THE INVERTED FILE TREE MACHINE:

EFFICIENT MULTI-KEY RETRIEVAL FOR VLSI

Hans-Peter Kriegel, Rita Mannss and Mark Overmars

RUU-CS-85-8
march 1985

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

THE INVERTED FILE TREE MACHINE:

EFFICIENT MULTI-KEY RETRIEVAL FOR VLSI

Hans-Peter Kriegel, Rita Mannss and Mark Overmars

Technical Report RUU-CS-85-8
march 1985

Department of Computer Science
University of Utrecht
P.O. Box 80.012
3508 TA Utrecht
the Netherlands

THE INVERTED FILE TREE MACHINE:

EFFICIENT MULTI-KEY RETRIEVAL FOR VLSI

Hans-Peter Kriegel
Lehrstuhl für Informatik I
Universität Würzburg
D-8700 Würzburg
West Germany

Rita Mannss
Institut für Informatik
ETH Zürich
CH-8092 Zürich
Switzerland

Mark Overmars
Vakgroep Informatica
Universiteit Utrecht
NL-3508 TA Utrecht
The Netherlands

Abstract:
In this paper we review index structures suggested for implementation on silicon. These structures do not efficiently support multi-key retrieval. We present two variants of the inverted file tree machine as efficient VLSI solutions to this problem improving query time of previous machines at least by the factor number of attributes specified in the query. Concerning non-redundant and redundant insertions and deletions plus multiple deletions, we suggest two phase methods to handle these update operations while continuously pipelining following operations. Both variants are compared with respect to their performance and space requirement. Furthermore, we describe a large class of multi-key queries all of which are handled on our machine with the speed up factor number of specified attributes.

1. Introduction

The problem of retrieving all the records satisfying a query involving a multiplicity of attributes, known as multi-key retrieval or associative retrieval, is a major concern in physical database organization. Physical database organization deals with the assignment of physical data records into pages of the secondary storage and the methods of retrieving the associated pages and records in response to a given query. The most commonly used method to support retrieval is an index or directory which relates combinations of attributes values to the physical records which have these value combinations. Various software methods for the construction of such indexes for different types of queries are at the disposal of the database designer, for a survey see [Kri 82], [Kri 84] and [NHS 84].
Search for hardware-oriented solutions to database problems, in the context of design and implementation of the so-called database machines, has been around for more than ten years. With recent advances in Very Large Scale Integration (VLSI) technology, the focus has shifted to hardware solutions directly implementable on silicon.
In this paper we will review index structures suggested for implementation on silicon. None of these structures efficiently supports multi-key retrieval. We present the inverted file tree machine as an efficient VLSI solution to this problem.

For completeness sake, let us review some terminology. In the following, we consider a collection of N records which we call a file or a database. Each record, more exactly k-dimensional record, consists of an ordered k-tuple $x = (x_1, \ldots, x_k)$ of values and some associated information. x_i is called value of attribute A_i, $1 \leq i \leq k$. An exact match query specifies a value for each attribute.

2. Survey of previous work

VLSI technology offers the potential of implementing complex structures and algorithms directly in hardware. With VLSI circuitry increasing in speed and densitiy at an amazing rate, there is ample motivation in developing customized designs of algorithms implementable on silicon. However, since computation is cheap in VLSI and communication determines the performance, structures and algorithms have to be reconsidered. A lot of attention has been paid to the problem of designing an index structure implementable on silicon. Bentley and Kung [BeKu 79] were the first to suggest a tree machine which can solve all of the decomposable searching problems. Let us consider a file of one-dimensional records consisting of a key (primary key) and a location where the record is stored on external storage. The following dictionary operations are of basic interest: determine whether the record with a given key is a MEMBER in the file, INSERT a new record in the file and DELETE an existing record from the file. The tree machine suggested in [BeKu 79] consists of a collection of processors connected as a binary tree, see figure 1. There are three kinds of processors in the machine: circles, squares and triangles. Each record (i.e. the key and the location) resides in a square which is provided with some logic to carry out a limited repertoire of instructions. The circles broadcast streams of instructions and data to the squares where the instructions are executed in parallel. The squares compute preliminary results which are then combined by the triangles to produce the final result being output through the root triangle. The structure of the tree machine is that of two complete binary trees, one being the mirror image of the other. Consider now the problem of performing the operation "Is the record with key 17 a MEMBER in the file?" We accomplish this

Proc. FODO 1985, Kyoto

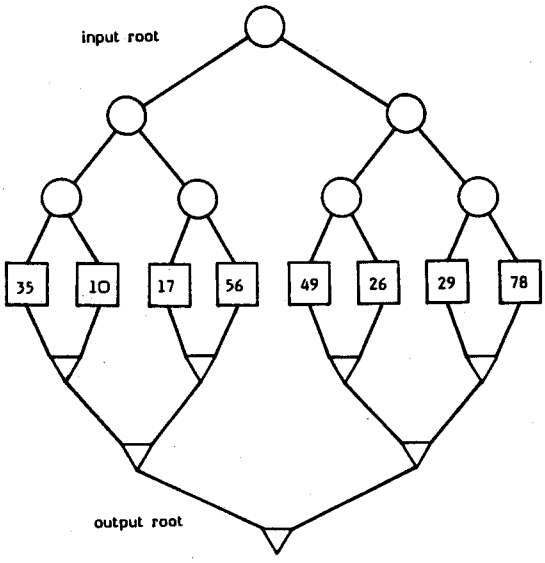

Figure 1

by inserting 17 into the input root and broadcasting it together with a code for the MEMBER operation down the tree - $\log_2 N$ steps later the value 17 and the operation code will arrive at all of the N squares. At that point we compare the key stored in each square to 17 and set a bit to one if the value is equal to 17 and zero otherwise. We can now combine the bits together through the bottom portion of the tree machine by letting each triangle compute the logic or of its two inputs. So after a total of $2 \log_2 N$ time units have passed since the query was posed, a single bit emerges from the output root telling whether or not the record with key 17 is a MEMBER of the file. For a single MEMBER operation, the performance of the tree machine is not exciting. However, if a sequence of MEMBER operations has to be performed, then we can take advantage of the very regular data flow by pipelining the process of answering those queries. As the key of the first operation is going down the tree, the next operation can follow one step behind, and so on. Thus a sequence of m MEMBER operations can be answered in $2 \log_2 N+m$ time units where N is the number of squares in the tree machine. In [BeKu 79] it is shown how the tree machine can solve all of the decomposable searching problems.

Since 1979, many improvements have been suggested to the Bentley-Kung machine. A major drawback is that the Bentley-Kung machine does not allow deletions to be pipelined. Suggesting a new space allocation scheme, this problem was solved by Song [Son 81] besides many practical implementation problems. However, Song's suggestion does not allow pipelining of redundant insertions and deletions. We call an insertion redundant, if it tries to add an already existing key and we call a deletion redundant, if it tries to remove a not existing key. The problem of handling redundant insertions and deletions was independently solved by [AtKo 81] and [ORS 82]. In additon to handling redundant insertions and deletions, both machines exhibit logarithmic performance in the problem size and not in the hardware size thus mimicing the performance of software implementations more closely.

There is one problem, however, for which (with one exception, see remark at the end of this paper) no major improvements have been suggested since the Bentley-Kung machine: the problem of multi-key retrieval. Consider an exact match query with k attributes specified where we want to count the satisfying records. In [BeKu 79] the following approach is suggested: we store the k-dimensional records in the (larger) square processors and sequentially broadcast all the k specified attribute values down the tree keeping track in each square processor whether it has satisfied all the attribute values shipped so far. We load a one if all conditions have been satisfied and a zero otherwise and combine the partial results by having the triangles sum their inputs. Thus m exact match queries with k attributes specified can be performed in $2 \log_2 N+m \cdot k$ time units. This is obviously not very efficient. The idea of serially feeding the k specified attributes into the input root clearly contradicts the paradigm of VLSI algorithms of exploiting parallelism as best as possible. On top of that, the square processors are at least k times as large as in the case of single-key retrieval and need some additional logic to ensure comparing values of the same attribute. This contradicts the design guideline that the logic and storage needed at each processor be as small as possible [Kun 79].

3. The inverted file tree machine

In this section, we present the inverted file tree machine (IFTM) as an efficient solution to the multi-key retrieval problem. For records with k attributes the IFTM consists of k attribute trees which are top parts of normal tree machines, each responsible for one attribute, see figure 2.
Now the k attibute values of the ith record are stored in the jth square processor of each of the k attribute trees, $1 \leq j \leq N$. Initially, we have $j = i$. For each i, $1 \leq i \leq N$, all the ith square processors of the k attribute trees are the leaves of a binary tree of logical ANDs, the record tree. Each root of a record tree contains the memory location of the record stored in its leaves. The N roots of these record trees are then the leaves of the result tree, a binary tree of triangles identical to the bottom part of the Bentley-Kung machine. Now, given an exact

match query, the k specified attribute values are fed into the input roots of the attribute trees in parallel. After arriving at the squares and being compared with the stored attribute values, the record tree of each record reports a one if the record satisfied all k specified values.

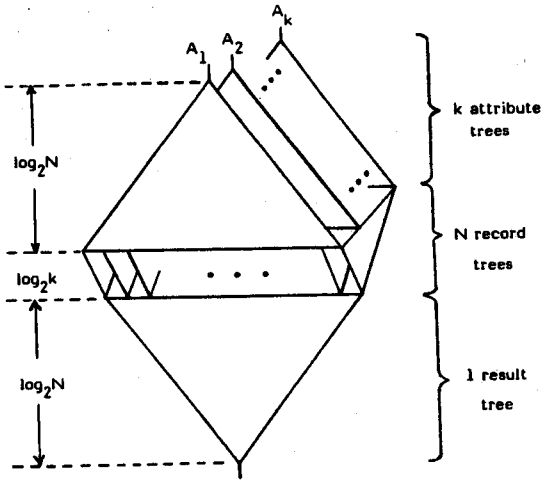

Figure 2

The binary tree of triangles then combines the bits by letting each triangle compute the sum of its two inputs, exactly as in the Bentley-Kung machine. Figure 3 shows an IFTM for two attributes storing the records (a,1), (a,2), (b,3) and (c,5).

Let us now compare the IFTM and the Bentley-Kung machine storing k-dimensional records with respect to their performance. In the IFTM we can perform m exact match queries with k attributes specified in $2\log_2 N + \log_2 k + m$ time units, which is an improvement by the factor k. Furthermore, the "inverted file time units" are much shorter because the square processors are much simpler and k times smaller. If exact match queries with at most one answer ask for the location of the satisfying record, the root of the record tree reports the location instead of a one and the location flows through the result tree. The price that has to be paid for the at least factor k improvement in time is more space needed by the k attribute trees. However, the IFTM needs less than k times the space of the Bentley-Kung machine storing k-dimensional records. Similar as in the Bentley-Kung machine, we can perform more types of queries, which will be handled in section 6. The next section shows how the IFTM handles insertions and deletions.

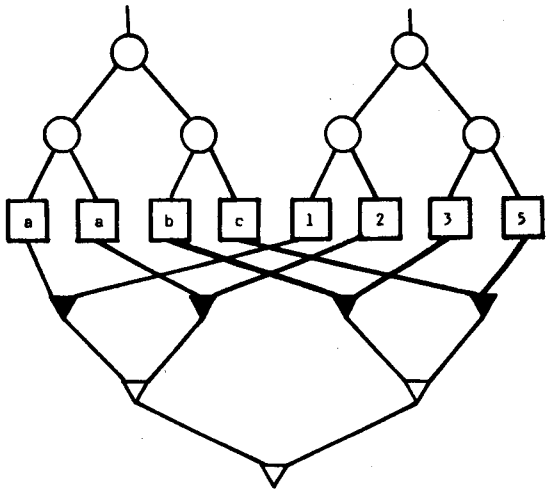

Figure 3

4. Insertions and Deletions

The only update operation which can be carried out with the present tools without creating the problem of inconsistency is the non-redundant insertion. In this section, we will describe how the IFTM can handle non-redundant and redundant insertions as well as all of the following types of deletions: non-redundant deletions, redundant deletions and multiple deletions where one DELETE operation is supposed to remove a list of records e.g. specified by a partial match query. The following method for update operations is based on Song's space allocation scheme [Son 81] which the reader is assumed to be familiar with. However, Song's scheme is only capable of handling non-redundant insertions and deletions. To include the more complicated update operations, we introduce two phases for each operations.

4.1 Deletions

The design of the IFTM implies the problem that records which are supposed to be deleted can be identified only at the root of the record trees. The records have to be removed, however, from the square processors of the attribute trees. Thus all operations following a deletion operate on an inconsistent file for a short time, if pipelining is not turned off after a deletion. Obviously, turning off pipelining is really crucial to the efficiency of VLSI algorithms. In the following, we will suggest a two phase method for deletions which allows continued pipelining after a DELETE operation by modifying the re-

sults of following operations until the record(s) has (have) been physically removed from the square processors.

Our method necessitates the following extensions to the IFTM:

(i) Each square processor of the attribute trees accomodates in addition to an attribute value a d-bit which is set in order to indicate that the attribute value will be physically removed.

(ii) It is possible to broadcast information in the record trees from the square processors to the roots of the record trees and vice versa. Thus the record trees have two strata: top-down logical AND is performed, bottom-up we apply "AND FALSE" to partial results of following operations on their way top-down in order to negate positive partial results and we broadcast an instruction which sets the d-bit in the square processors.

(iii) Each root of a record tree contains in addition to the location of a record an f-bit.

(iv) The result trees have two strata: top-down the results of operations are computed, bottom-up physical deletion requests are broadcast from the root of the result tree to the roots of the record trees using the location as a primary key.

With the extensions (i) - (iv) implemented in the IFTM a DELETE operation is performed in the following way: each DELETE is carried out in two phases: (1) the identification phase and (2) the removal phase.

(1) The identification phase:
The identification phase identifies the record(s) to be deleted, takes measures to guarantees consistency of the file and modifies results of operations directly following the DELETE operation. This is accomplished in the following way. A code for the DELETE operation together with the attribute value is broadcast down each of the attribute trees. The results of the comparisons in the square processors are AND - combined in the record trees. Reaching the roots of the record trees, the record(s) to be deleted is (are) identified. Each root of a record to be deleted broadcasts two instruction streams. One stream sends the location of the record down to the root of the result tree. The other stream traverses the record trees upwards, negates positive partial results of operations directly following the DELETE and sets the d-bit in the square processors indicating to following operations that the marked attribute value is determined to be removed. Using two different strata in the record trees, the bottom-up stream and the top-down stream meet at the AND processors. In order to guarantee that partial results of all following operations (not just of every second following operation) are negated, two consecutive "AND FALSE waves" traverse the record trees upwards. The second wave is initiated by the f-bit in the root of the record tree which is set at the time the first wave leaves the root on its way upwards.

(2) The removal phase:
In the removal phase the attribute values marked with a set d-bit are erased. This is done by broadcasting the location of the record upwards in the result tree. After identifying the root of the appropriate record tree using the location as a primary key, the square processors in this record tree are freed using Song's space allocation scheme for the simple tree machine with one tree controller and one FIRSTFREE count for the IFTM. In order to guarantee correct behavior of the space allocation scheme, the removal phases may not start in parallel at the roots of the record trees, but have to be sequentialized at the root of the result tree.

Thus if b records have been found in the identification phase of one DELETE operation, b consecutive pipelined removal phases are started from the root of the result tree. This ensures that the attribute values of one record are in the square processors with the same number in all the attribute trees and thus they are in the leaves of one record tree.

Summarizing, we can state that the suggested method for deletions supports continued pipelining of operations following an arbitrary deletion and it preserves the locality property of the data flow by starting the removal phase where the identification phase ends: at the root of the result tree.

4.2 Insertions
Non-redundant insertions can be easily handled in the IFTM by applying Song's scheme for the simple tree machine to the IFTM using one tree controller for all k attribute trees. The method suggested here additionally supports redundant insertions by making use of two phases:
(1) the combined identification and write phase and (2) the removal phase.

(1) The combined identification and write phase:
This phase identifies the possibly existing record and treats the record in the same way as the identification phase of the deletion. In parallel, the record is non-redundantly inserted into the IFTM.

(2) The removal phase:
The removal phase is identical to the one described for deletions. If the first phase has identified a record to be already existing, the removal phase is applied. If not, we have a do nothing or inactive removal phase.

Obviously, the above method handles non-redun-