TECHNICAL REPORT NO. 287

Processing Queries in ANDA: A Nested Relational Database System

by

José A. Blakeley and Anand Deshpande

August 1989

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

Processing Queries in ANDA: A Nested Relational Database System

José A. Blakeley Anand Deshpande*
Computer Science Department
Indiana University
Bloomington, IN 47405, USA
{blakeley,deshpand}@iuvax.cs.indiana.edu

August 17, 1989

Abstract

This paper presents the query processing architecture of ANDA, a prototype nested relational database system developed at Indiana University. The main features of ANDA are: (a) the extensive use of *structured tuple-ids* which allows greater opportunities for query processing in main memory. The transformation of queries into ANDA programs offers the opportunity to apply compiler optimization techniques to the database query optimization process; and (b) the application of rule-based query optimization techniques to provide a *flexible architecture*. Other features of the ANDA prototype are: the capability to efficiently compute joins, and the use of a programmable, flexible access language which provides direct access to the data structures of the system and is suitable for optimization. Queries are illustrated using Nested SQL, an upward compatible extension of SQL for nested relations.

1 Introduction

Researchers have argued about the advantages and disadvantages of the nested relational model over the conventional (*flat*) relational model [1]. At least, there seems to be agreement on the fact that there is a large class of real world applications whose data is hierarchically organized which would benefit considerably by systems that supported nested relations. Also, the nested relational model does not compromise the fundamental advantages of the relational model, namely, *data independence* and *nonprocedural query languages*. Thus making it a natural extension to the relational model.

*Authors' addresses: J.A. Blakeley, Information Technologies Laboratory, Texas Instruments Incorporated, P.O. Box 655474, MS 238, Dallas, Texas 75265; A. Deshpande, Hewlett-Packard Laboratories, 1501 Page Mill Road, Building 3U, P.O. Box 10490, Palo Alto, California 94303-0969.

Recently, there have been several other efforts to develop database systems based on the nested relational model. Some of these efforts include the AIM project at IBM Heidelberg Scientific Center [8, 9], the VERSO project at INRIA [4, 16], the DASDBS project at the Technical University of Darmstadt [27], the R^2D^2 system at Universität Karlsruhe, and the KAPPA project at ICOT [32]. Collectively, all these efforts have shown that building a nested relational database system is feasible.

This paper presents a query processing architecture for ANDA,[1] a prototype nested relational database management system developed at Indiana University. The ANDA project goes a step further than previous nested relational system projects by showing not only that such systems are feasible but also that they can process queries efficiently. ANDA uses a low-level access language amenable to nested relations as well as a query processing architecture that allows the translation of nonprocedural queries to programs in the access language. The architecture is unique in that it adapts the ideas of rule-based query processing used in extensible database systems to the implementation of a query processor for a nested relational system. Specifically, our design is distinct from previous nested relational database system development efforts for the following reasons.

Query processing based on tuple-ids. Query processing in ANDA consists of three basic stages. First, the user specifies a query by providing a predicate that contains the *values* that must be satisfied by some of the attributes of the tuples in the result. This is the usual way queries are specified in database systems providing nonprocedural interfaces. Using a data structure called the VALTREE, these values are mapped into a collection of tuple-ids of tuples that may potentially contribute to the result. Second, the collection of *tuple-ids* from the first stage are brought into main-memory and extensibly manipulated based on the logical connectors of the predicate as well as the list of attributes in the projection to obtain the result of the query in terms of sets of tuple-ids. Third, the sets of tuple-ids are mapped back to their corresponding database *values* using a data structure called the RECLIST. The VALTREE and RECLIST data structures where introduced by Deshpande and Van Gucht in [11, 12].

Tuple-id based query processing is feasible because of a judicious design of *structured tuple-ids*. In addition to uniquely identifying each value in the nested relational database, tuple-ids also *encode* the schema of their corresponding nested relation. Because our implementation of structured tuple-ids consists of a fixed-length bit pattern, it is possible to pack a large number of them in main memory. As a result of this, our system enjoys the opportunity of exploiting main memory query processing extensibly. This is in contrast to the strictly *value-oriented* query processing approach used by relational systems which rely, in some cases, on the use of temporary files to store intermediate results. In our system, intermediate results consist of sets of tuple-ids. To the best of our knowledge, no other relational nor nested relational system exploits tuple-ids for query processing to the extent that ANDA does.

Furthermore, the use of tuple-ids as a first class objects during query processing brings new opportunities for efficiency. For example, using the VALTREE data structure as a join index, it is possible to efficiently compute joins on single-valued, arbitrarily-nested attributes of two or more

[1] ANDA is an acronym formed from Architecture for Nested Database Applications.

nested relations. Also, the access language programs that result from the query optimization process which contain statements that manipulate and transform sets of tuple-ids can be further optimized using standard compiler optimization techniques.

A flexible query processing architecture. We feel that a flexible design is essential for two main reasons. First, since there have been several proposals for nested relational algebras each possessing different commutativity, distributivity, and associativity properties (e.g., [7, 13]), none of which has been widely adopted, we decided to allow the option to change it. This was achieved by building on the ideas of rule-based query optimization for extensible database systems [15, 17] The ability to change the algebraic properties of the nested algebra is provided by defining an *algebraic rule-base.*

Second, there is a large number of alternatives for implementing the storage-level interface for the nested model. Rather than relying on a single storage-level interface for nested relations as it is the case in relational implementations (e.g., System R's Relational Storage System [3]) ANDA supports several possible implementations of such an interface. We provide this flexibility by describing the expansion of each algebraic operation into a program in the target access language as rules. The ability to change the mapping from an algebraic operation to its corresponding sequence of statements in the access language is supported through what we call the *access language rule-base.* Currently, ANDA has been implemented using a single storage interface called the ANDA *access language processor.*

As an additional consideration, we want our development efforts to serve as building blocks of a testbed for research on query processing. Thus, we need the components of our query processor to be modular and reusable [5].

The paper is organized as follows. Section 2 presents a brief description of the nested relational model as well as the storage structures and commands of the ANDA access language processor. The ANDA access language processor represents the run-time component of the system. Section 3 gives an overview on how to program queries at the level of the access language as an introduction to later sections of the paper. This language is not intended as an end-user query language. Section 4 presents the query processing architecture of ANDA. This is the compile-time component of the system. Section 5 focuses on one aspect of the compilation process, namely, the rule-based approach to direct compilation of queries into ANDA access language programs. We have chosen to express user queries in Nested SQL, an upwards compatible extension of SQL for nested relations. Section 6 introduces our approach to access language program optimization using techniques widely used by optimizing compilers. Section 8 presents our conclusions, current status of the ANDA prototype, and directions for future research.

2 The ANDA Access Language Processor

In this section we describe the file structures, main memory component, and commands of the ANDA access language processor. Further information on ANDA can be found in [11, 12]. The design of the ANDA access language processor is motivated by two main considerations:

1. A large number of queries in the nested relational model are *value-oriented*, i.e., queries involve determining where a particular value exists in the database irrespective of the structure. Here by structure we mean the *schema* of the nested relation. However, there are some queries that are *structure-oriented*. Typically, data structures that are well suited for value-oriented operations are not well suited for structure-oriented queries and vice-versa. Hence ANDA uses two distinct data structures: the VALTREE for value-oriented operations and the RECLIST for structure-oriented operations, respectively.

2. For efficient query processing, it is important to exploit the large amounts of main-memory available in current machines. Main memory is exploited by using the CACHE for query processing. A structured, hierarchical tuple-id notation for identifying values makes it possible to process queries on tuple-ids rather than on values.

After introducing a database scheme used as a running example to be used throughout this paper, the following subsections describe the notion of structured tuple-ids, the VALTREE and RECLIST data structures, the CACHE or main memory component, and the ANDA access language commands.

2.1 An Example Database

In this subsection we describe a database of nested relations corresponding to the registration information that is stored in a typical university. There are three nested relations, (i) the Student relation, which stores personal information about students, (ii) the Faculty relation which stores personal information about faculty members who teach in the university and (iii) Course information that stores the offerings, prerequisites and enrollment for different courses offered by the university. We follow the convention that all relation names are capitalized. The nested relation *schemes* of the database are:

```
Course = (cname, dept, cno, credits, Prerequisite, Section)
   Prerequisite = (cno)
   Section = (secid, term, instructor, Enrollment)
      Enrollment = (sno, grade)
Student = (sid, sname, sdept, sage, ssex, sclass, sroomno, sdorm, smarried)
Faculty = (fid, fname, froomno, fdept, fsex, fage, fmarried)
```

A nested relational scheme can be represented by a scheme tree. Figure 1 illustrates the scheme tree for the Course nested relation. An *instance* of the Course relation is shown in Table 1.

2.2 Structured Tuple-ids

In ANDA each value in the database has a unique identifier, hereafter referred to as a tuple-id. Instead of generating unique, random tuple-ids, ANDA uses relation names tagged with subscripts and superscripts. The subscripts count instances of a particular subscheme in a nested relation. The

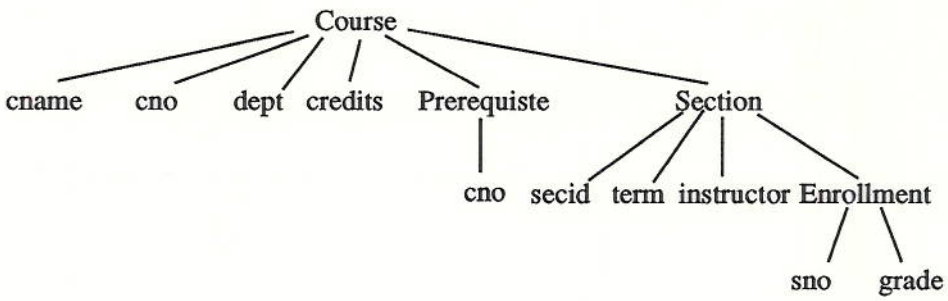

Figure 1: The scheme tree for the Course nested relation.

Course									
cname	cno	dept	credits	Prerequisite	Section				
				cno	secid	term	instructor	Enrollment	
								sno	grade
Database	C445	Computer Science	4	C311	1721	F88	Blake	149	B
				C343				284	A
								234	B
								635	C
								825	C
								261	F
								266	A
					1725	S89	Roberts	142	A
								261	B
								233	C
								634	D
Operating Systems	C435	Computer Science	4	C311	1127	F88	Dean	249	C
				C335				335	B
				C343				434	A
								535	D
								625	C
								761	B
								866	A

Table 1: An instance of the Course relation.

superscripts enumerate the different components (*attributes* and *subschemas*) of the nested relation. The organization of tuple-ids for hierarchies of nested relations have the following properties – (a) every value has a unique tuple-id, (b) given a deeply-nested tuple-id, it is possible to determine the tuple-ids of other neighboring components of the subtuple and the tuple-ids of the super-tuple, and (c) given two tuple-ids it is possible to compare them to determine if they belong to the same tuple or sub-tuple. Table 2 illustrates the tuple-ids corresponding to the Course nested relation of Table 1. The convention used to assign superscript names for tuple-ids of the Course relation is

shown in Figure 2.

Course									
cname	cno	dept	credits	Prerequisite	Section				
				cno	secid	term	instructor	Enrollment	
								sno	grade
$C_1{}^a$	$C_1{}^b$	$C_1{}^c$	$C_1{}^d$	$C_1{}^e{}_1{}^a$ $C_1{}^e{}_2{}^a$	$C_1{}^f{}_1{}^a$	$C_1{}^f{}_1{}^b$	$C_1{}^f{}_1{}^c$	$C_1{}^f{}_1{}^d{}_1{}^a$ $C_1{}^f{}_1{}^d{}_2{}^a$ $C_1{}^f{}_1{}^d{}_3{}^a$ $C_1{}^f{}_1{}^d{}_4{}^a$ $C_1{}^f{}_1{}^d{}_5{}^a$ $C_1{}^f{}_1{}^d{}_6{}^a$ $C_1{}^f{}_1{}^d{}_7{}^a$	$C_1{}^f{}_1{}^d{}_1{}^b$ $C_1{}^f{}_1{}^d{}_2{}^b$ $C_1{}^f{}_1{}^d{}_3{}^b$ $C_1{}^f{}_1{}^d{}_4{}^b$ $C_1{}^f{}_1{}^d{}_5{}^b$ $C_1{}^f{}_1{}^d{}_6{}^b$ $C_1{}^f{}_1{}^d{}_7{}^b$
					$C_1{}^f{}_2{}^a$	$C_1{}^f{}_2{}^b$	$C_1{}^f{}_2{}^c$	$C_1{}^f{}_2{}^d{}_1{}^a$ $C_1{}^f{}_2{}^d{}_2{}^a$ $C_1{}^f{}_2{}^d{}_3{}^a$ $C_1{}^f{}_2{}^d{}_4{}^a$	$C_1{}^f{}_2{}^d{}_1{}^b$ $C_1{}^f{}_2{}^d{}_2{}^b$ $C_1{}^f{}_2{}^d{}_3{}^b$ $C_1{}^f{}_2{}^d{}_4{}^b$
$C_2{}^a$	$C_2{}^b$	$C_2{}^c$	$C_2{}^d$	$C_2{}^e{}_1{}^a$ $C_2{}^e{}_2{}^a$ $C_2{}^e{}_3{}^a$	$C_2{}^f{}_1{}^a$	$C_2{}^f{}_1{}^b$	$C_2{}^f{}_1{}^c$	$C_2{}^f{}_1{}^d{}_1{}^a$ $C_2{}^f{}_1{}^d{}_2{}^a$ $C_2{}^f{}_1{}^d{}_3{}^a$ $C_2{}^f{}_1{}^d{}_4{}^a$ $C_2{}^f{}_1{}^d{}_5{}^a$ $C_2{}^f{}_1{}^d{}_6{}^a$ $C_2{}^f{}_1{}^d{}_7{}^a$	$C_2{}^f{}_1{}^d{}_1{}^b$ $C_2{}^f{}_1{}^d{}_2{}^b$ $C_2{}^f{}_1{}^d{}_3{}^b$ $C_2{}^f{}_1{}^d{}_4{}^b$ $C_2{}^f{}_1{}^d{}_5{}^b$ $C_2{}^f{}_1{}^d{}_6{}^b$ $C_2{}^f{}_1{}^d{}_7{}^b$

Table 2: The structured tuple-ids of the Course relation.

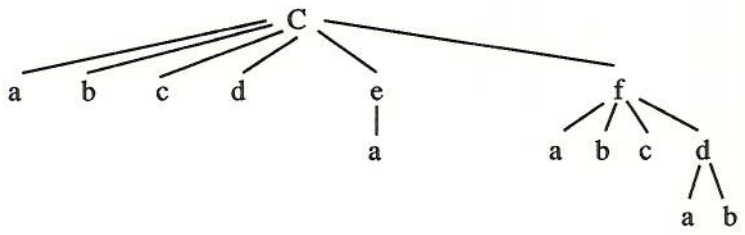

Figure 2: The superscripts for tuple-ids of the Course relation.

It is interesting to compare the notion of structured tuple-ids with the notion of object-identity in object oriented database systems. Khoshafian and Copeland [18] introduced the notion of strong identity for object-ids. In their definition, an implementation of object-ids is strong if it is *data independent* (i.e., identity is preserved through changes in either data values or structure) and *location independent* (i.e., identity is preserved through movement of objects among physical location or address spaces). In this paper we do not suggest using structured tuple-ids to implement object identity. However, if used for that purpose, then structured tuple-ids provide full location

and value independence, but do not provide full structure independence. Structured tuple-ids are independent from structural changes like addition or deletion of attributes but are dependent on changes like: splitting a nested relation into two or more nested relations, merging two or more nested relations into one, adding or removing levels of nesting within a nested relation.

To summarize, the main advantages of structured tuple-ids are: (1) to uniquely identify every single value in the database, (2) to allow more flexibility for in-memory query processing, and (3) to reduce (in some cases) disk accesses. Their main disadvantage is that they depend on the structure of the nested relations, so some schema changes may require a regeneration of tuple-ids.

2.3 The VALTREE

Traditional relational database management systems use indexing techniques to improve disk access time. Typically, indices are built on all or some of the attributes of a relation. ANDA's approach to indexing follows the domain based approach suggested by Missikoff and Scholl [25, 26] for relational databases. In their approach, every atomic value maps to a list of tuple identifiers of tuples in all relations in the database which contain that value. The VALTREE generalizes this approach by storing a mapping from each value to a list of all tuple-ids over the entire database. Hence, given an atomic value, the VALTREE provides a mapping to the list of hierarchical tuple-ids, which enables the system to determine directly which tuples or sub-tuples a value is stored in.

The VALTREE consists of five different levels as shown in Figure 3. The top-most level, the DOMAIN level, separates non-compatible domains into several subtrees. The second level, the VALUE level, stores all the atomic values of the database. This level is implemented in ANDA as a B^+-tree. The third level is the ATTRIBUTE level which stores all the attributes a particular value at the VALUE level belongs to. As the same attribute may belong to more than one relation, the fourth level called the RELATION level, contains all the relations associated with an attribute mentioned at the attribute level. Finally, the fifth and the lowest level of the VALTREE consists of all the tuple-ids that correspond to the the atomic value stored at the VALUE level; this is called the TUPLE-ID level. The advantage of using the VALTREE is that, given a value, it provides rapid access to the list of tuple-ids corresponding to all occurrences of the value throughout the entire database.

2.4 The RECLIST

The objective of the RECLIST is to provide an efficient mapping from a structured tuple-id to its corresponding value. ANDA implements the RECLIST using a linear hashing file structure [24]. The tuple-id is used as a key for the hashing scheme. <tuple-id, value> pairs are stored as the data value in the bucket obtained after applying the hash function on the tuple-id.

To insert a new tuple in the RECLIST, one has to generate a new tuple-id before the tuple can be mapped into an appropriate bucket. To keep track of the tuple-ids used and the next one available, a bitmap 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 for allocating tuple-ids during insertion and for reclaiming them during deletion.

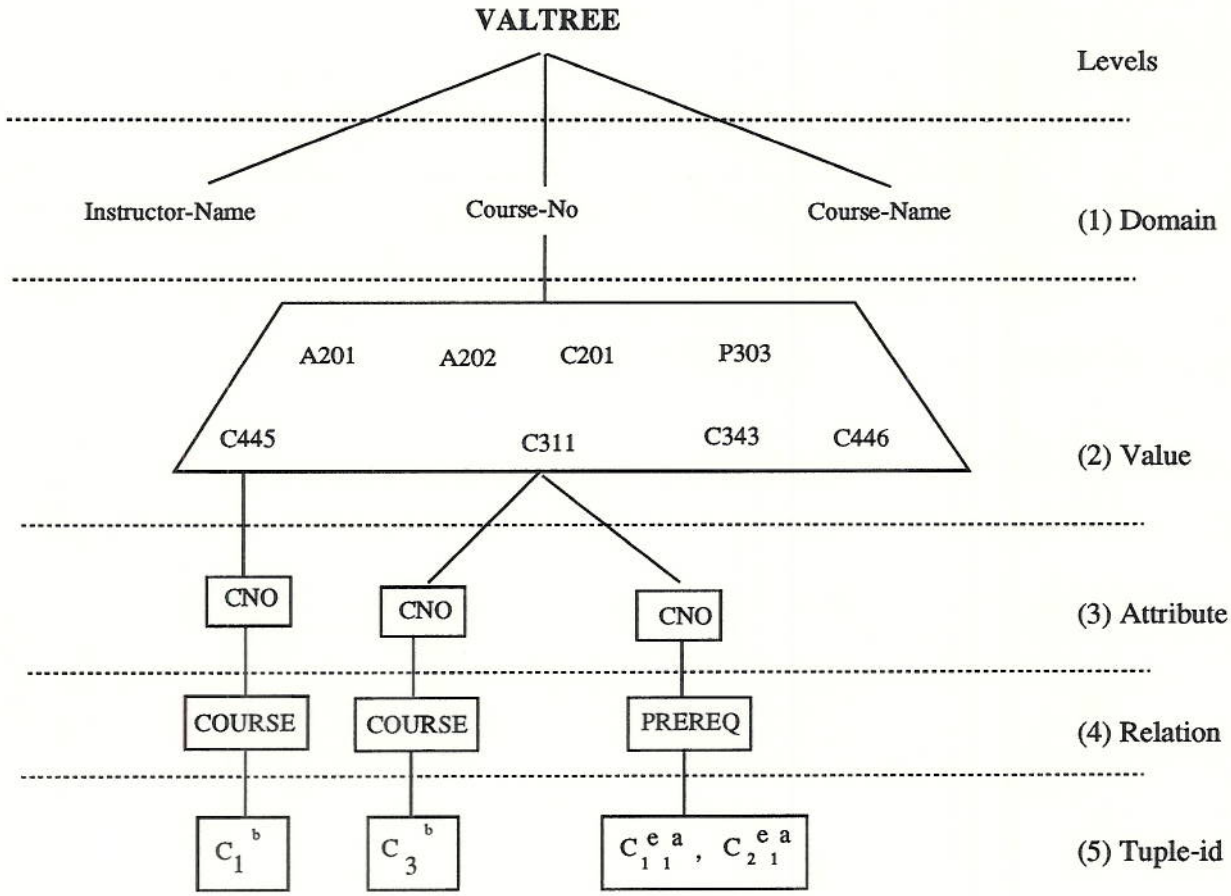

Figure 3: The VALTREE structure

2.5 The CACHE

The CACHE is the main-memory component of the ANDA access language processor. To improve performance of queries it is important to reduce disk accesses via the VALTREE and the RECLIST, and perform as many operations as possible in the CACHE.

Query processing in ANDA involves three basic steps. First, tuple-ids corresponding to conditions specified by the user query are obtained from the VALTREE and stored in the CACHE. This step provides the mapping from values to tuple-ids. Second, tuple-ids are manipulated extensively in the CACHE. After manipulating these tuple-ids in the CACHE the resulting tuple-ids are mapped back to their corresponding values using the RECLIST. This is the third step. We refer to the this transition from values → tuple-ids → values as the *VTV cycle*. A query may involve several iterations of the VTV cycle.

The tuple-ids described in Section 2.2 carry information regarding the exact location of a value within a database, nested relation, or tuple. Given any two tuple-ids, it is possible to determine if they belong to the same tuple or the same subtuple without having to map them to their

corresponding values.

In ANDA, the CACHE is implemented using stacks as the only data structures. We choose stacks because they provide a simple data structure with minimal pointer overhead and require no garbage collection. Typically, the CACHE stores only tuple-ids on stacks, however, in some cases we also store values as described in Section 3.3.2. As tuple-ids are implemented as bit patterns of fixed size, the CACHE stacks are uniform in size and are very simple to build. Elements of the stack are sets of tuple-ids and the majority of the CACHE operations on the stack use its top two elements as operands. Several interesting queries can be processed by simply comparing tuple-ids.

2.6 Access Language Commands

The access language commands abstract the functions of the storage structures. This language is used to specify and optimize access plans. Access language commands can be divided into five basic groups:

1. VALTREE commands: retrieve, insert, delete.
2. RECLIST commands: retrieve, insert, delete.
3. CACHE commands: basic stack commands, set oriented operations, filter, transform and copy commands.
4. DATA-DICTIONARY commands: query and insert commands.
5. Decision and Control statements.

An access language program may access several stacks, each stack can be perceived as a temporary variable. This is analogous to the use of temporary storage locations by the code generation process in compilers. For a reference list of access language commands the reader is referred to Appendix A.

3 Query Processing in the ANDA Access Language

This section provides an overview on how to specify programs in the ANDA access language. To introduce the concept we present a simple example in Section 3.1. This example takes an SQL query and shows the corresponding access plan. The access plan and the access language commands are explained in detail. This example is used as a brief introduction to the philosophy of query processing strategy of ANDA. This section also explains the notion of a VTV cycle in nore detail and concludes with another example which highlights some more facets of query processing in ANDA.

3.1 A Simple Example

To simplify the introduction of our approach to query compilation, we show an example on the following flat relational scheme.

```
Student = (sid, sname, sdept, sage, ssex, sclass, sroomno, sdorm, smarried)
```

The following SQL query retrieves the names and student-ids of all female students in the Computer Science (CSCI) department.

```
SELECT sname, sid FROM Student
WHERE ssex = "F" and sdept = "CSCI"
```

This query would be compiled into the following ANDA program.

```
1.  cache_create_stack(TEMP1)
2.  vt_retrieve(TEMP1,Dsex,F,F,ssex,Student,relation)
3.  vt_retrieve(TEMP1,Ddept,CSCI,CSCI,sdept,Student,relation)
4.  cache_intersection(TEMP1,S*)
5.  cache_transform(TEMP1,S*a)
6.  cache_copy(TEMP1,TEMP1)
7.  cache_transform(TEMP1,S*b)
8.  cache_union(TEMP1,S**)
9.  cache_sort(TEMP1)
10. rl_retrieve(TEMP1)
```

Statement 1 initializes the stack named TEMP1.[2] Statement 2 performs the retrieval of tuple-ids corresponding to tuples representing information about female students. In that statement, Dsex represents the sex domain, ssex represents the attribute name, Student represents the relation name, and the last parameter indicates the *granularity* of each member stored in the stack. Generally speaking, the relation granularity gives us a way to partition the set of all tuple-ids of the relation being accessed into equivalence classes. The vt_retrieve function accepts a range of values as parameters of the search. For each value in the range of values specified as parameters of the vt_retrieve statement, there is a set of tuple-ids associated with this value representing all occurrences of that value in the database. Of course, a set is empty when there is no occurrence of a value in the database. Each set of tuple-ids forms a class that is stored as one element in the stack. In this example, the range of the search specifies a single value, namely, F. Hence, the stack may contain at most one nonempty element consisting of the set of tuple-ids of tuples referring to females. Other possible granularity values that can be used as a parameter of the vt_retrieve function are domain and tuple-id. Similarly, Statement 3 performs the retrieval of tuple-ids from

[2] A cache_create_stack function call is needed to initialize each stack used in an ANDA program. This step is similar to a variable declaration in imperative languages. Hereafter, we omit cache_create_stack function calls in our programs to simplify our discussion.

the Student relation corresponding to records from students in the Computer Science department. After executing statements 2 and 3, the value of the TEMP1 stack could contain two elements as follows: [3]

```
TEMP1 = {<S2c,S3c,S10c,S14c,S94c>,<S3e,S4e,S14e,S87e>}
```

The structure of the tuple-ids shown in this example requires some explanation. For each tuple-id shown in the above stack, the letter S indicates that the tuple-ids refer to values from the Student relation. The numeric part indicates the tuple number of the matching tuple. Tuples are arbitrarily numbered in ascending order but once a number has been assigned its number stays valid until all the elements of that tuple are deleted from the database. The letters e and c indicate that the stack contains sets of tuple-ids corresponding to values from ssex and sdept, respectively.

We can see from this that tuple-ids with subscripts 3 and 14 correspond to tuples from the Student relation that satisfy the whole predicate in the WHERE clause. Statement 4 computes the intersection of the two topmost elements of the stack. That second parameter of that function, namely S*, is referred to as the *transform format.* Here, the transform format indicates that the result of the intersection will contain all tuple-ids matching on their first two components. Thus, in this example we are interested in S3c and S3e as they both belong to the same tuple S3. The system tries to retain as much information as possible and hence the large or the two values S3c and S3e, in this case S3e is retained.

The state of the stack after this statement is performed will contain a single element as follows:

```
TEMP1 = {<S3e,S14e>}
```

Now that the tuple-ids of tuples that satisfy the predicate in the WHERE clause have been found, we need to transform such tuple-ids to a format that will permit the retrieval of values from the sname and sid attributes. This is performed by statements 5–9. Statement 5 transforms the tuple-ids in the topmost element of the stack to:

```
TEMP1 = {<S3a,S14a>}
```

Statement 6 copies the topmost element of the stack to the same stack.

```
TEMP1 = {<S3a,S14a>,<S3a,S14a>}
```

Statement 7 transforms the topmost element of the stack to:

```
TEMP1 = {<S3a,S14a>,<S3b,S14b>}
```

Statement 8 computes the union of the two topmost elements of the stack resulting in:

```
TEMP1 = {<S3a,S14a,S3b,S14b>}
```

[3] In the examples that show the state of the stack we present tuple-ids in a flat format rather than in a format that shows superscripts and subscripts (as shown for example in Table 2). This is the way ANDA prints the contents of stacks.

Again the second argument of the union specifies the window which must be used to compare the two stack elements. Finally, Statement 9 sorts the members in the top element of the stack to bring together the tuple-ids for values for the `sname` and `sid` attributes of each matching tuple. The resulting stack is:

```
TEMP1 = {<S3a,S3b,S14a,S14b>}
```

Statement 10 retrieves the values stored in the RECLIST corresponding to each tuple-id in the topmost element of the stack. We can see, from the above description, how a simple SQL query can be transformed into an ANDA program containing one VTV cycle. Statements 2 and 3 provide the mapping from the value world to the tuple-id world. Statement 4–9 manipulate tuple-ids in main-memory-based tuple-id world (the CACHE), and Statement 10 provides the mapping from the tuple-id world to the value world.

3.2 Value World and Tuple-id World

In the relational model the user specifies the query in terms of relations and expects relations as results. Each relational algebra operation maintains this closure property. Conceptually, the same is true with nested relations and nested algebra. Thus, in the user's view, each query is expressed in terms of nested relations or values and the expected result is nested relations or values. We refer to this view of the user as the *value world.*

While the user may view the query in terms of values, it is more efficient to do query processing in terms of tuple-ids as tuple-ids are compact and can be manipulated efficiently in main-memory. We refer to this phase of query processing as the *tuple-id world.*

Thus, a typical query in ANDA consists of a sequence of "*value→tuple-id→value*" (VTV) cycles. The user expresses the query in terms of nested relations (values), these values are used to extract tuple-ids for those tuples that participate in the query. These tuple-ids are manipulated extensively in the CACHE. Finally, the results are presented by materializing tuple-ids that correspond to the result.

Unfortunately, it is not possible to evaluate any arbitrary query by means of a single VTV cycle. A query might involve a sequence of VTV cycles. To connect adjacent VTV cycles efficiently, the system provides a set of "glue" operations which are discussed in Section 3.3

The `vt_retrieve` command of the VALTREE described in Section 2.3 is used to transform from user's "value-world" to the "tuple-id world." These tuple-ids are manipulated extensively in the main-memory-based CACHE, until we obtain tuple-ids that correspond to the result of the query. Finally, to return to the value world, the `rl_retrieve` command is used to map tuple-ids to values from the RECLIST as described in Section 2.4.

3.3 Gluing Adjacent VTV Cycles

When manipulating tuple-ids in the CACHE, we lose the correspondence between tuple-ids and values. Queries where it is possible to defer materializing tuple-ids until the final result can be processed

by one VTV cycle. We need multiple VTV cycles when it is not possible for query processing to proceed with tuple-ids alone, and one needs to know the values associated with the tuple-ids in the stack. This situation occurs for example when translating a SQL statement containing nested subqueries (see Section 5.4).

Conceptually, values generated at the end of the VTV cycle can be inserted into the database as a temporary relation. However, in ANDA, this involves inserting values in the RECLIST, generating new tuple-ids and inserting values and tuple-ids in the VALTREE. In most cases, the temporary relation is used as an intermediate step, and values from this relation are used in the next VTV cycle. Clearly, creating a new relation and updating all the data-structures is an expensive process and should be avoided whenever possible. Thus, to make query processing in this model effective, it is important to "efficiently connect" two adjacent VTV cycles.

Let $\cdots T_i \rightarrow V_i \rightarrow V_{i+1} \rightarrow T_{i+1} \cdots$ be two adjacent VTV cycles. We are interested in providing a smooth interface between the set of values V_i and V_{i+1}. In general, since the only function to convert values to tuple-ids discussed so far is the `vt_retrieve` function, a transition $\cdots T_i \rightarrow V_i \rightarrow V_{i+1} \rightarrow T_{i+1} \cdots$ between two adjacent cycles would require building a VALTREE for values V_i so that the transformation $V_{i+1} \rightarrow T_{i+1}$ could be performed.

A more efficient way to interface two VTV cycles is to design *gluing* functions that allow an access plan to carry out the transition between cycles without having to create an elaborate data structures. We support two strategies to facilitate the connection of two adjacent VTV cycles.

1. Use a temporary variable to store some "value" during transition. This is done by the `assign` and `assign_value` functions.

2. Use a *value stack*. The value stack is much like the tuple-id stack but instead of storing only tuple-ids, it stores <tuple-id, value> pairs. The access language provides functions to translate to and from the tuple-id stack and the value-stack. Functions to filter and sort are also provided on the value stack. The value stack also provides the access language with added flexibility to specify alternative query plans.

The next subsections describe variables and the value stack in more detail.

3.3.1 Variables for Values

As discussed in the previous section, in ANDA's query processing strategy, we lose the correspondence between tuple-ids and their values after some transformations on tuple-ids in the tuple-id stack. When the access plan requires the use of a value for a short while, typically for use in the next `vt_retrieve` statement, the `assign_value` statement, which has stack-name and a variable-name as arguments, assigns the value corresponding to first tuple-id in the top element of the stack to the variable-name. This variable is dereferenced by placing a $ sign in front of the variable-name.

For example, if we are interested in finding the list of courses taken by "John", we must know the student-id so that it can be used in the `vt_retrieve` operation. Thus this query involves two

VTV cycles, the first cycle for getting the Student-id for "John" and the second VTV cycle for getting all the grades. These two cycles are connected by variable ID as shown in the following access plan.

```
# VTV Cycle 1
1. vt_retrieve(S1,Dname,John,John,Sname,Student,s)
2. cache_transform(S1,S*a)
3. assign_value(S1,ID)
# VTV Cycle 2
4. vt-retrieve(S2,Did,$ID,$ID,sno,Course,s)
5. cache_transform(S1,C*a)
6. rl-retrieve(S1)
```

3.3.2 The Value Stack

The value stack is very similar to the tuple-id stack and serves three basic purposes:

- Variables are convenient if there are a few temporary values that need to be stored. If we are interested in a set of values, the value stack can be used as glue.
- The value stack stores both tuple-ids and values and the access language provides functions to access and filter the value stack much the same way as the tuple-id stack. These features allow us to write alternative access plans using the value-stack. A discussion of how access plans can use the value stack for efficiency are discussed in Section 7.
- When performing aggregate operations like average, max, min etc., we are interested in values. These functions are defined on the value stack. Examples of aggregate functions are discussed in Section 5.5.

The access language provides functions for converting from a tuple-id stack to the value-stack and vice versa. Conversion from a tuple-id stack to the value stack involves accesses to the RECLIST and, therefore, is an expensive operation. However, this operation is considerably cheaper than creating temporary tables and rebuilding all the indices.

3.4 Another Query Example

This example shows a Nested SQL query which, when compiled into an ANDA program, is formed from two VTV cycles which are glued together by variables. The example is based on the Course and Student schemes introduced in Section 2.1. Suppose we want to find all the courses taken at the same time by Jones and Smith. This query can be formulated in Nested SQL as follows:

```
SELECT cno, cname, Section.term FROM Courses C
WHERE C.Section.Enrollment.sno
```

```
CONTAINS (SELECT sno FROM Student
          WHERE sname = "Jones" OR sname = "Smith")
```

This Nested SQL query will be compiled into the following ANDA program:

```
# VTV Cycle 1
# value-world:
  1.  vt_retrieve(TEMP1,Name,Jones,Jones,sname,Student,relation)
  2.  vt_retrieve(TEMP1,Name,Smith,Smith,sname,Student,relation)
#tuple-id-world:
  3.  cache_union(TEMP1,S*)
  4.  cache_transform(TEMP1,S*a)
  5.  cache_explode(TEMP1)
# Glue between VTV Cycle1 and VTV Cycle 2
  6.  assign(TEMP2,STUDENT1)
  7.  cache_pop(TEMP2)
  8.  assign(TEMP2,STUDENT2)
# VTV Cycle 2
  9.  vt_retrieve(TEMP3,Sid,$STUDENT1,$STUDENT1,sid,Course,relation)
 10.  vt_retrieve(TEMP3,Sid,$STUDENT1,$STUDENT2,sid,Course,relation)
# tuple-id-world:
 11.  cache_intersection(TEMP3,C*f*)
 11'. cache_intersection(TEMP3,C*)
 12.  cache_copy(TEMP3,RESULT)
 13.  cache_transform(RESULT,C*a)
 14.  cache_copy(TEMP3,RESULT)
 15.  cache_transform(RESULT,C*c)
 16.  cache_copy(TEMP3,RESULT)
 17.  cache_transform(RESULT,C*f*b)
 18.  cache_union-all(RESULT)
 19.  cache_sort(RESULT)
# value-world:
 20.  rl_retrieve(RESULT)
```

As usual, the names TEMP1, TEMP2, TEMP3, and RESULT refer to named stacks. This is similar to the temporary storage locations generated in the code generation stage of a compiler. This program shows several key ideas used in our system.

- As we mentioned before, an ANDA access program may contain several VTV cycles. This example contains two.
- Statement 1, 2, 6, 8, 9, 10 and 20 involve functions that retrieve values from secondary storage (i.e., VALTREE and RECLIST). Such statements should be minimized when possible.

- Blocks of statements 3–5, and 11–19 represent cache operations performed in main memory. Two observations can be made at this point: (1) it is possible to eliminate some of the cache operations. For instance, by performing the transformation in statement 4 after the `vt_retrieve` it is possible to avoid the `cache_union` and `explode` operations of statements 3 and 6; and (2) it is possible to change the semantics of the query to obtain all courses taken by Smith and Jones not necessarily at the same time by using statement 11' rather than 11 which changes the "window" of the intersection operation.

- Statements 6–8 represent a way of gluing two adjacent VTV cycles. Statement 6 saves the value of the `sid` corresponding to one of the students in variable $STUDENT1 which is invoked in statement 9.

All the above ideas are explained in more detail in the rest of the paper.

4 Query Processing

Query processing in ANDA consists of five major stages: parsing, algebraic rule-based optimization, rule-based translation into a program in the target access language, code optimization using compiler-like techniques, plan storage and execution. Figure 4 illustrates the query processing stages in our system.

During the parsing stage, a Nested SQL[4] statement is translated into an internal graph representation. The graph representation is an annotated graph resulting from the parse tree similar to the query graph model in Starburst [22, 23]. As part of the graph annotation process, references to attributes and nested relations in the high-level query are resolved to their corresponding internal names by accessing the data dictionary. Information regarding cardinalities of relations and distribution of values for the attributes and relations mentioned in the query is also extracted from the data dictionary and included in the query graph.

The query rewrite component takes the query graph resulting from parsing and transforms it into a graph leading to a better plan using the algebraic rules. The rule-base plan generator takes the query graph that results from the query rewrite component and, using the access language rule-base, generates a program in the ANDA access language that executes the query. Data-flow analysis similar to the one performed by optimizing compilers is performed on the plan produced by the plan generator [2]. The resulting plan is then stored for future execution.

4.1 Query execution

The ANDA access language processor (ANDA's run-time system) is illustrated in Figure 5 and consists of three basic components – the VALTREE, the CACHE and the RECLIST. The VALTREE, stores an efficient mapping from values of the database to a set of tuple-ids that correspond to all occurrences

[4] We have developed a Nested SQL query language interface for nested relations similar to SQL/NF [28, 30] and Laurel [20] as part of this project.

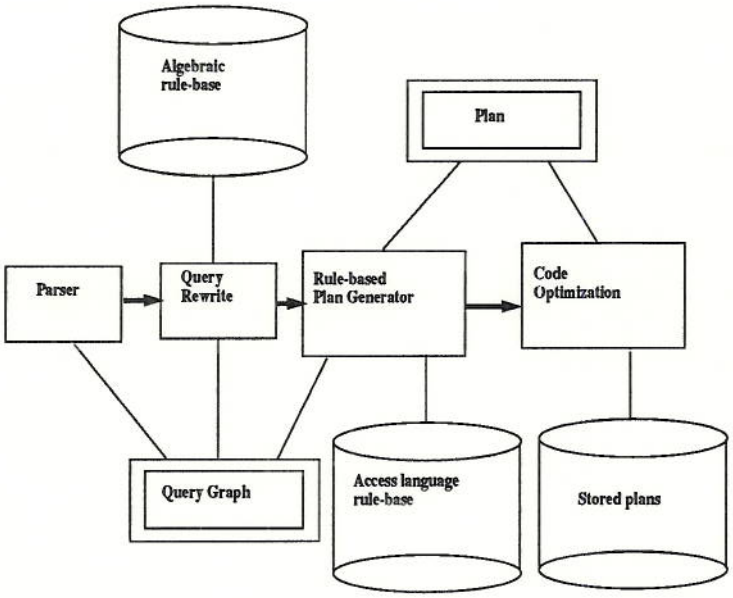

Figure 4: Stages of query processing in ANDA.

of the values in the entire database. The RECLIST provides a mapping from tuple-ids to values. Retrievals on the VALTREE convert the query from the value-world to the tuple-id world and the RECLIST converts from the tuple-id world to the value world. The CACHE which is the main memory component of the database manipulates tuple-ids in the tuple-id world.

5 Rule-based Plan Generation

This section focuses on the plan generation aspect of the query processing architecture described in the previous section. We start by showing in Subsection 5.1 how to generate a plan for a SQL query involving select and project on flat relations. We then extend it in Subsection 5.2 to queries on nested relations. Subsection 5.3 describes two ways of generating plans for queries involving joins. Subsections 5.4 describes plan generation for nested SQL subqueries, and finally, Subsection 5.5 shows plan generation for queries involving aggregate functions.

5.1 Rules for Select-Project Queries on Flat Relations

For simplicity of presentation we start our description of plan generation by showing how plans for queries involving select and project on flat relations are generated. A SQL query involving select

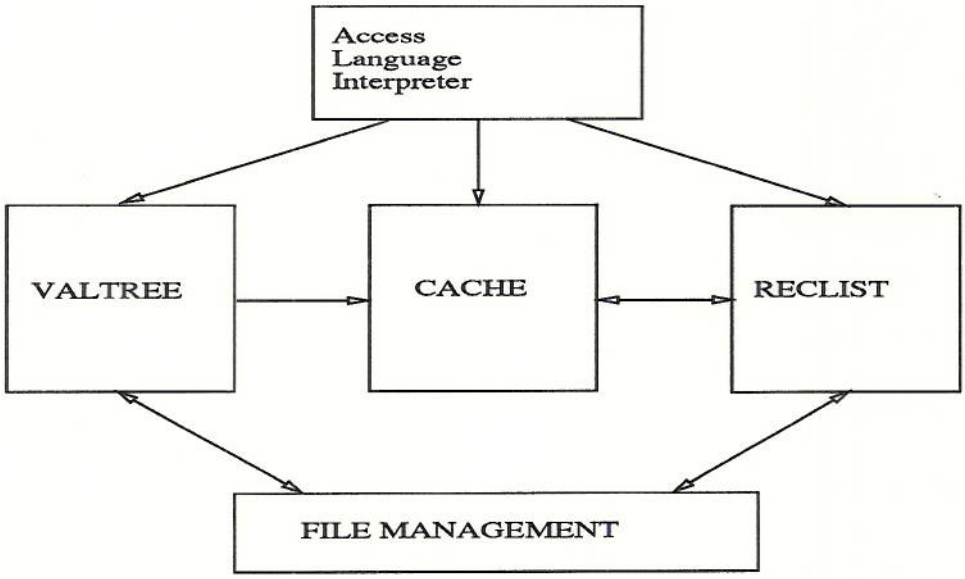

Figure 5: Access Language Processor.

and project on a single relation can be translated into an ANDA program that includes the three basic stages of a VTV cycle: (1) resolving the condition in the WHERE clause in terms of tuple-ids, (2) transforming the set of tuple-ids to the appropriate set of tuple-ids required to present the result of the query as indicated in the list of attributes, and (3) assembling the tuples of the result by retrieving values from the database that correspond to the tuple-ids obtained in the previous stage. The first example given in Section 3.1 on page 10 is a select-project query whose ANDA program contains the above three stages. Statements 2 and 3 correspond to first stage, statements 4–9 correspond to the second stage and statement 10 corresponds to the third stage.

More generally, suppose that we want to compute the following SQL statement on a single relation whose clause involves a conjunctive Boolean expression:

```
SELECT A1, A2, ..., An
FROM R
WHERE X1 = C1 and X2 = C2 and...and Xm=Cm
```

The corresponding program in the ANDA access language is shown in Figure 6. operations for this query are: In that program, RESULT is the stack containing the resulting tuple-ids. The operation cache_intersection_all converts an m-element stack into a one-element stack by performing the intersection of all the m elements of the stack. It is straightforward to generate the above plan by traversing the query graph resulting from the parsing stage of the query compilation process in postorder.

```
            vt_retrieve(RESULT,Dom_X1,C1,C1,X1,R,relation)
m times     ...
            vt_retrieve(RESULT,Dom_Xm,Cm,Cm,Xm,R,relation)
            cache_intersection_all(RESULT,R*)
            cache_transform(RESULT,R*A1)
            cache_copy(RESULT,RESULT)
            cache_transform(RESULT,R*A2)
n times     ...
            cache_transform(RESULT,R*An)
            cache_union_all(RESULT,R*)
            cache_sort(RESULT)
            rl_retrieve(RESULT)
```

Figure 6: ANDA access language template for conjunctive Boolean expressions.

5.2 Rules for Select-Project Queries on Nested Relations

The only complication in the nested case comes from the fact that queries can have path expressions as names for attributes in both the attribute list and the WHERE clause of a Nested SQL statement. Consider the following query on the Course relation to find the prerequisites of courses offered by the computer science department in the Spring of 1989.

```
SELECT cname, cno, Prerequisite FROM Course
WHERE dept = "CSCI" AND Course.Section.term = "S89"
```

This query can be translated into the following ANDA program:

```
1.  vt_retrieve(RESULT,Dterm,S89,S89,term,Course,relation)
2.  vt_retrieve(RESULT,Ddept,CSCI,CSCI,dept,Course,relation)
3.  cache_transform(RESULT,C*f)
4.  generate_tid(RESULT,C*f?b)
5.  cache_intersection(RESULT,C****)
6.  cache_transform(RESULT,C*a)
7.  cache_copy(RESULT,RESULT)
8.  cache_transform(RESULT,C*b)
9.  cache_copy(RESULT,RESULT)
10. cache_transform(RESULT,C*d)
11. generate_tid(RESULT,C*d?a)
12. cache_union(RESULT)
13. cache_union(RESULT)
14. cache_sort(RESULT)
15. rl_retrieve(RESULT)
```

This ANDA program introduces the generate_tid function. As we know, nested relational schemes are hierarchical and the structured tuple-ids reflect this hierarchy. Figure 7 shows the Course nested relation scheme an corresponding naming hierarchy used to name the superscripts of tuple-ids. For instance, the tuple-id for the Spring term is $C_1{}^f{}_2{}^b$. This is the general form of the the tuple-ids obtained after executing Step 1 of the above ANDA program.

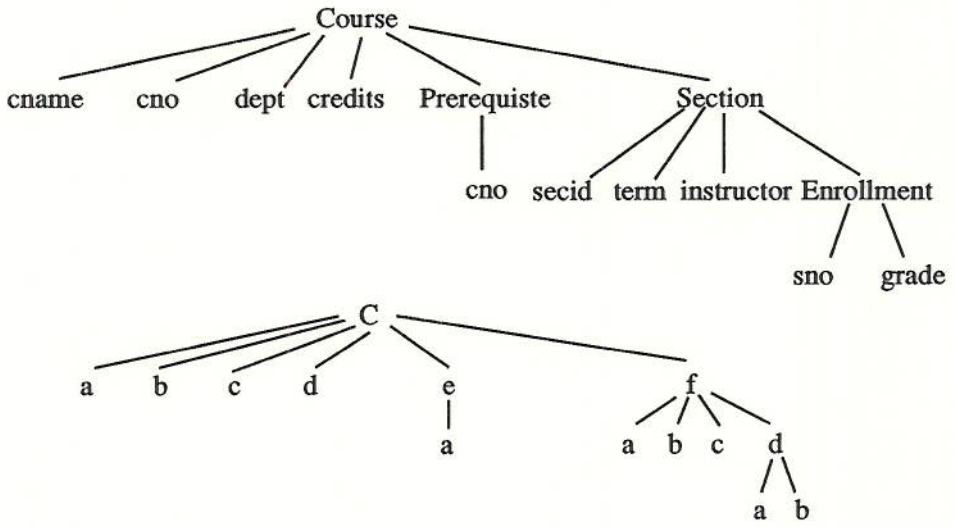

Figure 7: The Course nested relational scheme and its tid naming hierarchy.

Step 2 of the above program will yield the set of tuple-ids $\{C_1{}^c, C_2{}^c\}$ whose format is incompatible with the format $\{C_1{}^f{}_2{}^b\}$ in the sense that their intersection will yield the empty set. The reason for the incompatibility is that we are dealing with tuple-ids representing values at different nesting levels in the relation scheme. Furthermore, after manipulating sets of tuple-ids in the stack, an element may contain tuple-ids referring to an internal node in the naming hierarchy (e.g., $\{C_2{}^f, C_{11}{}^f, C_{23}{}^f\}$ refers to Section). These tuple-ids name whole subrelations rather than particular values. Often, it is necessary to know the exact tuple-ids that correspond to a branch of the subtree rooted at the particular internal node of the scheme hierarchy. In the above query, given that we have in the stack the tuple-id C_2^f, we might want to know all the tuple-ids corresponding to the values in the Section.term attribute of that tuple. In other words, we need the system to "generate the tuple-ids" corresponding to the branch Section.term of the subtree rooted at C_2^f. This is achieved through the operation generate_tid($S1, C_2{}^f{}_?{}^b$), where S1 is the name of the stack, and $C_2{}^f{}_?{}^b$ is the *generation pattern.* The generation pattern matches all tuple-ids that start with $C_2{}^f$ in the topmost element of stack S1 and generates the tuple-ids for the branch Section.term. The ? character indicates the elements of the tuple-id that must be generated. Another possible generation pattern might be $C_*{}^f{}_?{}^d{}_?{}^a$ which generates all tuple-ids for the branch of the schema Section.Enrollment.sno; the asterisk is a wild-card pattern. When generating tuple-ids one needs to know which tuple-ids exist. This information is available in the RECLIST and

hence `generate_tid` involves access to secondary storage.

In general, to compute a Nested SQL query based on a conjunctive condition involving attribute names at different levels in the scheme hierarchy, it may be necessary to transform tuple-ids to a common format. In the ANDA program described above, since an intersection of the two topmost elements of the RESULT stack needs to be computed by statement 5, we need to make sure that tuple-ids in both stack elements have compatible formats. That is, we need to transform the tuple-ids obtained from the `vt_retrieve` operation in statement 2 which have the format $C_*{}^c$ into the format $C_*{}^f{}_?{}^b$. This transformation is achieved by statements 3–4. Plan generation for queries whose WHERE clause involves more general Boolean expressions is also possible in our system.

5.3 Rules for Joins

In the nested relational model the join operation may be given several meanings. Two of the most common ones are the "equality-join" and the "intersection-join." See Roth and Kirkpatrick [29].

In this system, there are two approaches to compute a join: one is *iterative* and the other uses the VALTREE as a *join-index*. Both methods may be easily adapted to provide any of the different join semantics associated with joins in the nested relational model.

In general, an ANDA program to compute a join has the following three VTV-cycle stages: (1) for each value in the domain of the joining attributes that appears in the database, retrieve the set of tuple-ids of the joining relations sharing the value; (2) manipulate each set of tuple-ids depending on the join semantics desired (e.g., in the relational model perform the cartesian product of the tuple-ids in the set); and (3) retrieve the values from the database. The next two sections describe the two join methods currently supported in ANDA.

5.3.1 The VALTREE as a Join Index

This section describes a method to compute joins using features already built into the VALTREE. The VALTREE helps to efficiently compute joins since it is equivalent to a join index. Recall that the VALTREE is defined over all the values and their attributes appearing in the database. Given a particular value stored in the VALTREE for some domain, there may exist several tuple-ids from different relations associated with it that correspond to the occurrences of that value in the relations. Hence, all tuple-ids from different relations associated with the same value are clustered together in the VALTREE. This makes possible to compute a join using a method similar to the join index method [31]. For simplicity, we illustrate the join method on two flat relations. Suppose that schemes of the relations are

```
Student = (sid, sname, class, school, age, sex, dorm, room)
Roster  = (cno, sno, grade)
```

with the obvious meanings, and suppose that we want to print rosters that also include the name of the student and the course number. Therefore, we need to join the relations Student and Roster on the attributes sid and sno. The query formulated in SQL would be

```
SELECT R.cno, S.name FROM Student S, Roster R
WHERE S.sid = R.sno;
```

Following the three-stage approach to computing a join described above, it is necessary to first group tuple-ids according to the joining values. This is achieved using the following variation of a `vt_retrieve` operation

```
vt_retrieve(TEMP1,Dsno,low_value,high_value,[(sid,Student),(sno,Roster)],relation).
```

This operation traverses the `VALTREE` starting from the lowest value (`low_value`) to the highest value (`high_value`) in the domain `Dsno`. For each value in the `VALTREE`, if its leaf node points to a set of tuple-ids referring to the attributes and relations mentioned as parameters of the `vt_retrieve` operation, then the set of tuple-ids is pushed onto the stack. A possible snapshot of the stack resulting from this operation could be

```
TEMP1 = {<S2a>,<R24b,R37b,R98b>, <S3a>,<R15b,R29b>}
```

assuming that there are only two students taking 3 and 2 courses, respectively.

The second stage in the approach is concerned with presenting the final results of the query. All the tuple-ids on the stack `TEMP1` correspond to `sid` values, however, we are interested in values corresponding to the attributes student name and course number. So we need to transform the tuple-ids in the stack to a format that corresponds with the attributes of the result of the query. Observe that the `RESULT` stack would be in the nested relational form.

```
RESULT = { <S3b,{R15a,R29a,R24a}>, <S2b,{R37a,R98a}> }
```

This requires transforming the tuple-ids in the `RESULT` stack to the appropriate form and then retrieving the the values from the database. This is accomplished with the following `ANDA` code.

```
while_not_empty(TEMP1)
  cache_transform(TEMP1,S*b)
  rl_retrieve(TEMP1)
  cache_pop(TEMP1)
  cache_transform(TEMP1,R*a)
  rl_retrieve(TEMP1)
  cache_pop(TEMP1)
od
```

Computing joins of nested relations on deeply nested attributes involves the same three stages as in the flat relation case, however, the transformations in the `ANDA` program would need to be modified in its second stage to reflect the right join semantics.

To be able to perform a join of two nested relations on attributes deeply nested within structures Korth [19] defines a new join operator for nested relations called the unnested join operator. Here we show that with the access language we are able to simply and succinctly define different kinds of joins without the need for unnest.

5.3.2 Iterative Join

Consider again the same SQL example of the previous section. The iterative method differs from the join-index method in the first stage. That is, just in the way the tuple-ids of the joining relations sharing a value are collected. In the join-index method this is accomplished using the more general form of the `vt_retrieve` operation. Here, we do it by iterating on the `VALTREE` as follows

```
1.  vt_retrieve(TEMP1,Dsid,low_value,high_value,sid,Student,relation)
2.  while_not_empty(TEMP1)
3.    assign_value(TEMP1,VAR1)
4.    vt_retrieve(TEMP2,Dsid,$VAR1,$VAR1,sno,Roster,relation)
5.    if ( TEMP2 != EMPTY )
6.    then
7.      cache_copy(TEMP2,RESULT)
8.      cache_copy(TEMP1,RESULT)
9.    else
10.     cache_pop(TEMP1)
11.   endif
12. od
```

The above program contains several VTV cycles, one for each iteration of the outer loop. In statement 3 we find the `assign_value` operation which transforms the tuple-id in the top element of the stack into its corresponding value and assigns it to the variable `VAR1`. This is one of the operations used to glue adjacent VTV cycles. The example also introduces some of the loop constructs in `ANDA`: `label`, `if_not_empty`, and `while_not_empty` with the obvious meaning.

At first glance it may not appear advantageous to try to perform the above sequence of `ANDA` operations just to collect the joining tuple-ids when we have a more general `vt_retrieve` operation that does the same job. However, we will see that when the join is combined with a complex select operation, it is possible to restrict the set of joining values and avoid having to traverse the whole `VALTREE` from a `low_value` to a `high_value` (which may involve many secondary storage access in large domains). In that case, it is better to build a small set of tuple-ids corresponding to the restriction and then to iterate on the elements of this set. Also, the idea of iterative joins is applied when transforming Nested SQL subqueries (e.g., those involving `IN`, `EXISTS`, `ANY`, `CONTAINS`) into `ANDA` programs. Transformation of Nested SQL subqueries into `ANDA` programs is discussed in the next section.

Consider the following Nested SQL query to find the names and grades of students in the database class.

```
SELECT S.sname, C.Section.Enrollment.grade FROM Student S, Course C
WHERE C.cname = "Database" AND C.Section.Enrollment.sno = S.sid
```

This SQL statement can be compiled into the following `ANDA` program.

```
 1.  vt_retrieve(TEMP1,Dcname,Database,Database,cname,Course,relation)
 2.  generate_tid(TEMP1,C*f?d?a)
 3.  cache_explode(TEMP1)
 4.  while_not_empty( TEMP1 )
 5.    assign_value(TEMP1,VAR1)
 6.    vt_retrieve(TEMP2,Dsid,$VAR1,$VAR1,sid,Student,relation)
 7.    cache_transform(TEMP2,S*b)
 8.    rl_retrieve(TEMP2)
 9.    cache_pop(TEMP2)
10.    cache_transform(TEMP1,C*f*d*b)
11.    rl_retrieve(TEMP1)
12.    cache_pop(TEMP1)
13.  od
```

We have shown the use of iteration statements in the iterative join. In section 5.4 we show that iteration statements are also useful for the support of nested subqueries.

5.4 Rules for Subqueries

In the relational model, a SQL *subquery* is a SELECT expression nested within another SELECT expression connected through the keywords IN, EXISTS, NOT IN, NOT EXISTS. Nested SQL also provides the CONTAINS keyword. Subqueries typically involve set-valued predicates.

Consider the query: Find all courses having the data structures course as prerequisite.

```
SELECT cname FROM Course
WHERE Prerequisite.cno
CONTAINS (SELECT cno FROM Course
          WHERE cname = "data structures");
```

Notice that this query can also be formulated as

```
SELECT cname FROM Course
WHERE (SELECT cno FROM Course
       WHERE cname = "data structures")
IN Prerequisite.cno;
```

The corresponding ANDA program is:

```
1.  vt_retrieve(TEMP1,Dcname,data structures,data structures,cname,Course,relation)
2.  cache_transform(TEMP1,C*b)
3.  assign_value(TEMP1,VAR1)
4.  vt_retrieve(RESULT,Dcno,$VAR1,$VAR1,Prerequisite.cno,Course,relation)
5.  cache_transform(RESULT,C*a)
6.  rl_retrieve(RESULT)
```

First of all, these query involves a SQL query nested within another SQL query. This implies that the plan generated will contain two VTV cycles. Plans for each individual SQL query are generated using the rules for select-project queries and if necessary, using the rules for joins. The important aspect becomes how to glue the two VTV cycles in such a way that it leads to an efficient execution plan. For this case we make use of the *value variables* available in the CACHE through the assign_value function. In some cases, depending on the cardinality estimations, it might be advantageous to use a value stack instead.

Statements 1–3 correspond to the computation of the inner SELECT and statements 4–6 correspond to the outer SELECT. Statements 1–3 form the first VTV cycle whereas statements 4–6 form the second VTV cycle. The first statement retrieves the tuple-ids for the course named data structures. It is expected that there will be only one course with that name. The result of this query will push one element on the TEMP1 stack of the form <C32a>. Since we want to know the course number of the data structures course we need to transform the stack to the form <C32b>. This is done by statement 2. Once we have the tuple-id of the course number we need to know the actual value of he course number. This is obtained by using the assign_value operation which assigns the numeric value of the top element of the stack to the variable VAR1 Statement 4 retrieves all tuple-ids of values in the database that match the value in VAR1. The last two statements transform the tuple-ids found to the format C^a_* and retrieve the actual course name. For additional examples of plan generation for nested subqueries the reader is referred to Appendix B.

5.5 Rules for Aggregate Functions

Generation of plans for queries involving aggregate functions can be done in a straightforward way. Consider again the example introduced in section 3.1 which retrieves the names and student-ids of all female students in the Computer Science (CSCI) department.

```
SELECT sname, sid FROM Student
WHERE ssex = "F" and sdept = "CSCI"
```

For convenience, we repeat the corresponding ANDA program here.

```
1.  cache_create_stack(TEMP1)
2.  vt_retrieve(TEMP1,Dsex,F,F,ssex,Student,relation)
3.  vt_retrieve(TEMP1,Ddept,CSCI,CSCI,sdept,Student,relation)
4.  cache_intersection(TEMP1,S*)
5.  cache_transform(TEMP1,S*a)
6.  cache_copy(TEMP1,TEMP1)
7.  cache_transform(TEMP1,S*b)
8.  cache_union(TEMP1,S**)
9.  cache_sort(TEMP1)
10. rl_retrieve(TEMP1)
```

Now suppose that we modify the query slightly by asking instead for the number of female students in the Computer Science Department. This query can be formulated using aggregate functions as follows:

```
SELECT COUNT(S.sid) FROM Student S
WHERE ssex = "F" and sdept = "CSCI"
```

A plan to execute this query would be:

```
1. cache_create_stack(TEMP1)
2. vt_retrieve(TEMP1,Dsex,F,F,ssex,Student,relation)
3. vt_retrieve(TEMP1,Ddept,CSCI,CSCI,sdept,Student,relation)
4. cache_intersection(TEMP1,S*)
5. cache_top_card(TEMP1)
```

Notice that the two compiled programs are very similar. They differ only in two aspects. One is that it is no longer necessary to perform the tuple-id transformations. The other is that instead of having a `rl_retrieve` command as a last statement we have the `cache_top_card` command which provides the number of tuple-ids in the top element of the stack.

As an additional example, consider the query: Find the average mark obtained by students enrolled in section 1721.

```
SELECT AVERAGE(Section.Enrollment.grade) FROM Course
WHERE Section.secid = 1721;
```

The corresponding ANDA program would be

```
1. vt_retrieve(TEMP1,Dsecid,1721,1721,secid,Course,relation)
2. cache_copy(TEMP1,TEMP1)
3. generate_tid(TEMP1,C*f*d?a)
4. tuple_id_to_value(TEMP1,VAL1)
5. value_avg(VAL1)
```

Similarly, computing the maximum, minimum, or sum of the grades can be obtained by replacing statement 5 in the above program by the CACHE functions `value_max`, `value_min`, or `value_sum`, respectively.

6 Compiler-like Optimizations

So far we have shown how to generate ANDA programs from Nested SQL statements. The programs shown are not always the most efficient ANDA program that could be generated to execute the corresponding query. Sometimes, it is possible to improve the execution of an ANDA program by performing techniques similar to code optimization in compilers. Consider the following SQL query to find all 24 or 25 year old female students.

```
SELECT S.sname FROM Student S
WHERE (sex="F" and age=24) OR
      (sex="F" and age=25);
```

and corresponding ANDA program

```
1.  vt_retrieve(TEMP1,Dsex,F,F,sex,Student,relation)
2.  vt_retrieve(TEMP1,Dage,24,24,age,Student,relation)
3.  cache_intersection(TEMP1,S*)
4.  vt_retrieve(TEMP1,Dsex,F,F,sex,Student,relation)
5.  vt_retrieve(TEMP1,Dage,25,25,age,Student,relation)
6.  cache_intersection(TEMP1,S*)
7.  cache_union(TEMP1,***)
8.  cache_transform(TEMP1,S*b)
9.  cache_sort(TEMP1)
10. rl_retrieve(TEMP1)
```

Just like in previous database query optimization research, one of the main optimization objectives in our system is to reduce the number of secondary storage accesses. In this case, the only operations involving disk accesses are the file structure operations `vt_retrieve`, and `rl_retrieve`, and the CACHE operation `tuple-id_to_value`.

In the previous example, we can see that statements 1 and 4 are identical `vt_retrieve` operations. It is possible to eliminate one of these operations by storing the tuple-ids resulting from the first `vt_retrieve` in a different stack. This optimization produces the following ANDA program

```
1.  vt_retrieve(TEMP1,Dsex,F,F,sex,Student,relation)
2.  cache_copy(TEMP1,TEMP2)
3.  vt_retrieve(TEMP1,Dage,24,24,age,Student,relation)
4.  cache_intersection(TEMP1,S*)
5.  cache_copy(TEMP2,TEMP1)
6.  vt_retrieve(TEMP1,Dage,25,25,age,Student,relation)
7.  cache_intersection(TEMP1,S*)
8.  cache_union(TEMP1,***)
9.  cache_transform(TEMP1,S*b)
10. cache_sort(TEMP1)
11. rl_retrieve(TEMP1)
```

which (although longer) is more efficient since it avoids unnecessary disk accesses to retrieve tuple-ids for female students (see statement 4 in the first program). This example serves to illustrate an optimizing transformation on the generated program that eliminates *redundant I/O operations* by using extra main-memory storage.

Also, in the second program we can see that statements 3 and 6 are VALTREE accesses to retrieve tuple-ids referring to the same domain (`Dage`) and attribute (`age`). It is possible to combine these

two operations to retrieve all tuple-ids for students whose ages are in the range from 24 to 25 using a single VALTREE access. This is shown in the next program:

```
1. vt_retrieve(TEMP1,Dsex,F,F,sex,Student,relation)
2. vt_retrieve(TEMP1,Dage,24,25,age,Student,relation)
3. cache_intersection(TEMP1,S*)
4. cache_union(TEMP1,***)
5. cache_transform(TEMP1,S*b)
6. cache_sort(TEMP1)
7. rl_retrieve(TEMP1)
```

The vt_retrieve operation of Statement 2 retrieves all tuple-ids of students who are 24 or 25 years old. This transformation shows how by knowing *target language idioms* it is possible to apply powerful optimizing transformations. Here, we have been able to eliminate: a VALTREE access, the need to store temporarily the result of Statement 1, and one intersection and one union operations.[5] This example clearly shows that there is potential for applying optimization techniques similar to the ones widely applied in optimizing compilers.

We now give a more detailed description of the kinds of optimizations that can be performed on ANDA programs. Most of the material in the rest of this section is based from Chapter 10 of the book by Aho, Sethi, and Ullman [2]. In the same way as in code generation for compilers, an ANDA program can be partitioned into *basic blocks*. A basic block is a sequence of consecutive statements in which flow of control enters at the first statement in the block and leaves at the last statement without halt or possibility of branching except at the end of the block. An algorithm to partition an ANDA program into blocks can be found in Aho, Sethi, and Ullman (*Algorithm* 9.1). For example, the ANDA program shown in Section 5.3.2 can be partitioned into 4 basic blocks. Statement 1 forms the first block, Statements 3–5 form the second block, Statements 6–9 form the third block, and Statements 10–13 the fourth block.

Transformations can be applied to basic blocks without changing the semantics of the block. Transformations can be applied *locally* within a basic block or *globally* among a group of blocks. There are two important classes of local transformations that can be applied to basic blocks, namely, *structure-preserving* transformations and *algebraic* transformations. Examples of structure-preserving transformations are: common subexpression elimination, dead-code elimination, renaming of temporary stacks or variables, and interchange of two adjacent independent statements.

Peephole optimization can also be applied to ANDA programs. This is an effective technique for locally improving the target program by examining a window of adjacent commands called the peephole and replacing this window of commands by a faster (perhaps shorter) sequence. Example transformations that are characteristic of peephole optimization are: redundant instruction elimination, use of target language idioms, flow-of-control optimization, and algebraic simplifications.

[5] Actually, since we do not care about the order in which the the results are presented, Statement 7 (cache_sort) is unnecessary.

The optimizations performed to the ANDA programs corresponding to the queries given at the beginning of this section fit some of the categories just described. The following example serves to illustrate optimizations obtained by detecting common subexpressions.

Consider a the query to find all sophomore or freshman students of different sex sharing the same room.

```
SELECT DISTINCT S1.sname, S1.sex, S1.age, S1.dorm, S1.room
FROM Student S1, Student S2
WHERE (S1.class="SOPH" OR S1.class="FRES") AND
      (S2.class="SOPH" OR S2.class="FRES") AND
      (S1.room=S2.room) AND
      (S1.sex != S2.sex);
```

The code generated by the query processor for the predicate (S1.class="SOPH" or S1.class="FRES") would be:

```
vt_retrieve(TEMP1,Dclass,SOPH,SOPH,class,Student,relation)
vt_retrieve(TEMP1,Dclass,FRES,FRES,class,Student,relation)
cache_union(TEMP1)
```

The code generated for the predicate (S2.class="SOPH" or S2.class="FRES") would be exactly the same three statements as above. Clearly, the sequence of three function calls are identical and would be unnecessarily repeated. This example shows an instance where common subexpressions (redundant code) may appear in an ANDA program. After detecting this opportunity for optimization, it is possible to produce more efficient code. Describing details of all possible compiler-like optimizations is beyond the scope of this paper. We just want to point out that our approach to extensive manipulation of tuple-ids in main memory opens up many opportunities for optimization not previously explored in database query optimization.

7 Cost Based Query Optimization

The ANDA access language provides a mechanism to specify access plans. Several access plans can be specified for the same query. While they all yield the same result, the steps involved in obtaining the result may be quite different. Thus different access plans for the same query may have a wide disparity in the cost. This raises several important questions:

1. How does one determine the cost for an access plan?
2. Is the cost independent of the variables of the query?
3. Can we find an optimal plan?

Most of this paper so far, discusses strategies for optimization where it is possible to heuristically determine which of the two access plans are better by means of the rules specified in the rule base. However, this may not always be case and information regarding the selectivities of values in the database may have to be used to determine the optimal plan. ANDA stores information about the number of unique values for each attribute and the total number of values for each attribute in the DATA-DICTIONARY. These values can be accessed by the access language and the rule-based optimizer to make decisions about choosing the optimal access strategy. This is being done at compile time.

Unfortunately, all of the optimization results cannot be evaluated at compile time and some of these decisions can be best made at run time. ANDA access language has commands to count the number of elements on the stack at run time and it is possible to generate plans that multiple options embedded in the plan and the decision about the most optimal one is made at run time.

7.1 Cost Comparisons for Two Access Plans

Let us reconsider the example where we want to list the names of all the female Computer Science students.

```
SELECT sname, sid
FROM Student
WHERE ssex = "F" and sdept = "CSCI"
```

```
1.  cache_create_stack(TEMP1)
2.  vt_retrieve(TEMP1,Dsex,F,F,ssex,Student,relation)
3.  vt_retrieve(TEMP1,Ddept,CSCI,CSCI,sdept,Student,relation)
4.  cache_intersection(TEMP1,S*)
5.  cache_transform(TEMP1,S*a)
6.  cache_copy(TEMP1,TEMP1)
7.  cache_transform(TEMP1,S*b)
8.  cache_union(TEMP1,S**)
9.  cache_sort(TEMP1)
10. rl_retrieve(TEMP1)
```

Assume that there are 250,000 female students, 200 computer science students and 25 female computer science students in our university database. Also assume that: it takes 5 disk accesses (B^+-tree index page accesses) to get to tuple-ids in a VALTREE, one page stores 125 tuple-ids and one tuple-id is 8 bytes, and it takes 2 disk accesses to retrieve one value from the RECLIST.

The cost of a query in ANDA can be computed in terms of the number of disk accesses made, the amount of main memory used (maximum CACHE size), and the order of complexity for processing the query in terms of number of tuple-ids processed in the stack. The estimated cost for the above query would be:

Step 2. There are 250,000 female students. Therefore, to retrieve all tuple-ids we need (250000 / 125) + 5 = 2005 disk accesses. And the stack space required would be (25K * 8) = 200K bytes.
Step 3. There are 200 computer science students. Therefore to retrieve all tuple-ids we need (200/125) + 5 = 7 disk accesses. And the stack spaces required would be (200 * 8) 1.6K bytes.
Step 4. This step is performed in the CACHE. The order of complexity of performing the intersection is 25000 + 200. Assume tuple-ids are stored sorted.
Step 5–9. There are only 25 elements on the stack and these steps are all in memory. These steps do not signifacntly affect the cost.
Step 10. This step accesses values for 25 names. Therefore, the number of disk accesses for this step is 25 * 2 = 50 disk accesses.

The total cost for this access plan is:

1. Total Disk Accesses = 2005 + 7 + 50 = 2062 disk accesses.
2. Maximum Stack Required = 200K + 1.6K = 201.6K bytes.
3. Approximate number of tuple-ids touched in main-memory = 25K.

A better strategy that uses value-stack would be to determine the ssex attribute for all computer science students and discard all those that are not female. The access plan for this query is:

```
 1.  cache_create_stack(TEMP1)
 2.  vt_retrieve(TEMP1,Ddept,CSCI,CSCI,sdept,Student,relation)
 3.  cache_transform(TEMP1,S*e)
 4.  tid_to_value(TEMP1,VALUE2)
 5.  value_filter(VALUE2,F)
 6.  value_to_tid(VALUE2,TEMP3)
 7.  cache_transform(TEMP3,S*a)
 8.  cache_copy(TEMP3,TEMP3)
 9.  cache_transform(TEMP3,S*b)
10.  cache_union(TEMP3,S**)
11.  cache_sort(TEMP3)
12.  rl_retrieve(TEMP3)
```

The cost for this access plan can be computed as follows:
Statement 2. There are 200 computer science students. Therefore to retrieve all tuple-ids we need (200/125) + 5 = 7 disk accesses. And the stack spaces required would be (200 * 8) = 1.6K bytes.
Statement 3. There are 200 tuple-ids that are transformed.
Statement 4. To convert 200 tuple-ids to the value stack, we need 200 * 2 disk accesses. And the stack space required would be 200 * 8 bytes for tuple-ids and 200 * 4 bytes for the value field = 2.4 Kbytes.

Statement 5. This operation touches all the elements of the value stack and hence is of the order of 200.
Statements 6–11. This steps are performed in main memory and for our purposes of this discussion, this cost is not significant.

Statement 12. This is step accesses values for 25 names. Therefore, the number of disk accesses for this step is 25 * 2 = 50 disk accesses.

The total cost for this access plan is:

1. Total Disk Accesses = 7 + 400 + 50 = 457 disk accesses.
2. Maximum Stack Required = 1.6K + 2.4K = 4K bytes.
3. Approximate number of tuple-ids touched in main-memory = .4K.

Clearly, the second access plan is much better. Unfortunately, going to the value-stack may not always be the best strategy, and this decision at times can only be made by determining the costs of each of the commands and by knowing the selectivity of relations stored in the database.

7.2 Cost Evaluation at Run-Time

In some cases it may better to evaluate the cost of the query at run-time rather than at compile time. Without adding any significant additional cost to the query, the following access plan determines the actual number of tuples retrieved and uses that information to determine which access plan to use at run time.

```
 1.  cache_create_stack(TEMP1)
 2.  assign_value(CUTOFF,some_value)
 3.  vt_retrieve(TEMP1,Dsex,F,F,ssex,Student,relation)
 4.  if (cache_top_card(TEMP1) > $CUTOFF)
 5.  then
 6.    vt_retrieve(TEMP1,Ddept,CSCI,CSCI,sdept,Student,relation)
 7.    cache_intersection(TEMP1,S*)
 8.  else
 9.    cache_transform(TEMP1,S*e)
10.    tid_to_value(TEMP1,VALUE2)
11.    value_filter(VALUE2,F)
12.    value_to_tid(VALUE2,TEMP1)
13.  endif
14.  cache_transform(TEMP1,S*a)
15.  cache_copy(TEMP1,TEMP1)
16.  cache_transform(TEMP1,S*b)
17.  cache_union(TEMP1,S**)
```

```
18.  cache_sort(TEMP1)
19.  rl_retrieve(TEMP1)
```

A detailed analysis about the complexity of each of the access language commands is given in [11]. We are actively looking at other experimental results which are beyond the scope of this paper and will be discussed in a subsequent paper.

8 Conclusions and Future Research

We have presented the query processing architecture of ANDA, a prototype nested relational database system developed at Indiana University. The main features of ANDA are: (a) the extensive use of *structured tuple-ids* which allows greater opportunities for query processing in main memory. The transformation of queries into ANDA programs offers the opportunity to apply compiler optimization techniques to the database query optimization process; and (b) the application of rule-based query optimization techniques to provide a *flexible architecture*. Other features of the ANDA prototype are: the capability to efficiently compute joins, and the use of a programmable, flexible access language which provides direct access to the data structures of the system and is suitable for optimization.

Currently, the ANDA access language and the Nested SQL parser are fully implemented in the C programming language. The prototype runs on Sun 3 workstations under the Unix operating system. The query processor has been partly implemented. So far, rules for select-project queries on nested relations are operational. The rest of the rule-based plan generation described in this paper and the optimizer are currently under development. All the queries illustrated in this paper have been run and tested using the ANDA access language.

The ANDA prototype is evolving in several directions. We are not entirely convinced that a variation of Nested SQL is the best query language for this system, a graphical query language for ANDA is discussed in [11]. This query language permits direct manipulation of nested relations. The graphical query language has not been implemented and is currently under development. Also, some benchmarking and comparisons between contending access plans is also being done.

We regard the ANDA prototype as an intermediate learning step toward building more ambitious database systems (i.e., object-oriented database systems). The design of object-oriented database systems may benefit form the ANDA effort in two major ways. First, several object-oriented database systems are being implemented on top of existing relational database systems [14]. 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 relational databases [6]. This is because the nested relational model includes the hierarchical construction of aggregation and set formation which is fundamental to the structural component of object-oriented systems.

Second, some problems faced by designers building object-oriented systems ground up [10] are similar to the problems faced by designers of nested relational systems. In particular, data-structures like the VALTREE and the RECLIST, with some modifications, could be used by object-

oriented systems. Also, the design of structured tuple-ids in ANDA offers a new perspective for the design of object-ids in object-oriented systems.

We identify two main limitations in the current implementation of structured tuple-ids in ANDA that need to be removed in order to make them useful for object-oriented systems. First, currently they can only represent objects whose structure is represented by a hierarchy. The ability to represent more complex object structures is important in object-oriented systems. Second, because the nested relational model is entirely value oriented, there is no support for reference tuple-ids. Support of reference attributes in object-oriented systems is very important for sharing of information. We feel that ameliorating these two limitations is feasible and that this will bring new opportunities for query processing in object-oriented systems.

Appendix

A The Access Language Commands

A.1 VALTREE Retrieve

```
vt_retrieve(Stack,Domain,Low_Value,High_Value,Attribute,Relation,Granularity)
vt_retrieve_join(Stack,Domain,Low_Value,High_Value,
                                Attribute_Relation_pairs,Granularity)
```

These two functions extract tuple-ids that satisfy the conditions specified by the arguments of the function from the VALTREE and place the resulting set of tuple-ids on the Stack. In vt_retrieve_join, the set of tuple-ids for both Attribute and Relation arguments must be non-empty. The granularity of the elements placed on the stack is determined by the Granularity argument. The resulting group of tuple-ids are placed on the stack.

A.2 RECLIST Retrieve

```
rl_retrieve(Stack)
```

This function outputs database values corresponding to every tuple-id that is in the top element of the stack.

A.3 Basic CACHE Functions

```
cache_create_stack(Stack)
value_create_stack(Stack)
cache_dest_stack(Stack)
value_dest_stack(Stack)
cache_push(Stack,Element)
cache_pop(Stack)
value_pop(Stack)
cache_empty?(Stack)
value_empty?(Stack)
cache_top_card(Stack)
value_top_card(Stack)
```

The above functions are basic CACHE functions. As the CACHE is implemented as a set of stacks, the stack functions for the tuple-id CACHE and the value CACHE are defined. The value CACHE is implemented as an intermediate stack and hence does not have a user accessible push function.

A.4 Binary CACHE Functions

```
cache_union(Stack,Tid-Window)
cache_intersection(Stack,Tid-Window)
cache_difference(Stack,Tid-Window)
cache_product(Stack,Tid-Window)
```

The above functions operate on the top two elements of the stack. If there is only one element on the stack, the stack remains unchanged.

A.5 Unary CACHE Functions

```
cache_filter(Stack,Filter-Format)
value_filter(Stack,Filter-Format)
cache_transform(Stack,Transform-Format)
gen_tids(Stack,Tid-Format)
cache_sort(Stack)
value_sort(Stack)
cache_one_of(Stack)
cache_explode(Stack)
```

The above functions operate on the top element of the stack.

A.6 Tuple-id to Value Stack Conversion

```
tid_to_value(TupleStack,ValueStack)
value_to_tid(ValueStack,TupleStack)

cache_copy(FromStack,ToStack)
cache_val_copy(FromStack,ToStack)
```

The above functions copy or convert the top element of the first argument to the second stack. The top element of the first stack remains unchanged.

A.7 Variable Assignment

```
assign(Variable_Name,Value)
assign_value(Stack,Variable_Name)
```

The assign function assigns value to a variable. The assign_value determines a value for one of the tuple-ids from the top elements of the stack and assigns the value to the variable name.

A.8 Conditional and Jump Functions

```
ifempty(Condition,Label1,Label2)
if_not_empty(Condition,Label1,Label2)
while (Condition)
od
```

These functions are standard control sequences that are used for writing access plans.

B Example Queries Using Nested SQL

This section presents our Nested SQL language interface using several examples. The examples are based on the following database scheme:

```
Course = (cname, dept, cno, credits, Prerequisite, Section)
Prerequisite = (cno)
Section = (secid, term, instructor, Enrollment)
Enrollment = ( sno, grade )
Student = (sname, sid, class, school)
```

Q1. Find prerequisites of courses offered by the computer science department in the fall of 1989.

```
SELECT cname, cno, Prerequisite
FROM Course
WHERE dept = "CSCI" AND
      Course.Section.term = "F89"
```

```
 1. cache_create_stack(TEMP1)
 2. vt_retrieve(TEMP1,Ddept,CSCI,CSCI,dept,Course,relation)
 3. vt_retrieve(TEMP1,Dterm,F89,F89,term,Section,relation)
 4. cache_intersection(TEMP1,C*)
 5. cache_transform(TEMP1,C*e)
 6. gen_tids(TEMP1,C*e?a)
 7. cache_copy(TEMP1,TEMP1)
 8. cache_transform(TEMP1,C*e*b)
 9. cache_union(TEMP1,C****)
10. cache_sort(TEMP1)
11. rl_retrieve(TEMP1)
```

Q2. Find the students enrolled in computer science courses during the fall 1988 who received a grade of 80 or higher.

```
SELECT cno, cname, (SELECT Enrollment.sno, Enrollment.grade
                    FROM Section
                    WHERE term = "F88" AND
                          Enrollment.grade >= 80)
FROM Course
WHERE dept = "Computer Science"
```

Another way of expressing this query is:

```
SELECT cno, cname, Enrollment.sno, Enrollment.grade
FROM Course
WHERE dept = "CSCI"  AND
      Section.term = "F88" AND
      Section.Enrollment.grade >= 80
```

```
 1. vt_retrieve(TEMP1,Dgrade,80,~,grade,Enrollment,domain)
 2. vt_retrieve(TEMP1,Dterm,F88,F88,term,Section,relation)
 3. cache_intersect(TEMP1,C***)
 4. vt_retrieve(TEMP1,Ddept,CSCI,CSCI,dept,Course,relation)
 5. cache_intersection(TEMP1,C*)
 6. cache_copy(TEMP1,TEMP1)
 7. cache_transform(TEMP1,C*f*a)
 8. cache_copy(TEMP1,TEMP1)
 9. cache_transform(TEMP1,C*b)
10. cache_copy(TEMP1,TEMP1)
11. cache_transform(TEMP1,C*a)
12. cache_union(TEMP1,C******)
13. cache_union(TEMP1,C******)
14. cache_union(TEMP1,C******)
15. rl_retrieve(TEMP1)
```

If we want to include not only the student number but also the name of the student then we need the following query:

```
SELECT cno, cname, SINFO(Enrollment.sno, Student.sname, Enrollment.grade)
FROM Course, Student
WHERE dept = "Computer Science"  AND
      Section.term = "F88" AND
      Section.Enrollment.grade >= 80 AND
      Section.Enrollment.sno = Student.sid
```

```
 1. vt_retrieve(TEMP1,Dgrade,80,~,grade,Enrollment,domain)
 2. vt_retrieve(TEMP1,Dterm,F88,F88,term,Section,relation)
 3. cache_intersect(TEMP1,C***)
 4. vt_retrieve(TEMP1,Ddept,CSCI,CSCI,dept,Course,relation)
 5. cache_intersection(TEMP1,C*)
 6. cache_copy(TEMP1,TEMP1)
 7. cache_transform(TEMP1,C*f*a)
 8. cache_copy(TEMP1,TEMP1)
 9. cache_transform(TEMP1,C*b)
10. cache_copy(TEMP1,TEMP2)
11. cache_transform(TEMP2,C*a)
12. cache_explode(TEMP2)
13. while (not_empty? TEMP2) do
14.   assign_value(SID,TEMP2)
15.   vt_retrieve(TEMP2,Did,$SID,$SID,sid,Student,relation)
16.   cache_transform(TEMP2,S*b)
17.   cache_union(TEMP2,S**)
18.   cache_copy(TEMP2,TEMP1)
19.   cache_union(TEMP1,******)
20.   cache_pop(TEMP2)
21. od
22. cache_union(TEMP1,C******)
23. cache_union(TEMP1,C******)
24. cache_union(TEMP1,C******)
25. rl_retrieve(TEMP1)
```

Q3. Find courses where both students Jones and Smith are enrolled.

This query can be posed as:

```
SELECT cname, cno
FROM Course
WHERE (SELECT sno
       FROM Student
       WHERE sname = "Jones" OR
             sname = "Smith")
      IN Course.Section.Enrollment.sno
```

or more naturally as:

```
SELECT cname, cno
```

```
FROM Course
WHERE Course.Section.Enrollment.sno
CONTAINS (SELECT sno
          FROM Student
          WHERE sname = "Jones" OR
                sname = "Smith")
```

```
1. cache_create_stack(S1)
2. cache_create_stack(S2)
3. vt_retrieve(S1,Dname,Jones,Jones,sname,Student,relation)
4. vt_retrieve(S1,Dname,Smith,Smith,sname,Student,relation)
5. cache_union(S1,S**)
6. cache_transform(S1,S*a)
7. cache_explode(S1)
8. assign_value(S1,SMITH)
9. cache_pop(S1)
10. assign_value(S1,JONES)
11. vt_retrieve(S2,Did,$JONES,$JONES,sno,Enroll,relation)
12. cache_transform(S2,C*f*a)
13. vt_retrieve(S2,Did,$SMITH,$SMITH,sno,Enroll,relation)
14. cache_transform(S2,C*f*a)
15. cache_intersection(S2,C*f*a)
16. cache_transform(S2,C*a)
17. rl_retrieve(S2)
```

Q4. Find all students enrolled in some section of the database course in the fall of 1988.

```
SELECT C.Section.Enrollment.sno
FROM Course C
WHERE cname = "Database" AND
      Section.term = "F88"
```

Here we show that projection has "union semantics". That is, since the database course might have been offered in several sections during fall 1988, each matching section will yield a set of students, therefore, we may get a set of sets as a result. When this is the case, SELECT will automatically union these sets. This is in contrast to other approaches like the one in LOOQ, where they explicitly use a COLLAPSE function applied to the result of the select [21].

```
1. cache_create_stack(S1)
2. value_create_stack(V1)
```

```
3. vt_retrieve(S1,Dcourse,Database,Database,cname,Course,relation)
4. vt_retrieve(S1,Dterm,F88,F88,sterm,Section,relation)
5. cache_intersection(S1,C****)
6. gen_tids(S1,C*f*d?a)
7. tid_to_value(S1,V1)
8. remove_duplicates(V1)
9. print_stack(V1)
```

Q5. Find the number of sections of a database course offered in the fall of 1988.

```
SELECT COUNT(C.Section)
FROM Course C
WHERE cname = "Database" AND
      Section.term = "F88"
```

```
1. cache_create_stack(S1)
2. vt_retrieve(S1,Dcourse,Database,Database,cname,Course,relation)
3. vt_retrieve(S1,Dterm,F88,F88,sterm,Section,s)
4. cache_intersection(S1,C****)
5. cache_top_card(S1)
```

Q6. Find the average grade obtained by students enrolled in database courses in the fall of 1988.

```
SELECT  AVERAGE(C.Section.Enrollment.grade)
FROM Course C
WHERE cname = "Database" AND
      Section.term = "F88"
```

```
1. cache_create_stack(S1)
2. value_create_stack(V1)
3. vt_retrieve(S1,Dcourse,Database,Database,cname,Course,relation)
4. vt_retrieve(S1,Dterm,F88,F88,sterm,Section,relation)
5. cache_intersection(S1,C****)
6. gen_tids(S1,C*f*d?b)
7. tid_to_value(S1,V1)
8. value_average(V1)
```

Q7. Find all courses having some prerequiste.

```
SELECT cname FROM Course
WHERE Prerequiste.cno != EMPTYSET;
```

```
 1.  vt_retrieve(TEMP1,Dcno,low_value,high_value,cno,Course,tuple-id)
 2.  while_not_empty(TEMP1)
 3.    cache_copy(TEMP1,TEMP2)
 4.    cache_transform(TEMP2,C*e)
 5.    generate_tuple_ids(TEMP2,C*e?a)
 6.    if (not_empty(TEMP2)) then
 7.      cache_transform(TEMP2,C*a)
 8.      rl_retrieve(TEMP2)
 9.      cache_pop(TEMP2)
10.    else
11.      cache_pop(TEMP1)
12.    endif
13. od
```

Q8. Find the names of all sophomore students that share the same room.

```
SELECT DISTINCT S1.Sname, S1.Ssex, S1.Sage, S1.Sdorm, S1.Sroomno
FROM Student S1, Student S2
WHERE S1.Sclass="SOPH" AND
      S2.Sclass="SOPH" AND
      S1.Sroomno=S2.Sroomno;
```

or alternatively

```
SELECT S1.sname, S1.room FROM Student S1
WHERE S1.class="SOPH"
AND EXISTS (SELECT S1.sname FROM Student S2
            WHERE S2.class="SOPH" AND S2.room=S1.room);
```

```
 1.  vt_retrieve(TEMP1,Dclass,SOPH,SOPH,class,Student,tuple-id)
 2.  while_not_empty(TEMP1)
 3.    cache_copy(TEMP1,TEMP2)
 4.    cache_transform(TEMP2,S*h)
 5.    assign_value(TEMP2,VAR1)
 6.    vt_retrieve(TEMP3,Droom,$VAR1,$VAR1,room,Student,relation)
 7.    vt_retrieve(TEMP3,Dclass,SOPH,SOPH,class,Student,relation)
 8.    cache_intersection(TEMP3,S*)
 9.    if (not_empty? (TEMP3)) then
10.       cache_transform(TEMP3,S*a)
11.       cache_copy(TEMP2,TEMP3)
```

```
12.        cache_union(TEMP3)
13.        rl_retrieve(TEMP3)
14.       cache_pop(TEMP3)
15      else
16.        cache_pop(TEMP2)
17.        cache_pop(TEMP1)
18.     endif
19. od
```

References

[1] Special issue on nested relations. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 11* (Sept. 1988).

[2] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers, principles, techniques, and tools.* Addison-Wesley Publishing Company, Reading, Ma, 1986.

[3] Astrahan, M., Blasgen, M., Chamberlin, D., Eswaran, K., Gray, J., Griffiths, P., King, W., Lorie, R., McJones, P., Mehl, J., Putzolu, G., Traiger, I., Wade, B., and Watson, V. System R: Relational approach to database management. *ACM Transactions on Database Systems 1*, 2 (1976), 97–137.

[4] Bancilhon, F., Richard, P., and Scholl, M. On line processing of compacted relations. In *Proceedings of the Eighth International Conference on Very Large Data Bases, Mexico City* (Sept. 1982), pp. 263–269.

[5] Batory, D. S. A molecular database systems technology. Tech. Rep. TR-87-23, Department of Computer Sciences, University of Texas at Austin, June 1987.

[6] Blakeley, J., Celis, P., Colby, L., Deshpande, A., Jeng, I., Kitchel, S. W., Martin, N., Plaisier, D., and Van Gucht, D. An object-oriented database using a nested relational backend. Position Paper at OOPSLA'88, Apr. 1988.

[7] Colby, L. S. A recursive algebra and query optimization for nested relations. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data* (May 1989), pp. 273–283.

[8] Dadam, P. Advanced information management (AIM): Research in extended nested relations. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 11*, 3 (Sept. 1988), 4–14.

[9] Dadam, P., Küspert, K., Andersen, F., Blanken, H., Erbe, R., Guenauer, J., Lum, V., Pistor, P., and Walch, G. 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., 1986), pp. 356–367.

[10] DAYAL, U., MANOLA, F., BUCHMANN, A., CHAKRAVARTHY, U., GOLDHIRSCH, D., HEILER, S., ORENSTEIN, J., AND ROSENTHAL, A. 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* (Apr. 1987), Springer-Verlag.

[11] DESHPANDE, A. *An Implementation for Nested Relational Databases.* PhD thesis, Indiana University, Bloomington, Indiana 47405, June 1989.

[12] DESHPANDE, A., AND VAN GUCHT, D. An implementation for nested relational databases. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases* (Aug. 1988), pp. 76–87.

[13] DESHPANDE, V., AND LARSON, P. An algebra for nested relational databases. Technical report, University of Waterloo, Waterloo, Canada, Nov. 1987.

[14] FISHMAN, D. H., BEECH, D., , CATE, H., CHOW, E., CONNORS, T., DAVIS, J., DERRETT, N., HOCH, C., KENT, W., LYNGBAEK, P., MAHBOD, B., NEIMAT, M., RYAN, T. A., AND SHAN., M. IRIS: An object-oriented database management system. *ACM Transactions on Office Information Systems 5*, 1 (Jan. 1987), 48–69.

[15] FREYTAG, J. C. A rule-based view of query optimization. In *Proceedings of ACM-SIGMOD 1987 Annual Conference* (San Francisco, CA, May 1987), pp. 173–180.

[16] GAMERMAN, S., AND SCHOLL, M. Hardware versus software data filtering: The VERSO experience. In *Proceedings of the Fourth International Workshop on Database Machines, Grand Bahama Island* (1985), Springer-Verlag, pp. 112–136.

[17] GRAEFE, G., AND DEWITT, D. J. The EXODUS optimizer generator. In *Proceedings of ACM-SIGMOD 1987 Annual Conference* (San Francisco, CA, May 1987), pp. 160–172.

[18] KHOSHAFIAN, S. N., AND COPELAND, G. P. Object identity. In *OOPSLA '86 Proceedings* (Sept. 1986), pp. 406–416.

[19] KORTH, H. F. Optimization of object-retrieval queries. In *Advances in Object-Oriented Database Sytems, 2nd International Workshop on Object-Oriented Database Systems* (Bad Münster am Stein-Ebernburg, FRG, Sept. 1988), Springer-Verlag, pp. 352–357.

[20] LARSON, P. The data model and query language LauRel. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 11*, 3 (Sept. 1988), 23–30.

[21] Lécluse, C., Delobel, C., and Richard, P. LOOQ: A query language for object-oriented databases, informal presentation. Rapport Technique Altair 22-88, GIP Altair, Rocquencort, B.P. 105, 78153 Le Chesnay Cedex, France, Sept. 1988.

[22] Lee, M. K., Freytag, C., and Lohman, G. M. Implementing an interpreter for functional rules in a query optimizer. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases* (Aug. 1988), pp. 218–229.

[23] Lindsay, B. G., McPherson, J., and Pirahesh, H. A data management extension architecture. In *Proceedings of ACM-SIGMOD 1987 Annual Conference* (San Fransisco, CA, May 1987), pp. 220–226.

[24] Litwin, W. Linear hashing: A new tool for file and table addressing. In *Proceedings of the Sixth International Conference on Very Large Data Bases* (Montreal, Canada, 1980).

[25] Missikoff, M. A domain based internal schema for relational database. In *Proceedings of ACM-SIGMOD 1982 International Conference on Management of Data* (1982), pp. 215–224.

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

[27] Paul, H.-B., Schek, H.-J., Scholl, M. H., Weikum, G., and Deppisch, U. Architecture and implementation of the Darmstadt database kernel system. In *Proceedings of ACM-SIGMOD 1987 Annual Conference* (San Francisco, CA, May 1987), pp. 196–207.

[28] Pistor, P., and Anderson, F. Designing a generalized NF^2 model with an SQL-type language interface. In *Proceedings of the Twelfth International Conference on Very Large Data Bases, Kyoto* (Aug. 1986), pp. 278–285.

[29] Roth, M. A., and Kirkpatrick, J. E. Algebras for nested relations. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 11*, 3 (Sept. 1988), 39–47.

[30] Roth, M. A., Korth, H. F., and Batory, D. S. SQL/NF: A query language for ¬1NF relational databases. *Information Systems 12*, 1 (1987), 99–114.

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

[32] Yokota, K., Kawamura, M., and Kanaegami, A. Overview of the knowledge base management system (KAPPA). In *Proceedings of the International Conference on Fifth Generation Computer Systems* (Tokyo, Japan, 1988), ICOT, pp. 252–263.