A-TREES: AN INDEXING ABSTRACTION FOR ORDERED AGGREGATES

by

W. Bradley Rubenstein

Memorandum No. UCB/ERL M86/77

12 September 1986

COVER PAGE

A-TREES: AN INDEXING ABSTRACTION FOR ORDERED AGGREGATES

by

W. Bradley Rubenstein

Memorandum No. UCB/ERL M86/77

12 September 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

A-TREES: AN INDEXING ABSTRACTION FOR ORDERED AGGREGATES

by

W. Bradley Rubenstein

Memorandum No. UCB/ERL M86/77

12 September 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Table of Contents

List of Figures

A-trees: An Indexing Abstraction for Ordered Aggregates

W. Bradley Rubenstein

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

ABSTRACT

A number of current proposals that extend B-trees to handle ordered relations are generalized into a single data structure, the A-tree. A-trees are shown to support a wide variety of *ordered aggregates*, such as line numbers and running totals, over ordered relations. The extensions to the relational database system INGRES, and its query language QUEL are presented. The performance of A-trees is shown to be similar to that of B-trees. The issue of supporting complex orderings and graphs using this structure is also explored.

1. Introduction

Ordered relations have been developed to model data in which order among objects plays a fundamental role. Several independent proposals have been made that provide similar facilities for supporting this type of information. Examples are OB-trees [StR80], time-line indices [Rub85], and large storage objects [CDR86].

Unlike sorted relations, ordered relations determine an ordering among records that is not necessarily dependent on the data contained within a particular record. For example, the ordinate position (i.e. position within an ordering in an ordered relation) of a record depends on the number of records prior to it in the ordering, not necessarily on the contents of that or any previous record.

This last example demonstrates a common feature of records within the ordered relation. They often possess attributes which are an aggregate function of other records in the ordering. In the above example, the ordinate position of a record can be determined by applying the "count" aggregate to all previous records in the ordering. An aggregate used to provide such a virtual attribute to records within an ordered relation is known as an *ordered aggregate*.

This paper presents a system for specifying and manipulating ordered relations, and efficiently processing user defined ordered aggregates over those relations. The remainder of the introduction gives examples of ordered aggregates as they have been proposed in previous approaches to the problem of maintaining ordered relations. It then reviews the proposal for user-defined aggregates in INGRES[Han84] and Ordered B-trees[Lyn82] which form the basis for our proposal.

This research was sponsored by the National Science Foundation under grant DMC-8504633.

Section 2 then presents the proposed A-tree data structure. The actual structure is very similar to the B-tree [BaM72], and familiarity with that structure is assumed.

Section 3 discusses the modifications to the database system needed to support A-trees. User-defined ordered aggregates are supported by allowing user-coded routines to be registered with the database system. The system also implements the control structure necessary to apply these routines to the user's A-trees. The algorithms for growing and shrinking the tree structure upon insertion and deletion are also outlined here.

Various extensions to the query language QUEL are given in section 4. These include a **define aggregate** operation to register ordered aggregates, a **modify** command to instantiate the A-tree structure, and a new version of the **append** command to perform ordered insertions into an ordered relation. Familiarity with QUEL [HSW75,Rel84] is assumed.

A simulation of the A-tree system shows that performance of this data structure, in both time and space efficiency, is similar to that of B-trees. These results are discussed in section 5.

There are two more complex cases of ordered data which we also consider in section 6. First, we discuss the case where a relation can be cast into multiple orderings at one time (for example, a relation ordered one way by one key field, and another way by a different key field). Section 6.1 discusses the use of A-trees as secondary indices to support these multi-orderings.

Then, in section 6.2, the concept of *hierarchical ordering* is explored, where the data in a relation is ordered within partitions. Logically, each record in such a relation is subordinate to a parent (all the records under a single parent constitute a partition). The ordering of records is maintained independently under each parent. Examples of the use of this construct are presented, along with extensions to the A-tree structures necessary to support them.

An alternative implementation to supporting multi-orderings is currently being explored. This involves storing a multi-ordered relation as an indexed graph. The advantages of this are twofold: the graph representation is almost always more space efficient, and graph partitioning techniques may result in superior query performance. These issues are detailed in section 7.

Section 8 briefly discusses other issues currently under investigation. These include additional implementation alternatives, performance optimizations and concurrency control issues. Finally, conclusions and suggestions for future research are presented in section 9.

1.1. Three examples of Ordered Aggregates

Ordered relations have been suggested as a means of representing lines of text in a relation [SSK82]. In this proposal, each line of text is stored as a record in the TEXT relation. The line number (LID) is viewed as an attribute associated with each line of text. Rather than having the user update every LID attribute whenever an insertion or deletion is performed, a special purpose extension of the B-tree, called an OB-tree [StR80] is used to maintain correct LID attributes for each record without user intervention.

The OB-tree is built over the records of the TEXT relation, which are stored in order. Each internal record of the OB-tree maintains a count of the records below it. Therefore, by traversing the OB-tree from root to leaf, the ordinate position of any record, that is, the LID, can be quickly ascertained. Furthermore, upon insertion of a leaf record, the counts can be efficiently updated, again by traversing the the tree from root to leaf.

An almost identical approach is used in the EXODUS system [CDR86] for storing large data objects. A large data object consists of a variable length string of bytes split among several disk pages. An OB-tree index provides efficient access to any substring at a given ordinate position, and algorithms for inserting blocks of bytes at an arbitrary point in the string were developed. In this case, the records of the index contain counts of the number of bytes in the leaf pages below.

In both of these systems, an aggregate function is being applied to the records of the ordered relation. In particular, for each record, the **count** aggregate function is calculated over all records previous to it in the ordering. The OB-tree index allows this aggregate value to be determined dynamically (so as to reflect the current state of the relation) and efficiently (since the entire set of records preceding a particular record need not be accessed).

In a proposal for managing events and processes, the author presented a similar extension of B-trees [Rub85]. Each event in the ordered EVENT relation is stored along with the amount of time until the start of its succeeding event (the *delay*). In this relation, the start time of any event can be calculated by summing the delays from the beginning of the ordering up to the given event. Each record of the B-tree index contains the sum of all the delay field values under it. By traversing the tree from root to leaf, the sum up to a given leaf record can be ascertained. The start times for each event can thus be maintained dynamically and efficiently by the system.

In addition to the count and sum ordered aggregates, other more application specific aggregates might be useful. For example, consider the relation RUNQUEUE that contains the length of a system queue of runnable processes, sampled at regular intervals. We define the *load average* at a given point in time to be the exponentially weighted average of queue length from time zero to the given point. The value of an exponential average is given by:

$$\bar{x}_i = (1-s)\sum_{k=1}^{i} s^{k-i} a_k$$

where:

a is the ordered set of attributes to be averaged,
$\bar{x}_i$ is the exponential average at a_i,
s is the scale factor for weighting the average ($s < 1$).

The system can efficiently provide the load average at each sample point by using an index which contains in each internal record the weighted average of the run-queue lengths in the subtree under that record. Given the appropriate algorithms for accumulating this internal state information, this index provides dynamic calculation of the exponential average aggregate function. The load average values will remain correct in the face of arbitrary updates and deletions to the ordered relation itself.

1.2. User-Defined Aggregates

From the above discussion, it is clear that ordered aggregates are application dependent, and that it would be disadvantageous to hard-wire a predetermined set of indices into the database system. We offer a more general solution to the management of ordered relations, which admits the above proposals as special cases, along with a number of other interesting cases. The problem of maintaining ordinate information over a set of data is seen as a special case of maintaining user-defined aggregate information over ordered sets.

The proposals described in this paper find their basis in various extensions to the INGRES relational database system [HSW75]. This system has been extended to provide a facility for defining abstract data types and abstract data type operators [Fog82, Ong82]. The result, ADT INGRES, has been further extended to incorporate user defined aggregates, including aggregates over abstract data types [Han84]. Our proposal further develops this work by extending ADT INGRES to support user-defined ordered aggregates over ordered relations. First, we outline the proposal presented in [Han84].

Determining aggregate values is performed by scanning sequentially through the records of the relation. At each record, particular values are passed to a routine which accumulates state information used to calculate the aggregate value.

By allowing the user to define the nature of the state information, as well as the routine used to incrementally accumulate this state based on a succession of data values (called the *Next()* routine), a wide variety of aggregate functions can be flexibly implemented by the user, with only a general control system embedded into the data manager. For user-defined aggregates, this general control system allows the user to initialize a scan, presents selected fields of the data records in sequence to the Next() routine, and finally calculates a result based on the resultant accumulated state information.

By extending this general control system to accommodate traversals through an index, the user is given the flexibility to incorporate arbitrary aggregate state information into the internal records of the index. This provides just the functionality necessary to implement ordered aggregates. The resulting index, whose functionality is largely under user control, is known as an A-tree.

2. Implementation of the Index

2.1. The Data Structure

The data structure used to implement A-trees is a tree of disk pages. These disk pages are divided into *internal* and *leaf* pages. Data records are stored at the leaf nodes of the tree, and internal nodes contain control information determined by the *modify* command that generated the tree. Every page except the root contains a pointer to its parent page in the tree. Every record in an internal node contains a pointer to a child in the tree. Finally, leaf node pages are doubly linked together. Figure 1 shows a small example of this configuration.

2.2. Operating Characteristics

The data structure has the following characteristics:

All leaf nodes are at the same distance from the root of the tree. This is guaranteed by the insertion and deletion algorithms, as with B-trees.

All pages are at least half full, except possibly the root. In other words, if p is the number of bytes on a page, and r_i is the size of the i'th record (the records need not all be the same size), then the total size of records on a non-root page is bounded above and below:

$$\frac{p}{2} \leq \sum_i r_i \leq p$$

The data records are ordered, although there need not be a key attribute that generates the ordering. This is made possible by allowing the user to explicitly specify the position of an

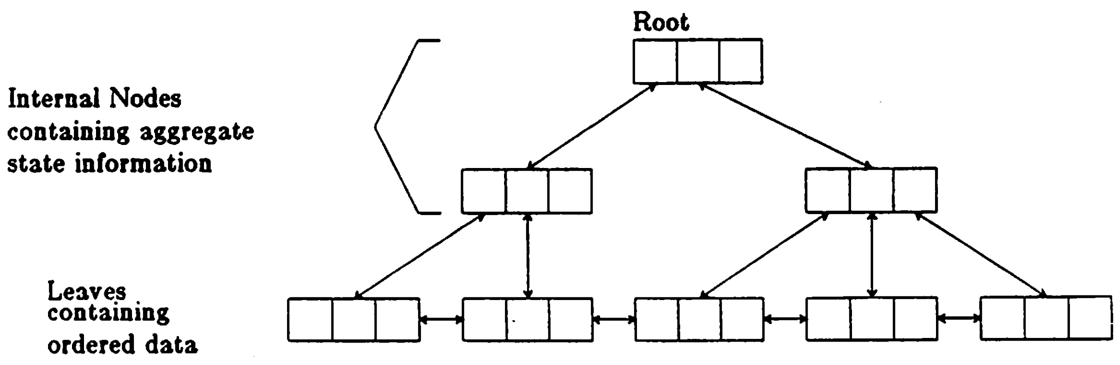

Root
Internal Nodes containing aggregate state information
Leaves containing ordered data

Figure 1. The A-tree Data Structure

inserted record (via the **append before** and **append after** commands, described below).

Any aggregate value computable via the A-tree index is also computable by a linear scan of the relation. The index serves as a mechanism to improve on the amount of computation and on the number of page accesses required by the linear scan. The internal nodes, in effect, contain summary information, allowing a number of pages worth of leaf data to be incorporated into an aggregate calculation without actually accessing those leaf pages.

3. Extensions to the Database System

3.1. User-coded Routines

User coded routines allow the customization of the A-tree data structure to a variety of applications. These routines are called on every access to a routine that refers to aggregate attributes, and on every modification of a relation with an A-tree index. The routines that must be provided by the user in our implementation are summarized in figure 2.

3.2. Control Flow for Traversing the A-tree

There are two types of tree traversal which the system must implement, using the user-coded routines: top down and bottom up.

In top down traversal, a search value is provided, and the system needs to find the leaf page on which that value would be found, or should be inserted. Starting at the root, records are scanned sequentially using NextLeaf or NextInner (as appropriate) to accumulate state information of the aggregate. When a record causes the cumulative state to exceed the search value, the child page of this record is then traversed. This can continue until a leaf record is found.

Bottom up traversal is used for two purposes. Given a particular record in a leaf page, it can determine the value of any aggregate attributes on that record. The second use is whenever a page is modified by insertion, deletion, or splitting, a bottom up traversal is needed to update the summary state information in the internal nodes of the A-tree.

To calculate the aggregate value for a given leaf record, we scan the page on which the record lies up to that record. The state information is accumulated using NextLeaf. The parent record of the page is determined, and the page on which this parent record lies is sequentially scanned up to the parent record. This continues up the tree until the root is scanned. The resulting accumulated state determines the aggregate value for the leaf record.

Name	Function
InitializeScan(State)	Initialize the cumulative state.
NextInner(State,Constants,NextState)	Accumulate more state.
NextLeaf(State,Constants,Variables)	Accumulate a leaf value.
Result(State,ResultValue)	Determine resultant aggregate value.
Compare(Value1,Value2)	Compare and order two aggregate values.

Figure 2. User Routines for Ordered Aggregates

To correct state information when a page is modified, we merely rescan the page with NextLeaf or NextInner. This results in a cumulative state that is then inserted into the parent record for that page. The parent page is thus modified, and the correction must percolate recursively to the top of the tree.

These top-down and bottom-up traversal routines suffice to maintain the A-tree under retrieval and update. Two additional routines are necessary for insertion and deletion, namely those for splitting and merging pages.

3.2.1. Splitting Pages

It may happen, after an insertion of a record onto a page, that the set of records now on the page is larger than the page size. In such an instance, the page must be split. This is done as follows:

Split a Page

Given: an overfull page c.

1. If c is the root, then split the root, then go on to step 2.
2. Determine r_c, the record in the parent of c that points to c. This must exist, since c is not the root.
3. Get a new page c'.
4. Move the last half of the records on c onto c'.
5. Make an internal record $r_{c'}$ that points to c'.
6. Point c' to the parent of c.
7. Insert $r_{c'}$ after r_c (recursively).
8. Fix the summary information from page c.

When making the internal record in step 5 for the new page c', the system must initialize a scan of c' and call NextLeaf or NextInner repeatedly for each record. The resulting state information is stored in the internal record $r_{c'}$. Later, in step 8, the only the state information for the c need be updated, since c and c' have a common parent.

Notice that the recursive step 7 may itself cause the parent page to split. The recursion occurs because **append after** may call **split page** which in turn calls **append after**.

In the case that the root is over full, it must be split. This can be cleanly accomplished by the following algorithm:

Split the Root

Given: the root page c.

1. Get a new page, which will be the new root, p.
2. Make an internal record r_c that points to c.
3. Point c to p.
4. Insert r_c into the (empty) page p.

At this point, if c, that was previously the root, is overfull, it can be split as usual, since it is no longer a root node. Again, step 2 requires that a scan be performed on page c in order to determine the state information to be stored in r_c.

3.2.2. Merging Pages

The merge operation for deletion is analogous to the splitting operation for insertion. It operates as follows:

Merge Page

Given: a page c that is less than half full.

1. Set c' to be the page before c.
2. Set r_c to be the parent record that points to c.
3. Set $r_{c'}$ to be the parent record that points to c'.
4. Take all the records of c and append them to the end of c'.
5. Delete (recursively) r_c from the parent of c.
6. If c' is overfull, split it.

The effect of this procedure is to make sure, after deletion, that all pages are at least half full. If the result of merging a page with its predecessor c' (step 4) is to overfill c', then splitting that page will redistribute the records so that the resulting pages are at least half full.

Analogous to the operation of splitting the root on insertion, is the case where a root can be deleted. This happens when the root is not a leaf, and after deletion contains only a single internal record. The algorithm is:

Delete Root

Given: A root page p that is not a leaf, containing a single record.

1. Set c to be the sole child of p.
2. Free page p, and set the new root to be c.

4. Extensions to the Query Language

Ordered Relations may be generated in two ways. First, the system may take an existing relation, and build an A-tree on top of it. This requires that the records of the relation are already ordered. For example, there could be an existing ordering index such as a B-tree on the relation, or the system might guarantee that a particular ordering can be maintained by the user. This could be accomplished by saying that **appends** to a relation always place records at the end of a heap (the system promises not to reorder the records). Alternatively, as part of the specification of the A-tree, the user can indicate a sort key by which the records should be ordered prior to building the actual tree.

The second method of generating an ordered relation is to build an A-tree, either over an existing relation as above, or over an empty relation, and then grow the structure by performing successive ordered insertions.

In any case, there are three main steps to providing A-trees as a database service.

1) The ordered aggregates maintained by A-trees must be registered with the system, indicating the location of user-coded routines to be called by the system, as well as other parameter and type checking information.

2) The user must take a relation which has been registered with the system (in INGRES, this is done with the **create** command) and modify its storage structure to type A-tree. At this time, the user determines which aggregates are to be maintained for the particular relation.

3) Individual ordered insertions are performed by the user.

In addition to these operations, retrievals and deletions may be performed on an ordered relation just as with any other relation type.

4.1. Registering Ordered Aggregates: The define Command

This syntax is taken from Hanson [Han84].*

```
define [ ordered ] aggregate aggregate_name
    ( parameter { , parameter } )
    returns type
    file = "file name"
```

The **ordered** keyword is an extension to Hanson's proposal. It is used to indicate to the system that the aggregate will be used for an ordered aggregate index. For example,

```
define ordered aggregate osum
    (value = numeric)
    returns typeof(value)
    file = "~bradr/aggregates/osum.o"
```

Ordered aggregates are not used in queries. They are employed by the system to maintain aggregate values over an ordered set of records. They are therefore associated with a particular relation at the time the relation is defined (or modified) to be of type A-tree. Because of this, ordered aggregates, from the system's point of view, have two syntactic restriction over ordinary aggregates.

1) They never serve as aggregate functions. In other words, they are not partitionable via a by-clause within the parameter specification.

2) The parameters to the aggregate are either constants, or generic attribute names of the ordered relation. They are not "range_variable.attribute" terms, as with regular aggregates.

In fact, the only command in which ordered aggregates are referenced is the following **modify** command.

4.2. Creating Ordered Relations: The modify Command

A relation can be modified to be of type A-tree with the **modify** command. Here is its syntax:

```
modify_command ::=
    modify relationname to A-tree
        ( virtual_attribute_list )
        [ on field_name_list ]

virtual_attribute_list ::=
    field_name = aggregate_name ( parameter_list )
    { , field_name = aggregate_name ( parameter_list ) }

field_name_list ::= field_name { , field_name }

parameter_list ::= parameter { , parameter }

parameter ::= field_name | constant
```

The **modify** command takes a physical relation, and logically orders it. It also registers with the system what ordered aggregates are to be maintained over the relation. Pictorially, we can represent this logical ordering by drawing arrows from each record to its successor, as in figure 3. This does not mean to imply that these pointers are necessarily stored in the ordered relation (although we will see that this is one possible implementation).

* In syntax specifications, square brackets indicate optional clauses, and curly braces indicate elements that may be repeated zero or more times.

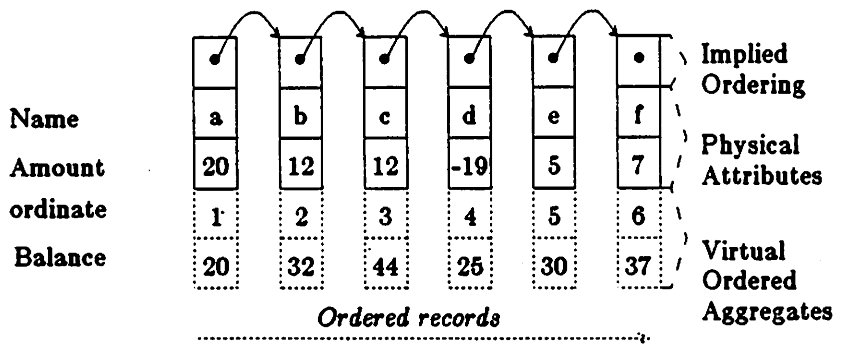


Figure 3. Records in an Ordered Relation

The **modify** command has two clauses:

• (*virtual_attribute_list*)

This clause makes ordered aggregates available within the relation. The values of these aggregates are maintained by the system. They appear to the user as read-only attributes associated with each record in the relation. The aggregates have been registered with the system by *aggregate_name* via the **define aggregate** operator. The *parameter_list* consists of a list of attribute names and constants. The attribute names must be already defined over the relation to be modified.

• on *field_name_list*

The given field names are to serve as a sort key, defining an implicit ordering among the records of the relation. The relation is first sorted on the list of fields, then linked according to the resulting sort order. In fact, providing sort keys in this way can be implemented as just another aggregate function.

The two methods of determining the position of an inserted record upon insertion into an ordered relation, either by key or by explicitly specifying the position, are unfortunately not compatible. If keys are specified in an on-clause of the **modify** command, then explicit positioning of records is disallowed, since the user would then be free to violate the sort order of keys by explicitly inserting records out of order. If the modify command has no on-clause, then only explicit ordering can be used (this is done by the **append before** and **append after** commands described below).

4.3. The append Command

For insertion into ordered relations, the syntax of the append command is extended as follows:

```
append
    [ [to] table_name |
      after range_variable |
      before range_variable ]
    ( target_list )
    where qualification
```

The standard form, **append to** *table_name* now has two alternatives, to append before or after records in an ordered relation. For example, to insert a line of text prior to the eighth line of text, one would say:

```
range of t is TEXT

append before t (text = "inserted line of text")
    where t.Lid = 8
```

Although the qualification in the above example produces a single record, this need not be the case. For example, we might want to insert a line of text prior to every line that has the word "special" in it:

```
append before t (text = "mark following line")
    where t.Text = "* special *"
```

4.4. Reference to Ordered Aggregate Attributes

The syntax of appends, deletes, replace, and retrieve is otherwise unmodified, except that one may refer to any of the virtual attributes introduced by the **modify** command. These attributes function syntactically just like physical attributes in the relation, subject to the constraint that they cannot be written.

For an example, consider a command to identify adjacent lines of text that are identical. The appropriate query is:

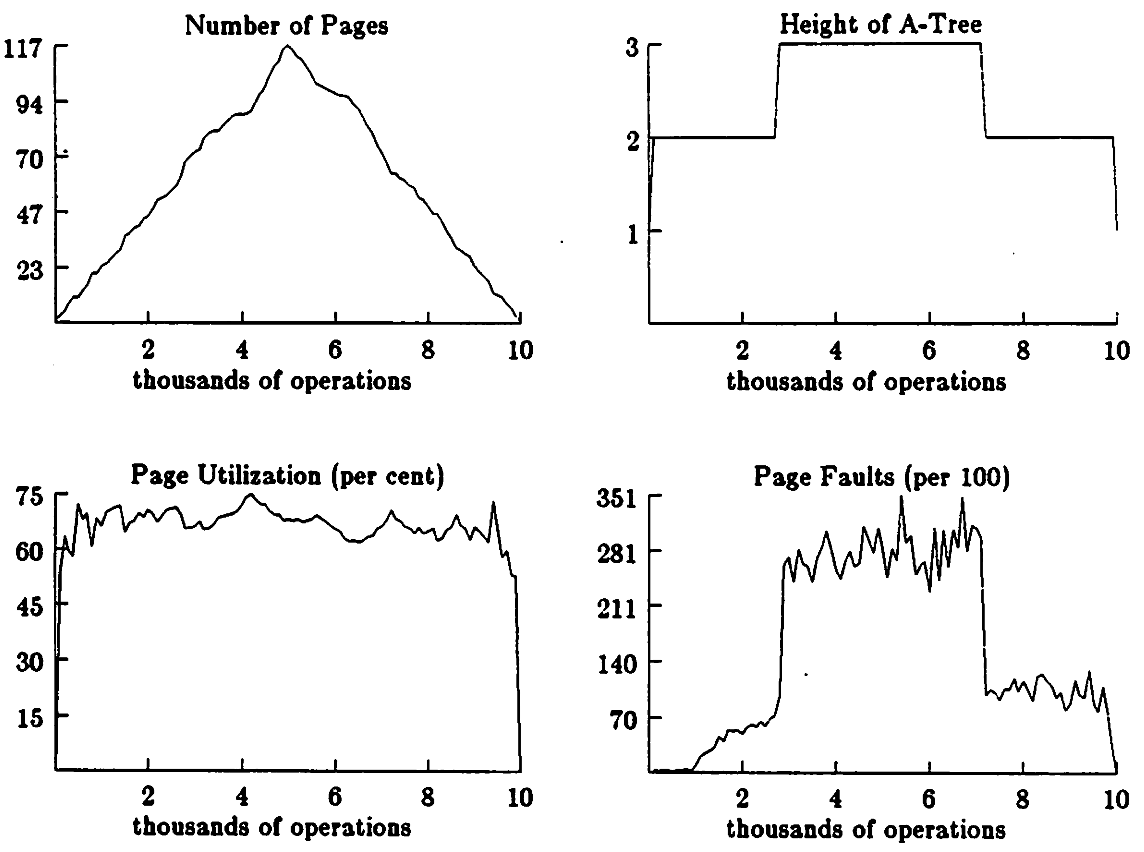
```
range of t1,t2 is TEXT

retrieve (t1.Lid)
    where t1.Lid + 1 = t2.Lid
    and t1.Text = t2.Text
```

A slightly more complicated example is to insert context markers into the TEXT relation. We want to insert a special line 5 lines before every line containing the word "special": Here is the command:

```
    append before t2 (Text = "marker")
        where t1.Text = "* special *"
        and t2.Lid + 5 = t1.Lid
```

5. Index Performance

We expected the performance of A-trees, in terms of storage utilization and overhead, to be equivalent to that of B-trees. A simulation of the A-tree algorithms was coded in LISP, in order to better understand the operation of the algorithms and determine this performance. This initial hypothesis was born out. The experiment performed was to perform 10000 A-tree operations: first to insert 5000 random records, then to delete those records in random order. At each step, page utilization and page fault behavior were monitored. Typical results are shown in figure 4, which show the number of pages, tree height, utilization, and number of faults per operation over the course of the experiment.



5000 insertions followed
by 5000 deletions

Page size = 1024
Size of leaf record = 16
Size of inner record = 16
Size of buffer pool = 20 pages

Figure 4. A-tree Performance

Except when the tree is nearly empty, the page utilization of the A-tree remains fairly constant, in the vicinity of 69%. This agrees with the analytical result for B-trees derived by Yao [Yao78].

6. Supporting More Complex Ordering Abstractions

6.1. Using Secondary Indices for Multi-ordering

There might be a number of useful orderings among the records of a relation. In existing systems such as INGRES, these are supported by allowing the user to create secondary indices whose leaf records contain key information and pointers to primary data records. These leaf records can be resorted along a different ordering than the primary data records. Thus, each relevant ordering of the data has its own A-tree index. Of course, each A-tree index supports any number of ordered aggregates, all defined over over a single ordering of the data.

With slight modifications to INGRES indexing techniques, the A-tree facility can be used to support ordered aggregates over a single relation that are to be computed using different record orderings.

Using the **index** command, an index relation is created that contains a pointer (tid) to each record of the base relation, plus copies of the values of a particular set of attributes from the base relation. In the existing system, the specific set of attributes to be copied form the key of the index. The index relation is automatically sorted on that key.

In the proposed system, the specified set of attributes must contain all the parameters of ordered aggregates to be calculated using this index. This may or may not include keys. Additionally, the index is not sorted. The order of records in the index must match the order of records in the base relation. Modification of this ordering is then left under user control.

For example, consider a set of points in 2-space. The records have an x-coordinate and a y-coordinate. Suppose we wish to maintain count information over each of these, so as to be able to refer to "the n'th object along the x-axis" or "the m'th object along the y-axis". The base relation is called POINT, and the two secondary indices are XORDER and YORDER. The indices are modified to type A-tree, each containing **ocount** aggregates to provide the ordinate information required.

When the **modify** command is executed on a secondary index, the system creates the virtual attributes, and adds them to the *attribute* system catalog under both the primary and indexing relations. In this example, the POINT relation inherits two new virtual attributes, say, x_ordinate and y_ordinate.

The append command, when adding records, must now update these secondary indices also. However, the syntax of the **append** command must be further extended to manage this properly. An example should make this clear. Suppose we now wish to insert a point so that it follows the fifth point along the x axis, and follows the tenth point along the y axis. The command would be:

```
range of p1,p2 is POINT

append to POINT,
    after p1 in XORDER,
    after p2 in YORDER
    (PointID = "New Point ID")
    where p1.x_ordinate = 5
    and p2.y_ordinate = 10
```

One **before** or **after** clause is needed for each A-tree that exists, either primarily or secondarily, on the base relation. Suppose the user does not specify such a clause (or leaves out one of many). Then the system must arbitrarily place the object in the ordering. This is similar to the case where an insertion is made into a sorted relation where the inserted record was not assigned a key value. The default location in the ordering could be before the first record. This would be similar to the current treatment of null values in the INGRES system.

6.2. Hierarchical Ordering

The concept of hierarchical ordering involves orderings within a relation that are partitioned, much like aggregate functions are calculated over partitions of the records. For example, in a music database that stored notes and chords, a note has an ordinate position under a particular chord. Consider the NOTE relation in figure 5. We define the aggregate **hocount** to perform hierarchical ordering. The first NoteID under a given parent will have ordinate position 1, the

NOTE		
	NoteID	The name of the note.
	ChordID	The name of the chord to which the note belongs.

Figure 5. The NOTE Relation

next 2, and so on. There are as many notes with ordinate position 1 as there are chords. The ordered relation can be thus created:

```
define ordered aggregate hocount
    (parent = I4)
    returns I4
    file = "~bradr/agg/hocount.o"

modify NOTE to A-tree
    with ordinate = hocount(ChordID)
    on ChordID
```

This command creates the A-tree such as is shown in figure 6

In the **ocount** aggregate, the state for any internal record consisted of a count of the leaf records in the subtree rooted at that internal record. For the **hocount** aggregate, this state consists of two pieces of information: the chord ID of the last data record in the subtree, and its count value.

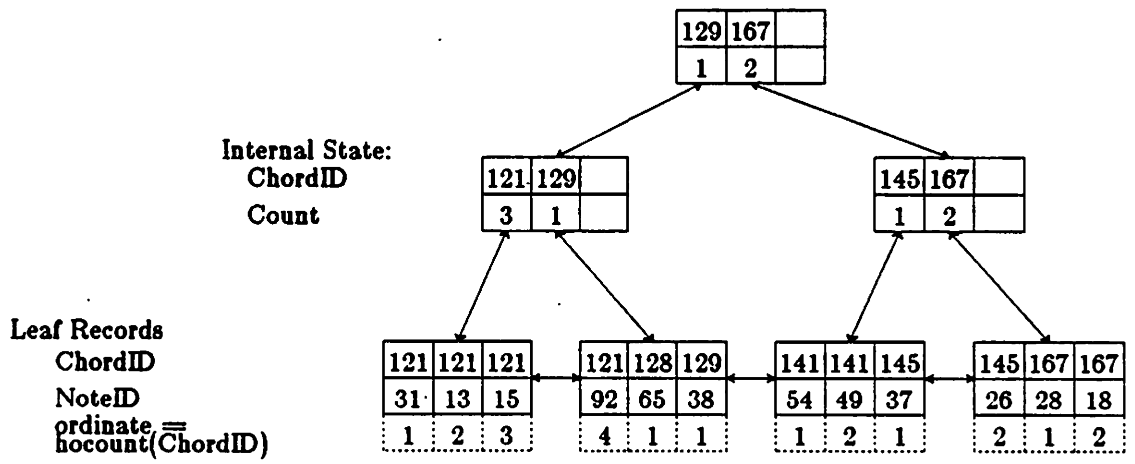


Figure 6. An A-tree for Hierarchical Ordering

There is a semantic problem with this approach. The **modify** command specifies a key, because the records must indeed be sorted (more precisely, partitioned) by ChordID. However, the user still wants to be able to use **append before** to place a given note, with ChordID=x, at a particular position within the list of notes under chord x. This must be allowed, without allowing the insert of this note after a record with a different chord value.

Our solution to this problem is to introduce a partitioning clause into the **modify** command. We call this the **by**-clause, after the notation used for partitioning aggregate functions in QUEL. The syntax of the above example would become:

```
modify NOTE to A-tree
    with ordinate = hocount(ChordID)
    by ChordID
```

The attributes in the **by**-clause become a primary sort key (any attributes in an **on**-clause would be used as a secondary sort key). When a query asks for a retrieval of an aggregate attribute of a particular value, such as,

```
/* retrieve the first note in every chord */
retrieve (NOTE.NoteID)
    where NOTE.ordinate = 1
```

every partition must be searched. Again, this problem arises because in hierarchically ordered data, the aggregate function values are not globally ordered on the relation. Rather, they are only locally ordered within each **by**-partition.

Under these circumstances, the A-tree retrieve logic becomes more complicated. The value of the aggregate attribute is no longer monotonically increasing over the course of the relation, rather it is increasing only within each partition. When a top-down traversal is performed, every partition must be searched. In those cases where a partition spans several disk pages, the state information of the internal nodes can be used by the system to accelerate this search.

7. Storing the Data as Graphs

When the data in a relation participates in more than one ordering, the issue is raised as to how the data itself should be ordered so that access via each ordering is efficiently performed.

The simplest approach to this problem, and the one taken by existing data base systems, is to select one of the orderings as primary, and store the records of the relation in that order. Any access of the records via this order will thereby incur a minimum amount of paging activity. Of course, sequential access via any secondary ordering will cause considerable paging activity, since records that are physically near each other in this ordering are not necessarily nearby each other in the primary ordering.

An alternative implementation would be to represent the orderings by actual pointers from a data record to each of its successors records (one successor per ordering). The A-tree is maintained over this graph, as before, except that the leaf level of the secondary index A-tree (which previously determined the ordering of the records) is now absorbed into the base relation, where the ordering is represented by pointers. Figure 7 compares these two approaches. The pointers from an object to each of its successors are edges in the graph.

There are two advantages to this new storage strategy. First, it is more space efficient, since the parameter information need no longer be copied into each secondary index. More importantly, the data can now be effectively clustered.

We wish to cluster the data so as to efficiently support access via the pointers. The data may be clustered with respect to all orderings, rather than only a single (primary) one. In those cases where no one ordering can be seen as primary, especially in those cases where the various orderings are correlated, graph clustering would be expected to provide a significant advantage.

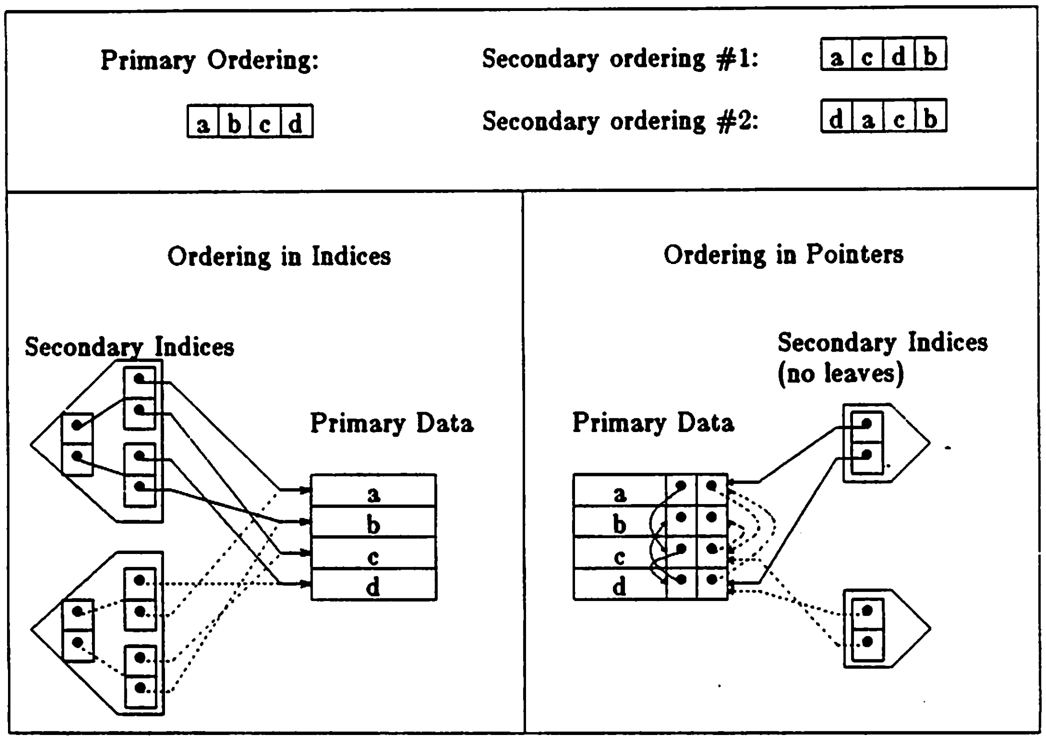


Figure 7. Secondary Indices vs. Successor Pointers

Their currently exists a large body of research on graph partitioning that may be brought to bear on how such data should be clustered. Although the optimization problem is NP-complete, several interesting heuristics have been developed [AJM84,KeL70,Mac78]. Whether they can be practically applied in the database environment is the subject of future research.

8. Additional Issues

There are several ways in which the current implementation could be made more general or more efficient. These ideas remain to be further explored.

8.1. User Access to Tuning Parameters

The affect of various tuning parameters, such as initial utilization of pages (i.e. the fill factor) remain to be explored.

For graph clustered implementations, a number of parameters are subject to further research, for example the nominal length of linked lists in a particular ordering under one parent (summary) record can be varied by the user to improve performance.

It remains to be seen what other tuning parameters might usefully be set by the user.

8.2. More Efficient Tree Traversal

Each internal record contains information that summarizes the contents of the data records in the leaves of its subtree. For top down search over ordered aggregates, every page on the path from root to leaf is scanned from left to right. For each internal record scanned, a call to a user routine is made to update the state.

In the case where retrievals are much more frequent than updates, it might be more efficient to precalculate the values of the state at each record of a page, and store this cumulative state value, rather than each individual summary record. Once this is calculated, a scan of the records can take place without any calls to user code. However, upon update to a record, the cumulative state at each of the subsequent records is invalidated.

It is not clear that this will provide a significant performance improvement in CPU usage.

8.3. Relative Access via Aggregate Value

In the case where a user requests access to a field via an aggregate value relative to another aggregate value a simple query optimization is possible. For example, the user, editing the end of the TEXT relation, executes a command such as:

```
append before t1 (Text = "Inserted text line")
    where t1.Lid = t2.Lid - 6
    and t2.tid = Record_ID
```

to insert a line of text six lines prior to a particular line whose tid is known.

A sophisticated query optimizer would notice that to get from Lid to Lid - 6 does not require the absolute value of the Lid attribute. The system could simply scan backwards 6 records. If the relative move were larger, then the system could take advantage of low levels of the tree to jump over larger portions of the relation. The point is that the system need not traverse all the way back up to the root, locking all those resources along the way, in order to satisfy this query.

The problem of concurrent access in A-trees is complicated by the fact that, unlike B-trees, updates to a relation, in general, have global impact on the value of aggregate attributes. Because every insertion into the ordered relation is guaranteed to cause a write to every page on the path from leaf to root, the root will typically be locked for every insertion. This means that an update will lock out all other concurrent access to the relation. Taking advantage of relative accesses prevents this relation-level lockout, increasing potential concurrency.

It remains to be seen how to allow a query engine to determine when these "relative access" shortcuts are possible, and what user-code is necessary to implement them.

8.4. Parent Pointers

The above discussion of A-trees assumes that every page of the data structure contains a pointer to its parent page (except, of course, the root page). This parent pointer is used in bottom-up traversal of the tree. Such a traversal would be required to perform a query such as:

```
retrieve (TEXT.Lid)
    where TEXT.Text = "A line of text"
```

Some search (possibly a linear scan) of the TEXT relation would find a record containing the appropriate line of text, and then a bottom-up traversal would determine that line's ordinate position, *Lid*.

In B-tree implementations where the leaf records are sorted on a key field, such parent pointers are unnecessary, since the path from leaf to root is easily determined by taking the key of a leaf record and performing a top-down traversal based on that key. The successive page addresses at each level of the tree can then be saved for a subsequent bottom-up traversal.

This approach does not work when there is no key field stored in the leaf records. For example, in the OB-trees developed by Lynn [Lyn82], the absence of parent pointers disallows the retrieval of the previous example. It is not possible to determine the Lid of a given line of text, unless that line was found by performing a top-down traversal of the index.

Although this problem is solved by maintaining parent pointers, the cost of this feature may be quite high. In particular, any time an internal page is split or merged, a large set of child pages must have there parent pointers updated. Each such update requires a page access.

This issue can be resolved in three ways:

- Do not maintain parent pointers, and disallow accesses that require them.
- Support parent pointers on child pages, and tolerate the excess cost of maintaining them.
- Support parent pointers, but rather than keeping them on child pages, store them in a separate data structure.

This last alternative proves to be a desirable choice. Our simulation showed a full tree of height three (1.4 Megabytes of data) to have a parent pointer relation of less than 16K bytes.

The cost of maintaining parent pointers can be effectively minimized by separating them from the actual tree structure, and maintaining them in main memory. As an aside, it should be noted that this internal structure need not be crash recoverable, since it can easily be reconstructed by traversing all the internal nodes of the A-tree.

9. Conclusions

A-trees provide a general approach for supporting user-defined indexing structures over abstract data types. They are modeled after B^+-trees, with the information stored in the internal nodes of the tree placed directly under user control. By supplying a small number of routines a user can easily simulate existing ordered access methods, such as B^+-trees and OB-trees.

Additionally, the A-tree supports aggregate values calculated over ordered relations. Of particular interest are thoses aggregates, known as ordered aggregates, which supply for each record of a relation an aggregate value based on that record and all previous records in the ordering. Examples of this approach are found in databases that maintain ordinate record numbers and those that maintain a per record running total over an attribute of the relation.

It has been demonstrated that the performance of these A-trees under dynamic insertions and deletions is the same as for B^+-trees.

By including a method of partitioning the data stored in an A-tree, hierarchical orderings are supported. By permitting A-trees to be used as secondary indices, multi-orderings are supported.

In the case where multi-orderings are present, the data may be stored as a graph. This results in a space savings, and potentially a time savings as well. In this case, the issue of how the data (i.e. the nodes of the graph) should be clustered needs to be addressed. It remains to be seen whether general purpose graph partitioning algorithms may be brought to bear to solve this problem.

References

[AJM84] Aragon, C., Johnson, D., MeGeoch, L. and Schevon, C., "Optimization By Simulated Annealing: An Experimental Evaluation", Technical Report Draft, September 1984.

[BaM72] Bayer, R. and McCreight, E., "Organization and maintenance of large ordered indexes", *Acta Informatica 1* (1972), 173-189, Springer-Verlag.

[CDR86] Carey, M., DeWitt, D., Richardson, J. and Shekita, E., "Object and File Management in the EXODUS Extensible Database System", *Proceedings of the International Conference on Very Large Data Bases*, Kyoto, August 1986.

[Fog82] Fogg, D., "Implementation of Domain Abstraction in the Relational Database System INGRES", Masters Report, Department of Electrical Engineering and Computer Science, University of California Berkeley, Berkeley, CA, November 1982.

[Han84] Hanson, E., "User-Defined Aggregates in the Relational Database System INGRES", Masters Report, Computer Science Division, University of California Berkeley, Berkeley, CA, December 1984.

[HSW75] Held, G., Stonebraker, M. and Wong, E., "INGRES - A Relational Database System", *Proceedings of the National Computer Conference*, Anaheim, CA, May 1975, 409-416.

[KeL70] Kernighan, B. and Lin, S., "An Efficient Heuristic Procedure for Partitioning Graphs", *Bell System Technical Journal 49*, 2 (February 1970), 291-307.

[Lyn82] Lynn, N., "Implementation of Ordered Relations in a Data Base System", Masters Report, Department of Electrical Engineering and Computer Science, University of California Berkeley, Berkeley, CA, September 1982.

[Mac78] MacGregor, R., *On Partitioning a Graph: A Theoretical and Empirical Study*, Ph.D. Dissertation, Computer Science Division, University of California Berkeley, Berkeley, CA, June 1978.

[Ong82] Ong, J., "The Design and Implementation of Abstract Data Types in the Relational Database System INGRES", Masters Report, Department of Electrical Engineering and Computer Science, University of California Berkeley, Berkeley, CA, September 1982.

[Rel84] Relational Technology Incorporated, *INGRES Reference Manual, Version 2.1*, Relational Technology Incorporated, Alameda, CA, July 1984.

[Rub85] Rubenstein, W. B., "Indices for Time-Ordered Data", Masters Thesis, Computer Science Division, University of California Berkeley, Berkeley, CA, May 1985.

[StR80] Stonebraker, M. and Rowe, L., "Database Portals: A New Application Program Interface", Electronic Research Laboratory Memorandum M82/80, University of California Berkeley, Berkeley, CA, November 1980.

[SSK82] Stonebraker, M., Stettner, H., Kalash, J., Guttman, A. and Lynn, N., "Document Processing in a Relational Data Base System", Electronic Research Laboratory Memorandum M82/32, University of California Berkeley, Berkeley, CA, May 1982.

[Yao78] Yao, A., "On Random 2-3 Trees", *Acta Informatica 9*, 2 (1978), 159-170, Springer-Verlag.