An Implementation for Nested Relational Databases

by

Anand Deshpande and Dirk Van Gucht
Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 243

An Implementation for Nested Relational Databases
by Anand Deshpande and Dirk Van Gucht

February, 1988

A condensed version of this report appeared in F. Bancilhon and D. J. DeWitt (Eds.), *Proceedings of the 14th International Conference on Very Large Data Bases,* (Morgan Kaufmann Publishers, 1988), pp. 76–87.

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

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 implementaion are:

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

1. Introduction

In 1977 Makinouchi [29] proposed to generalize the relational model by removing the first normal form assumption. Jaeschke and Schek [21] introduced a generalization of the ordinary relational model by allowing relations with set-valued attributes and adding two restructuring operators, the *nest* and the *unnest* operators, to manipulate such (one-level) nested relations. Thomas and Fischer [46] generalized Jaeschke and Schek's model and allowed nested relations of arbitrary (but fixed) depth. Roth, Korth and Silberschatz [38] defined a calculus like query language for the nested relational database model (NRDM) of Thomas and Fischer. Since then numerous SQL-like query languages [28, 35, 36, 37], graphics-oriented query languages [19] and datalog-like languages [3, 4, 7, 26] have been introduced for this model or slight generalizations of it. It should also be mentioned that other researchers [1,8,16,50] have addressed similar issues for a more restricted class of nested relations, called hierarchical structures.

Several attempts at implementing the nested relational model directly are being made, notable are among them are the AIM project at IBM-Heidelberg [12,35,36], DASDBS at TH-Darmstadt [14, 33, 34, 41], and the VERSO project at INRIA [5, 8] It is also important to mention some other efforts at building object-oriented databases and post-relational systems [6, 9, 13, 18, 10, 23, 42, 43, 44, 45, 52].

In this paper we propose an implementation for the 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, nested algebra operations like select, join and nest are 'value-driven' while project and unnest are not. To implement the value-driven operations it is crucial to be able 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 in order to perform queries more efficiently.

We chose to implement the NRDM from scratch rather than map it to some other existing database implementations for the following reasons:

- Tuple components are not necessarily atomic, making the mapping to the relational model difficult [42].
- Query optimizations that exploit the nested relational model cannot be used when the underlying storage structure is relational [8].
- Selections are often made on components deeply nested within tuples [16, 28, 35, 36, 37].
- Hierarchical and network models were not developed with high level non-procedural languages in mind [11].

In Section 2 of this paper, we briefly discuss the nested relational model and describe an algebra for it. In Section 3 we discuss the architecture of our implementation. In Section 4 we describe a notation for tuple identification and then discuss the three major components of our implementation – `VALTREE`, `RECLIST` and `CACHE`. In Section 5 we discuss the operations on these three components and define an access language. In Section 6 we demonstrate how some typical nested algebra queries could be implemented. Finally, in Section 7 we discuss important observations about this implementation and discuss issues that need further investigation.

2. The Nested Relational Database Model

Let us consider two nested relational structures, `SCHEDULE` and `AIRLINE-INFO`, as shown in Figure 1 and Figure 2. The structure `SCHEDULE` stores information about universities their nearest airports and a list of their away football games while the `AIRLINE-INFO` structure stores information about cities, flights departing the city and airlines for which the city is a hub. These structures have been chosen to illustrate some typical features of our approach and will be used as examples throughout this paper.

The NRDM defines data as a set of nested relational structures, i.e., a component of a tuple in such a structure can be an atomic value or a nested relational structure itself. This is in contrast with the classical relational model where a database is defined as a set of flat relations in which components of the tuple are always atomic values.

The class of nested structures we have restricted ourselves to in our implementation is referred to as the class of hierarchical structures [1,8,16,50]. The difference between hierarchical structures and general nested structures is that in a hierarchical structure a combination of atomic attributes form the key at each level of the structure, whereas

TEAM	NEAREST-AIRPORTS	TEAMS-TO-PLAY
Indiana	Indianapolis Cincinnati Louisville	Purdue Michigan Wisconsin
Purdue	Indianapolis 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 1: The SCHEDULE structure

this restriction is not imposed on a general nested structure. The algebra used for manipulating hierarchical structures is an extension of mechanisms used to manipulate flat relations, with the addition of some restructuring operators.

Union ($\bigcup^e$) : The union operator has to be extended to ensure that the resulting structure is hierarchical, i.e. the key property is maintained. The extended union operator involves a union of all tuples and then a fusion of set components with identical atomic values.

Difference ($-^e$) : The difference operator is extended in a similar manner as the union operator.

Project (π^e) : The standard project operator does not maintain hierarchical structures. Therefore project, like union, fuses set components that have identical key values. See Example 11 in the Appendix.

Select (σ^e_C) : The select operator though very similar to the relational model is far more expressive in this model. Selects can be performed at any level of the structure. For convenience a template for filtering tuples is provided [8, 16]. Besides standard conditional operators ($=, \neq, >, \geq, <, \leq$) which can be used for comparing atomic values we also need ($\in, \notin, \subset, \not\subset, \subseteq, \not\subseteq$) to compare sets and elements of the set.

	FLIGHTS		
CITY	DESTINATION	AIRLINES	AIRLINE-HUBS
Indianapolis	Chicago	Transworld United	Transworld United
	NewYork	United Eastern	
	St. Louis	Transworld Continental	
St. Louis	Chicago	Transworld United	Transworld Eastern
	New York	Transworld Eastern	
	Indianapolis	Transworld Continental	
	Detroit	Northwest Transworld	
Chicago	Indianapolis	United Eastern Northwest	United Eastern
	New York	Transworld Northwest United	
	St. Louis	Transworld	
	Detroit	Northwest	
	Los Angeles	United	
New York	Indianapolis	Transworld Eastern	Transworld United Eastern Delta
	St. Louis	Transworld	
	Detroit	Northwest	
	Cincinnati	Delta Eastern	
	Atlanta	Delta Eastern	

Figure 2: The AIRLINE-INFO structure

Join ($\bowtie$) : Joins are only allowed when the pivot attributes in the two structures are atomic. An example of the join operator is shown in Figure 11 in the Appendix.

Nest (ν) : The nest operator is restricted in such a way that the result of a nest operation yields a structure with only one set-valued attribute at the highest level. To

construct structures with multiple set-valued attributes one needs to use a combination of nest and join operations. See Figure 9 in the Appendix.

Unnest (μ) : The unnest operator is the inverse of the nest operator.

For a more detailed and a mathematical description of the algebra the reader is referred to [1,8,16, 50]. Examples showing how this set of algebraic operators manipulates hierarchical structures shown in Figure 1 and Figure 2 are described in the Appendix.

3. Architecture for the Implementation

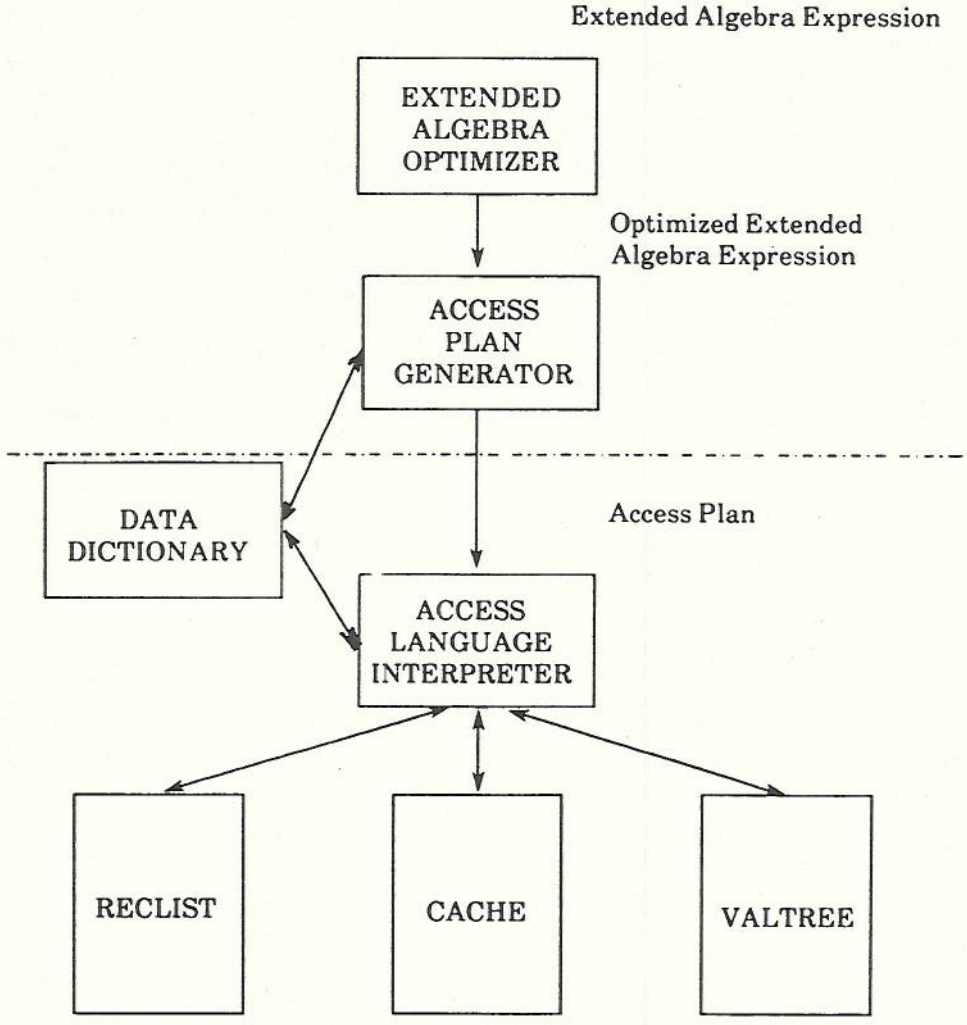

Figure 3: The proposed design for a nested relational database.

In this section we describe the architecture proposed to implement the nested relational database. As shown in Figure 3 our implementation has the following major components:

- 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,
- ACCESS PLAN GENERATOR – converts optimized extended relational algebra expressions to access plans, and
- EXTENDED ALGEBRA OPTIMIZER – optimizes the given extended algebra expression into an optimized expression.

The typical execution of a query expressed in the extended algebra would involve the following steps:

1. An extended relational algebra query is specified to the system.
2. The EXTENDED ALGEBRA OPTIMIZER will optimize this query into an optimized algebra expression. While research in the nested algebra optimization is still in its infancy, several results from relational algebra optimization [22, 25] can be naturally extended to the nested relations. [8, 3240]
3. The optimized algebra 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 ACCESS PLAN GENERATOR 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 this implementation we have considered only extended relational algebra operations, however the trend is to provide the user with an SQL like interface [35, 36, 37] or graphics oriented interface [19, 20, 53]. It will be possible for us to provide these interfaces by mapping SQL queries to algebra expressions or rewriting the ACCESS PLAN GENERATOR

module to convert from the user interface to the access language. In the current implementation we plan to maintain a loose coupling between the ACCESS PLAN GENERATOR and the ACCESS LANGUAGE INTERPRETER. The ability to specify queries directly in the access language would provide us with tools necessary to experimentally test contending access plans and develop heuristics for the ACCESS PLAN GENERATOR.

In the first phase of the implementation we have built a prototype of the ACCESS LANGUAGE INTERPRETER in Scheme [17] 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 in the secondary storage.

In Section 4 we shall discuss the details of the VALTREE, the RECLIST, and the CACHE. In Section 5 we discuss the ACCESS LANGUAGE. The EXTENDED ALGEBRA OPTIMIZER and the ACCESS PLAN GENERATOR are actively being researched and will not be discussed in this paper.

4. Components of the Implementation

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

4.1. A Notation for Tuple and Component Identification

In the nested relational model queries and updates can be performed on values that are deeply nested. For example, in the structure AIRLINE-INFO of Figure 2 we could select all cities that have flights by 'Northwest'. In this case, selections are to be performed on the AIRLINE attribute which is nested within the FLIGHTS attribute. 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. Also, as some of the components of the tuple could be sets, which in turn could have sets as their components, tuple identifiers cannot be flat but must be hierarchical.

In this section, we introduce a notation for identifying tuples and their components. 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.

Example 1: We will illustrate our notation on the (CITY (DESTINATION AIRLINES*)* AIRLINE-HUBS*)* structure. Thus, for 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 4, 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 or component identifier belongs to by going through the superscript strings and the subscript strings.

The length of the subscript and superscript strings depends on the depth of the hierarchical schema of the database. As the schema of the database is fixed, the lengths of the subscript and superscript strings are also fixed.

4.2. The Value-Driven Indexing Scheme

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 [30] and Missikoff and Scholl [31] 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

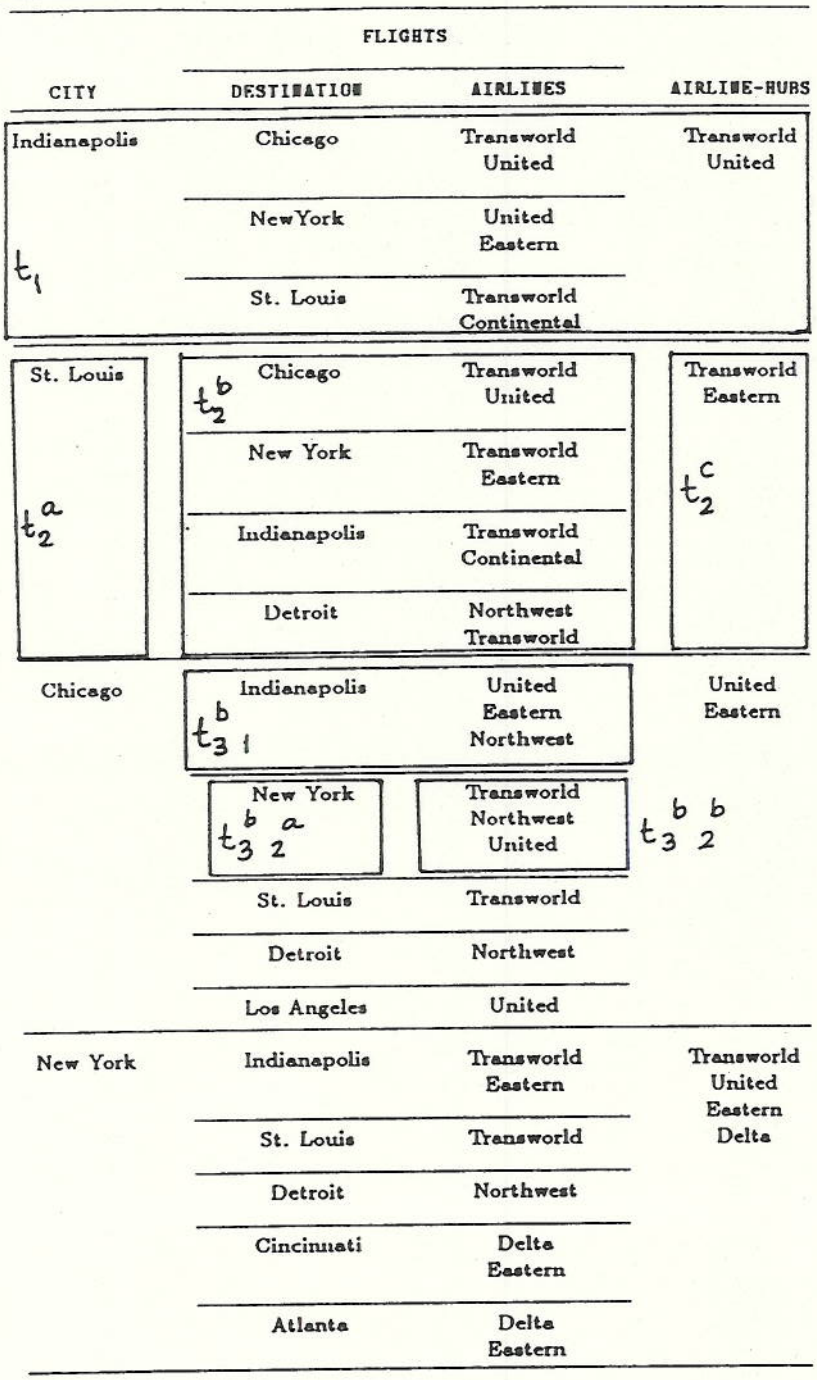

Figure 4: Tuple Notation for AIRLINE-INFO structure

indexed attribute, our scheme has only one tree, denoted VALTREE, that spans over all the atomic values of the database.

Valduriez, Khoshafian and Copeland [48, 49] 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 [2]. 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.

We now describe the VALTREE in more detail. VALTREE is made up of five different levels. 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.

The following observations can be made regarding the VALTREE structure:

1. An indexing technique like a B+Tree would be an appropriate data-structure at the VALUE level.
2. The granularity of the domain level depends on the database administrator and may depend on the installation. Typically, all compatible attributes belong to the same domain.
3. Values for attributes which do not participate actively in value-driven queries (i.e. remarks fields) need not be stored in the VALTREE.
4. It is possible to merge the attribute level and structure level into one if we avoid conflicts in attribute names over structures.

Example 2: In Figure 5, we show parts of the VALTREE for the structures s corresponding to the SCHEDULE structure and t corresponding to the AIRLINE-INFO structure.

4.3. The Record-List 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.

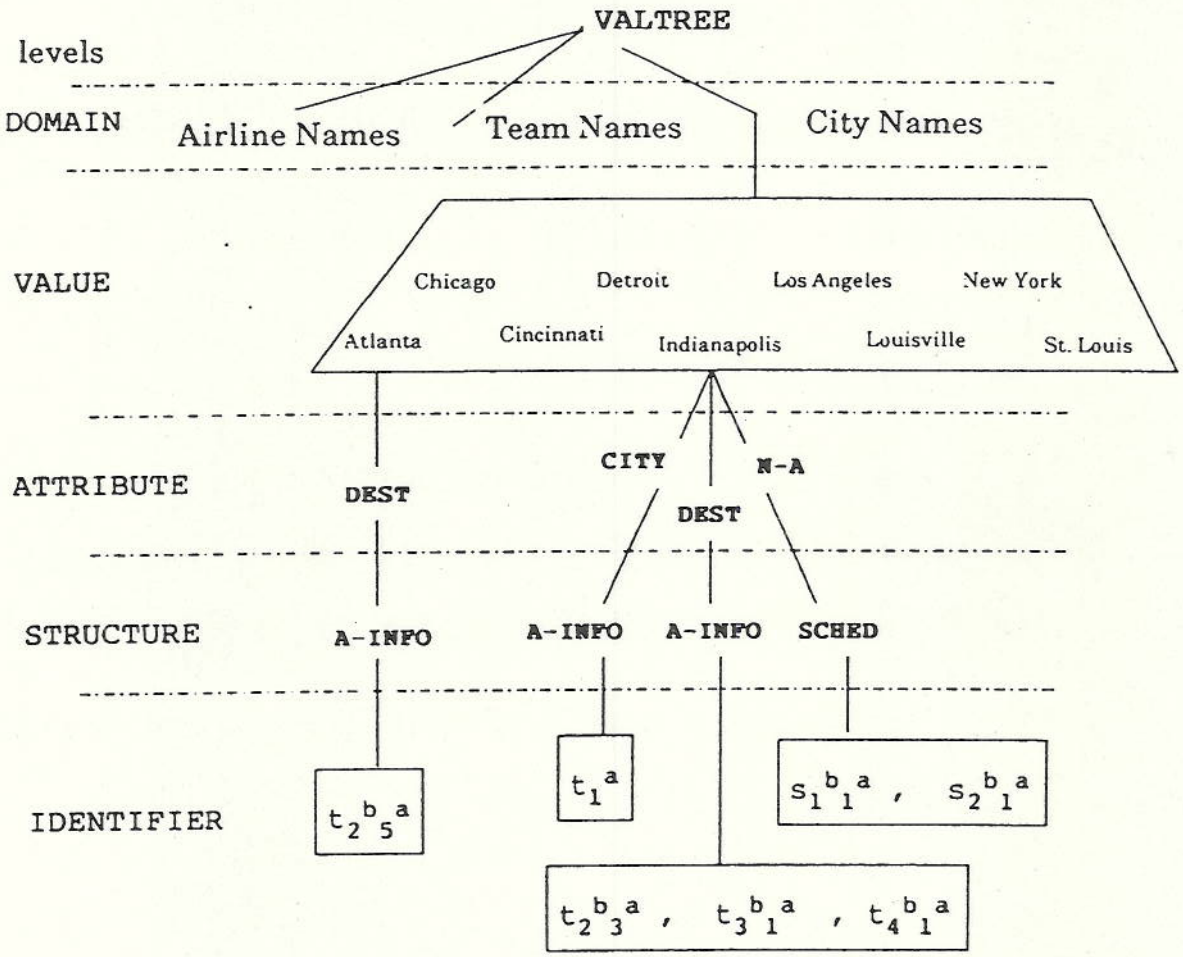

Figure 5: The VALTREE structure

- 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 [47] and Wiederhold [51] have a detailed description of some of these structures. Carey, DeWitt et.al. [9] have also suggested a storage structure for the Exodus project. Our implementation of the RECLIST is inspired by some of the storage structures proposed by Dadam et.al. and Deppisch et.al. [12, 14].

In this subsection we discuss three possible schemes for the implementation of the RECLIST. The first one is a simple scheme and is described to motivate the other two schemes. The second scheme is based on the implementations of Dadam et.al. and Deppisch et.al. [12, 14] and descriptions from Tsichritzis and Lochovsky [47]. The third scheme is the proposed storage scheme for the RECLIST and is compared with the second

scheme.

4.3.1. Simple Pointer Scheme

The simple pointer scheme consists of storing the tuples and sub-tuples as a linked list. Insertions to the linked lists are done at the end of the linked list and deletions are performed by flagging. Other variations of the linked list like double pointers and coral rings could be used to improve performance. However, to access a tuple when the hierarchical tid is known is tedious as a sequential traversal through linked lists is required, thus violating our goal of minimizing the disk accesses when the tid is given.

4.3.2. Array Pointer Scheme

In the simple pointer scheme it is difficult to access a sub-tuple when we have the hierarchical address because to get to the sub-tuple one has to traverse through all the intermediate pointers. To circumvent this problem all pointers within a tuple are moved to one single block – the structure block which stores only the structural information. Data pages now contain uninterpreted data [12, 14]. The Array-Pointer scheme for parts of the `AIRLINE-INFO` structure is illustrated in Figure 6. In this scheme it is possible to get to any value of the sub-tuple in exactly two disk accesses.

Two important advantages of the Array-Pointer scheme are:

1. This structure allows the relocation of data pages without having to alter tuple-identifiers. This is very important because tuple-identifiers are also used by the `VALTREE` and their stability is very critical.
2. Structure nodes for several tuples that would be required frequently could be cached thereby reducing disk accesses to the structure node. The `VALTREE` provides an indication of which tuples may be required by the query.

There are several issues that must be considered when using this scheme:

1. If we keep a pointer for each sub-tuple in the structure node it may become too large to fit in one disk block.
2. As the cardinality of a set is not fixed the structure nodes are dynamic. Therefore, it is not possible to predetermine the size of the structure node accurately.

These problems can be handled as follows:

1. Instead of having a pointer for each sub-tuple, several sub-tuples can be grouped together on one page, and instead of storing a separate pointer for each sub-tuple, one pointer is stored for a group of sub-tuples stored contiguously in the data pages.

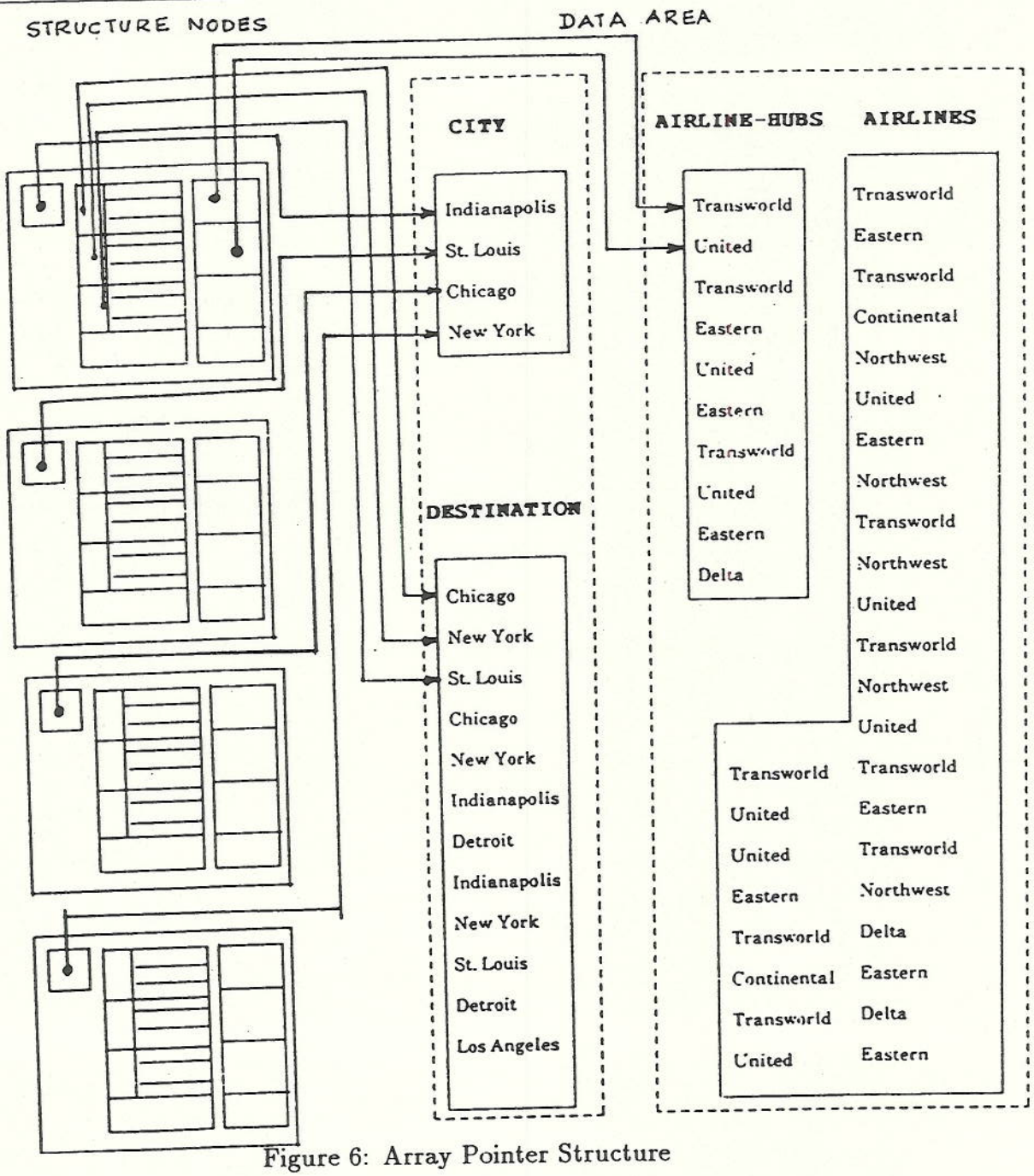

Figure 6: Array Pointer Structure

The exact location can be computed by using the base pointer and a displacement computed using the size of the sub-tuple. This reduces the number of pointers in the structure node but forces the data pages to store sub-tuples of the same kind.

2. It is not unreasonable to ask the DBA to specify typical cardinality for the set components at the time of the definition of the schema. This could be used as a 'guide' when designing the sizes of the pointer arrays. In case the array overflows,

a pointer to link to the next pointer array is also stored.

3. In case the structure nodes become too large it is possible to have a hierarchy of structure nodes. This structure makes the file look like an Indexed Sequential File.

4.3.3. Hashing Scheme

The objective of the structure node in the Array-Pointer scheme is to map the hierarchical tuple-identifier to the actual physical address. It is natural to consider a hashing scheme for this problem. Hashing could be done on the key values of the sub-tuple or on the tuple-identifiers of the sub-tuple.

Hashing on key values is not necessary as the `VALTREE` takes care of value-driven indexing. Since we are also interested in the ability to traverse all tuples and sub-tuples of the structure, hashing on key-values is not an appropriate scheme to do so.

The second alternative is hashing on tuple-identifiers. This scheme needs to be considered in the context of the granularity of the data and search, insert, delete and traverse operators.

Granularity : It is convenient to have uniform buckets so that the slots in the buckets are of the same size. This could be done by storing different kinds of sub-tuples in different hash areas. This is not a serious problem and it is a trade-off between the number of disk-accesses required to reconstruct a tuple and the granularity of the operations on parts of the sub-tuple. Thus if we have sub-tuples whose components would seldom be accessed individually then it makes sense to store all components of the sub-tuple in one slot of a bucket. This issue is discussed again in the section about granularity of operations in this model.

Search : To search a sub-tuple given by a tid, say ${t_x}^y{}_z$, we generate a hash-value from the tid which maps us to the bucket that contains the appropriate sub-tuple. As several tids map to the same bucket, it will be difficult to associate the appropriate sub-tuple to the tid unless that tid is stored along with the corresponding sub-tuple. This is at the cost of extra storage space. It must however be noted that this extra storage space is in the bucket and does not cost any more disk accesses.

Insert : To insert a tuple in the `RECLIST` structure one has to generate a new hierarchical tid before the tuple can be mapped to the appropriate bucket by the hash function. Hence it is required to save at each tuple and sub-tuple level the last generated tid for that level. Doing this bookkeeping will require some additional space. We propose the following two suggestions to handle this additional space problem:

1. We can have a structure node similar to the Array-Pointer scheme which stores counters instead of pointers for each tuple.
   - This is better than the Array-Pointer scheme because counters are not required for search and delete. They would be only be required for insert and for traverse.
   - The structure node for the hashing scheme is less dynamic than the Array-Pointer scheme since instead of keeping pointers for each sub-tuple all we need to store is the count. It should be noted however, that this structure is not totally static because we need one counter for each set of sub-tuples, in addition the sub-tuple can in turn have many sets of sub-tuples.
   - As the counters that need to be stored are smaller than pointers, the structure node for the Hashing scheme would be smaller than the Array-Pointer scheme. Hence, it should be possible to cache more structure nodes, thus reducing the number of disk accesses.
2. Instead of having all counters stored together in one structure node the counters could be placed along with the data pages. For example, if we number all tuples and sub-tuples starting with 1, the 0 value could be used to map to the slot in the data area which stores a counter instead of data-values. Thus the slot that corresponds to t_0 stores the counter for the number of tuples in the structure t.

Delete : It is not possible to reclaim tids by compression as this will involve updating all the tids of the `VALTREE`. Thus deletions are performed by flagging. However, it is possible to reclaim tids when inserting new sub-tuples.

Traverse : There are two kinds of traversals possible. Traversal within a tuple, i.e. the reconstruction of the tuple, and traversals across all tuples of the structure. Both kinds of traversals are performed by generating tids in order, until the maximum value is reached and then reconstructing the entire tuple.

Delete flags create holes in the sequence generated for traversal. Each time the tuple-identifier comes across a deleted sub-tuple a disk-access is wasted. A solution to this problem would be to store the delete flags in the structure node instead of storing them with the sub-tuples. This is done by storing a bitmap, where a bit of the bitmap is set if the corresponding sub-tuple is valid – stored and not deleted. Since the counter value can be computed by checking the largest set bit all counters can be replaced by bitmaps. Again both the schemes proposed to store counters could be used to store the bitmaps. An interesting observation can be made here: the layout of a structure node for the bitmaps is very similar to the Array-Pointer scheme, with a pointer field replaced by a bit to store the status of the corresponding sub-tuples. Hence all the observations made

about the structure node in the Array-Pointer scheme apply to bitmaps associated with the hashing scheme.

Now, traversal is performed by generating tuple-ids for sub-tuples where the delete flag shows the existence of a valid sub-tuple.

Deletions in this scheme can be performed by simply toggling the appropriate bit in the bitmap and not reclaiming the appropriate space immediately. This space could be reclaimed later by a background process or it can be reclaimed when a new tuple is inserted. This is possible because the tuple-identifier will always correspond to the same hash bucket.

To summarize, the linked list method is not really suitable because it is too slow. However, either the Array-Pointer or the the Hashing method would be appropriate. The Hashing method is better than the Array-Pointer method for the following reasons:

1. Search does not need an access to the structure node.
2. The number of disk-accesses required for the Hashing scheme when the structure node is required is exactly the same as the the number of accesses for the Array-Pointer scheme.
3. The structure node in all variations of the the Hashing scheme is smaller than the structure node of the Array-Pointer scheme.

4.4. 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 our implementation 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:

1. Retrieve tuple-ids from the VALTREE and place them in the cache,
2. Process tuple-ids in the cache, and
3. Retrieve appropriate parts from the RECLIST.

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

5. 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.

5.1. VALTREE functions

As discussed in Section 4.2 the nested relational structures are indexed by the VALTREE which consists of a B+Tree at the VALUE level of the VALTREE. Therefore operations on the VALTREE closely correspond to standard B+Tree operations.

5.1.1. Valtree Retrieve

You can imagine the vt-retrieve to be a procedure that does a pre-order traversal of the entire VALTREE (This is not actually the case, the vt-retrieve procedure makes judicious calls to the B+Tree tree and accesses only those parts of the VALTREE that are necessary). The arguments to the procedure are used to prune the VALTREE appropriately.

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

The arguments to this 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.

5.1.2. Valtree Insert

This procedure inserts appropriate information to the valtree. If a particular domain or a value does not exist the value will be added otherwise the tuple-id will be unioned to the set corresponding to the given ⟨ domain, value, attribute, structure ⟩ values.

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

The arguments to this procedures are:

- `domain` : If domain exists then the ⟨ value, attribute, structure, tid ⟩ are inserted, otherwise a new domain is created and then the rest of the information is added.
- `value` : If the value exists then the ⟨ attribute, structure, tid ⟩ are added, else the value is added.

- `attribute` : If the attribute exists then ⟨ structure, tid ⟩ are added, else the new attribute is added and then structure and tids are inserted.
- `structure` : If the structure-name exists then the tid is added otherwise a new structure-name is added before inserting the tid.
- `tid` : union tid to the existing set of tids.

5.1.3. Valtree Delete

Delete is very similar to the insert operation. A value, attribute or a structure are not deleted in the VALTREE until all its subtrees have been deleted.

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

5.2. RECLIST functions

We use the hashing scheme with bitmaps stored along with data values as discussed in Section 4.3.3 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.

5.2.1. 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)
```

5.2.2. Reclist Insert

This inserts the template in the appropriate structure ensuring that the hierarchical database properties are maintained. This procedure also generates a list of commands that correspond to `vt-inserts` 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)
```

5.2.3. Reclist Delete

This deletes the template in the appropriate structure ensuring that the hierarchical database properties are maintained. This procedure also generates a list of commands that corresponds to deletes in the valtree.

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

5.3. CACHE functions

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

5.3.1. Standard Stack Operations

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

5.3.2. Set Operations

Union : This forms the union of the top two elements (sets of tids) of the stack and places the result on the top of the stack.

```
union (stack, tid-window-format)
```

Intersection : This forms the intersection of the top two elements (sets of tids) of the stack and places the result on the top of the stack.

```
intersection (stack, tid-window-format)
```

Difference : This result of this operation is a set of tuple-ids that belong to the top element of the stack but do not belong to the second element of the stack.

```
difference (stack, tid-window-format)
```

- `tid-window-format` : determines the window for the union operation. 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.

Product : This takes the top two elements of the stack and forms the cartesian product of the top two elements and places the result as the top element of the stack. Thus, if the top element of the stack has n_1 tuple-ids and the next element on the stack has n_2 tuple-ids then, the resulting element on top of the stack has $n_1 \times n_2$ pairs of tuple-ids.

`product (stack)`

5.3.3. 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 or 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 teh 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.

6. Access plans for some NRDM operations

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

Example 3: *Select* CITIES *having 'Eastern'* FLIGHTS *to 'Indianapolis'.*

In this example we have two constant values DESTINATION = 'Indianapolis' and AIRLINES = 'Eastern' and finally we want to project the CITY component.

Following is the access plan:

1. vt-retrieve(City-Name, Indianapolis, DESTINATION, AIRLINE-INFO, structure,S1)
2. vt-retrieve(Airline-Name, Eastern, AIRLINES, AIRLINE-INFO, structure,S1)
3. transform(S1, $t_*{}^b{}_*{}^a$)
4. intersection(S1, $t_*{}^b{}_*{}^a$)
5. transform(S1,$t_*{}^a$)
6. rl-retrieve(S1)

The corresponding state of the CACHE for the above query:

1. $S1 = \langle\{t_2{}^b{}_3{}^a, t_3{}^b{}_1{}^a, t_4{}^b{}_1{}^a\}\rangle$
2. $S1 = \langle\{t_1{}^b{}_2{}^b{}_2{}^a, t_2{}^b{}_2{}^b{}_2{}^a, t_3{}^b{}_1{}^b{}_2{}^a, t_4{}^b{}_1{}^b{}_2{}^a, t_4{}^b{}_4{}^b{}_2{}^a, t_4{}^b{}_5{}^b{}_2{}^a\}, \{t_2{}^b{}_3{}^a, t_3{}^b{}_1{}^a, t_4{}^b{}_1{}^a\}\rangle$
3. $S1 = \langle\{t_1{}^b{}_2{}^a, t_2{}^b{}_2{}^a, t_3{}^b{}_1{}^a, t_4{}^b{}_1{}^a, t_4{}^b{}_4{}^a, t_4{}^b{}_5{}^a\}, \{t_2{}^b{}_3{}^a, t_3{}^b{}_1{}^a, t_4{}^b{}_1{}^a\}\rangle$
4. $S1 = \langle\{t_3{}^b{}_1{}^a, t_4{}^b{}_1{}^a\}\rangle$
5. $S1 = \langle\{t_3{}^a, t_4{}^a\}\rangle$
6. $[t_3{}^a = Chicago, t_4{}^a = New\ York]$

Example 4: *Select* CITY, DESTINATION *pairs having 'Eastern'* FLIGHTS *leaving from 'Transworld'* 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 'Transworld' 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, Transworld, 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. $[t_1{}^a = Indianapolis, t_1{}^b{}_2{}^a = New\ York,$
    $t_2{}^a = St.\ Louis, t_2{}^b{}_2{}^a = New\ York,$
    $t_4{}^a = New\ York, t_4{}^b{}_1{}^a = Indianapolis, t_4{}^b{}_4{}^a = Cincinnati, t_4{}^b{}_5{}^a = Atlanta]$

Example 5: *For each airport in the* SCHEDULE *structure, list all teams close to it.*

In this example, for each city we are interested in collecting teams which include the city as one of the nearest airports.

This query is shown in Example 10 in the Appendix. Algebraically, it can be expressed as

$$\nu_{\text{TEAM}}(\mu_{\text{NEAREST-AIRPORT}}(\pi^e_{\text{TEAM,NEAREST-AIRPORT}}(\text{SCHEDULE})))$$

The access plan for this rather complex query is as follows:

1. vt-retrieve(City-Name, *, NEAREST-AIRPORTS, SCHEDULE, structure, S1)
2. **repeat**
   2.1 copy(S1,S1)
   2.2 one-of(S1)
   2.3 rl-retrieve(S1)
   2.4 transform(S1,$s_*{}^a$)
   2.5 rl-retrieve(S1)
2. **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.**
   - 2.1 $S1 = \langle\{s_2{}^b{}_2{}^a, s_3{}^b{}_1{}^a, s_6{}^b{}_1{}^a\}, \{s_2{}^b{}_2{}^a, s_3{}^b{}_1{}^a, s_6{}^b{}_1{}^a\}, \{s_1{}^b{}_2{}^a\}, \{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\}, \{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{s_1{}^b{}_3{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$
   - 2.2 $S1 = \langle\{s_2{}^b{}_2{}^a\}, \{s_2{}^b{}_2{}^a, s_3{}^b{}_1{}^a, s_6{}^b{}_1{}^a\}, \{s_1{}^b{}_2{}^a\}, \{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\}, \{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{s_1{}^b{}_3{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$
   - 2.3 $[s_2{}^b{}_2{}^a = Chicago]$
     $S1 = \langle\{s_2{}^b{}_2{}^a, s_3{}^b{}_1{}^a, s_6{}^b{}_1{}^a\}, \{s_1{}^b{}_2{}^a\}, \{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\}, \{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{s_1{}^b{}_3{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$
   - 2.4 $S1 = \langle\{s_2{}^a, s_3{}^a, s_6{}^a\}, \{s_1{}^b{}_2{}^a\}, \{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\}, \{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{s_1{}^b{}_3{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$
   - 2.5 $[s_2{}^a = Purdue, s_3{}^a = Nortwestern, s_6{}^a = Illinois]$
     $S1 = \langle\{s_1{}^b{}_2{}^a\}, \{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\}, \{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{s_1{}^b{}_3{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$

Example 6: *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 is shown in Example 12 in the Appendix. Algebraically, this query is

$$\nu_{\text{TEAM}}(\mu_{\text{NEAREST-AIRPORT}}(\pi^e_{\text{TEAM,NEAREST-AIRPORT}}(\text{SCHEDULE}))) \bowtie \pi^e_{\text{FLIGHTS}}(\text{AIRLINE} - \text{INFO})$$

The access plan for this complex query is as follows:

1. `vt-retrieve (City-Name, *, (and DESTINATION NEAREST-AIRPORT), *, structure, S1)`
2. **repeat**
   - 2.1 `copy(S1,S1)`
   - 2.2 `one-of (S1)`
   - 2.3 `rl-retrieve(S1)`
   - 2.4 `transform(S1,` $t_*{}^b{}_*{}^b$`)`
   - 2.5 `rl-retrieve(S1)`
   - 2.6 `transform(S1,` s_*^a`)`
   - 2.7 `rl-retrieve(S1)`
2. **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.**

2.1 $S1 = \langle \{t_1{}^b{}_1{}^a, t_2{}^b{}_1{}^a\}, \{t_1{}^b{}_1{}^a, t_2{}^b{}_1{}^a\}, \{s_2{}^b{}_2{}^a, s_3{}^b{}_1{}^a, s_6{}^b{}_1{}^a\}, \{t_4{}^b{}_4{}^a\},$
$\{s_1{}^b{}_2{}^a\}, \{t_2{}^b{}_4{}^a, t_3{}^b{}_4{}^a\}, \{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\}, \{t_2{}^b{}_3{}^a, t_3{}^b{}_1{}^a, t_4{}^b{}_1{}^a\},$
$\{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{t_1{}^b{}_3{}^a, t_3{}^b{}_3{}^a, t_4{}^b{}_2{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$

2.2 $S1 = \langle \{t_1{}^b{}_1{}^a\}, \{t_1{}^b{}_1{}^a, t_2{}^b{}_1{}^a\}, \{s_2{}^b{}_2{}^a, s_3{}^b{}_1{}^a, s_6{}^b{}_1{}^a\}, \{t_4{}^b{}_4{}^a\},$
$\{s_1{}^b{}_2{}^a\}, \{t_2{}^b{}_4{}^a, t_3{}^b{}_4{}^a\}, \{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\}, \{t_2{}^b{}_3{}^a, t_3{}^b{}_1{}^a, t_4{}^b{}_1{}^a\},$
$\{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{t_1{}^b{}_3{}^a, t_3{}^b{}_3{}^a, t_4{}^b{}_2{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$

2.3 $[t_1{}^b{}_1{}^a = Chicago]$
$S1 = \langle \{t_1{}^b{}_1{}^a, t_2{}^b{}_1{}^a\}, \{s_2{}^b{}_2{}^a, s_3{}^b{}_1{}^a, s_6{}^b{}_1{}^a\}, \{t_4{}^b{}_4{}^a\}, \{s_1{}^b{}_2{}^a\}, \{t_2{}^b{}_4{}^a, t_3{}^b{}_4{}^a\},$
$\{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\}, \{t_2{}^b{}_3{}^a, t_3{}^b{}_1{}^a, t_4{}^b{}_1{}^a\}, \{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{t_1{}^b{}_3{}^a, t_3{}^b{}_3{}^a, t_4{}^b{}_2{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$

2.4 $S1 = \langle \{t_1{}^b{}_1{}^b, t_2{}^b{}_1{}^b\}, \{s_2{}^b{}_2{}^a, s_3{}^b{}_1{}^a, s_6{}^b{}_1{}^a\}, \{t_4{}^b{}_4{}^a\}, \{s_1{}^b{}_2{}^a\}, \{t_2{}^b{}_4{}^a, t_3{}^b{}_4{}^a\},$
$\{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\}, \{t_2{}^b{}_3{}^a, t_3{}^b{}_1{}^a, t_4{}^b{}_1{}^a\}, \{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{t_1{}^b{}_3{}^a, t_3{}^b{}_3{}^a, t_4{}^b{}_2{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$

2.5 $[t_1{}^b{}_1{}^b = \{Transworld, United\},$
$t_2{}^b{}_1{}^b = \{Transworld, United\}]$
$S1 = \langle \{s_2{}^b{}_2{}^a, s_3{}^b{}_1{}^a, s_6{}^b{}_1{}^a\}, \{t_4{}^b{}_4{}^a\}, \{s_1{}^b{}_2{}^a\}, \{t_2{}^b{}_4{}^a, t_3{}^b{}_4{}^a\}, \{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\},$
$\{t_2{}^b{}_3{}^a, t_3{}^b{}_1{}^a, t_4{}^b{}_1{}^a\}, \{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{t_1{}^b{}_3{}^a, t_3{}^b{}_3{}^a, t_4{}^b{}_2{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$

2.6 $S1 = \langle \{s_2{}^a, s_3{}^a, s_6{}^a\}, \{t_4{}^b{}_4{}^a\}, \{s_1{}^b{}_2{}^a\}, \{t_2{}^b{}_4{}^a, t_3{}^b{}_4{}^a\}, \{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\},$
$\{t_2{}^b{}_3{}^a, t_3{}^b{}_1{}^a, t_4{}^b{}_1{}^a\}, \{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{t_1{}^b{}_3{}^a, t_3{}^b{}_3{}^a, t_4{}^b{}_2{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$

2.7 $[s_2{}^a = Purdue, s_3{}^a = Northwestern, s_6{}^a = Illinois]$
$S1 = \langle \{t_4{}^b{}_4{}^a\}, \{s_1{}^b{}_2{}^a\}, \{t_2{}^b{}_4{}^a, t_3{}^b{}_4{}^a\}, \{s_4{}^b{}_1{}^a, s_5{}^b{}_1{}^a\}, \{t_2{}^b{}_3{}^a, t_3{}^b{}_1{}^a, t_4{}^b{}_1{}^a\},$
$\{s_1{}^b{}_1{}^a, s_2{}^b{}_1{}^a\}, \{t_1{}^b{}_3{}^a, t_3{}^b{}_3{}^a, t_4{}^b{}_2{}^a\}, \{s_6{}^b{}_2{}^a\}\rangle$

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 nested algebra to access plan is hardly a trivial task. Efforts are being made to determine if algebra is the right intermediate step for optimization. It may be possible to design query languages that generate better access plans.

7. Discussion

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

7.1. 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.

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	$\{s_4{}^{b}{}_{1}{}^{a}, s_5{}^{b}{}_{1}{}^{a}\}$
	Indianapolis	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 7: The VALTREE as a nested relational structure

7.2. 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 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 relational databases, though not entirely trivial, is much cleaner than the mapping to a relational model. 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 [13] 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.

7.3. 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.

7.4. Integrity Checking

When we perform an insert or delete we have to check if all the integrity constraints have been satisfied. This checking is based on a value-driven approach. For instance, when we say that $A \rightarrow B$, we mean that when the values of attribute A match in two tuples they must match in values of B. This can be checked by looking for each value corresponding to attribute A in the `VALTREE`. We get a set of tids, say S_A. We pick any one tid from the set S_A and then using the `RECLIST` find a corresponding value, v_B for attribute B. Now we can go to the `VALTREE` and extract the set of tids, S_B that

correspond to the value v_B and attribute B. The integrity constraint is satisfied if the set of identifiers associated with the B-value, S_B is a super-set of the list of tuple-identifiers of the A-value S_A. Other constraints also require that values for certain attribute obey a set of criteria. Conditions can therefore be verified while performing inserts in the VALTREE.

7.5. 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.

7.6. 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.

7.7. 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.

7.8. Computing Transitive Closures

Example 7: Let us consider the standard manager subordinate example. Let us say we pick the chairman who has some subordinates. Each of the subordinates in turn have some subordinates and so on. Assume we want to find all the subordinates of the chairman. To handle this example in our scheme, we first select the chairman. Now as the chairman is the manager of some subordinates, his name must appear as the manager attribute of those tuples. We get the tuple identifiers for all the subordinates as soon as we search for the chairman in the value-driven tree. Once we have the tuple identifiers for each of the subordinate tuples we can extract the names of the subordinates from the RECLIST structure. Once we have the names of the subordinates at the first level, we can, possibly in parallel, search all their sub-ordinates in a similar manner.

Example 8: : Determine all cities which have direct or indirect flights to St. Louis.

This problem can be solved as follows:

1. Since we are interested in all cities connected to city of 'St. Louis', we must find all cities that have flights with 'St. Louis' as the destination. To do this we look up the VALTREE and collect all tids where 'St. Louis' is the destination.
2. Now, for each tid obtained from the previous step we extract the tid for the CITY component and extract the city name from the RECLIST.

Now for each of the cities obtained from step 2 we repeat step 1 and 2, alternating selections between RECLIST and VALTREE. If the relationship between these sets is an arbitrary graph then it is possible to repeat indefinitely. To avoid this problem we need to keep a list of all cities included and include a new city only if it is not already there. The algorithm stops when no more cities can be generated. Notice that it is convenient to store a list of tids corresponding to the city component rather than storing city names.

7.9. Logic Programming Interface

There are numerous advantages to coupling logic programming with relational databases[24]. Beeri et.al. have proposed a logic language with sets which could be mapped to the

NRDM[7]. Most deductive systems are based on resolution and unification principles which are value-driven in nature. We believe that our storage structure would be appropriate for supporting such deductive systems.

8. References

1. S. Abiteboul, N. Bidoit, "Non First Normal Form Relations to Represent Hierarchically Organized Data", Proc. Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1984, 191-200.
2. S. Atre, *Data Base: Structured Techniques for Design, Performance, and Mangement*, Second Edition, John Wiley and Sons, Inc, 1988
3. F. Bancilhon, "A Logic-Programming/Object-Oriented Cocktail," SIGMOD Record, Vol. 15, No. 3 (Sept. 1986), pp. 11-20.
4. F. Bancilhon, S. Khoshafian, "A Calculus for Complex Objects" Proc. Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1986, 53-59.
5. F. Bancilhon, P. Richard, M. Scholl, "On Line Processing of Compacted Relations", *Proc.8th VLDB*, 1982, 263-269.
6. J. Banerjee, W. Kim, H.-J. Kim, H.F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases" *Proc. ACM SIGMOD*, San Fransisco, May 1987, pp. 311-322
7. C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, S. Tsur, "Sets and Negation in Logic Database Language (LDL1)" *Proc. 6th PODS*, San Diego, 1987, pp. 21-37.
8. N. Bidoit, "Efficient Evaluation of Relational Queries Using Nested Realtions", *Rapports de Recherche, no 480*, INRIA, 1986
9. M.J. Carey, D.J. DeWitt, J.E. Richardson, E.J. Shekita, "Object and File Management in the EXODUS Extensible Database System", *Proceedings of the Twelfth International Conference on Very Large Data Bases*, Kyoto, August 1986, pp 91-100.
10. G. Copeland, and D. Maier, "Making Smalltalk a Database System," Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass. June 1984
11. C.J. Date, "Why Is It So Difficult to Provide a Relational Interface to IMS?", *InfoIMS*, Vol. 4, No. 4, (4th Quarter 1984), PO Box 20651, San Jose, California 95160.
12. P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, G. Walch, "A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies", *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Washington, D.C., 1986, 356-366.
13. U. Dayal, F. Manola, A.Buchmann, U. Chakravarthy, D. Goldhirsch, S. Heiler, J. Orenstein, A. Rosenthal, "Simplifying Complex Objects: The PROBE Approach to Modelling and Querying Them", *Proc. GI Conf. on Database Systems for Office Automation, Engineering and Scientific Applications*, Darmstadt, April 1987

14. U. Deppisch, H.-B. Paul, H.-J. Schek, "A Storage System for Complex Objects", Proc. of the Int'l Workshop on Object-Oriented Database System, Pacific Grove, 1986, pp. 183-195.

15. A. Deshpande, D. Van Gucht, "A Storage Structure for Unnormalized Realtions", *Proc. GI Conf. on Database Systems for Office Automation, Engineering and Scientific Applications*, Darmstadt, April 1987, pp. 481-486.

16. V. Deshpande, P. Larson "An Algebra for Nested Relational Databases" Waterloo Technical Report

17. R. Kent Dybvig, *The Scheme Programming Language*, Prentice-Hall, 1987.

18. Dittrich K. and Dayal, U., eds. ,1986 International Workshop on Object Oriented Database Systems, Pacific Grove, Ca, Sept 23-26, 1986.

19. G. Houben, J. Paredaens, "The R^2-Algebra: An Extension of an Algebra for Nested Relations", *Tech. Rep.*, Tech. University, Eindhoven, 1987

20. B.E. Jacobs, C. Walczak, "A Generalized Query By Example Data Manipulation Language Based on Database Logic", *IEEE Transactions on Software Engineering SE-9*, 1 (January 1983), 40-57.

21. G. Jaeschke, H.-J. Schek, "Remarks on the the Algebra on Non First Normal Form Relations", Proc. ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1982, 124-138.

22. M. Jarke, J. Koch "Query Optimization in Database Systems" *Computing Surveys*, Vol. 16, No. 2, June 1984, pp. 111-152.

23. P. Kachhawaha R. Hogan, "LCE: An Object-Oriented Model for a Scientific Environment", Scientific Software Products, Indianapolis, 1987.

24. L. Kerschberg ed., *Expert Database Systems - Proceedings from the First International Workshop*, Benjamin/Cummings Publishing Company, Inc. 1986.

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

26. G.M. Kuper, "Logic Programming With Sets", *Proc. 6th PODS*, San Diego, 1987, pp. 11-20

27. G.M. Kuper, M.Y. Vardi, "A New Approach to Database Logic", Proc. Third SCM SIGACT-SIGMOD Symposium on the Principles of Database Systems, 1984, 86-96.

28. V. Linnemann, "Non First Normal Form Relations and Recursive Queries: An SQL-Based Approach", *Proc. 3rd IEEE Int. Conf. on Data Engineering*, Los Angeles, 1987

29. A. Makinouchi, "A Consideration of Normal Form of Not-Necessarily-Normalized Relations in the Relational Data Model", Proc. 5th Int'l Conf. on Very Large Data Bases, 1977, 447-453.

30. M. Missikoff, "A Domain Based Internal Schema for Relational Database Machines", Proc. ACM SIGMOD Int'l Conf. on Management of Data, 1982, 215-224.

31. M. Missikoff and M. Scholl, "Relational Queries in Domain Based DBMS", Proc. ACM SIGMOD Int'l Conf. on Management of Data, 1983, 219-227.

32. J. Paredans, D. Van Gucht, "Possibilities and Limitations of Using Flat Operators in Nested Algebra Expressions" *Proc. of ACM-PODS*, Austin, March 1988, to appear

33. H.-B. Paul, H.-J. Schek, M.H. Scholl, G. Weikum, U. Deppisch, "Architecture and Implementation of Darmstadt Database Kernel System" *Proc. Ann SIGMOD Conf.*, San Fransisco, 1987, pp. 196-207.

34. H.-B. Paul, A. Soder, H.-J. Schek, G. Weikum, "Unterstutzung der Buro-Ablage-Service durch ein Datenbankkernsystem" GI-Fachtagung Datenbanksysteme in Buro, Technik und Wissenschaft, Darmstadt, 1987, pp. 198-211

35. P. Pistor, F. Andersen, "Designing a Generalizied NF^2 Model with an SQL-Type Language Interface", *Proc. 12th VLDB*, Kyoto, Japan, 1986, pp. 278-288.

36. P. Pistor, R. Traunmueller, "A Database Language for Sets, Lists and Tables", *Information Systems 11:4*, 1986, pp. 323-336

37. M.A. Roth, H.F. Korth, D.S. Batory, "SQL/NF: A Query Language for ¬1NF Relational Databases", *Information Systems 12:1*, 1987, pp. 99-114.

38. M.A. Roth, H.F. Korth, A. Silberschatz, "Theory of Non-First-Normal-Form Relational Databases", *Tech. Report TR-84-36 (Revised January 1986)*, University of Texas at Austin, 1984.

39. H.-J. Schek, M.H. Scholl, "The Relational Model with Relation-Valued Attributes", Information Systems (11:2), 1986

40. M.H. Scholl, "Theoretical Foundations of Algebraic Optimization Utilizing Unnormalized Relations" *Proc. 1st ICDT*, Rome, Italy, Sept. 1986, in *Lecture Notes in Computer Science, 243*, G. Ausiello and P. Atzeni, eds., Springer Verlag, pp. 380-396.

41. M.H. Scholl, H.-B. Paul, H.-J. Schek "Supporting Flat Relations by a Nested Relational Kernel" *Proc. 13th VLDB*, London, 87.

42. K.E. Smith, S.B. Zdonik "Intermedia: A Case Study of the Difference Between Relational and Object-Oriented Database Systems" *Proc. of OOPSLA*, Miami, Fl. 1987

43. M. Stonebraker, *The INGRES Papers*, Addison Wesley, 1986.

44. M. Stonebraker, "Future Tresnds in DAtabase Systems", *Proc. of the Fourth Int'l. Conf. on Data Engineering*, Los Angeles, February, 1988

45. M. Stonebreaker, L.A. Rowe, ed. "The Postgres Papers", Memorandum No. UCB/ERL M86/85, Electronics Research Laboratory, University of California, Berkeley, 1987

46. S.J. Thomas, P.C. Fischer, "Nested Relational Structures", *Advances in Computing Research III, The Theory of Databases*, P.C. Kanellakis, ed., JAI Press, 1986, pp. 269 - 307.

47. D.C. Tsichritzis, F.H. Lochovsky, *Data Base Management Systems*, Academic Press, Inc., 1977

48. P. Valduriez, S. Khoshafian, G. Copeland "Implementation Techniques of Complex Objects" *Proceedings of the Twelfth International Conference on Very Large Data Bases*, Kyoto, August 1986, pp 101-109.

49. P. Valduriez, "Join Indices", *ACM TODS*, vol. 12, no. 2, June 1987, pp 218-246.

50. D. Van Gucht, P.C. Fischer, "High Level Data Manipulation Languages for Unnormalized Relational Database Models", Tech. Report, Indiana University, 1986.

51. G. Wiederhold, *File Organization for Database Design*, McGraw-Hill Book Company, 1987

TEAMS	NEAREST-AIRPORT
Indiana Purdue	Indianapolis
Purdue Northwestern Illinois	Chicago
Indiana	Cincinnati
Michigan Michigan State	Detroit
Indiana	Louisville
Illinois	St. Louis

Figure 9: $\text{SCHEDULE}'' = \nu_{\text{TEAM}}(\text{SCHEDULE}')$

52. C. Zaniolo, "The Database Language GEM", Proc. ACM SIGMOD Conference, 1983, 207-218.
53. M.M. Zloof, "Query-by-Example: Operations on Hierarchical Data Bases", IBM Research Report RC5491.

9. Appendix

The following examples illustrate some operators of the algebraic query language for the NRDM.

Example 9: $\text{SCHEDULE}' = \mu_{\text{NEAREST-AIRPORTS}}\pi^{e}_{\text{TEAM,NEAREST-AIRPORTS}}\ (\text{SCHEDULE})$

This example projects the teams and the nearest airports and then unnests it as shown in Figure 8.

TEAM	NEAREST-AIRPORT
Indiana	Indianapolis
Indiana	Cincinnati
Indiana	Louisville
Purdue	Indianapolis
Purdue	Chicago
Northwestern	Chicago
Michigan	Detroit
Michigan State	Detroit
Illinois	Chicago
Illinois	St. Louis

Figure 8: $\text{SCHEDULE}' = \mu_{\text{NEAREST-AIRPORT}}\pi^{e}_{\text{TEAM,NEAREST-AIRPORT}}\ (\text{SCHEDULE})$

Example 10: $\text{SCHEDULE}'' = \nu_{\text{TEAM}}(\text{SCHEDULE}')$

This example nests the SCHEDULE′ structure from the previous example and nests the TEAM component as shown in Figure 9. The resulting structure SCHEDULE″ lists cities and the teams that are close to them.

Example 11: FLIGHTS$' = \pi^{e}_{\text{FLIGHTS}}$ (AIRLINE-INFO)

This example projects the structure FLIGHTS from the structure AIRLINE-INFO as shown in Figure 10. Notice how the project operator causes the sets of AIRLINES corresponding to a CITY to merge, e.g observe the AIRLINES set for 'Indianapolis'.

DESTINATION	AIRLINES
Chicago	Transworld United
NewYork	United Eastern Transworld Northwest
St. Louis	Transworld Continental
Indianapolis	Transworld Continental United Eastern Northwest
Detroit	Northwest Transworld
Los Angeles	United
Cincinnati	Delta Eastern
Atlanta	Delta Eastern

Figure 10: FLIGHTS$' = \pi^{e}_{\text{FLIGHTS}}$ (AIRLINE-INFO)

Example 12: TEAM-AIRLINE = SCHEDULE$''$ $\bowtie$ FLIGHTS$'$

In this example we join the structure SCHEDULE$''$ with the FLIGHTS$'$ structure as shown in Figure 11. In our algebra, joins ($\bowtie$) are only defined when the pivot attributes are atomic and the domains are compatible. In this example a join is performed on the NEAREST-AIRPORT attribute of SCHEDULE$''$ structure and the DESTINATION attribute of the FLIGHTS$'$ structure.

TEAMS	NEAREST-AIRPORT	AIRLINES
Indiana Purdue	Indianapolis	Transworld Continental United Eastern Northwest
Purdue Northwestern Illinois	Chicago	Transworld United
Indiana	Cincinnati	Delta Eastern
Michigan Michigan State	Detroit	Northwest Transworld
Illinois	St. Louis	Transworld Continental

Figure 11: TEAM-AIRLINE $=$ SCHEDULE$''$ $\bowtie$ FLIGHTS$'$