer, it should obviously perform the required operations better than the general-purpose computer can. If the types of operations are simple, a sequential computer can perform fairly well, however. For example, by using a simple hashing scheme a sequential computer can perform the operations **Insert**, **Delete**, and **Member** in $O(n)$ time in the worst case, but in practice hashing techniques come very close to constant time performance. If we add **ExtractMin** to the list of operations, then a sequential computer can perform any operation in $O(\log n)$ time in the worst case, by using any of the familiar balanced search-tree data structures. But the cost of performing more complex operations on a sequential computer can become quite prohibitive. For example, dynamic d-dimensional orthogonal range queries take $O(\log^d n)$ time and $O(n \log^{d-1} n)$ space [10] on a sequential computer, and there is little hope for improving this so long as a sequential computer is used [5].

One way that some dictionary machines excel over sequential machines is in their use of parallelism, especially their ability to pipe-line operations. A host computer using such a machine sends it a stream of operations, and gets back a stream of responses. In this context there are two useful measures for evaluating a machine's performance: its *latency* time and its *interval* time (or *period*). The latency time of a machine is the time it takes one operation to flow through the machine, from the input of the operation to the output of its first result. The interval time of a machine is the time that elapses between consecutive responses.

One design which makes considerable use of this type of parallelism is the tree-structured dictionary machine [1,2,3,6,8,9], depicted in Figure 1. Such a machine is structured as a binary tree of processing elements (PE's), and stores data items internally, each PE having $O(1)$ storage registers. There is an actual database machine which was recently introduced and is based on this kind of structure [7]. In each of the previous designs data operations are sent to the dictionary machine in a pipe-lined fashion, the execution of an operation beginning when it is input as a single instruction to the tree's root processor. (These high-level instructions should not be confused with the primitive instructions which comprise the instruction set for the CPU in a processing element (PE).) A PE executes the instruction on its contents, then divides the instruction into two instances, called *bubbles*, and sends them to its children, including its response (if there was one) with one of the bubbles, and its parent's response with the other. The instruction continues to be broadcast down the tree in this manner, PE's broadcasting bubbles to their children. As the bubbles from this instruction move down the tree, being divided into more and more bubbles, they carry along the responses that have been generated so far. When bubbles from this instruction reach the last level of the tree still storing data items they stop being broadcast down the tree, and start moving back up the tree to be output, possibly combining with each other as they move up. When a (response) bubble arrives at the root, the root simply sends it back to the host.

Much of the previous work on tree-structured dictionary machines [1,2,3,6,8,9] has dealt with efficient ways of implementing simple operations, such as inserting a data item, deleting an item, finding an item, etc. Because so much work has been spent optimizing the simple operations, one might be lead to the misconception that a tree-structured dictionary machine cannot perform more complex operations, such as rank-based queries. Some of these papers describe machine designs in which data items are stored close to the top of the tree, thus optimizing the latency time of the machine. Some describe machine designs which allow for graceful recovery from redundant **Insert** and **Delete** operations. That is, the machine can recover gracefully if an **Insert** operation tries to insert something that is already in the dictionary machine or if a **Delete** tries to remove something that is not in the machine. In [9] Song describes a tree-structured dictionary machine which can perform complex operations, such as multi-selection, sorting, and join. Still, the machine described does not support rank-based queries, it requires that data items be stored only at the leaves of the tree, and operations cannot be redundant.

In this paper we show how to extend the previously proposed tree-structured dictionary machines to be able to perform order statistics and other data operations, such as range-queries and sorting, even in the presence of redundant operations. If we were to disallow redundant **Insert**'s and **Delete**'s, then rank-based queries would be straight-forward to implement, for we could easily maintain a rank field for each data item in the dictionary machine. This becomes non-trivial, however, in the presence of redundant operations, especially if we allow for an operation **Extract**(j), which simultaneously selects and deletes the j-th smallest data item. This **Extract**(j) operation is a generalization of the **ExtractMin** and **ExtractMax** operations that some of the previous designs support [1,6,8]. In section 3 we describe how to maintain data item ranks while allowing for an **Extract**(j) operation as well as redundant **Insert**'s and **Delete**'s. In section 4 we show that even if we allow for queries which could have a variable number of responses (such as range-queries) we need not sacrifice any of the desirable qualities of the tree-structured dictionary machine. That is, our design does not require that response pipe-lining ever be turned off for such operations, nor does it allow such operations to corrupt data item ranks. The extension guarantees that the results from different operations will never become "inter-mixed", and that the order of operation results is the same as the order in which the operations were sent to the machine.

2 The Machine Model

The machine model we will be using incorporates the common properties of previous tree-structured dictionary machine designs [1,2,3,6,8,9]. Some of the previously proposed models have data items stored only at the leaves of the tree, but for the purpose of more generality we will assume that a data item can be stored at any node in the tree. Restricting items to the leaves is then a simple special case.

We define the last level of the tree still containing data items to be the *effective last level* of the tree. All previous designs execute instructions as described earlier, instructions being executed as they flow down the tree to the effective last level. An exception is reference [8] where instructions can be executed as they are going up the tree. The algorithms we describe in this paper assume that instructions execute as they travel down the tree. Modifying our model so that instructions execute on the way up would be fairly straightforward.

A data item is represented as a key-record pair (k, r), where k is a key uniquely identifying the item, and r is a data record. In a database it is often useful for k to be a tuple $(k_1, k_2, \ldots, k_d)$. Machine instructions are denoted by i. An instruction instance, or bubble, in the tree consists of an instruction–response pair (i, b), where i is the instruction being executed and b is a response from some PE which has executed i on its contents. We place no restrictions on the response b, but typically it will be $\emptyset$ (the null response), "yes," "no," or some (k, r) pair.

There are six primitive operations which previously proposed machines can perform:

1. **Insert**(k, r): insert a (k, r) pair;

2. **Delete**(k): delete the (k', r) pair for which $k' = k$;

3. **Update**(k, r): replace (k, r') by (k, r);

4. **Member**(k): return the (k', r) pair for which $k' = k$ if there is such a pair, otherwise, return "no";

5. **Min**(): return the (k, r) pair which has the minimum k value among all those stored in the machine;

6. **ExtractMin**(), **ExtractMax**(): simultaneously select and delete the (k, r) pair with smallest (largest) k value [1,6,8].

Since the implementation details for operations 1, 2 and 3 above differ from machine model to machine model, we will not describe their implementation in any detail. For the details of various implementations see the references at the end of this paper. The only thing we assume is that, like all operations, content-modifying operations execute while travelling down the tree. We will describe how to perform the operations 4, 5, and 6 in this paper, since we describe generalizations of these three operations.

An **Insert**(k, r) is *redundant* if there is already a pair (k, r') in the tree, and a **Delete**(k) or **Update**(k, r) is redundant if there is no pair (k, r') in the tree. Not all machine models allow for redundant operations, and some of the ones that do allow for them require that all redundant **Insert**'s be treated like **Update**'s. Our solution to the redundancy problem is to change the way instructions are pipelined through the machine similar to the method used by Atallah and Kosaraju in [1], thus allowing for general kinds of recovery from redundant operations. Instead of only going up and down the tree once, instructions will now go up and down the tree twice. In the first round trip the content-modifying instructions will be tagged either "redundant" or "non-redundant" (all non-content-modifying instructions are trivially non-redundant). Then, in the second round trip, the non-redundant instructions will perform just as before when we assumed all instructions were non-redundant. The redundant instructions will perform some error recovery procedure (usually a no-operation) in their second round trip. For example, a redundant **Insert**(k, r) could act as a non-redundant **Update**(k, r) or it could simply do nothing, allowing the (k, r') pair already in the machine to remain unchanged. This is, of course, more general than a "hard-wired" redundancy-handling mechanism.

If we restrict the content-modifying instructions to **Insert**(k, r), **Delete**(k), and **Update**(k, r), then the original scheme of Atallah and Kosaraju [1] will correctly tag an instructions as "redundant" or "non-redundant" by the end of the first round trip through the machine. We will review their scheme here, and then in the next section show how to extend it for our model. Their scheme called for instructions traveling up the tree in their first round trip to examine both the (k, r) pairs in the tree and the instructions traveling down the tree in their second round trip (see Figure 2) and execute the following tagging rules as they went.

Tagging Rules:

1. A non-redundant **Insert**(k, r) becomes redundant, or a redundant **Delete**(k) or **Update**(k, r) becomes non-redundant, by finding a pair (k, r') in the tree or by seeing a non-redundant **Insert**(k, r') instruction making its way down the tree in its second round trip.

2. A redundant **Insert**(k, r) becomes non-redundant, or a non-redundant **Delete**(k) or **Update**(k, r) becomes redundant, by seeing a non-redundant **Delete**(k) in its second round trip down the tree.

Initially, all **Insert**(k, r) instructions are tagged non-redundant, and all **Delete**(k) and **Update**(k, r) instructions are tagged redundant. On the way up the tree in the first round trip the tags are combined according to the following rule. If the tags for the two instances of the same **Delete**(k) (or

Update(k,r)) instruction differ when they are sent to a parent PE, then the combined bubble is tagged non-redundant. If, on the other hand, the tags for two instances of the same **Insert**(k,r) differ, then the combined bubble is tagged redundant. Note that in each case we are tagging the instruction opposite to the tag the instruction had when it was input to the machine.

Unfortunately, if we are to implement rank-based queries, including an operation **Extract**(j) which deletes an item based on its rank, then the redundancy scheme presented above is not sufficient to correctly tag operations as "redundant" or "non-redundant." This is because an **Extract**(j) can cause a redundant **Insert**(k,r) to become non-redundant or a non-redundant **Delete**(k) or **Update**(k,r) to become redundant, and there is currently no way to tell if the instruction is affected by the **Extract**(j) coming down the tree in its second round trip. We solve this problem in the next section.

3 Order Statistics

There are a number of useful data operations which use the ranks of data items as operands: selecting the median element, extracting the j-th smallest element, or selecting the j-th largest element, to name a few. In this section we show how to implement these types of rank-based queries. The first operation we describe is selecting the j-th smallest data item in the dictionary tree. If we add a rank field to the $[k,r]$ register of each PE, to store the rank of k, then we can add a new instruction **Select**(j), which returns the (k,r) pair with j-th smallest k value. A PE storing (k,r) can easily update the rank of k by incrementing its rank for every non-redundant **Insert**(k',r') with $k' < k$ and decrementing it for every non-redundant **Delete**(k') with $k' < k$. If the key k is really a tuple $(k_1, k_2, \ldots, k_d)$, then the rank of k can also be a tuple $(r_1, r_2, \ldots, r_d)$. The algorithms we present below consider the rank to be a single integer, but changing them for the case of a tuple of ranks should be obvious.

Adding the operation **Select**(j) does not corrupt the redundancy checking mechanism presented in the previous section, because it does not alter the contents of the dictionary machine. However, if we include the instruction **Extract**(j), which simultaneously selects and deletes the (k,r) pair with the j-th smallest k value, then, as we have already observed, the previous scheme is not enough.

Let i be some content-modifying instruction, using the key k, which is in its first round trip moving up the machine. Of course, as it is moving through the machine it exists as a number of instances (bubbles). We define the *relative rank* of k for a bubble of i which is in its first round trip moving up the tree and is currently at some PE P to be the rank of k if we were to complete executing all the instructions which have already passed through P in their second (executing) round trip moving down the machine (see Figure 3). To rectify the problem introduced by **Extract**(j) we add a rank field to each bubble of every content-modifying instruction i. In

i's first round trip through the machine this rank field will store the relative rank of k, and in i's second round trip through the machine this field will store the actual rank of k. The rank field is used in the instruction's first round trip through the machine so that it will be able to tell if it is affected by an **Extract**(j) moving down the machine in its second round trip. This rank field is also used in the second round trip to help compute the ranks of bubbles of instructions in their first round trip. Now, when an instruction i is moving up the tree in its first round trip it will be compared against those instructions coming down the tree in their second round trip by examining their rank fields as well as their redundancy tags. We denote the rank field of a bubble $B = (i, b)$ by $B.rank$. For a non-redundant bubble $B = (i, b)$ which is moving up the tree in its first round trip we use the following rules to update its rank field (note that similar rules should be used to update the rank field of a (k, r) pair in the tree):

1. Upon encountering a non-redundant **Insert**(k, r) with rank field less than $B.rank$, then increment $B.rank$ by 1.

2. Upon encountering a non-redundant **Delete**(k) with rank field less than B.rank, then decrement $B.rank$ by 1.

3. Upon encountering an **Extract**(j) where $j < B.rank$ then decrement $B.rank$ by 1.

Computing the initial rank of a **Delete**(k) or **Update**(k, r) bubble is straight-forward. When one of these bubbles becomes non-redundant, it sets its rank to the rank of what ever made it non-redundant: either a (k, r) pair in the tree or an **Insert**(k, r') instruction, which by an inductive argument has a correct rank field. But, since an **Insert**(k, r) is non-redundant as soon as it is input to the machine, computing its initial rank is not as easy.

Initially, an **Insert**(k, r) instruction will have rank 1. As it travels down the tree in its first round trip if a bubble B of i is sent to a PE storing a (k', r') pair with $k' < k$, then set $B.rank :=$ $\max(B.rank, \text{rank}(k') + 1)$. This rank is then propagated down to the children PE's.

Lemma: Let i be some **Insert**(k, r) instruction. At least one bubble for i will contain the correct relative rank of k by the time the bubble reaches the effective last level of the tree.

Proof: The correct relative rank of k for a bubble of i =**Insert**(k, r) at the effective last level of the tree is one greater than the maximum rank of all (k', r') pairs, $k' < k$, which were in the tree when the bubbles of i flowed past them. Since bubbles for the instruction i pass by every PE in the tree storing a (k', r') pair, then there is certainly some bubble for i which will pass through the PE storing the immediate predecessor of (k, r) (if there was one). ■

Since the correct relative rank of k for a bubble of i =**Insert**(k, r) is the maximum over all the bubbles of i, in combining instances as they move up the tree, we take the rank of a combined bubble to be the maximum rank of its two children bubbles.

Note that a bubble of i =**Insert**(k,r) with the correct relative rank of k might not have the correct actual rank of k over all data items in the tree at any particular time before reaching the top of the tree, however. This is because there may be a **Delete**(k') instruction, $k' \leq k$, in the tree ahead of i but a bubble of i went by (k', r') before it was deleted. This is not a problem, however, because i will encounter the **Delete**(k') coming down the tree as i is moving up the tree.

Lemma: By the time any content-modifying instruction has completed its first round trip through the machine, with all its first-trip instances being combined back into one bubble, it contains the correct rank of the (k,r) pair it is to affect.

Proof: Let i be some content-modifying instruction. We have established that if i =**Insert**(k,r), then as bubbles for i start up the tree in i's first round trip there is at least one with the correct relative rank of k. Also, if i =**Delete**(k) or i =**Update**(k,r) then as soon as a bubble of i becomes non-redundant it contains the correct relative rank of k. If a bubble of i with correct relative rank encounters an instruction coming down the tree in its second round trip then the ranking rules given above are clearly sufficient to correctly update the relative rank for this bubble. Finally, since we have made sure that the correct relative rank of k for i moves up the tree with i, then by the time i arrives back at the root as one bubble the rank of that bubble will be the relative rank of k. The instruction i is now the next instruction to be sent down for execution; hence, the relative rank of k for i is in fact the actual rank of k. ■

So, we add the following rule to the redundancy rules:

3. A redundant **Insert**(k,r) becomes non-redundant, or a non-redundant **Delete**(k) or **Update**(k,r) becomes redundant, if it encounters an **Extract**(j) instruction travelling down the machine in its second round trip with j equal to the the rank field for that instruction.

The **Extract**(j) instruction trivially stores the rank of the item it will affect, so we need not add any rank field to it. Moreover, the only time **Extract**(j) can be redundant is if there are less than j elements being stored in the machine. But, as should be apparent by the way **Extract**(j) affects other instructions, the fact that $n < j$ does not corrupt the instruction of the machine, and is discovered by the time the response bubbles from the **Extract**(j) instruction reach the root processor.

Theorem: An instruction is correctly tagged as redundant or non-redundant by the time it is finished travelling through the tree for the first time, even in the presence of **Extract**(j) instructions.

Proof: We have already established the theorem for the basic three content-modifying instructions. So we need to prove that an instruction is correctly tagged in the presence of **Extract**(j) instructions. From the previous lemma, for an instruction i moving up the tree in its first round trip, where i is a non-redundant **Insert**(k, r) or a redundant **Delete**(k) or **Update**(k, r), there is at least one bubble of i with the correct relative rank of k. If i on its way up encounters an **Extract**(j) on its way down with j equal to its relative rank, then it is affected by this **Extract**(j). Note that the redundancy rule 3 above correctly handles this case. Since we have already established the theorem for the other content-modifying instructions, this completes the proof. ∎

3.1 Using the Value of n

Another important piece of information that should be allowed to be included in machine instructions is n, the number of (k, r) pairs in the machine. This makes operations such as extracting the maximum element, selecting the median, or selecting the j-th largest element possible. One might be tempted to require that every PE store the value of n to be able to implement these operations, but the only PE which needs to store the value of n is the root processor. Any instruction that uses the value of n as an argument can have the actual value substituted as soon as the instruction reaches the root after its first round trip through the machine. This is because the root processor will store the correct value of n for that instruction, having processed all content-modifying instructions which precede it. For example, when an instruction such as **SelectMedian**() or **SelectMax**() reaches the root PE the root simply changes the instruction into the appropriate **Select**(j) instruction. Similarly, the root can convert an **ExtractMax**() or **ExtractMedian**() operation to an **Extract**(j) in this way, as well. After this change, the instruction proceeds in its second round trip as before.

4 Multiple Responses

In most of the previous tree machine designs an instruction moved from level to level in the tree at the same time. But this is not necessary to assure that response pipe-lining never be disabled. In fact, it is not even to be desired if the machine is to perform instructions which can have multiple responses. So, to provide for multiple responses, we distinguish between two kinds of responses that a PE in our model can produce: *atomic* and *fusing*. Atomic responses are responses which are in their final form as soon as they are created, and are never combined with other responses. For example, the responses from a range-query instruction could be a collection of data items, each of which needs to be output as an atomic entity. Fusing responses are responses which need to be combined (fused) with other responses before the answer is in its final form. For example, the response from a **Delete**(k) instruction could be that each PE indicates by responding with a "yes" or "no" whether it deleted the specified item or not. For the host to know if the item was deleted all of the responses would need to be combined into just one answer: "yes" or "no." As we will show, by being able to designate whether an operation requires atomic or fusing responses we never have to disable response pipe-lining.

Each instruction will be accompanied by two extra bits, identifying their "type" when they are travelling through the machine: the color bit and the bubble-type bit. The color bit is assigned by the root processor to each instruction as it is input to the machine, assigning 0 and 1 to each one in an alternating fashion, so that consecutive instructions always differ in this bit. This is used to help maintain the order of instructions moving through the machine, as well as making sure that the bubbles of two different instructions never become inter-mixed. The bubble-type bit identifies whether the response to an instruction is atomic or fusing.

A simple communication rule can be used here to allow for multi-response instructions and still maintain response pipe-lining, accompanied by some way to prevent instruction inter-mixing. Simply put, a PE will not send a message to another PE unless the latter is ready to receive a message. Also, a PE must send messages to its two children simultaneously: it cannot send a message to one child without sending a message to the other (and therefore both children must be ready). Instruction inter-mixing is prevented by having null atomic responses (with $b = \emptyset$) act as fusing bubbles. We describe the scheme in detail below:

Input algorithm for an interior node PE:

> An interior node PE repeats the process of reading an instruction–response (i, b) pair from its parent (see Figure 4a), computing its response b' to i, and sending an instruction–response pair to each of its children (see Figure 4b). If the PE has no response to this instruction, then it sets $b' := \emptyset$. After its two children become ready, then the PE sends (i, b) to its left child and (i, b') to its right, simultaneously.

Input/Output algorithm for an effective last level PE:

> An effective last level PE repeats the process of reading an instruction–response pair (i, b) from its parent, computing its response b' to i, and then performing the output algorithm below as if it were an interior node PE which had just read b and b' from its children.

Output algorithm for an interior node PE:

> An interior node PE repeats the process of reading instruction–response pairs from its two children, and sending instruction–response pairs to its parent. It always completes processing the bubbles from one instruction before moving on to the bubbles from another. Note that a processor can test this just by checking the color bit of incoming bubbles. In particular, suppose the PE has just read two bubbles with the same color from its children (as in Figure 5a). The PE proceeds as follows:

- If the bubbles are fusing bubbles, then the PE combines the two bubbles and sends the result to its parent (see Figure 5b).
- If one of the bubbles is the null response ($\emptyset$), then the PE sends the other bubble to the parent and discards the null response. If they are both the null responses, then one null response is sent to the parent.
- If both bubbles are atomic, then the PE chooses one and sends it to the parent (see Figure 5c). This causes one of the PE's input registers to become empty, so the PE reads another bubble into that register (from the connected child PE). The PE then repeats this process as long as both bubbles are the same color. When one of the children sends a bubble from the next instruction, then the PE sends bubbles from the other child to its parent until it is done processing bubbles for this instruction (and both bubbles it has read from its children are the same color again).

Observation: Along any path from input root to output root, instruction bubbles will be in the exact order in which they were input.

Proof: As an instruction is moving down the tree, a PE sends the two bubbles from that instruction to its children simultaneously. So instruction bubbles cannot become out of order as they are moving down the tree. Bubbles cannot become out of order as they are moving up the tree, either. If an instruction calls for fusing responses, then its bubbles only combine with other fusing bubbles from the same instruction. If an instruction calls for atomic responses then the $\emptyset$ responses prevent the responses from two different instructions from becoming inter-mixed. ■

There is always some response that is being output at the root, so the machine is pipe-lining responses as fast as is possible: the interval time for the machine is constant. Note that this does not mean that instructions can be sent to the machine at a constant rate, however, since the time delay for an instruction depends on the number of bubbles that must be output for every instruction which precedes it.

With the added power of these techniques, providing for multi-selections becomes straightforward. We define a multi-selection operation, adding a new instruction, **Select**(F), to our instruction list. The instruction **Select**(F) returns all the (k, r) pairs (as atomic responses) for which the boolean function $F(k)$ is true. The only restriction we place on F is that a PE storing (k, r) receiving a **Select**(F) instruction can compute $F(k)$ without communicating with any other PE's. If $F(k)$ is true, then the PE generates an atomic response $b' = (k, r)$; otherwise, it generates $b' = \emptyset$, which is discarded as it moves up the tree (see the output algorithm above). Note that this is a generalization of the **Select**(j) instruction of the previous section, since the function F can use the rank of k as one of its arguments.

It is not hard to see that the rank-maintaining and redundancy-checking mechanisms of the previous section will work even in the presence of multi-response instructions.

Now, to perform, say, a d-dimensional range-query one encodes the query as a boolean function involving a key k, which is actually a tuple $(k_1, k_2, \ldots, k_d)$, to test if each of the d fields (coordinates) of k are within specified ranges. Since operation pipe-lining is never turned off, m consecutive d-dimensional range queries can be performed in $O(\log n + m + t)$ time, where t is the total number of (k, r) pairs which need to be output. This is a substantial improvement over what is possible on a sequential computer, in which the problem requires $O(m \log^d n + t)$ time with $O(n \log^{d-1} n)$ space [5].

Note that in the output algorithm for interior node PE's the method for choosing between two atomic bubbles from the same instruction was never specified. If we make the choice criterion that of always selecting the minimum (or maximum) of the two, then the output will be sorted, for each PE will be merging two sorted lists coming from its children. Being able to sort the output from an operation allows many other data operations to be realizable in the tree machine. For example, Song noted in [9] that once a tree-structured dictionary machine can sort its output, it can also perform projections and various grouping operations.

5 Conclusion

We have presented ways of extending any of the proposed tree-structured dictionary machines to be able to perform order statistics, such as selecting or extracting the j-th smallest element, and other data operations, such as multi-responses and sorting. Moreover, our model allows for general recovery from redundant operations, and allows multiple responses from single instructions, without compromising machine performance by disabling response pipe-lining.

References

[1] M. J. Atallah and S. R. Kosaraju, "A Generalized Dictionary Machine for VLSI," *IEEE Trans. on Computers*, Vol. C-34, no. 2, February 1985, pp. 151–155.

[2] J. L. Bentley and H. T. Kung, "A Tree Machine for Searching Problems," *Proc. 1979 Int. Conf. on Parallel Processing*, pp. 257–266.

[3] M. A. Bonuccelli, E. Lodi, F. Luccio, P. Maestrini, and L. Pagli, "A VLSI Tree Machine for Relational Data Bases," *10-th ACM/IEEE Int. Symp. on Computer Architecture*, Stockholm, Sweden, 1983, pp. 67–73.

[4] A. L. Fisher, "Dictionary Machines with a Small Number of Processors," *11th Int. Symp. on Computer Architecture*, Ann Arbor, MI., 1984, pp. 151–156.

[5] M. L. Fredman, "A Lower Bound on the Complexity of Orthogonal Range Queries," *Journal of the ACM*, Vol 28, 1981, pp. 696-706.

[6] T. A. Ottmann, A. L. Rosenberg, and L. J. Stockmeyer, "A Dictionary Machine (for VLSI)," *IEEE Trans. on Computers*, Vol. C-31, No. 9, September 1982, pp. 892–897.

[7] J. Shemer and P. Neches, "The Genesis of a Database Computer," *Computer*, Vol. 17, No. 11, November 1984, Pp. 42–56.

[8] A. K. Somani and V. K. Aggarwal, "An Efficient Unsorted VLSI Dictionary Machine," *IEEE Trans. on Computers*, Vol. C-34, No. 9, September 1985, pp. 841–852.

[9] S. W. Song, "A Highly Concurrent Tree Machine for Database Applications," *1980 IEEE Int. Conf. on Parallel Processing*, August 1980, pp. 259–268.

[10] D. E. Willard and G. S. Lueker, "Adding Range Restriction Capability to Dynamic Data Structures," *Jour. ACM*, Vol. 32, No. 3, July 1985, pp. 597–617.

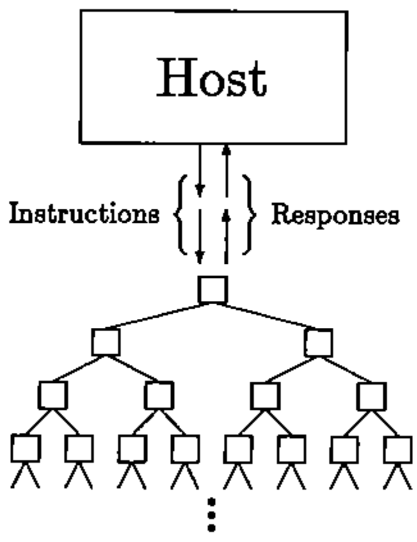

Figure 1: Instruction pipe-lining in the dictionary machine.

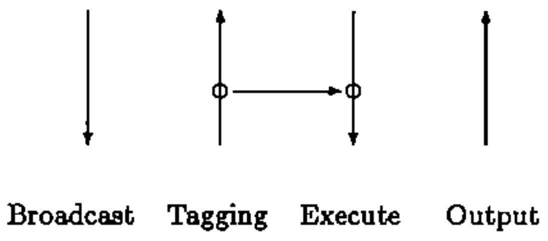

Figure 2: The phases of instructions flowing through the machine. The first round trip consists of a Broadcast phase and a Tagging phase; the second round trip consists of an Execute phase and an Output phase.

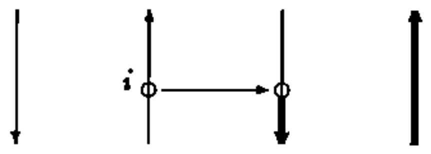

Figure 3: The relative rank of i, currently in some PE P, is the rank it would have after executing all the instructions (indicated by a heavy line) which are past P in their second round trip.

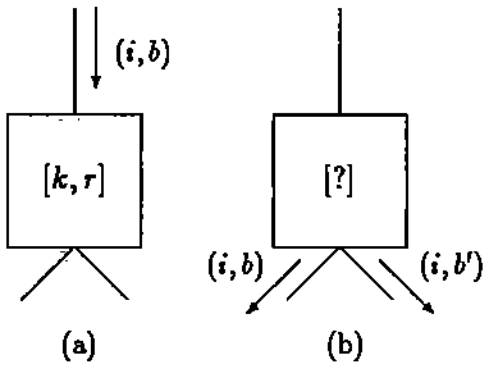

Figure 4: Input cycle. (a) a processor recieves an (i, b) pair from its parent; (b) a processor executes i on (k, r) and sends (i, b) left and (i, b') right.

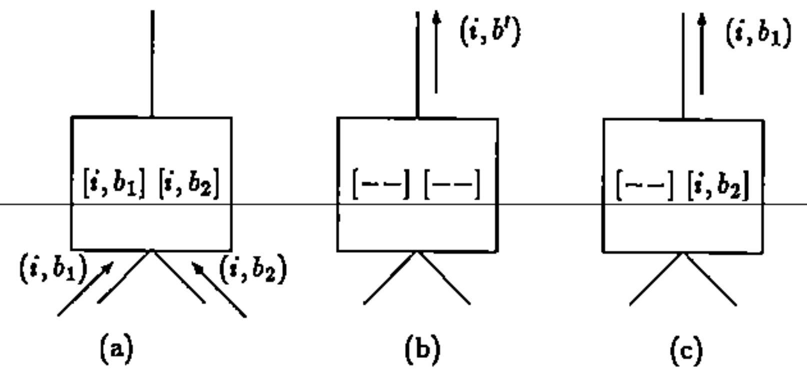

Figure 5: Output cycle. (a) input two response bubbles from children; (b) if i calls for fusing responses, combine b_1 and b_2 and send result b' to parent; (c) if i calls for atomic responses, pick one of b_1 and b_2 to send to parent.