An Implementation for Nested Relational Databases

Anand Deshpande
Dirk Van Gucht

Computer Science Department
Indiana University
Bloomington, IN 47405, USA

deshpand@iuvax.cs.indiana.edu
vgucht@iuvax.cs.indiana.edu

Abstract

We propose an architecture for implementing nested relational databases. In particular, we discuss the storage structures, their organization and an access language for specifying access plans.

The features of our implementation are:

- A notation for hierarchical tuple identification.
- One value-driven indexing structure (VALTREE) for the entire database.
- A main-memory based component (CACHE) for manipulating hierarchical tuple-identifiers.
- A hashing scheme (RECLISTs) for fast access to data specified by tuple-identifiers.
- An access language based on the VALTREE, the RECLIST and the CACHE to define access plans for execution of queries.

1 Introduction

In this paper we propose an implementation, ANDA[1] for the Nested Relational Data Model (NRDM). In particular, we discuss the storage structures, their organization, and an access language for specifying access plans. The motivation for our design comes from these observations:

- In the NRDM, select, join and nest are 'value-driven' operations while project and unnest are not. To implement the value-driven operations it is crucial to efficiently determine which attribute and tuples are associated with a particular 'value'. In contrast, for 'structure-oriented' operations like project and unnest, it is required to efficiently access tuples and their components irrespective of the values contained in them. Data-structures that are well suited for project and unnest are unfortunately not always suitable for the value-driven operations. Hence our proposal for two storage structures where one supports value-driven requests effectively, while the other supports structure-oriented operations.
- Primary storage on computers has become fairly inexpensive while a disk access is still considerably more expensive than a memory access. We exploit this availability of main memory and indicate methods that use the cache to perform queries more efficiently.

We chose to implement the NRDM from scratch rather than map it to some other existing database implemen-

[1] ANDA – Acronymn for a Nested Database Architecture. In Hindi, ANDA means an egg, something you are likely to find in a nest.

tations for the following reasons:

- Tuple components are not necessarily atomic, making the mapping to the relational model difficult [22].
- Query optimizations that exploit the nested relational model cannot be used when the underlying storage structure is relational [3].
- Selections are often made on components deeply nested within tuples [12, 19, 20].
- Hierarchical and network models were not developed with high level non-procedural languages in mind [8].

In Section 2 we discuss the architecture of ANDA. In Section 3 we describe a notation for tuple identification and then discuss the storage components of ANDA – VALTREE, RECLIST and CACHE. In Section 4 we discuss the operations on these three components and define our access language. In Section 5 we demonstrate how some queries could be implemented in the access language. Finally, in Section 6 we discuss important observations about this implementation and discuss issues that need further investigation.

2 The ANDA Architecture

The major components of ANDA as shown in Figure 1 are:

- VALTREE – a tree structure storing all the atomic values present in the tuples and sub-tuples of the database,
- RECLIST – record-list structures which store data as tuples and sub-tuples,
- CACHE – the main memory component of the implementation where tuple-ids are manipulated,
- DATA-DICTIONARY – stores all the important information about structure definitions,
- ACCESS LANGUAGE INTERPRETER – interprets instructions described in the access language,
- QUERY LANGUAGE OPTIMIZER – this optimizes the query language expression into an access plan,
- GRAPHICAL QUERY LANGUAGE – this is the one of the user interfaces which maps to the optimizer,

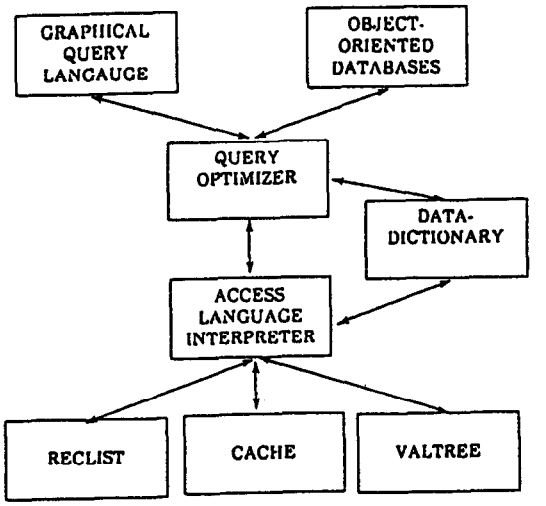

Figure 1: Components of ANDA

- OBJECT-ORIENTED INTERFACE – This uses the NRDM as the back-end.

The typical execution of a query would involve the following steps:

1. A query is specified to the system by the user in the OBJECT-ORIENTED INTERFACE or in the GRAPHICAL QUERY LANGUAGE.
2. The QUERY LANGUAGE OPTIMIZER will optimize this query into an optimized access plan. While research in the nested algebra optimization is still in its infancy, several results from relational algebra optimization [13, 14] can be extended to nested relations. [3, 18, 21]
3. The optimized query obtained from the previous step is mapped to an access plan in the access language of the system. As it is possible to come up with several alternative access plans the QUERY LANGUAGE OPTIMIZER uses information from the DATA DICTIONARY and knowledge of the data structures to come up with an optimal access plan. Heuristics for generating access plans for nested relational databases will have to be collected.
4. The access plans specified in the access language are interpreted by the ACCESS LANGUAGE INTERPRETER. The interpreter communicates with the VALTREE, the RECLIST, the DATA DICTIONARY, and the CACHE.

In the first phase of the implementation we have built a prototype of the ACCESS LANGUAGE INTERPRETER in Scheme with the VALTREE, the RECLIST, the DATA DICTIONARY and the CACHE in the main memory. This provided us with insights that were valuable in designing the access language.

In the second phase we are currently building these four components in C, this time with VALTREE and the RECLIST on the secondary storage.

3 Components of ANDA

This section defines a notation for tuple and component identification and describes the VALTREE, the RECLIST, and the CACHE.

3.1 A Notation for Tuple and Component Identification

In the nested relational model queries and updates can be performed on values that are deeply nested. To efficiently handle this request, it is important for tuple-identifiers at the sub-tuple level to be logically related to the tuple-identifiers of their super-tuples. Some of the components of the tuple could be sets, which in turn could have sets as their components therefore tuple identifiers cannot be flat but must be hierarchical. For the purpose of of identifying tuples and their components, we introduce the following notation. Let the database consist of a finite set of nested relational structures $\{r, s, t, ...\}$. The notation for the identification of tuples and their components uses these relation names tagged with subscripts and superscripts. The subscripts take us down the tuples and the superscripts take us across the components.

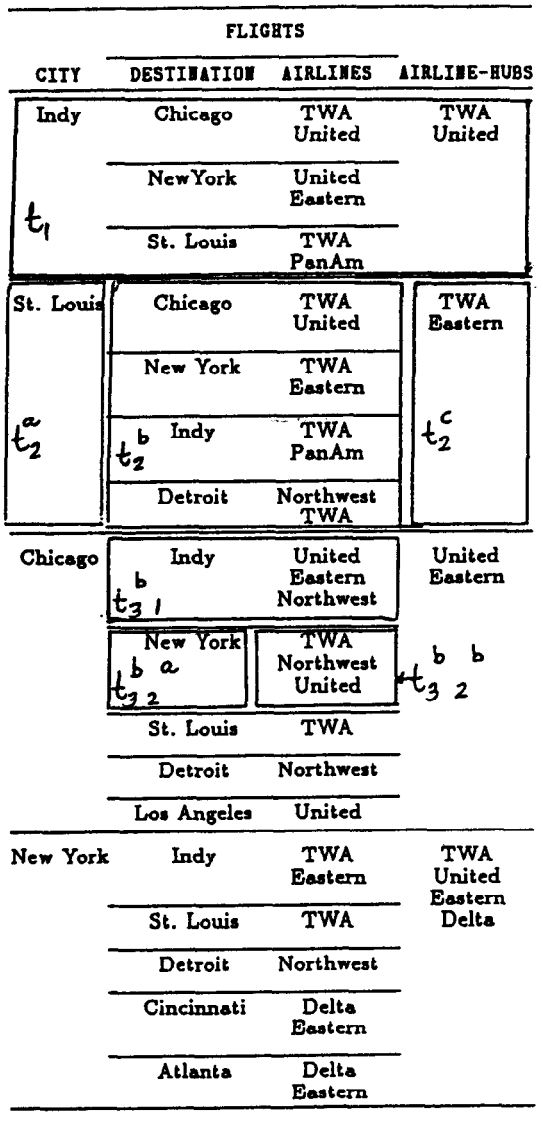

Figure 2: The AIRLINE-INFO structure

We introduce two nested structures, AIRLINE-INFO and SCHEDULE as shown in Figure 2 and Figure 3. The AIRLINE-INFO structure stores information about cities, flights departing the city and airlines for which the city is a hub, while the SCHEDULE structure stores information about universities their nearest airports and their away football games.

Thus, for the structure t corresponding to the AIRLINE-INFO structure of Figure 2 the tuples would be identified as t_1, t_2, t_3 and t_4 Each tuple is made up of three components: a CITY component, a FLIGHT component and AIRLINE-HUBS component. Thus, the first tuple t_1 has three components ${t_1}^a$, ${t_1}^b$ and ${t_1}^c$, where ${t_1}^a$ corresponds to the CITY component, ${t_1}^b$ corresponds to the FLIGHTS component and ${t_1}^c$ corresponds to the AIRLINE-HUBS component. Each of these components is either an atomic value or a structure. In our example ${t_1}^a$ is an atomic value, whereas ${t_1}^b$ and ${t_1}^c$ are structures. The structures ${t_1}^b$ and ${t_1}^c$ consists of sub-tuples, so we need to descend one level. The tuples of the structure ${t_1}^b$ are identified as ${t_1}^b{}_1$, ${t_1}^b{}_2$ and ${t_1}^b{}_3$. The identifiers for the components of the tuple ${t_1}^b{}_1$ are ${t_1}^b{}_1{}^a$ and ${t_1}^b{}_1{}^b$ corresponding to the DESTINATION and AIRLINES components.

In Figure 2, the notation is illustrated on the AIRLINE-INFO structure. An interesting feature of this notation is that once we get a tuple or component identifier, we can trace which tuples or sub-tuples the tuple

TEAM	NEAREST-AIRPORTS	TEAMS-TO-PLAY
Indiana	Indy Cincinnati Louisville	Purdue Michigan Wisconsin
Purdue	Indy Chicago	Minnesota Iowa Michigan
Northwestern	Chicago	Ohio State Iowa Minnesota
Michigan	Detroit	Micigan State Ohio State Wisconsin
Michigan State	Detroit	Indiana Purdue Iowa
Illinois	Chicago St. Louis	Indiana Ohio State Northwestern

Figure 3: The SCHEDULE structure

or component identifier belongs to by going through the superscript strings and the subscript strings.

3.2 The VALTREE Structure

Traditional relational database management systems use indexing techniques to improve access time. Typically, indexes are built on all or some of the attributes of a relation. A value of the index maps to a list of tuple-identifiers of tuples that contain the value of the indexed attribute. Our approach to indexing follows the domain based approach suggested by Missikoff [16] and Missikoff and Scholl [17] for relational databases. In their approach, an atomic value maps to a list of tuple identifiers of tuples in all relations in the database which contain that value. We generalize this approach by storing in the VALTREE, a mapping from a value to a list of all tuple identifiers of tuples in all structures and sub-structures in the nested relational database which contain that value. Hence, given an atomic value, the VALTREE returns a set of hierarchical tuple-identifiers, which enables us to determine directly which tuples or sub-tuples the value is stored in. Unlike the conventional database scheme where we have a separate tree for each indexed attribute, our scheme has only one tree, denoted VALTREE, that spans over all the atomic values of the database.

Valduriez, Khoshafian and Copeland [25, 24] have suggested processing 'Join Indices' in main memory to improve the performance of joins in relational systems; 'Inverted Files' have been used by the ADABAS database management system [1]. Our implementation of the VALTREE is a generalization of both these schemes. The CACHE allows us to perform some of the operations in main memory to improve performance.

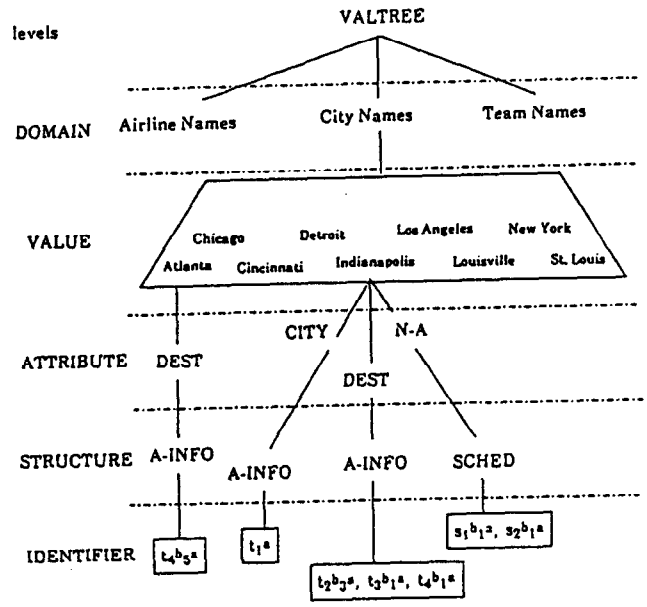

Figure 4: The VALTREE structure

The VALTREE is made up of five different levels as shown in Figure 4. The top-most level is called the DOMAIN level. This level separates the non-compatible domains into separate sub-trees. The second level, the VALUE level, stores all the atomic values of the database. The third level is the ATTRIBUTE level. At this level, we store all the attributes that a particular value of the VALUE level belongs to. As the same attribute may belong to more than one structure, we have the fourth level called the STRUCTURE level. Finally, the fifth and the lowest level consists of all the tuple-identifiers (tid) that correspond to the the atomic value stored at the VALUE level; this level is called the IDENTIFIER level. The advantage of using the VALTREE is that given a value it provides us rapid access to the list of tuple-identifiers corresponding to all occurrences of the value throughout the entire database. For further details the user is refered to [11]

3.3 The RECLIST Structure

As the VALTREE is a suitable data structure for performing value-driven operations. RECLIST structures have the following requirements:

- Each structure in the nested relational database has a separate RECLIST structure.
- Given a tuple-identifier and an attribute, the number of disk accesses to access the component of the tuple associated with the tuple-identifier and attribute should be minimal.
- The number of disk accesses to retrieve an entire tuple should be minimal.
- It should be easy to do structure-oriented operations.
- It should be possible to traverse through the entire structure a tuple at a time.

There are several storage structures that can be used to achieve these goals. Tsichritzis and Lochovsky [23] and Wiederhold [26] have a detailed description of some of these structures. Carey, DeWitt et.al. [6] have also suggested a storage structure for the Exodus project. The RECLIST in ANDA is inspired by some of the storage structures proposed by Dadam et.al. and Deppisch et.al. [7, 10].

The objective of the RECLIST is to provide a mapping from the hierarchical tuple-identifier to the actual physical address. ANDA uses Linear Hashing Scheme [15] for the RECLIST. Tuples are stored into buckets obtained by hashing tuple-ids. Tuples from a structure may be divided into fragments depending on the granularity of the desired operations. It is convenient, though not necessary to have uniform bucket sizes. Breaking the tuple into smaller fragments allows the user to get to a value more directly and inserts do not have to account for overflows. Smaller fragments, however require multiple disk accesses to reconstruct the entire tuple. A decision on the granularity of the tuples is made by taking these trade-offs into account.

To insert a new tuple in the RECLIST, one has to generate a new hierarchical tuple-id before the tuple can be mapped into an appropriate bucket. To keep track of the tids used and the next one available, a bitmap with bits corresponding to existing sub-tuples is associated with each sub-tuple and is stored along with the sub-tuple. Deletions are performed by toggling the bitmap. The bitmap allows a space efficient method allocating tuple-ids and reclaiming them during deletion. The reader is referred to Deshpande and Van Gucht [11] for details of the hashing scheme used for ANDA.

3.4 The Cache

The CACHE is the primary storage component of the implementation. To improve performance of nested algebra expressions it is important to reduce accesses to the secondary storage components – VALTREE and RECLIST, and perform as many operations as possible in the cache.

We have the following goals for the CACHE:

- The storage utilization of the CACHE should be as efficient as possible.
- Since the total amount of available primary storage space is limited the cache should only store as much information as required.
- The organization of the cache should be simple yet should allow the flexibility to handle complex operations.
- Complex reorganization and garbage collection should be kept to a minimum.

The tuple-ids described in the previous subsection carry information regarding the exact location of the value in a tuple. Given two tuple-ids it is possible to determine if they belong to the same tuple or the same sub-tuple. For this reason, we store only tuple-ids obtained from the VALTREE in the CACHE and never store actual values or tuples. After manipulating these tuple-ids in the CACHE the results corresponding to the tuple-ids are extracted from the RECLIST.

In ANDA the CACHE consists of a set of stacks. We choose stacks to be the cache as stacks provide a simple implementation with minimal pointer overhead, and no garbage collection.

Several interesting queries can be processed by comparing tuple-ids as will be shown in the next section. Ideally, our queries are processed in the following three steps:

- Retrieve tuple-ids from the VALTREE and place them in the cache,
- Process tuple-ids in the cache, and

- Retrieve required parts of the structure corresponding to the tuple-ids in the CACHE is retrieved from the RECLIST.

Details of some of the important CACHE operations are described in the next section.

4 The Access Language

In this section we discuss some important operations on the storage structures. The access language used for specifying the access plans consists of instructions for operations on the storage structures and condition and iteration statements.

4.1 VALTREE functions

The Access Language has functions to insert, delete and retrieve from the VALTREE. To maintain consistency the insert and delete functions of the VALTREE are not used directly by the user but are used by the insert and delete functions of the RECLIST. Conditions can be specified with vt-retrieve to allow complex selections. The vt-insert and vt-delete functions ensure that there are no duplicates adn no empty sub-trees.

```
vt-retrieve (domain, value-cond,
             attribute-cond, structure-cond,
             granularity, stack)

  vt-insert (domain, value, attribute,
             structure, tid)

  vt-delete (domain, value, attribute,
             structure, tid)
```

The arguments to these procedure are:

- domain : corresponds to the domain at the DOMAIN level to be selected.
- value-cond : If the value at the VALUE level satisfies the value-cond then this value is selected and the subtree corresponding to the ATTRIBUTE, the STRUCTURE and the TUPLE levels is traversed otherwise the subtree corresponding to this value is ignored.
- attribute-cond : If the attribute at the ATTRIBUTE level for a selected value at the VALUE level satisfies the attribute-cond then this attribute is selected and the subtree corresponding to the STRUCTURE and TUPLE level is traversed otherwise the subtree corresponding to this attribute is ignored.
- structure-cond : If the structure-name at the STRUCTURE level satisfies the structure-cond then the tuple-ids corresponding to the selected (domain, value, attribute, structure) are placed on the stack otherwise the tuple-ids corresponding to the structure are ignored.
- stack : This is the name of the stack in the cache that is used to store the result – the set of tuple-ids, obtained from this retrieve.
- granularity : This can be one of domain, value, attribute, structure or tuple. This specifies the granularity of the elements placed on the stack. If structure is used as the granularity for the retrieve then each set of selected tuple-ids obtained at the STRUCTURE level is placed on the stack as a separate element. If value is used as the granularity then all the sets of tids obtained for different attributes and structures for this particular value are unioned together and stored as one element on the stack.

4.2 RECLIST functions

We use the hashing scheme with bitmaps stored along with data values as discussed in Section 3.3 for our implementation of the RECLIST. As we have two data structures – the VALTREE and the RECLIST, it is important to keep the information in both the data structures consistent at all times. To ensure consistency all inserts and deletes are performed on the RECLIST. These procedures in turn call the vt-insert and vt-delete procedures. To ensure that inserts and deletes are done at the correct place in the RECLIST, the RECLIST procedures call the vt-retrieve procedure.

4.2.1 Reclist Retrieve

This procedure returns sub-tuples from appropriate RECLIST that correspond to the tuple-ids in the top element of the stack

```
rl-retrieve (stack)
```

4.2.2 Reclist Insert/ Delete

This inserts/deletes the template in the appropriate structure ensuring that the hierarchical database prop-

erties are maintained. The template is constructed after consulting the DATA-DICTIONARY. This procedure also generates a list of commands that correspond to vt-inserts/ vt-deletes in the VALTREE. While it is possible to have null values in the template, these values are not allowed for key values at any level.

```
rl-insert (structure, template)
rl-delete (structure, template)
```

4.3 CACHE functions

The cache is organized as a collection of stacks in the main memory.

4.3.1 Standard Stack Operations

```
create-stack
destroy-stack
push (element, stack)
pop (stack)
empty? (stack)
full? (stack)
```

4.3.2 Set Operations

These are binary operations and are performed on the top two elements (sets of tids) of the stack and places the result on top of the stack. The tid-window-format specifies parts of the tuple-id that are considered when performing the operation.

```
union (stack, tid-window-format)
intersection (stack, tid-window-format)
difference (stack, tid-window-format)
product (stack)
```

- tid-window-format : specifies the window for these operators. Wild-cards like '*' are used to indicate 'don't-care' values. When performing a union of tuple-ids with different formats but matching tid-window-format the tuple-ids with more information (subscripts/superscripts) is retained. Thus, if we specify the tid-window-format to be $t^a{}_*$, then when forming the union, tuple-ids with common first subscript and 'a' as the first superscript merge in the union. Therefore the union of $t^a{}_2{}^b{}_1$ and $t^a{}_2{}^b{}_2$ is $t^a{}_2$ but the union of $t^a{}_2$ and $t^a{}_2{}^b{}_2$ is $t^a{}_2{}^b{}_2$. In other words the tid-window-format tells you which parts of the tids should be used for the union. This operation performs a union and project in one step. The utility of this operation will be clear after observing the examples in the next section.

4.3.3 Filter, transform and copy instructions

This group of stack commands allow us to perform some miscellaneous functions to make the specification of the access plans easier.

```
filter (stack, filter-format)
```

Given a filter-format this procedure removes tids that do not satisfy the format from the top element of the stack.

```
transform (stack transformed-format)
```

This procedure transforms the tids from the top of the stack and transforms them to correspond to the format specified by the transformed-format. If the structure is defined as a structure tree then then it is possible to transform the tuple into its siblings, its ancestors, and siblings of its ancestors. This procedure is used for projections.

```
copy (stack1, stack2)
```

This instruction copies the top element of stack1 and copies it to the top of stack2. If both stack1 and stack2 are the same, then this stack would have two identical elements as the top two elements of the stack.

```
sort (stack)
```

This procedure sorts all the tuple-ids in the top element of the stack.

```
one-of (stack)
```

This procedure picks one tuple-id from the top element of the stack and discards the rest.

Several procedures like for-each (element of the stack) do, repeat until, if - then - else and functional combinators etc. may be required.

5 Access plans for some NRDM operations

The following examples apply to the AIRLINE-INFO and SCHEDULE structures.

Example 1 *Select* CITY, DESTINATION *pairs having 'Eastern'* FLIGHTS *leaving from 'TWA'* AIRLINE-HUBS.

In this example, different parts of the query are not at the same level. We have to project the CITY and the appropriate DESTINATION pairs where 'Eastern' is one of the AIRLINES and 'TWA' is one of the AIRLINE-HUBS.

The access plan for this query is:

1. vt-retrieve(Airline-Name, Eastern, AIRLINES, AIRLINE-INFO, structure, S1)
2. transform(S1, $t_*{}^b{}_*{}^a$)
3. vt-retrieve(Airline-Name, TWA, AIRLINE-HUBS, AIRLINE-INFO,structure, S1)
4. transform(S1,$t_*{}^b$)
5. intersection(S1, $t_*{}^b$)
6. copy(S1,S1)
7. transform(S1, $t_*{}^a$)
8. union(S1, $t_*{}^*{}_*{}^*$)
9. sort(S1)
10. rl-retrieve (S1)

The corresponding state of the CACHE:

1. S1 $= \langle\{t_1{}^b{}_2{}^b{}_2{}^a, t_2{}^b{}_2{}^b{}_2{}^a, t_3{}^b{}_1{}^b{}_2{}^a, t_4{}^b{}_1{}^b{}_2{}^a, t_4{}^b{}_4{}^b{}_2{}^a, t_4{}^b{}_5{}^b{}_2{}^a\}\rangle$
2. S1 $= \langle\{t_1{}^b{}_2{}^a, t_2{}^b{}_2{}^a, t_3{}^b{}_1{}^a, t_4{}^b{}_1{}^a, t_4{}^b{}_4{}^a, t_4{}^b{}_5{}^a\}\rangle$
3. S1 $= \langle\{t_1{}^c{}_1{}^a, t_2{}^c{}_1{}^a, t_4{}^c{}_1{}^a\}, \{t_1{}^b{}_2{}^a, t_2{}^b{}_2{}^a, t_3{}^b{}_1{}^a, t_4{}^b{}_1{}^a, t_4{}^b{}_4{}^a, t_4{}^b{}_5{}^a\}\rangle$
4. S1 $= \langle\{t_1{}^b, t_2{}^b, t_4{}^b\}, \{t_1{}^b{}_2{}^a, t_2{}^b{}_2{}^a, t_3{}^b{}_1{}^a, t_4{}^b{}_1{}^a, t_4{}^b{}_4{}^a, t_4{}^b{}_5{}^a\}\rangle$
5. S1 $= \langle\{t_1{}^b{}_2{}^a, t_2{}^b{}_2{}^a, t_4{}^b{}_1{}^a, t_4{}^b{}_4{}^a, t_4{}^b{}_5{}^a\}\rangle$
6. S1 $= \langle\{t_1{}^b{}_2{}^a, t_2{}^b{}_2{}^a, t_4{}^b{}_1{}^a, t_4{}^b{}_4{}^a, t_4{}^b{}_5{}^a\}, \{t_1{}^b{}_2{}^a, t_2{}^b{}_2{}^a, t_4{}^b{}_1{}^a, t_4{}^b{}_4{}^a, t_4{}^b{}_5{}^a\}\rangle$
7. S1 $= \langle\{t_1{}^a, t_2{}^a, t_4{}^a\}, \{t_1{}^b{}_2{}^a, t_2{}^b{}_2{}^a, t_4{}^b{}_1{}^a, t_4{}^b{}_4{}^a, t_4{}^b{}_5{}^a\}\rangle$
8. S1 $= \langle\{t_1{}^a, t_2{}^a, t_4{}^a, t_1{}^b{}_2{}^a, t_2{}^b{}_2{}^a, t_4{}^b{}_1{}^a, t_4{}^b{}_4{}^a, t_4{}^b{}_5{}^a\}\rangle$
9. S1 $= \langle\{t_1{}^a, t_1{}^b{}_2{}^a, t_2{}^a, t_2{}^b{}_2{}^a, t_4{}^a, t_4{}^b{}_1{}^a, t_4{}^b{}_4{}^a, t_4{}^b{}_5{}^a\}\rangle$
10. $[t_1{}^a = Indianapolis, t_1{}^b{}_2{}^a = New\ York, t_2{}^a = St.\ Louis, t_2{}^b{}_2{}^a = New\ York, t_4{}^a = New\ York, t_4{}^b{}_1{}^a = Indianapolis, t_4{}^b{}_4{}^a = Cincinnati, t_4{}^b{}_5{}^a = Atlanta]$

In this example the two vt-retrieves extract the required tuple-ids. The transform functions bring the tuple-ids to a form appropriate for intersection. Steps 6–10 are required to output the right parts of the tuple. The copy function is required when projecting more than one attribute because the transform function is destructive.

Example 2 *For each airport in the* SCHEDULE *structure, list all teams close to it.*

This example involves a sequence of unnest and nest operations. The access plan for this query is as follows:

1. vt-retrieve(City-Name, *, NEAREST-AIRPORTS, SCHEDULE, structure, S1)
2. repeat
   (a) copy(S1,S1)
   (b) one-of(S1)
   (c) rl-retrieve(S1)
   (d) transform(S1,$s_*{}^a$)
   (e) rl-retrieve(S1)
   until empty?(S1)

The corresponding state of the CACHE:

1. S1 $= \langle\{s_2{}^b{}_2{}^a, s_3{}^b{}_1{}^a, s_6{}^b{}_1{}^a\}, \{s_1{}^b{}_2{}^a\}, \{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\}, \{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{s_1{}^b{}_3{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$
2. The state of the stack for the first iteration; these steps are repeated until the stack is empty
   (a) S1 $= \langle\{s_2{}^b{}_2{}^a, s_3{}^b{}_1{}^a, s_6{}^b{}_1{}^a\}, \{s_2{}^b{}_2{}^a, s_3{}^b{}_1{}^a, s_6{}^b{}_1{}^a\}, \{s_1{}^b{}_2{}^a\}, \{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\}, \{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{s_1{}^b{}_3{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$
   (b) S1 $= \langle\{s_2{}^b{}_2{}^a\}, \{s_2{}^b{}_2{}^a, s_3{}^b{}_1{}^a, s_6{}^b{}_1{}^a\}, \{s_1{}^b{}_2{}^a\}, \{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\}, \{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{s_1{}^b{}_3{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$
   (c) $[s_2{}^b{}_2{}^a = Chicago]$ S1 $= \langle\{s_2{}^b{}_2{}^a, s_3{}^b{}_1{}^a, s_6{}^b{}_1{}^a\}, \{s_1{}^b{}_2{}^a\}, \{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\}, \{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{s_1{}^b{}_3{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$
   (d) S1 $= \langle\{s_2{}^a, s_3{}^a, s_6{}^a\}, \{s_1{}^b{}_2{}^a\}, \{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\}, \{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{s_1{}^b{}_3{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$
   (e) $[s_2{}^a = Purdue, \quad s_3{}^a = Nortwestern, s_6{}^a = Illinois]$ S1 $= \langle\{s_1{}^b{}_2{}^a\}, \{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\}, \{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{s_1{}^b{}_3{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$

In this example the first vt-retrieve operation retrieves sets of tuple-ids. Each set corresponds to an airport and each element of the set corresponds to the teams that have that city as the NEAREST-AIRPORT. As we are interested in both the NEAREST-AIRPORT and TEAMS we need the copy operation.

Example 3 *For each city from the* SCHEDULE *structure find the teams close to it and the Airlines that can be used to reach that city.*

This query involves a join and then a restructuring of the structures. The VALTREE stores all ATTRIBUTES that corresponding to a value at the attribute level. Therefore, the join is essentially performed by traversing the VALTREE and extracting tuple-ids for values that have both the required attributes at the attribute level. The access plan for this query is as follows:

1. vt-retrieve (City-Name, *, ({ DESTINATION, NEAREST-AIRPORT} $\subseteq$ ATTRIBUTES), *, structure, S1)
2. repeat
   (a) copy(S1,S1)
   (b) one-of (S1)
   (c) rl-retrieve(S1)
   (d) transform(S1, $t_*{}^b{}_*{}^b$)
   (e) rl-retrieve(S1)
   (f) transform(S1, s_*^a)
   (g) rl-retrieve(S1)
   until empty?(stack)

The state of the CACHE:

1. S1 = $\langle\{t_1{}^{b}{}_1{}^{a}, t_2{}^{b}{}_1{}^{a}\}, \{s_2{}^{b}{}_2{}^{a}, s_3{}^{b}{}_1{}^{a}, s_6{}^{b}{}_1{}^{a}\},$
$\{t_4{}^{b}{}_4{}^{a}\}, \{s_1{}^{b}{}_2{}^{a}\},$
$\{t_2{}^{b}{}_4{}^{a}, t_3{}^{b}{}_4{}^{a}\}, \{s_4{}^{b}{}_1{}^{a}, s_5{}^{b}{}_1{}^{a}\},$
$\{t_2{}^{b}{}_3{}^{a}, t_3{}^{b}{}_1{}^{a}, t_4{}^{b}{}_1{}^{a}\}, \{s_1{}^{b}{}_1{}^{a}, s_2{}^{b}{}_1{}^{a}\},$
$\{t_1{}^{b}{}_3{}^{a}, t_3{}^{b}{}_3{}^{a}, t_4{}^{b}{}_2{}^{a}\}, \{s_6{}^{b}{}_2{}^{a}\}\rangle$

2. **The state of the stack for the first iteration; these steps are repeated until the stack is empty.**

(a) S1 = $\langle\{t_1{}^{b}{}_1{}^{a}, t_2{}^{b}{}_1{}^{a}\}, \{t_1{}^{b}{}_1{}^{a}, t_2{}^{b}{}_1{}^{a}\},$
$\{s_2{}^{b}{}_2{}^{a}, s_3{}^{b}{}_1{}^{a}, s_6{}^{b}{}_1{}^{a}\}, \{t_4{}^{b}{}_4{}^{a}\},$
$\{s_1{}^{b}{}_2{}^{a}\}, \{t_2{}^{b}{}_4{}^{a}, t_3{}^{b}{}_4{}^{a}\},$
$\{s_4{}^{b}{}_1{}^{a}, s_5{}^{b}{}_1{}^{a}\}, \{t_2{}^{b}{}_3{}^{a}, t_3{}^{b}{}_1{}^{a}, t_4{}^{b}{}_1{}^{a}\},$
$\{s_1{}^{b}{}_1{}^{a}, s_2{}^{b}{}_1{}^{a}\}, \{t_1{}^{b}{}_3{}^{a}, t_3{}^{b}{}_3{}^{a}, t_4{}^{b}{}_2{}^{a}\}, \{s_6{}^{b}{}_2{}^{a}\}\rangle$

(b) S1 = $\langle\{t_1{}^{b}{}_1{}^{a}\}, \{t_1{}^{b}{}_1{}^{a}, t_2{}^{b}{}_1{}^{a}\}, \{s_2{}^{b}{}_2{}^{a}, s_3{}^{b}{}_1{}^{a}, s_6{}^{b}{}_1{}^{a}\},$
$\{t_4{}^{b}{}_4{}^{a}\}, \{s_1{}^{b}{}_2{}^{a}\},$
$\{t_2{}^{b}{}_4{}^{a}, t_3{}^{b}{}_4{}^{a}\}, \{s_4{}^{b}{}_1{}^{a}, s_5{}^{b}{}_1{}^{a}\},$
$\{t_2{}^{b}{}_3{}^{a}, t_3{}^{b}{}_1{}^{a}, t_4{}^{b}{}_1{}^{a}\}, \{s_1{}^{b}{}_1{}^{a}, s_2{}^{b}{}_1{}^{a}\},$
$\{t_1{}^{b}{}_3{}^{a}, t_3{}^{b}{}_3{}^{a}, t_4{}^{b}{}_2{}^{a}\}, \{s_6{}^{b}{}_2{}^{a}\}\rangle$

(c) $[t_1{}^{b}{}_1{}^{a} = Chicago]$
S1 = $\langle\{t_1{}^{b}{}_1{}^{a}, t_2{}^{b}{}_1{}^{a}\}, \{s_2{}^{b}{}_2{}^{a}, s_3{}^{b}{}_1{}^{a}, s_6{}^{b}{}_1{}^{a}\},$
$\{t_4{}^{b}{}_4{}^{a}\}, \{s_1{}^{b}{}_2{}^{a}\},$
$\{t_2{}^{b}{}_4{}^{a}, t_3{}^{b}{}_4{}^{a}\}, \{s_4{}^{b}{}_1{}^{a}, s_5{}^{b}{}_1{}^{a}\},$
$\{t_2{}^{b}{}_3{}^{a}, t_3{}^{b}{}_1{}^{a}, t_4{}^{b}{}_1{}^{a}\}, \{s_1{}^{b}{}_1{}^{a}, s_2{}^{b}{}_1{}^{a}\},$
$\{t_1{}^{b}{}_3{}^{a}, t_3{}^{b}{}_3{}^{a}, t_4{}^{b}{}_2{}^{a}\}, \{s_6{}^{b}{}_2{}^{a}\}\rangle$

(d) S1 = $\langle\{t_1{}^{b}{}_1{}^{b}, t_2{}^{b}{}_1{}^{b}\}, \{s_2{}^{b}{}_2{}^{a}, s_3{}^{b}{}_1{}^{a}, s_6{}^{b}{}_1{}^{a}\},$
$\{t_4{}^{b}{}_4{}^{a}\}, \{s_1{}^{b}{}_2{}^{a}\},$
$\{t_2{}^{b}{}_4{}^{a}, t_3{}^{b}{}_4{}^{a}\}, \{s_4{}^{b}{}_1{}^{a}, s_5{}^{b}{}_1{}^{a}\},$
$\{t_2{}^{b}{}_3{}^{a}, t_3{}^{b}{}_1{}^{a}, t_4{}^{b}{}_1{}^{a}\}, \{s_1{}^{b}{}_1{}^{a}, s_2{}^{b}{}_1{}^{a}\},$
$\{t_1{}^{b}{}_3{}^{a}, t_3{}^{b}{}_3{}^{a}, t_4{}^{b}{}_2{}^{a}\}, \{s_6{}^{b}{}_2{}^{a}\}\rangle$

(e) $[t_1{}^{b}{}_1{}^{b} = \{TWA, United\},$
$t_2{}^{b}{}_1{}^{b} = \{TWA, United\}]$
S1 = $\langle\{s_2{}^{b}{}_2{}^{a}, s_3{}^{b}{}_1{}^{a}, s_6{}^{b}{}_1{}^{a}\}, \{t_4{}^{b}{}_4{}^{a}\},$
$\{s_1{}^{b}{}_2{}^{a}\}, \{t_2{}^{b}{}_4{}^{a}, t_3{}^{b}{}_4{}^{a}\}, \{s_4{}^{b}{}_1{}^{a}, s_5{}^{b}{}_1{}^{a}\},$
$\{t_2{}^{b}{}_3{}^{a}, t_3{}^{b}{}_1{}^{a}, t_4{}^{b}{}_1{}^{a}\}, \{s_1{}^{b}{}_1{}^{a}, s_2{}^{b}{}_1{}^{a}\},$
$\{t_1{}^{b}{}_3{}^{a}, t_3{}^{b}{}_3{}^{a}, t_4{}^{b}{}_2{}^{a}\}, \{s_6{}^{b}{}_2{}^{a}\}\rangle$

(f) S1 = $\langle\{s_2{}^{a}, s_3{}^{a}, s_6{}^{a}\}, \{t_4{}^{b}{}_4{}^{a}\}, \{s_1{}^{b}{}_2{}^{a}\},$
$\{t_2{}^{b}{}_4{}^{a}, t_3{}^{b}{}_4{}^{a}\}, \{s_4{}^{b}{}_1{}^{a}, s_5{}^{b}{}_1{}^{a}\}, \{t_2{}^{b}{}_3{}^{a}, t_3{}^{b}{}_1{}^{a},$
$t_4{}^{b}{}_1{}^{a}\}, \{s_1{}^{b}{}_1{}^{a}, s_2{}^{b}{}_1{}^{a}\}, \{t_1{}^{b}{}_3{}^{a}, t_3{}^{b}{}_3{}^{a}, t_4{}^{b}{}_2{}^{a}\},$
$\{s_6{}^{b}{}_2{}^{a}\}\rangle$

(g) $[s_2{}^{a} = Purdue, s_3{}^{a} = Northwestern, s_6{}^{a} = Illinois]$
S1 = $\langle\{t_4{}^{b}{}_4{}^{a}\}, \{s_1{}^{b}{}_2{}^{a}\}, \{t_2{}^{b}{}_4{}^{a}, t_3{}^{b}{}_4{}^{a}\},$
$\{s_4{}^{b}{}_1{}^{a}, s_5{}^{b}{}_1{}^{a}\}, \{t_2{}^{b}{}_3{}^{a}, t_3{}^{b}{}_1{}^{a}, t_4{}^{b}{}_1{}^{a}\},$
$\{s_1{}^{b}{}_1{}^{a}, s_2{}^{b}{}_1{}^{a}\}, \{t_1{}^{b}{}_3{}^{a}, t_3{}^{b}{}_3{}^{a}, t_4{}^{b}{}_2{}^{a}\},$
$\{s_6{}^{b}{}_2{}^{a}\}\rangle$

In this example the first `vt-retrieve` groups the tuple-ids into pairs of sets that participate in the join. The next steps in the loop rearrange the structure to the desired form. It is interesting to observe that the join operation is $O(n_1 + n_2)$ where n_1 is the number of elements at the VALUE level for the `City-Name` domain and n_2 is the number of elements that participate as pivot elements in the join.

Here are some observations about the access language:

1. The results obtained from the `rl-retrieve` procedure could be displayed on the screen in the appropriate format. Details of the display routines for outputing the results in the nested relational form have not been included in this paper. If the results obtained from the `rl-retrieve` procedure are temporary or need to saved in new structures then they could be inserted into new RECLISTS.

2. Conversion from the query language to an efficient access plan is hardly a trivial task. Efforts are being made to determine if algebra is the right intermediate step for optimization.

6 Discussion

In this section, we discuss how our storage structure is suitable to effectively handle some other important DBMS issues.

6.1 The VALTREE as a Nested Relational Structure

The VALTREE itself can be thought of as a nested relation as shown in Figure 7. This allows one to perform nested relational algebraic operations on the VALTREE. This allows for example to consider other indexing schemes like the standard (attribute, value) pairs by simply restructuring the VALTREE using the nested relational algebra.

If fast implementations for the RECLIST structure become available, the VALTREE can be implemented as a RECLIST and all the VALTREE operations can be performed by performing algebraic operations on VALTREE stored as a RECLIST.

6.2 Object-Oriented Databases

Object-Oriented Databases are becoming increasingly popular. Our research would be beneficial to the implementation of object-oriented databases in the following two ways:

1. Several current implementations of object-oriented databases [2] map the object oriented systems to relational databases. While this is possible, designers of such systems have problems mapping complex objects to flat relations. We feel that the mapping from object-oriented databases to nested

DOMAIN	VALUE	ATTRIBUTE	STRUCTURE	{IDENTIFIER}
City Name	Atlanta	Destination	Airline-Info	$\{t_4{}^{b}{}_{5}{}^{a}\}$
	Chicago	City	Airline-Info	$\{t_3{}^{a}\}$
		Destination	Airline-Info	$\{t_1{}^{b}{}_{1}{}^{a}, t_2{}^{b}{}_{1}{}^{a}\}$
		Nearest-Airports	Schedule	$\{s_2{}^{b}{}_{2}{}^{a}, s_3{}^{b}{}_{1}{}^{a}, s_6{}^{b}{}_{1}{}^{a}\}$
	Cincinnati	Destination	Airline-Info	$\{t_4{}^{b}{}_{4}{}^{a}\}$
		Nearest-Airports	Schedule	$\{s_1{}^{b}{}_{2}{}^{a}\}$
	Detroit	Destination	Airline-Info	$\{t_2{}^{b}{}_{4}{}^{a}, t_3{}^{b}{}_{4}{}^{a}\}$
		Nearest-Airports	Schedule	$\{s_4{}^{b}{}_{1}{}^{a}, s_5{}^{b}{}_{1}{}^{a}\}$
	Indy	City	Airline-Info	$\{t_1{}^{a}\}$
		Destination	Airline-Info	$\{t_2{}^{b}{}_{3}{}^{a}, t_3{}^{b}{}_{1}{}^{a}, t_4{}^{b}{}_{1}{}^{a}\}$
		Nearest-Airports	Schedule	$\{s_1{}^{b}{}_{1}{}^{a}, s_2{}^{b}{}_{1}{}^{a}\}$
	Los Angeles	Destination	Airline-Info	$\{t_3{}^{b}{}_{5}{}^{a}\}$
	Louisville	Nearest-Airports	Schedule	$\{s_1{}^{b}{}_{3}{}^{a}\}$
	New York	City	Airline-Info	$\{t_4{}^{a}\}$
		Destination	Airline-Info	$\{t_1{}^{b}{}_{2}{}^{a}, t_2{}^{b}{}_{2}{}^{a}, t_3{}^{b}{}_{2}{}^{a}\}$
	St. Louis	City	Airline-Info	$\{t_2{}^{a}\}$
		Destination	Airline-Info	$\{t_1{}^{b}{}_{3}{}^{a}, t_3{}^{b}{}_{3}{}^{a}, t_4{}^{b}{}_{2}{}^{a}\}$
		Nearest-Airports	Schedule	$\{s_6{}^{b}{}_{2}{}^{a}\}$
Airline Name	...	...	...	{...}
Team Name	...	...	...	{...}

Figure 5: The VALTREE as a nested relational structure

relational databases, though not entirely trivial, is much cleaner than the mapping to a relational model [4]. This is because the nested relational paradigm models sets which are fundamental to object-oriented systems.

2. The problems faced by designers building 'pure' object-oriented systems [9] are very similar to the problems that are faced in the representation of the nested relational model. Data-structures like the VALTREE and the RECLIST with some modifications could be used for designing object-oriented databases. The notation for tuple-identifiers is similar to the tagged notation used for creating object-identifiers.

Some more interesting solutions for problems with object-oriented systems, such as object sharing, can be handled by associating object-ids explicitly with objects and storing these object-identifiers in the VALTREE as though they were the values.

6.3 Granularity of the Database

While it may be ideal to save every atomic value in the VALTREE and have a pointer for each atomic value in the structure node of the RECLIST, this may not be appropriate or feasible. It is therefore left to the DBA to adjust the granularity of indexing. Thus tuples which are always accessed together and never as components may be stored as a single entity in the RECLIST and the key value for the tuple may be stored in the VALTREE instead of storing all individual values.

6.4 Intermediate Results

Most database queries are performed in stages, thus intermediate results are very important. As our algorithms depend on the use of two data structures it may be important to maintain the two data-structures for all partial results. We have not yet studied the issue of intermediate results in detail. Several approaches to this problem could include:

1. Do not maintain any new data-structures on partial results; use tids and extract from the same VALTREE and RECLIST all the values as and when needed.
2. Assume that the partial result is a new structure and store the structure as a RECLIST and add values to the existing VALTREE.
3. Generate new, small and temporary VALTREE and RECLIST structures which survive only until the expression has been evaluated.

6.5 Query Optimization

This is another issue that has not been studied in detail for nested relational models. The VALTREE and the RECLIST are an integral part of our storage scheme and they should be exploited to perform query optimization. Furthermore, while the algebraic properties for the nested algebra are fairly well understood, as was demonstrated in some of the examples of the previous section, alternate query plans for the same query are possible. We believe that query optimization should not only take into account the algebraic properties but should also consider heuristics and the current state of the database. We are currently involved in studying this problem. While it is possible to draw parallels from the query optimization techniques for the relational model, these techniques cannot be mapped directly to the NRDM as additional problems need to be addressed.

6.6 Partitioning and Parallelism

Nested structures inherently partition the data horizontally. Another level of partitioning of the data occurs in the VALTREE. For instance, the tuples in a structure are partitioned according to the values they have. We can effectively set up locks at each value level thereby allowing us to use concurrent processes to perform our operations. When we are performing an update, we need to lock only the concerned values and do not need to lock the entire database. This approach lets us localize in memory our most active and interacting processes. Furthermore, partitioning of the database allows us to perform several operations in parallel.

We have been investigating a parallel implementation of ANDA, PANDA on the massively parallel Data Structure Machine (DSM) which is being built at Indiana University [5].

7 Acknowledgements

We would like to think José Blakeley, Timothy Bridges, Umeshwar Dayal, Paul DeBra, Patrick Fischer, Hank Korth, Ravi Krishnamoorthy, Nancy Martin and Jan Paradaens for their suggestions. Jan Bond, Ng Chook, Vivian Howat, Ryan Hou, Shirley Lee, Edy Liongosari, Norma MacKay, Prab Sal, Lin-Long Shyu helped with the implementation.

References

[1] Shaku Atre. *Data Base: Structured Techniques for Design, Performance, and Management.* John Wiley and Sons, Inc, second edition, 1988.

[2] François Bancilhon. Object-oriented database systems. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Austin*, pages 152–162, March 1988.

[3] Nicole Bidoit. Efficient evaluation of relational queries using "nested relations". INRIA internal report, 1985.

[4] José Blakeley, Pedro Celis, Latha Colby, Anand Deshpande, Ieming Jeng, Sidney W. Kitchel, Nancy Martin, David Plaisier, and Dirk Van Gucht. An object-oriented database using a nested relational backend. An extended abstract submitted to the 2nd International Workshop on Object-Oriented Databases, April 1988.

[5] Timothy R. Bridges and Anand Deshpande. An efficient implementation of nested relational databases on the massively parallel data structure machine. submitted to the International Symposium on Databases in Parllel and Distributed Systems, May 1988.

[6] M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the Twelfth International Conference on Very Large Databases, Kyoto*, pages 91–100, August 1986.

[7] P. Dadam, K. Küspert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, and G. Walch. A DBMS prototype to support extended NF^2 relations: An integrated view on flat tables and hierarchies. In *Proceedings of ACM-SIGMOD '86 International Conference on Management of Data, Washington, D.C.*, pages 356–367, 1986.

[8] C.J. Date. Why is it so difficult to provide a relational interface to ims? *InfoIMS*, 4(4), 1984. 4th Quarter.

[9] Umeshwar Dayal, Frank Manola, Alejandro Buchmann, Upen Chakravarthy, David Goldhirsch, Sandra Heiler, Jack Orenstein, and Arnon Rosenthal. Simplifying complex objects: The PROBE approach to modelling and querying them. In *Proceedings of the Conference on Datenbank-Systeme für Büro, Technik und Wissenschaft, Darmstadt, Informatik Fachberichte*. Springer-Verlag, April 1987.

[10] U. Deppisch, H.-B. Paul, and H.-J. Schek. A storage system for complex objects. In *Proc. of the Int'l Workshop on Object-Oriented Database System, Pacific Grove*, pages 183–195, 1986.

[11] Anand Deshpande and Dirk Van Gucht. An implementation for nested relational databases. Technical Report 243, Computer Sceince Department, Indiana University, Bloomington, February 1988.

[12] Vinay Deshpande and Per-Ake Larson. An algebra for nested relational databases. Technical report, University of Waterloo, Waterloo, Canada, November 1987.

[13] M. Jarke and J. Koch. Query optimization in in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[14] W. Kim, D.S. Reiner, and D.S. Batory. *Query Processing in Database Systems*. Springer-Verlag, Berlin Heidelberg, 1985.

[15] Per-Ake Larson. Dynamic hash tables. *Communications of the ACM*, 31(4):446–457, April 1988.

[16] M. Missikoff. A domain based internal schema for relational database. In *Proceedings of ACM-SIGMOD 1983 International Conference on Management of Data, San Jose*, pages 215–224, 1982.

[17] M. Missikoff and M. Scholl. Relational queries in domain based DBMS. In *Proceedings of ACM-SIGMOD 1983 International Conference on Management of Data, San Jose*, pages 219–227, 1983.

[18] Jan Paredaens and Dirk Van Gucht. Possibilities and limitations of using flat operators in nested algebra expressions. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Austin*, pages 29–38, 1988.

[19] Peter Pistor and F. Anderson. Designing a generalized NF^2 model with an SQL-type language interface. In *Proceedings of the Twelfth International Conference on Very Large Databases, Kyoto*, pages 278–285, August 1986.

[20] Mark A. Roth, Henry F. Korth, and Don S. Batory. SQL/NF: A query language for $\neg$1NF relational databases. *Information Systems*, 12(1):99–114, 1987.

[21] Marc H. Scholl. Theoretical foundation of algebraic optimization utilizing unnormalized relations. In *International Conference on Database Theory, Rome (Lecture Notes in Computer Science 243)*, pages 380–396. Springer-Verlag, 1986.

[22] K.E. Smith and S.B. Zdonik. Intermedia: A case study of the difference between relational and object-oriented database systems. In *Proc. of OOPSLA, Miami, Fl.*, 1987.

[23] D.C. Tsichritzis and F.H. Lochovsky. *Data Base Management Systems*. Academic Press, Inc., 1977.

[24] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, June 1987.

[25] Patrick Valduriez, Setrag Khoshafian, and George Copeland. Implementation techniques of complex objects. In *Proceedings of the Twelfth International Conference on Very Large Databases, Kyoto*, pages 101–110, August 1986.

[26] Gio Wiederhold. *File Organization for Database Design*. McGraw-Hill Book Company, 1987.