The Design of the Triton Nested Relational Database System

Tina M. Harvey Craig W. Schnepf
Mark A. Roth
Department of Electrical and Computer Engineering (AFIT/ENG)
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433

Abstract

Unique database requirements of applications such as computer-aided design (CAD), computer-aided software engineering (CASE), and office information systems (OIS) have driven the development of new data models and database systems based on these new models. In particular, the goal of these new database systems is to exploit the advantages of complex data models that are more efficient (in terms of time and space) than their relational counterparts.

In this paper, we describe the design and implementation of the Triton nested relational database system, a prototype system based on the nested relational data model. Triton is intended to be used as the backend storage and access component of the aforementioned applications.

This paper describes the architecture of the Triton system, and compares the performance of the nested relational model versus the relational model using Triton. In addition, this paper evaluates the EXODUS extensible database toolkit used in the development of the Triton system including key features of the persistent programming language E and the EXODUS storage manager.

1 Introduction

In recent years, database research has focused on the development of database systems to support non-standard applications, such as computer-aided design (CAD), computer-aided software engineering (CASE), and office information systems (OIS). Requirements in these new application areas have driven the development of new data models and database systems based on these new models to efficiently manage large volumes of non-standard data (such as textural or pictoral information).

This paper presents the Triton nested relational database system [11, 17], a prototype database intended to be used as the backend storage component for non-standard applications. The Triton system is based on an extension of the relational model [5], called the *nested relational model* [20], which allows a hierarchical representation of complex objects. There were three goals to our research:

1. To implement a DBMS prototype (Triton) to process data within the context of the nested relational data model using the tools provided by the EXODUS extensible database toolkit [4],
2. To use Triton to evaluate the capability of the nested relational model versus the relational model, and
3. To evaluate the effectiveness of the EXODUS toolkit in the development of the Triton system.

We begin in Section 2 by discussing some background information on the nested relational data model and the EXODUS extensible database toolkit. Section 3 gives an overview of the Triton system by describing the components that make up its design. In Section 4, we demonstrate query processing within Triton through the use of an example. A preliminary discussion of the advantages of the nested relational data model versus the relational data model using Triton is presented in Section 5. Section 6 highlights our evaluation of the EXODUS toolkit in the development of Triton. Section 7 presents some related work in the realm of nested relational database implementations. Finally, we conclude this paper with a summary of our research and a discussion of future enhancements to Triton.

2 Background

This section provides some background regarding the development of the Triton system. First, the nested relational model is described; then we give a brief overview of the EXODUS extensible database toolkit.

2.1 Nested Relational Model

The nested relational model (NRM) is an extension of the relational model, relaxing the 1NF restriction

and allowing attributes of a relation to be a set of values [12], or possibly, another relation [20]. The NRM has two advantages over the relational model for the storage of hierarchical data: decreased storage requirements and increased processing speed. An intangible advantage of the NRM over the relational model is a more intuitive mapping of complex data; the relational model splits the data into 1NF "chunks", while the NRM allows data to remain in hierarchical form within the database schema. Thus the NRM may be a better underlying data model for mapping complex objects in support of an object-oriented database. See Section 4 for an example nested relation.

2.2 EXODUS Extensible Database Toolkit

In order to quickly prototype the Triton system, we utilized EXODUS [4], an extensible database "toolkit" developed at the University of Wisconsin. The goal of EXODUS is to provide extensibility without sacrificing performance [4]. To meet this goal, EXODUS either provides generic system components or furnishes a component *generator* to aid in the construction of the component. When neither approach is possible, EXODUS provides the tools to aid in the development of the component. The EXODUS tools used in the development of Triton include:

- the storage manager [3], which stores the physical data of the database and provides access to the data via procedural calls
- the E persistent programming language and its compiler [15]

At the present time, Triton's optimizer component has not been fully developed. However, the intention is to use the EXODUS optimizer generator [9] to generate this component. The EXODUS optimizer generator takes as input (1) a set of operators, (2) a set of methods that implement the operators, (3) transformation rules that describe equivalence-preserving transformations of query trees, and (4) implementation rules that describe how to replace an operator with a specific method. Using these rules, a specific optimizer is generated for the particular application.

We chose to use the EXODUS optimizer generator for Triton because the relational algebra used by Triton lends itself to EXODUS' rule-based method. This modular approach to database development will reduce the amount of code required for implementation of the Triton system. The only unique code Triton's developers will need to write will be the additional functions that are called by the optimizer when implementing a specific operator or access method. With accurate cost functions, we anticipate the generated optimizer will work as well as a custom built one.

3 Triton Architecture

The architecture of the Triton system is given in Figure 1. This diagram shows how queries are processed within Triton. First, the user (or application program) inputs a query that is written in SQL/NF [16], a query language based on an extension of SQL. SQL/NF enhances SQL's capabilities by modifying all SQL operators to allow the manipulation of nested relations.

Accessing the database schema via the catalog manager, the parser translates the query into an unambiguous algebra for computer manipulation. The algebra used in the Triton system was developed by Latha Colby [6] for nested relations. Colby algebra supports the retrieval of information from any level of nesting in a relation without first "flattening out" the relation to fit the relational model [6:276]. The Colby algebra query representation is contained in a data structure called a *query tree*.

Triton's parser component was implemented using the UNIX tools of YACC and LEX. The parser process is accomplished in two steps. In the first step, the query is scanned by the lexical analyzer (LEX) which assigns a "token" to the key words and other components of the query statement and returns a stream of these tokens. The second step is accomplished by YACC, which receives the stream of tokens and organizes them according to the input structure rules; when one of these rules is recognized, the C code in the *action* part of the rule is invoked.

At the present time, Triton's parser is able to parse all possible SQL/NF statements, but only builds query trees for the statements that create and delete items in the system catalogs, as well as query statements that directly translate into the appropriate algebraic operations of select, project, and cartesian product.

The query tree built by the parser is passed to the rule-based optimizer where it is transformed into a plan tree. In this process, the sequence and contents of the nodes in the query tree are changed into an optimized plan tree and relational operators are replaced with specific operator methods.

The E code generator traverses the plan tree in postorder and generates the E code to perform the query. The operator and access methods are designed using a recursive feature that takes advantage of the nested data model; whenever a nested relation is encountered in the plan tree, the procedure that generates the methods is recursively called. The methods currently implemented by Triton are a sequential filescan and a nested loops join method. These methods were selected because their simplicity made them easy to

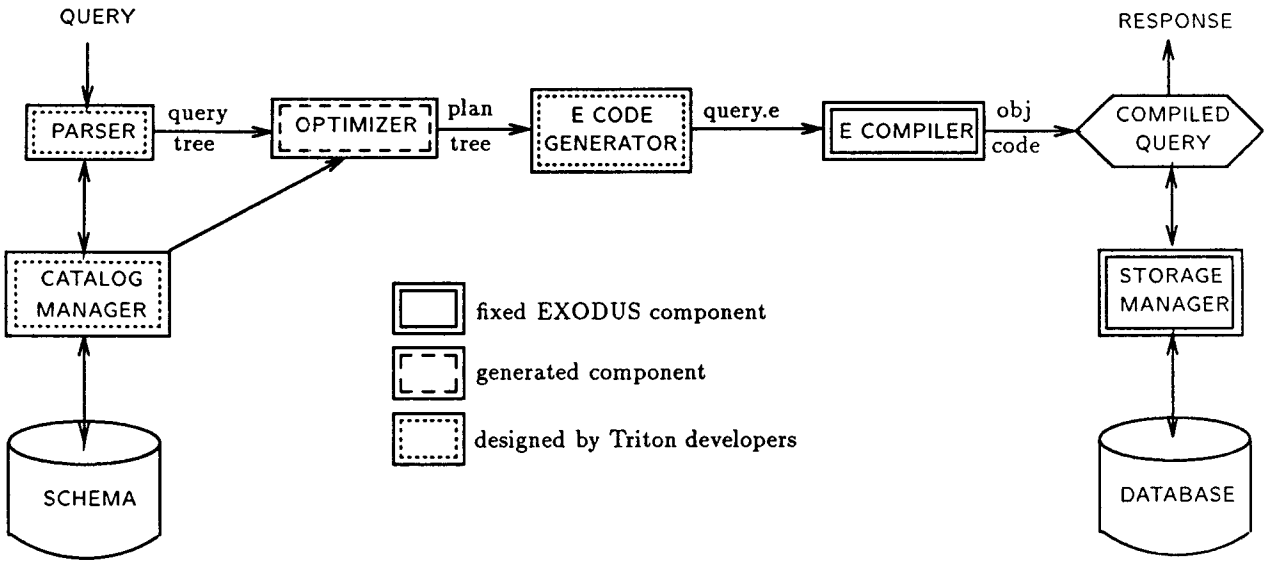

Figure 1: The Triton Nested Database System Architecture Using EXODUS

implement as we explored access to nested relations. The goal was to give Triton an initial capability to process queries of any type, and to lay the foundation for the development of more efficient access methods in the future.

The E compiler links the methods with the E code that specifies the relation definitions and the object code is executed to manipulate the information in the database via the storage manager provided by EXODUS. The manipulated data is returned to the user as the response to the query. When it came time to design how the Triton system would perform queries, two approaches were possible. The query implementor could either be (1) an *interpreter* that manipulates the database directly, or (2) a *code generator* that generates E code to implement the query.

There are advantages and disadvantages to both approaches. The interpreter approach would work well in an *ad hoc* query environment, since manipulation of the database is performed directly by the interpreter. A code generator requires substantial time to compile and run each generated query. The code generator approach works best when specific queries are known ahead of time, so that the code for those queries is already generated and compiled. The interpreter approach would take longer in this type of environment, since the query must be analyzed and executed on the fly. Because the intended environment for the Triton system will not have an *ad hoc* query capability, the code generator approach was chosen.

4 Query Processing Within Triton

In Triton, a nested relation definition is mapped directly into a persistent collection, which in turn can be made up of collections of objects. As an example, Figure 2 shows a nested relation that holds information on VHSIC Hardware Description Language (VHDL) systems[1]. Figure 3 gives the E code representation of the *Systems* relation.

To demonstrate how query processing is performed within Triton, this section follows a query through the Triton system. Let's assume we have the following SQL/NF query:

```
SELECT number, (SELECT name
                FROM ports
                WHERE mode = "in")
FROM systems
```

This query is asking for names of the "in" ports and the system number for each system in the database. The parser transforms the SQL/NF query into a query tree as depicted in Figure 4. The optimizer changes the query tree into the plan tree depicted in Figure 5 by exchanging operator names with method names. Notice that the project and select nodes are merged into a single node with "FILESCAN" as the method.

Finally, the E code generator walks the plan tree in postorder and generates the code in Figure 6 to perform the query. First, templates are set up to

number	*name*	*ports*				
		name	*mode*	*type*	*start_bit*	*stop_bit*
43191	COUNTER	STRT	in	BIT	0	0
		STROBE	in	BIT	0	0
		CON	in	BIT_V	0	1
		DATA_BUS	in	BIT_V	0	3
		CNT	out	BIT_V	0	3
14701	FULL_ADDER	X	in	BIT	0	0
		Y	in	BIT	0	0
		CIN	in	BIT	0	0
		Z	out	BIT	0	0
		COUT	out	BIT	0	0

Figure 2: The *Systems* Relation

```
dbstruct port {
  dbchar name[12];
  dbchar mode[4];
  dbchar type[6];
  dbint  start_bit;
  dbint  stop_bit;
public:
  port (char *, char *, char *, int, int);
  char * get_name();
  void change_name (char *);
  char * get_mode();
  void change_mode (char *);
  char * get_type();
  void change_type (char *);
  int get_start_bit();
  void change_start_bit (int);
  int get_stop_bit();
  void change_stop_bit (int);
  void print (port *);
};
```

```
dbstruct system {
  dbchar name[12];
  dbint  number;
  dbclass portRVA:collection[port];
  portRVA ports;
public:
  system (char *, int);
  char * get_name();
  void change_name (char *);
  int get_number();
  void change_number (int);
  void print (system *);
};

dbclass systemRVA:collection[system];
persistent systemRVA systems;
```

Figure 3: The E Code Representation of the *Systems* Relation

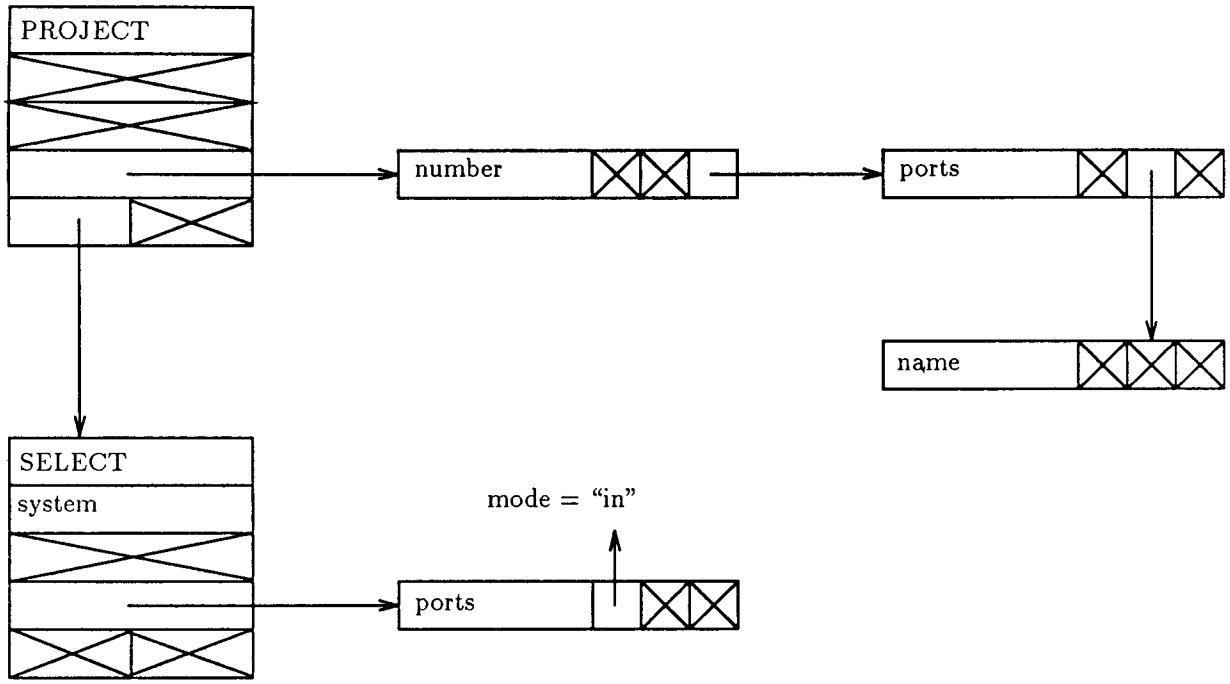

Figure 4: Query Tree

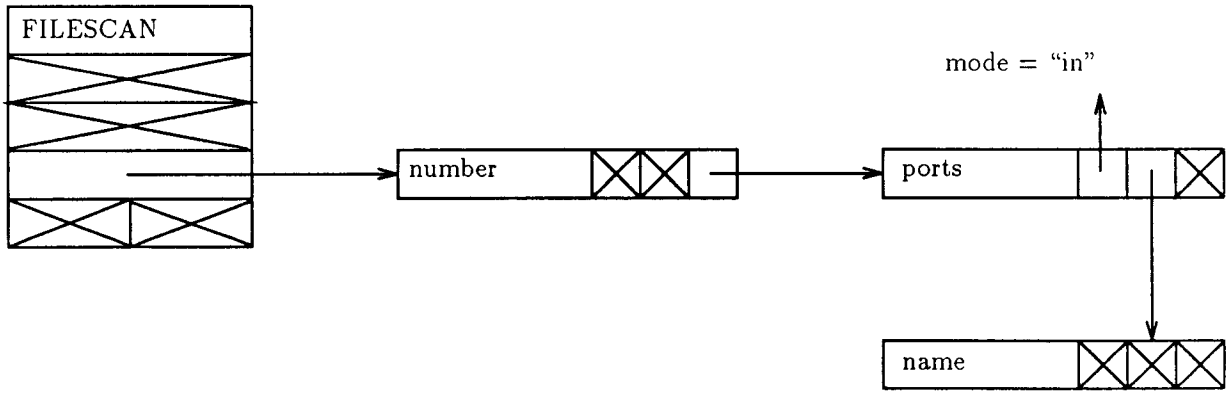

Figure 5: Plan Tree

```
#include <stream.h>
#include <stdio.h>

#ifndef SYSTEMS_H
#include "systems.h"
#endif

extern persistent systemRVA systems;

dbstruct temp1 {
  dbchar name[32];
public:
  temp1 (char *);
  char * get_name();
};

temp1::temp1 (char *nameAtom) {
  dbchar *dest;
  dest = name;
  while (*dest++ = *nameAtom++);
}

char *temp1::get_name() {
  dbchar *source = name;
  char *dest = new char[32];
  char *start = dest;
  while (*dest++ = *source++);
  return (start);
}

dbstruct temp2 {
  dbint number;
  dbclass temp1RVA:collection[temp1];
  temp1RVA temp1_rels;
public:
  temp2 (int);
  int get_number();
};

temp2::temp2 (int numberAtom) {
  number = numberAtom;
}

int temp2::get_number() {
  return (number);
}

iterator temp2 *filescan_temp2()
  {
  iterator (system *tuple_ptr1 = systems.scan())
    {
    temp2 *t2 = new temp2 (tuple_ptr1->get_number());
    temp2 &temp2_ref = *t2;
    system &source_ref1 = *tuple_ptr1;
    iterate (port *tuple_ptr2 = source_ref1.ports.scan())
      if (strcmp (tuple_ptr2->get_mode(), "in") == 0)
        temp1 *t1 = in (temp2_ref.temp1_rels) new
                    temp1(tuple_ptr2->get_name());
    yield (t2);
    }
  }

main ()
  {
  iterate (temp2 *tuple_ptr = filescan_temp2())
    {
    cout << form ("System number: %d\n",
                  tuple_ptr1->get_number());
    temp2 &source_ref1 = *tuple_ptr1;
    iterate (temp1 *tuple_ptr2 = source_ref1.
                                 temp1_rels.scan())
      cout << form ("     Port name: %s\n",
                    tuple_ptr2->get_name());
    }
  }
```

Figure 6: Code to Implement Query

```
System number: 43191
     Port name: STRT
     Port name: STROBE
     Port name: CON
     Port name: DATA_BUS
System number: 14701
     Port name: X
     Port name: Y
     Port name: CIN
```

Figure 7: Results of SQL/NF Query

hold a tuple of the result. These appear as *dbstruct temp1* (which holds the port name) [lines 10–29] and *dbstruct temp2* (which holds the system number and a collection of port names) [lines 31–45]. An iterator [lines 47–60] is set up to process each tuple of the *systems* relation by first copying the system number into the *number* attribute of *temp2* [line 51]. If the port mode is "in" [line 55], the port name is copied into the *temp1_rels* attribute of *temp2* using the constructor function of *temp1* [lines 56–57]. Finally, the main program [lines 62–74] iterates through each of the resultant tuples and prints out the contents. The result of this particular query is given in Figure 7. Note that in all cases two levels of iterators are required to access both the top level attributes of the *systems* relation and the nested *ports* relation within each *systems* tuple.

5 Comparison of Query Performance

As stated earlier, the Triton system is intended to be the backend storage component for non-standard database applications, such as CAD, CASE, or OIS tools. The first application for the Triton system was the representation of a particular CASE methodology, the USAF IDEF$_0$ Structured Analysis (SA) language [13].[1] An example of an SA diagram is shown in Figure 8. Morris [14] defined a nested relational model and a relational model representation for IDEF$_0$ which we implemented in the Triton system. A normalized relational model required 40 relations to represent IDEF$_0$ data whereas the nested relational model required only one relation with one atomic attribute and four relation-valued attributes. The two representations were compared in terms of code generation time and query execution time. Of the four possible queries suggested by Morris, three were selected since the majority of the application program's queries will probably be one of these three. Each of the three queries were generated and executed twenty times, and the minimum, maximum, variance, and average were calculated for each of these twenty runs. All reported times are in seconds, and all runs were accomplished on a Sun 3 workstation at approximately the same level of workload. In terms of code generation time and query execution time, the nested representation outperformed the relational version; the results are dicussed in further detail below.

[1] IDEF$_0$ stands for ICAM Definition Method Zero. ICAM is the United States Air Force's Integrated Computer Aided Manufacturing program.

5.1 Comparison of Code Generation Times

Table 1 compares the code generation times for the relational version against the nested version for each of the three queries. Note that for each query, the generation time for the nested version takes less than one third the time for the equivalent query in the relational version. The difference in code generation times can be attributed to two factors: one, the relational version requires several subqueries to extract needed information and two, the increased complexity of the plan trees for the relational versions of the queries adversely impacts the speed of the code generator.

The fact that there are many subqueries needed in the relational version means that many functions of the code generator must be duplicated for each subquery, such as opening the output file to hold the generated query, checking to see if the main program should be generated, and, if it is, generation of the main program. These actions may seem trivial, but can increase the run time of the code generator for queries made up of many subqueries.

Perhaps the most significant contribution to the code generation time is the complexity of the query's plan tree. For both the relational and nested versions, each table involved in a query must have at least one filescan node in the plan tree. Since the relational model required normalization into 40 tables, the queries in the relational version are represented as multi-node plan trees, with filescans for each table in the query and join nodes connecting them. However, the plan trees in the nested version of the queries contain only one filescan node, since all information is selected and projected from one table. The code generator sets up a separate iterator for each filescan and join node in the plan tree. Clearly, for multi-node plan trees, the job of *codegen* is much more involved. A good assumption seems to be that this complexity accounts for a large portion of the higher code generation times for the relational version as compared to the nested version.

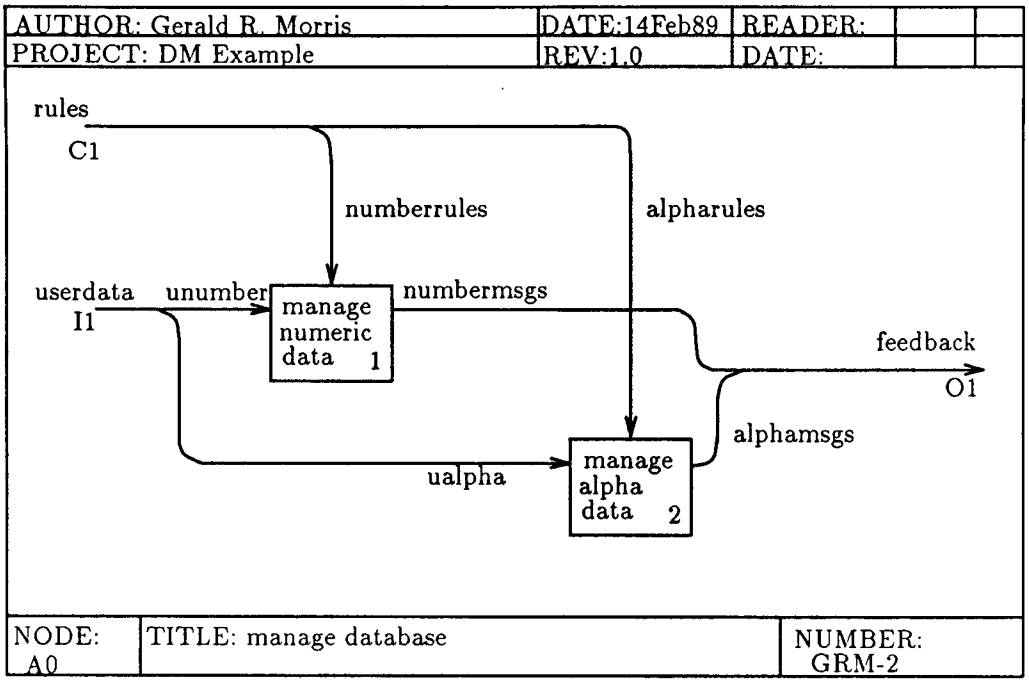

Figure 8: Structured Analysis Diagram

Table 1: Comparison of Code Generation Times

Query	Relational Generation Time	Nested Generation Time
Drawing Data	24.261	7.386
Activ Data Dict	17.311	3.131
Data Elem Data Dict	31.619	6.103

5.2 Comparison of Query Execution Times

Table 2 compares the query execution times for the relational version against the nested version for each of the three queries. As with code generation, the nested query exection time is less than the relational query execution time. The difference in query execution time can be attributed to two factors: one, the large number of joins in the relational version, and two, increased disk access time due to a lack of clustering in the relational version.

As stated earlier, all information was contained in one table in the nested version, whereas the relational version broke the $IDEF_0$ language data into 40 normalized tables. Each time data had to be correlated between two or more tables, a join was executed. While no joins were required in the nested version (since all data is contained in one table), some of the relational subqueries required as many as four joins. As stated earlier, the join method implemented by Triton is a nested loops join method, selected for its simplicity. A more efficient algorithm such as a sort-merge join or a hash join would probably speed up the execution time of the relational version. However, even with the speedup realized with a more efficient join algorithm, the execution time of the relational version would probably still exceed that of the nested relational version, since no joins are required in the nested relational version. The difference in execution times was not as great as expected, probably due to the limited number of tuples in the relational tables. As the size of the database grows, the execution times should increasingly favor the nested version of the database due to the absence of joins.

The second reason for the faster execution of the nested queries is that disk access times are longer for relational queries since tables are not automatically clustered on disk. The EXODUS storage manager automatically clusters tuples of a collection on disk to

Table 2: Comparison of Query Execution Times

Query	Relational Execution Time	Nested Execution Time
Drawing Data	0.660	0.551
Activ Data Dict	0.874	0.695
Data Elem Data Dict	1.263	0.324

decrease access time. Of course, this means that attributes in the tuples are clustered, including relation-valued attributes. Since the nested version has only one overall table (with several nested tables), all data in the nested version should theoretically be clustered on disk.

5.3 Summary of Comparison.

The goal of this comparison was to demonstrate the advantage of the NRM over the relational model for the storage and retrieval of complex data. This advantage was demonstrated by comparing the two models on code generation time and query execution time. In both areas, the nested version outperformed the relational version, particularly in code generation time. However, between generation and execution time, execution time is most important. This is because queries will be generated once, but will be executed many times. However, the performance of both activities must be considered, since both play a crucial role in determining the processing speed of the Triton system.

6 Evaluation of the EXODUS Toolkit[2]

Implementation of a DBMS is not a trivial undertaking, but the use of extensible systems has eased some of the burden of this task. The use of the EXODUS extensible toolkit reduced the development time of the Triton nested relational database system, and provided the means to evaluate the advantages and disadvantages of using extensible systems such as EXODUS and a persistent programming language to build application-specific database management systems.

6.1 E Programming Language

The nested relational data model is mapped very nicely using the *collection* generator class supplied by EXODUS in the E programming language; relation-valued attributes are represented using collections of collections. Unfortunately, EXODUS only provides the capability for sequential scanning of collections, making access via a search key slow for large relations. The only way around this shortcoming is to build indexes on every frequently accessed or sufficiently large relation and nested relation.

Implementing the relation definitions in the programming language enabled them to be compiled using the E compiler and linked with the object files containing the queries. The advantage of this method is that queries are executed quickly, since they are in machine code. However, the long compilation and linking time makes this method prohibitive for dealing with ad hoc queries.

A good performance aspect of the *collection* class is that items in a collection are clustered on disk, including subcollections. This automatic clustering reduces access time and frees the implementor of having to worry about the grouping of data on disk.

6.2 Storage Manager

The use of the EXODUS storage manager greatly reduced the development time of the Triton system by freeing the developer from worrying about Triton's storage component. The storage manager's procedural interface allows access to database objects without having to know how they are actually being manipulated. As an example, persistent objects are automatically mapped to permanent storage locations through the use of the **persistent** keyword. Data clustering is performed by using the *collection* class and is transparent to the developer. Access to collections of objects is provided via built in procedure calls to the storage manager.

7 Related Work

As stated in Section 1, one of the goals of this research was to implement the first Triton prototype to process data within the context of the nested relational data model, with the aim of making Triton the backend storage component for non-standard applications. This section presents some work closely related to Triton.

Like Triton, the Advanced Information Management Prototype (AIM-P) [7], is also intended to be the database implementation "backend" for design

[2] A more complete evaluation of EXODUS can be found in [10].

application tools and uses the nested relational data model to represent the underlying structure of the database. However, AIM-P's physical storage structure differs from Triton's in that Triton maps relation definitions into a programming language, while AIM-P uses a tree structure to hold the same information.

Deshpande and Van Gucht [8] implemented a nested relational database called ANDA. Of particular interest to this research are the access methods and mechanisms ANDA uses for the retrieval of data from the database. ANDA makes a distinction between value-driven operations, such as select, join, and nest, and structure-oriented operations, such as project and unnest. Speed of retrieval of information is maximized for these two types of operations by using two different storage structures.

The Verso DBMS [18] is a relational database system that stores data in nested form (called *V-relations*). Verso uses dedicated hardware to decrease query processing time. The Verso system utilizes a filter for on-the-fly processing of tuples, which can perform all algebraic operations except for restructuring actions. The filter is implemented as a finite state automaton (FSA), which scans the V-relation one byte at a time. The value of the current byte and the present state of the FSA determines the output function and the next state. The advantage of using such a filter is that query processing is much faster because it is mapped to a very low-level representation of the problem space on a dedicated machine. The disadvantage is that this low-level representation is complex and difficult to grasp, making the filter hard to modify and maintain.

8 Conclusions

The primary goal in developing the Triton nested relational database system is to test the theory that nested relations are a "good" extension to the relational model. Previous theoretical studies have indicated this possibility as the reason for studying nested relations. Next generation databases [19] must support complex data types which don't easily fit into the flat relational model, and yet there are significant advantages to the relational model (normal forms, set-oriented query languages, etc.). We believe that the nested relational model, while retaining the theoretical and mathematical benefits of the relational model, can better support the storage and retrieval of complex objects.

As of this point we have developed an SQL/NF-to-Colby-algebra translator for a major subset of SQL/NF, and some basic access methods (filescan and loops_join) to implement queries on nested relations. We made some initial comparisons of queries for a complex CASE application. By using Triton to store a nested and relational representation of the CASE data and using the methods implemented in this research to query that data, we found that the nested relational model outperforms the relational model in terms of code generation time and query execution time. These results demonstrate the viability of the nested relational model for the representation of complex data for at least one application. In the process we were able to evaluate the use of the EXODUS database toolkit in building a database system. The EXODUS toolkit provided several necessary tools that greatly reduced the development time of the Triton system. However, the use of the EXODUS toolkit did constrain this development, and future efforts must attempt to overcome these constraints.

In the future we will be designing additional operator methods, such as *merge_join* or *index_join* to more efficiently handle the join operator and developing indexing techniques and access methods to use them. Bertino and Kim's work in this area [2], and the indexing strategy of the ANDA system [8] are possibilities. We will also be using and evaluating the EXODUS optimizer generator to optimize Colby algebra expressions.

9 Acknowledgements

We gratefully acknowledge the work of the University of Wisconsin database group, especially David Dewitt and Michael Carey, for their work on the EXODUS system and their willingness to share the results.

References

[1] Armstrong, James R. *Chip-Level Modeling with VHDL.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

[2] Bertino, Elisa and Won Kim. "Indexing Techniques for Queries on Nested Objects," *IEEE Transactions on Knowledge and Data Engineering, 1*(2):196–214 (June 1989).

[3] Carey, Michael J. and others. "Storage Management for Objects in EXODUS." Computer Sciences Department, University of Wisconsin, 1989.

[4] Carey, Michael J. and others. "The EXODUS Extensible DBMS Project: An Overview." In Zdonik, Stanley B. and David Maier, editors, *Readings in Object-Oriented Database Systems,*

pages 474–499, San Mateo, CA: Morgan Kaufmann, 1990.

[5] Codd, E. F. "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, *13*(6):377–387 (June 1970).

[6] Colby, Latha S. "A Recursive Algebra and Query Optimization for Nested Relations." In Clifford, James, et al., editors, *Proceedings of the 1989 ACM-SIGMOD International Conference on the Management of Data*, pages 273–283, May 1989.

[7] Dadam, P., et al. "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies." In Zaniolo, Carlo, editor, *Proceedings of the 1986ACM-SIGMOD International Conference on the Management of Data*, pages 356–367, May 1986.

[8] Deshpande, Anand and Dirk Van Gucht. "An Implementation for Nested Relational Databases." In Bancilhon, Francois and David J. DeWitt, editors, *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, pages 76–87, August 1988.

[9] Graefe, G. and D. J. DeWitt. "The EXODUS Optimizer Generator." In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, May 1987.

10] Hanson, Eric N., et al. *Experiences in DBMS Implementation Using an Object-Oriented Persistent Programming Language and a Database Toolkit*. Technical Report AFIT/EN-TR-90-8, AFIT/ENG, Wright-Patterson AFB, OH 45433: Air Force Institute of Technology, December 1990.

11] Harvey, Capt Tina M. *Access and Operator Methods for the Triton Nested Relational Database System*. MS thesis, AFIT/GCS/ENG/90D-06, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

[12] Makinouchi, A. "A Consideration of Normal Form of Not-Necessarily-Normalized Relations in the Relational Data Model," *Proc. 3rd VLDB*, pages 447–453 (1977).

[13] Materials Laboratory, Air Force Wright Aeronautical Laboratories, Air Force Systems Command, Wright-Patterson AFB, OH 45433. *Integrated Computer-Aided Manufacturing (ICAM) Function Modeling Manual ($IDEF_0$)*, June 1981.

[14] Morris, Capt Gerald R. *A Comparison of a Relational and Nested Relational $IDEF_0$ Data Model*. MS thesis, AFIT/GCE/ENG/89D-5, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1990.

[15] Richardson, Joel E., et al. "The Design of the E Programming Language." Computer Sciences Department, University of Wisconsin, 1989.

[16] Roth, Mark A., et al. "SQL/NF: A Query Language for ¬1NF Relational Databases," *Information Systems*, *12*(1):99–114 (1987).

[17] Schnepf, Capt Craig W. *SQL/NF Translator for the Triton Nested Relational Database System*. MS thesis, AFIT/GCE/ENG/90D-05, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

[18] Scholl, M., et al. "VERSO: A Database Machine Based On Nested Relations." In Abiteboul, S., et al., editors, *Nested Relations and Complex Objects in Databases (Lecture Notes in Computer Science 361)*, pages 27–49, Springer-Verlag, 1989.

[19] *SIGMOD Record*, *19*(4) (December 1990).

[20] Thomas, Stan J. and Patrick C. Fischer. "Nested Relational Structures." In Kanellakis, P. C., editor, *Advances in Computing Research, Volume 3: The Theory of Databases*, pages 269–307, JAI Press, 1985.