SEPTEMBER 1988 VOL. 11 NO. 3

a quarterly bulletin of the IEEE Computer Society technical committee on

Data Engineering

CONTENTS

SPECIAL ISSUE ON NESTED RELATIONS

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC
IEEE

IEEE Computer Society

The LOTUS Corporation has made a generous donation to partially offset the cost of printing and distributing four issues of the Data Engineering bulletin.

Database Engineering Bulletin is a quarterly publication of the IEEE Computer Society Technical Committee on Database Engineering . Its scope of interest includes: data structures and models, access strategies, access control techniques, database architecture, database machines, intelligent front ends, mass storage for very large databases, distributed database systems and techniques, database software design and implementation, database utilities, database security and related areas.

Contribution to the Bulletin is hereby solicited. News items, letters, technical papers, book reviews, meeting previews, summaries, case studies, etc., should be sent to the Editor. All letters to the Editor will be considered for publication unless accompanied by a request to the contrary. Technical papers are unrefereed.

Opinions expressed in contributions are those of the individual author rather than the official position of the TC on Database Engineering, the IEEE Computer Society, or organizations with which the author may be affiliated.

Membership in the Database Engineering Technical Committee is open to individuals who demonstrate willingness to actively participate in the various activities of the TC. A member of the IEEE Computer Society may join the TC as a full member. A non-member of the Computer Society may join as a participating member, with approval from at least one officer of the TC. Both full members and participating members of the TC are entitled to receive the quarterly bulletin of the TC free of charge, until further notice.

Letter from the TC Chair

On behalf of the entire TC on Data Engineering membership, I would like to extend our warmest thanks and appreciation to Sushil Jajodia for his leadership during the last two years.

As the new TC Chair I feel very fortunate in that Sushil will be nearby to offer his advice and counsel. This Fall, Dr. Jajodia will be joining the faculty of George Mason University as Associate Professor of Information Systems and Systems Engineering. Welcome aboard Sushil!

In the June issue of the *Data Engineering Bulletin*, Sushil provided a status report of the major accomplishments of the TC. In brief they are:

- Sponsorship and co-sponsorship of conference, symposia and workshops of interest to our members,
- The timely and regular publication of the *Data Engineering Bulletin*, under the stewardship of Won Kim of MCC, and his Associate Editors, with the generous support of our corporate sponsor, the **Lotus Development Corporation**, and
- The active role of our TC in the development of the proposal that helped to establish the new *IEEE Transactions on Knowledge and Data Engineering.*

Our TC is best known for the *Data Engineering Bulletin* and our association with the *International Conference on Data Engineering.* Both provide forums for the publication, presentation, and discussion of research and development results. It is important for our members to be active participants in these forums and future ones.

My goals are to continue the good works for which our TC is known, and to explore new avenues of professional development, cooperation and growth. In organizing the *International Conferences on Expert Database Systems*, I have been fortunate in being able to draw upon the volunteer efforts of dozens of profesionals and several societies in the cooperative effort of forming the organization and program of a large and successful conference.

In working with the membership of our TC, I hope to be able to count on your volunteer efforts to help build the TC, and to make it a relevant resource within the IEEE Computer Society. We are asked to provide consultation and opinions on many topics, and I hope to get the membership involved in supporting these activities. In addition, if you have specific suggestions, please feel free to contact me at the address given below.

As Sushil mentioned in the June Issue of the Bulletin, the issue of TC membership dues is being addressed by the Society. Dr. Mario Barbacci, Vice Chair of the Technical Activities Board (TAB), told me that of the 90,000 IEEE Computer Society Members only about 3000 are members of a TC. The TC on Data Engineering has 1295 full members and correspondents. Clearly, we have a pool of potential members, and Dr. Barbacci is planning a TC membership drive to be initiated in November with ads and TC descriptions in key IEEE publications. Membership dues will allow our TC to establish a better financial basis to support our activities, such as the publication of the *Data Engineering Bulletin.*

Another method of obtaining funds is through the sponsorship of successful, i.e., money-making conferences. Our TC is currently co- sponsoring the Very Large Database Conference and will co-sponsor future Expert Database Systems Conferences. We will also try to complement these funds with contributions from our corporate benefactors.

Lastly, Won Kim and I have been concerned about the timely distribution of the *Data Engineering Bulletin.* In order to coordinate this process, I have appointed Mr. David Barber as the liaison for requests for back issues, and related matters.

Larry Kerschberg
August 16, 1988

Letter from the Editor

This issue of IEEE Data Engineering is devoted to Nested Relations. In a nested relation, attribute values are not necessarily atomic, but they can be relations themselves. A relation valued attribute may, in turn, have relations as attributes and so on. Although the original definition of a relation by Codd in 1970 does not restrict attribute values to be atomic (i.e. First Normal Form, 1NF, relations), most work on the relational model and most succesful relational database systems in the last decade have the assumption that all the relations are in 1NF.

The research in nested relations is motivated by the need to extend the utilization of relational database systems beyond the traditional data processing applications. For the so called nontraditional database applications such as engineering designs, office forms, image processing, information processing of textual data, the semantics of data objects, such as CAD objects, are too complex to model by flat 1NF relations. When relational database systems with flat relations are used for such applications, the database becomes unecessarily fragmented both conceptually and physically, which in turn has undesirable implications on query processing, data redundancy, and storage structures.

Makinouchi was first to recognize, in 1977, the need for relaxing the 1NF assumption, and extending relational model with nested relations to improve the utilization of relational databases for applications requiring complex objects. Since early 1980's there has been a growing number of database research groups working on nested relations. Several important research results on various aspects of nested relations, including design theory, algebra and calculus, query languages, and storage structures for nested relations, have been published in the literature. In parallel with the theoretical research, several research prototypes such as VERSO project of I.N.R.I.A., AIM-I, AIM-P of IBM Heidelberg, and Darmstadt Database Kernel System of University of Darmstadt have been implemented since then.

In April 1987, the first International Workshop on Nested Relations and Complex Objects was held in Darmstadt, West Germany. The Workshop Material edited by Scholl and Schek provides a list of position papers and a rich collection of research abstracts on nested relationa and complex objects. A book on Nested Relations and Complex Objects containing some selected papers from the workshop and some invited papers will be published by Springer Verlag, in early 1989. There was also a well attended, lively panel on Nested Relations, organized by

Patrick Fischer in the SIGMOD 88 Conference in Chicago. The title of the panel is "Nested Relations: A Step Forward or Backward?". My opinion is that panel's conclusion was nested relations is a step forward, or at least it is not a step backward.

First two papers in this issue are on research prototypes that have been implemented. In the first paper, Dadam describes the AIM-P, Advanced Information Management Prototype, project, and the work on the nested relations at the IBM Heidelberg Scientific Center. The second paper is by Abiteboul and Scholl, and describes the VERSO project and the research on the system design and theory for nested relations and complex objects at I.N.R.I.A. In the following paper, Larson gives a brief overview of the data model and the query language of Laurel, which is a prototype database system, being developed at the University of Waterloo, supporting nested relations and reference attributes. The fourth paper is by G. Ozsoyoglu and Hafez and describes the research at the Case Western Reserve University on storage structures for nested relations. Roth and Kilpatrick in the next paper gives an overview of the algebras for nested relations, and discuss methods for joining nested relations in various forms. Gyssens and Van Gucht's paper is on the expressive power of nested algebra, and they propose a powerset algebra as an alternative to the nested algebra. The final paper of the issue is titled "Nested Relations: A Step Forward or Backward?", and is based on Schek's presentation for the panel discussion at the SIGMOD 88 Conference panel with the same title.

As in the case of any special issue on a timely subject, only a few of the many current projects and research results on nested relations could be described here. I would like to thank the authors for their contributions and their cooperation on meeting the deadlines during the preparation of this issue. I hope that this issue of Data Engineering will help sitimulate interest in Nested Relations Research and Development.

Sincerely,

meral özsoyoğlu

Z. Meral Özsoyoğlu
August, 1988

Advanced Information Management (AIM): Research in Extended Nested Relations

Peter Dadam
IBM Heidelberg Scientific Center
Tiergartenstr. 15, D-6900 Heidelberg
West Germany

EARN/BITNET: DAD at DHDIBM1

1. Introduction and Background

One of the main missions of the IBM Heidelberg Scientific Center (HDSC) is to explore new application areas for computer assisted methods, especially by investigating the application requirements and product needs, developing and evaluating new concepts, thus contributing to the state of the art in the affected scientific areas. The theoretical or analytical studies are usually complemented by prototyping and by evaluating the main concepts in representative applications in order to gain further insight into the feasibility and applicability of the proposed solutions.

In 1979/80, based on previous research work in information retrieval /Sch78, KSW79, KW81/, the AIM department started to have a closer look on "integrated" applications in the office area and the adequate database support for it. As a first step, a prototype was developed combining information retrieval with relational /As76/ database technology. Though the general feasibility and applicability of this approach could be successfully demonstrated, the data model ("flat" relations) was felt to be too poor for an adequate treatment of structured text documents. Looking for appropriate solutions of this problem led - independently from other groups like VERSO /BRS82/ - to the idea of *nested relations* which were called NF^2 Relations (Non First Normal Form Relations). The most critical point was whether the extended relational model could be put on an equally sound theoretical basis as the original ("flat" relational) one. Consequently, at first we concentrated primarily on the theoretical issues of this data model, especially on its relationship to the relational design theory (functional and multi-valued dependencies). This led to scientific contributions by the HDSC to the theory of nested relations /Jae82, JS82, Jae85, Jae85a/. In parallel to this basic research work, conceptual work was started aiming to develop an extended SQL-like database language able to deal with the extended relational data model at the user's level /SP82, HHP82, PHH83, GP83/.

In 1982/83 the issue of "integration" of applications across formerly isolated application areas like office, manufacturing, engineering design, etc. and the understanding of related database requirements and problems became more and more important. We, therefore, decided to redirect our research and development activities and to look at database related issues on a broader scope. To be able to evaluate advanced database concepts in real (existing) advanced applications, we decided to develop an experimental type of a "next generation" DBMS - the **Advanced Information Management Prototype (AIM-P)** - as a vehicle for our research and development. The decision to use the NF^2 relations as base technology of the system was significantly influenced by our expectation of how the database supported integration of applications will take place. It did not only affect the selection of the data model for AIM-P (which goes beyond NF^2, see Sect. 3) but also the system's functionality and architecture. Hence, in the next section we describe at first our view of a "general purpose" DBMS for "large scale integration" of application areas and the resulting decisions for its design, and thus also for our main research areas.

2. Target and Scope of Research and Development

Up to now the situation in the area of data management is still characterized by having special purpose data management or file systems for every major application area. These systems differ in functionality, data representation ("data model", interfaces), real-time behavior (immediate update versus batch update), transaction management, recovery, and security aspects, thus making the required integration of applications a difficult task. Powerful database management systems handling the data across the different application areas in a uniform and consistent way could improve this situation drastically. Very likely, no single type of system will be able to adequately support all application types one can think of. However, to reach at least a coverage of say 80 to 90 percent of the main application types would already be a big gain. The target of the AIM research and development is to understand how closely this goal can be approached.

For that end, one prerequisite is that the DBMS supports both a *data oriented* (e.g. relational) as well as an *object oriented* view on the data. Probably one and the same kind of data (e.g. representing a computer board) may be accessed in an object oriented fashion by one application (e.g. board design) and in a data oriented fashion (e.g. bill of material) by another. The support of this *integrated view* on data significantly influenced our design decisions and research targets for the development of AIM-P. The most important ones can be summarized as follows:

Architecture: The DBMS should logically consist of two parts: The (logically) central **database server** and user/application front-ends, called **workstations** in our terminology. Special emphasis should be given to provide adequate data exchange mechanisms between server and workstation in order to keep the communication overhead low, especially in cases where large complex objects are involved. In other words, the overall architecture should support efficient **cooperative processing** - in particular of complex objects - in a workstation - server environment.

Database Language/Data Model: The *database server* should provide a homogeneous view on all the data (from flat relations to complex objects) to serve as "integration tool". That is **complex objects** should not be treated as "special animals" but be an integral part of the data model. All (or nearly all) operations defined for "normal" ('flat') data should be applicable to complex object data as well. The server should have a relational-like data model with set oriented, descriptive query capabilities to reduce the communication overhead between server and workstation. The *workstation* has to use this interface when requesting data ("check-out"). Which interface (data model) at the workstation is offered to the user or application program (the server's data model or e.g. flat relations, hierarchies, network structures) should be application dependent.

3. Heidelberg Data Base Language (HDBL)

The following description concentrates on the AIM-P data model and the relationship between HDBL and the operations of the NF^2 relational algebra. Due to lack of space, only a "flavor" of HDBL can be given here. A more comprehensive treatment of this issue can be found in /PT85, PT86/ and /PA86/. The currently supported language features are described in /ALPS88/.

3.1 Data Model

It is very easy to create a "monster" DBMS by trying to take the union of all nice data models and to put it into one DBMS. For AIM-P we decided to use a rather rigid (but powerful) data model based on NF^2 relations. The reason for this decision was the capability of the NF^2 data model to support both an object oriented and a data oriented view on the data, and because of its potential to treat complex objects and flat relations in a uniform way. However, the pure NF^2 model, only offering relations with relation valued attributes, is semantically too poor to provide a reasonable "general purpose" data model for the intended integration of a broad range of application areas. For engineering applications (but not restricted to them), an adequate representation of vectors and matrices (i.e. simple or nested lists) was missing. In addition, it is not possible to store *simple* objects (e.g. the highest invoice number used so far) in a *simple* fashion (i.e. not as a relation with one row and one column). We, therefore, decided to use an **extended** NF^2 data model for AIM-P.

This data model has the following basic types: **set** valued, **list** valued, **tuple** valued (composite), and **atomic**. Objects can be assembled from basic types in a fairly free fashion. Figure 1 illustrates the AIM-P data model and its relationship to the pure NF^2 and flat relational data model. Using this data model, a matrix of real numbers can be modelled, for example, as list of list of real, and this construct can either occur as attribute value within some list, set, or tuple valued object or subobject, or it can be used as singleton object identified by its own name. Atomic values and tuples can be used analogously.

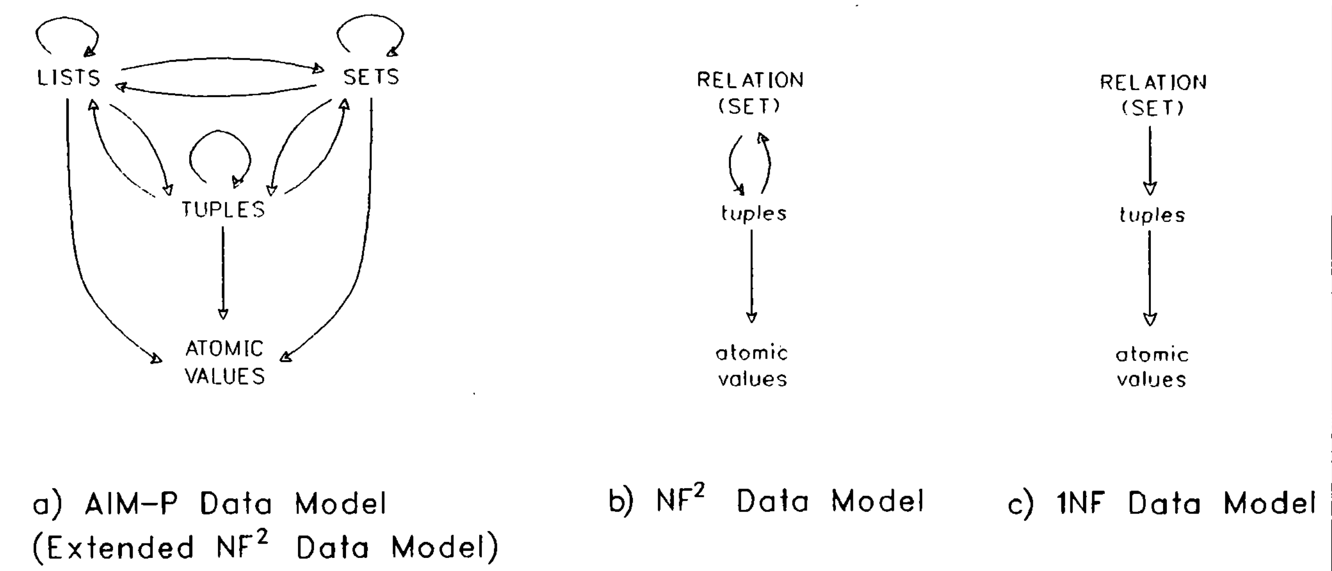


Figure 1. Comparison: AIM-P data model, pure NF^2, relational data model.

Though Figure 1 may create the impression that the *pure* NF^2 data model has become a very special case of the AIM-P data model, this is only true from a conceptual point of view. In the *usage* of the AIM-P data model, the concept of nested relations (unordered and ordered) turned out to be dominating in object definitions. Therefore, most emphasis in query processing has been put on supporting unordered and ordered nested relations efficiently.

In the sequel we concentrate on describing how NF^2 algebra operations are supported in AIM-P respectively in its database language (HDBL). Additional features also supported in HDBL respectively AIM-P like *time versions* (history data), *text*, and *user defined data types and functions* cannot described here because of space reasons (see Sect. 6 for references to the related literature).

3.2 Support of NF^2 Algebra Operations

HDBL is an SQL-like language developed at the HDSC to support the AIM-P data model. Like classical SQL it uses a SELECT...FROM...WHERE (SFW) construct to provide the facilities for expressing projections, restrictions, and joins. HDBL's SFW construct, however, is by far more powerful than the original one. It allows to dynamically construct nested tables out of flat tables (*nest*), to "flatten" a non-flat table into a flat one (*unnest*), or to restructure a nested table; and these operations are not only applicable to a relation (table) as a whole but also at the level of relation valued fields ("subrelations" /DK86/).

The latter effect is achieved by applying a SFW-expression to a relation valued field as illustrated in the following example (cf. Figure 2; the square brackets ([...]) in the query are *tuple constructors* /PT86/):

```
SELECT [x.DNO,
        Manuf_Cells: (SELECT [y.CID,
                              y.NC_MACH]
                      FROM   y IN x.MANUF_CELLS
                      WHERE  EXISTS (z IN y.NC_MACH): z.TYPE = 'Flex 200')]
FROM   x IN MANUF_DEPTS
```

For every manufacturing department, this query associates *DNO* with some information about those manufacturing cells (*MANUF_CELLS*) which have an NC machine (NC_MACH) of type 'Flex 200'.

If none of the department's manufacturing cells has such type of machine, the subrelation will be empty.

{ MANUF_DEPTS }								
DNO	DNAME	{ MANUF_CELLS }					{ STAFF }	
		CID	{ NON_NC_MACH }		{ NC_MACH }		ENO	FUNCTION
			QU	TYPE	QU	TYPE		
15	Shafts	C13	1	MLDX 300	1	NRP 5000	1217	NC Programmer
			1	MLDX 230	1	Flex 200	1494	NC Programmer
			1	Autex 77			1548	Supervisor
							1799	Supervisor
		C28	1	Varix 92	1	Speedy 5	1852	Laborer
			2	Varix 99	2	Preci 22	1877	Chief
			1	Autex 77			1938	Laborer
							1941	Laborer
22	Slabs	C11	2	MLDX 300	1	DSX 700	1199	Supervisor
			1	JRP 500	1	DSX 800	1292	Chief
			1	Autex 35			1385	NC Programmer
							1741	Laborer
							1855	Laborer

Figure 2. MANUF_DEPTS Information in NF^2 Representation

With the same subquery technique **nesting** can be formulated, too. Assume two flat tables

```
MDEPT (DNO, DNAME)
```

and

```
STAFF (DNO, ENO, FUNCTION).
```

Based on this input, the MANUF_DEPTS table could be partly (only DNO, DNAME, and STAFF) constructed using the following HDBL expression:

```
SELECT [x.DNO,
        x.DNAME,
        STAFF :  (SELECT [y.ENO,
                          y.FUNCTION]
                  FROM   y IN STAFF
                  WHERE  x.DNO = y.DNO)]
FROM    x IN MDEPT
```

Unnesting is formulated similar to a join. Assume we want to unnest the path from top to STAFF, while retaining the DNO, DNAME, ENO, and FUNCTION attributes (see Figure 2). This can be formulated in HDBL as follows:

```
SELECT [x.DNO,
        x.DNAME,
        y.ENO,
        y.FUNCTION]
FROM    x IN MANUF_DEPTS, y IN x.STAFF
```

Opposed to the strict NF^2 data model, HDBL is also capable to deal with lists (ordered relations). Therefore some additional rules are necessary in order to make predictable the result of a selection, projection, or join operation involving set-valued and list-valued tables. For HDBL, the following set of rules is applied during query processing (cf. /PT85/):

1. One unordered collection as input: Result is unordered.
2. One ordered collection as input: Result is ordered (processing observes order).
3. Join between lists: The result is ordered. Two nested loops may serve as a mental model for describing the iteration over the elements of the two operands. (This explains why a join of two lists is no longer commutative.)

4. Join between list and set: The result is ordered, but the sequence of the result elements is determined only to the extent that the relative order of the list elements is preserved. (Without preserving this order it would be impossible to recover input list elements from the join result in the right sequence.)
5. Join down an hierarchical path: Processing goes in parent-child sequence as described in the unnest example above. - Together with the previous rules on joins, unnesting semantics are well defined when lists are involved.

A more complex table - the ROBOTS table - which demonstrates some of the HDBL concepts which go beyond the pure NF^2 data model, and the corresponding CREATE statement are shown in Figure 3 and Figure 4. Besides relation valued attributes, the ROBOTS table shows also **list valued** attributes (AXES, DH_MATRIX) and **tuple valued** attributes (KINEMATICS, JOINT_ANGLE, DYNAMICS). *List valued* means that the values occurring e.g. in the AXES attribute are *ordered*. That is, there is a first axis, a second axis, etc. The *tuple valued* attribute DYNAMICS contains a composite attribute value, namely a value for MASS, and a value for ACCEL. Thus tuple valued attributes provide some structuring capabilities like the RECORD concept in many programming languages.

{ ROBOTS }								
ROB_ID	{ ARMS }						{ ENDEFFECTORS }	
	ARM_ID	< AXES >					EFF_ID	FUNCTION
		[KINEMATICS]			[DYNAMICS]			
		< DH_MATRIX >	[JOINT_ANGLE]		MASS	ACCEL		
			MIN	MAX				
Rob1	left	< 1, 0, 0, 1 > < 0, 0, 1, 0 > < 0, -1, 0,100 > < 0, 0, 0, 1 >	-180	180	50.0	1.0	E200 E150 E180	Gripper Welder Screw Driver
		< 1, 0, 0, 70 > < 0, 1, 0, 0 > < 0, 0, 1, 20 > < 0, 0, 0, 1 >	-250	60	37.25	2.0		
		< 0, 0, 1, 0 > < 1, 0, 0, 40 > < 0, 1, 0, -10 > < 0, 0, 0, 1 >	-80	250	10.4	6.0		
		< 0, -1, 0, 0 > < 0, 1, 0, 0 > < 0, 0, 1, 0 > < 0, 0, 0, 1 >	-180	180	2.0	6.0		
	right	.	.	.	.	.		
Rob2	left	.	.	.	.	.	.	.

Figure 3. ROBOTS Table: The attribute AXES contains a **list** (indicated by "<...>") of tuples. That is, it is **an ordered relation**. DH_MATRIX is also list valued, but the elements of this list are lists again. That is, it forms a list of lists (in this case a 4 x 4 matrix). KINEMATICS and DYNAMICS are **tuple valued** attributes (indicated by "[...]") of the (ordered) AXES relation. JOINT_ANGLE, in turn, is a tuple valued attribute of KINEMATICS.

To retrieve e.g. all robots which are able to use a 'Screw Driver' as endeffector and which have at least 2 arms, each of which having at least 4 axes, the following query could be issued:

```
SELECT  ro
FROM    ro IN ROBOTS
WHERE   COUNT(ro.ARMS) >= 2
          AND  (FORALL (ar IN ro.ARMS): COUNT(ar.AXES) >= 4 )
          AND  (EXISTS (ee IN ro.ENDEFFECTORS): ee.FUNCTION = 'Screw Driver')
```

```
CREATE robots
  {[ rob_id  :  STRING (6 FIX),
     arms    :
       {[ arm_id    :  STRING (12 FIX),
          axes      :
            <[ kinematics  :
                 [ dh_matrix     :  < 4 FIX < 4 FIX INTEGER >>,
                   joint_angle  :
                     [ min          :  REAL,
                       max          :  REAL ] ],
               dynamics    :
                 [ mass          :  REAL,
                   accel         :  REAL ] ] >,
     endeffectors :
       {[ eff_id    :  STRING (16 FIX),
          function  :  TEXT (1000) ]} ]}
```

Figure 4. **CREATE Statement for ROBOTS Table (Figure 3):** {...}, < ... >, [...] are set, list and tuple constructors (alternatively, set(...), list(...), and tuple(...) could be used).

4. Ad Hoc (On-line) and Application Program Interface

AIM-P supports an *on-line interface* for ad hoc queries as well as an *application program interface.* The **on-line interface** accepts input of HDBL statements (query, data manipulation, type and function definition), offers facilities for querying the catalogs (object, type, function), for editing and retrieving stored queries, and for browsing in query results[1].

The AIM-P **application program interface** (API) follows the same philosophy as the one supported by System R /Ch81/ respectively SQL/DS /IBM83/. An API *pre-compiler* takes *API language statements* embedded in the source code of the application program and translates them into respective subroutine calls to the *API run-time system* and appropriate type and variable declarations (mainly for parameter passing) of the target host programming language[2]. The API language constructs can be roughly divided into two groups: *declarative statements* and *operational statements.* Because of lack of space, we will only outline these language constructs with help of some selected examples. More detailed descriptions of this language can be found in /EW87, ESW87/.

The **declarative statements** are used to describe the database objects one wants to deal with (DECLARE RESULT), the application program variables which shall take values of the database objects (DECLARE DATA), and *cursors* for navigating on objects (DECLARE CURSOR), see Figure 5. In addition, there are also statements for exception handling (see Figure 6).

Cursors are declared using DECLARE CURSOR statements. In contrast to System R or SQL/DS, which support only 'flat' relations, AIM-P cursors are ordered in a hierarchy. In Figure 5 (based on the table shown in Figure 2) C_CURSOR and S_CURSOR are dependent on D_CURSOR, i.e. C_CURSOR can only operate on those manufacturing cells which belong to the manufacturing department on which D_CURSOR is currently positioned. Besides defining the scope for dependent cursors, a cursor also gives access to the data elements at its level. That is, D_CURSOR provides access to attributes DNO (atomic), DNAME (atomic), MANUF_CELLS (set valued), and STAFF (set valued); C_CURSOR provides access to attributes CID (atomic), NON_NC_MACH (set valued), and NC_MACH (set valued); etc.

---

[1] The HDBL facilities for *type definition* and *function definition* as well as the respective catalogs are related to the support of *user defined data types and functions* which are not described in this paper because lack of space. See /DKS88, LKDP88/ for more information on this subject.

[2] Currently only PASCAL is supported.

```
%INCLUDE celltup   /* embedding of PASCAL representation for CELLTUP */

    BEGIN DECLARE DATA;
      type
        staff_type = record
                       eno    : integer;
                       funct  : string(30)
                     end;

        staff_arr_type = array [1..50] of staff_type;

      var
        dno                  :  integer;
        dname                :  string(10);
        complex_cell_data    :  CELLTUP; /* variable of user defined data type CELLTUP */
        staff                :  staff_arr_type;
        staff_info           :  AIM_RESULT_DESCR;
    END DECLARE DATA;

    BEGIN DECLARE CURSOR;
      DECLARE RESULT manudept FOR UPDATE FROM QUERY_STATEMENT
        SELECT [ x.dno, x.dname, x.manuf_cells, x.staff ]
        FROM   x IN MANUF_DEPTS;

      DECLARE CURSOR  d_cursor                     WITHIN  manudept;
      DECLARE CURSOR  c_cursor   FOR  manuf_cells  WITHIN  d_cursor;
      DECLARE CURSOR  s_cursor   FOR  staff        WITHIN  d_cursor;
    END DECLARE CURSOR;
```

Figure 5. Examples of DECLARE statements

```
      WHENEVER END LEAVE;

      repeat
        MOVE d_cursor; /* on (next) manufacturing department */
        GET  d_cursor ATTR_WISE dno, dname INTO dno, dname;

        /* here processing of DNO and DNAME in the application program */

        repeat
          MOVE c_cursor; /* on (next) manufacturing cell */

          GET  c_cursor OBJECT_WISE INTO complex_cell_data;

          /* one manufacturing cell with all its non-nc machines     */
          /* and nc machines has now been transferred into           */
          /* program variable complex_cell_data                      */

          /* here processing of non-nc machines and nc machines data */

        until false;

        repeat
          MOVE s_cursor  /* on (next) staff member */
          GET  s_cursor BY 50 TUPLE_WISE INTO staff : staff_info;

          /* the actual number of retrieved staff tuples is returned in */
          /* staff_info.n_units_ret                                     */

          /* here processing of STAFF array in the application program */

        until false;


       /* here additional manuf. department related processing */

     until false;
```

Figure 6. Examples of MOVE and GET statements with object oriented data transfer

The **operational statements** are used to "drive" the API via the application program. In essence, there are query execution and update propagation related statements, cursor related statements, and session and transaction oriented statements. To open a session (to "connect to the database") a BEGIN SESSION statement providing a user identification and a password has to be issued. Transactions can subsequently be started with BEGIN TRANSACTION and closed using a COMMIT TRANS-

ACTION or ABORT TRANSACTION statement. Finally, a session can be closed via END SESSION.

The statement 'EVALUATE manudept' triggers both the execution of the query shown in Figure 5 and the materialization of the result[3]. On this result, cursors can be opened to transfer the data into application program variables (and vice versa). After having issued the statement 'OPEN CURSOR d_cursor' this cursor and all dependent cursors are open and can be positioned by MOVE statements.

For transferring data into application program variables, the GET statement is provided. An example for using these functions is given in Figure 6. The GET statements in this figure deserve some further comments; they demonstrate that *data transfer* can be performed in several ways:

- The option 'ATTR_WISE' specifies that the *atomic* attribute values accessible via the respective cursor shall be transferred into *individual* application program variables (in our example attributes DNO and DNAME are transferred into program variables DNO and DNAME).
- The option 'TUPLE_WISE' tells the system that *all atomic* attribute values at the current cursor position shall be taken as a unit (tuple) and be assigned to a corresponding (type compatible) record variable. The specification BY n ($n > 1$) triggers that more than one instance (tuple) at a time is transferred. - An optional variable of type AIM_RESULT_DESCR (see Figure 5 and Figure 6 ("staff_info")) can be used to tell the user how many data instances (tuples) have actually been transferred into the application program.
- The option 'OBJECT_WISE' means - in contrast to the keywords TUPLE_WISE and ATTR_WISE - that *all* the atomic *and* non-atomic data at the current cursor position shall be transferred into the application program: A complete complex object is transferred at a time (by a single call to the API run-time system). Of course, an appropriate program variable must be provided to deliver all these data, especially the non-atomic (set-valued, list-valued) data[4].

The WHENEVER clause in Figure 6 is used to specify **exception handling**. In this particular case, the handling of an END condition is specified (LEAVE is a PASCAL/VS statement to leave the current REPEAT ... UNTIL loop; cf. /IBM85/).

If a result has been declared FOR UPDATE as in Figure 5 then it can both be read and modified. To that end, the cursors have to be positioned in the same way as it is done for reading (GET, FETCH). UPDATE, INSERT, or DELETE statements (which are syntactically very close to the GET statement) can be issued to modify the data /EW87, ESW87/. After modification, the PROPAGATE statement is used to transfer the modified or new data from the workstation (back) to the database server. This, however, does not mean that the changes are already committed. At the server site, the modifications are only performed in the transaction's private workspace. It is still at the user's disposal to subsequently perform either a COMMIT or an ABORT.

5. *Summary, Status, and Outlook*

AIM-P is a research and development effort of the HDSC to provide a better understanding of database technology adequate for integrated applications. AIM-P, the research prototype under development at the HDSC, bases in its core on the concept of nested relations. The experiences from applications or analytical studies with extended NF^2 relations are very promising /GP83, PT85, KW87, DH88, Mi87/. Some areas are still not adequately covered in the current implementation, however, and therefore further developments of HDBL to support e.g. references for data sharing and recursive queries /Li87, Li87a/ are envisaged for the near future.

In this brief project description only few aspects of the overall AIM-P effort could be highlighted. Because of lack of space, the issue of **time version** support (which has been deeply integrated into AIM-P;

[3] The EVALUATE and OPEN CURSOR statements are not shown in Figure 6.

[4] Appropriate type declarations for these program variables (e.g. CELLTUP in Figure 5) can be obtained from AIM-P's *type compiler* in conjunction with the support of *user defined data types* (see /DKS88, LKDP88/ for details).

cf. /Lu84, DLW84/ and especially /Pi87/ for a detailed discussion of this aspect), the integration of **text** search capabilities, and the important aspect of **cooperative processing** in a workstation - server environment /DGKOW86, KDG87, Pi87/ had to be omitted completely. The same holds also for the support of **user defined data types and functions** which was the most significant effort during the last year in order to contribute to the R^2D^2 project[5] /DDKL87, DDKL88, DH88, DKS88, KLW87, KWD88, LKDP88/.

The first rather complete version of AIM-P became operational end of 1986. Since then the system has been installed in a couple of places inside and outside of IBM for research and evaluation purposes. The current implementation status (Release 2.0) can be summarized as follows:

- Support of flat and nested relations, both unordered and ordered. Legal attribute types are atomic, flat and nested relations, and lists or sets of atomic values. Sets of sets or lists of lists are not supported yet.
- Large subset of HDBL operational but only rudimentary query optimization performed so far; also view support is still missing.
- Access to historical data in ASOF fashion /DLW84/.
- Access to HDBL facilities both in on-line mode and via application program interface.
- Support of user defined data types and functions
- Support of textual data (text search capabilities)
- Workstation - server support
- Basic transaction support (abort/commit) in a single-user environment.

Work has been started to improve AIM-P's query processing by integrating indexes for extended NF^2 tables, and to develop rules for query transformation and optimization. In addition, work has been started on sorting and duplicate elimination (which shall also provide the basis for the envisaged support of recursive queries). Our main target, however, remains to understand the database *requirements* in advanced, integrated application areas (cf. Sect. 2). We therefore will increase the number of case studies performed in such areas using our prototype. Direction and emphasis of our future research work will - as in the past - be heavily influenced by the requirements and open problems discovered there.

Many research groups are dealing with the theory of nested relations at present in the one way or another (cf. e.g. /SS87, OOM87/), and a lot of interesting results have been achieved already. However, as already pointed out in Sect. 3, the *pure* NF^2 data model seems to be too limited for being directly useful as data model for an advanced DBMS. Therefore, the development of commercial systems using pure NF^2 relations as data model is not very likely. Opposed to that, systems based on a semantically richer (extended) NF^2 model like that of AIM-P, for example, look very promising. Systems based on such a data model seem to have a realistic chance. Hence, in order to make theoretical work in NF^2 relations more beneficial for the development of "real" systems, it should broaden its scope to support a more general data model like the one outlined above, for example.

Acknowledgements

The development of AIM-P was (and is) a cooperative effort of IBM scientists, visiting scientists, and students. At present, the AIM-P group consists of P. Dadam, R. Erbe, U. Herrmann (ds[6]), U. Keßler (vs[6]), K. Küspert, V. Linnemann, P. Pistor, E. Roman, G. Saake (vs), and N. Südkamp. Prior project members who have made major contributions to the AIM-P development are V. Lum who managed the project from September 1982 to August 1985, J. Günauer, and G. Walch as well as our visiting scientists F. Andersen, H.-D. Werner, and J. Woodfill. The basic research work performed by H.-J. Schek (manager of the AIM department prior to V. Lum) and G. Jaeschke prior to this project has

[5] R^2D^2 stands for **R**elational **R**obotics **D**atabase with Extensible **D**ata Types and is a joint research project with the robotics and database research groups at the University of Karlsruhe and the AIM group at the Heidelberg Scientific Center.

[6] vs = visiting scientist, ds = doctoral student

provided the stimulating factor and was an important part of the theoretical basis for the whole development effort.

Also joint research with partners at the *University of Darmstadt* in the area of *workstation - server cooperation* /KDG87, DGKOW86, DO87/, at the *University of Hagen* on *engineering design version support* /KSW86, Wi87/, and with our partners in the R^2D^2 project (*University of Karlsruhe*, see above) have influenced and contributed to our work. In addition, a lot of valuable conceptual work for AIM-P or AIM-P related problems has also been performed by our visiting scientists H. Blanken, B. Hansen, M. Hansen, M.R. Scalas, H.-J. Schneider, J. Teuhola, R. Traunmüller, H. Wedekind, and L. Wegner. All these contributions are gratefully acknowledged.

The author wants to thank K. Küspert and P. Pistor for carefully reading an earlier version of this paper and valuable suggestions which helped to improve the presentation. Thanks for their comments also to V. Linnemann, E. Roman, and G. Saake.

6. References

ALPS88	F.Andersen, V.Linnemann, P.Pistor, N.Südkamp: Advanced Information Management Prototype: User Manual for the Online Interface of the Heidelberg Data Base Language (HDBL) Prototype Implementation. Technical Note TN 86.01, Heidelberg Scientific Center, Febr. 1988
As76	M.M.Astrahan et al.: System R: Relational Approach to Database Management. ACM TODS, Vol 1(2), June 1976, pp. 97-137
BRS82	F.Bancilhon, P.Richard, M.Scholl: On Line Processing of Compacted Relations. Proc. VLDB 82, Mexico, September 1982, pp. 263-269
Ch81	D.D.Chamberlin et al.: Support for Repetitive Transaktions and Ad Hoc Queries in System R. ACM TODS, Vol. 6, No. 1, March 1981, pp. 70-94
DDKL87	P.Dadam, R.Dillmann, A.Kemper, P.C.Lockemann: Objektorientierte Datenhaltung für die Roboterprogrammierung. Informatik Forschung und Entwicklung, Springer-Verlag, Heidelberg, Vol. 2, 1987, pp. 151-170
DDKL88	P.Dadam, R.Dillmann, A.Kemper, P.C.Lockemann: Object-Oriented Databases for Robot Programming. In /KM88/, pp. 289-303
DGKOW86	U.Deppisch, J.Günauer, K.Küspert, V.Obermeit, G.Walch: Überlegungen zur Datenbank-Kooperation zwischen Server und Workstations (Considerations About the Cooperation Between Database Server and Workstations). Proc. 16th GI Jahrestagung, Berlin, October 1986, Springer-Verlag, Informatik Fachberichte No. 126, pp. 565-580 (in German)
DH88	R.Dillmann, M.Huck: R^2D^2: An Integration Tool for CIM. In /KM88/, pp. 355-372
DK86	P.Dadam, K.Küspert, F.Andersen, H.Blanken, R.Erbe, J.Günauer, V.Lum, P.Pistor, G.Walch: A DBMS Prototype to Support Extended NF^2-Relations: An Integrated View on Flat Tables and Hierarchies. Proc. ACM SIGMOD Conf., Washington, May 1986, pp. 356-367
DKS88	P.Dadam, Küspert, N.Südkamp, R.Erbe, V.Linnemann, P.Pistor, G.Walch: Managing Complex Objects in R^2D^2. In /KM88/, pp. 304-331
DLW84	P. Dadam, V. Lum, H.-D. Werner: Integration of Time Versions into a Relational Database System. Proc. VLDB 84, Singapore, Aug. 1984, pp. 509-522
DO87	U. Deppisch, V. Obermeit: Tight Database Cooperation in a Server- Workstation Environment. Proc. 7th Int. Conf. on Distributed Computing, Berlin, Sept. 1987, pp. 416-423
ESW87	R.Erbe, N.Südkamp, G.Walch: An Application Program Interface for a Complex Object Database. Proc. 3rd Int. Conf. on Data and Knowledge Bases, Jerusalem, Israel, June 1988 (also available as Technical Report TR 87.10.008, Heidelberg Scientific Center, Oct. 1987)
EW87	R.Erbe, G.Walch: An Application Program Interface for an NF^2 Database Language or How to Transfer Complex Object Data into an Application Program. Heidelberg Scientific Center, Technical Report TR 87.04.003, April 1987
GP83	L.Gründig, P.Pistor: Landinformationssysteme und ihre Anforderungen an Datenbankschnittstellen. In /Sch83/, pp. 61-75
HHP82	B.Hansen, M.Hansen, P.Pistor: Formal Specification of the Syntax and Semantics of a High Level User Interface to an Extended NF2 Data Model (unpublished, 1982)
IBM83	SQL/Data System, Application Programming, IBM Corporation, SH24-5018-2, Aug. 1983
IBM85	PASCAL/VS Language Reference Manual, IBM Corp., Program No. 5796-PNQ, 3. Edition, 1985
Jae82	G. Jaeschke: An Algebra of Power Set Type Relations. Technical Report TR 82.12.002, Heidelberg Scientific Center, Dec. 1982

Jae85	G. Jaeschke: Nonrecursive Algebra for Relations with Relation Valued Attributes. Technical Report TR 85.03.001, Heidelberg Scientific Center, March 1985
Jae85a	G. Jaeschke: Recursive Algebra for Relations with Relation Valued Attributes. Technical Report TR 85.03.002, Heidelberg Scientific Center, March 1985
JS82	Jaeschke, G., Schek, H.-J.: Remarks on the Algebra of Non First Normal Form Relations. Proc. ACM SIGACT-SIGMOD Symp. on Principles of Data Base Systems, Los Angeles, Cal., March 1982, pp. 124-138
KDG87	K. Küspert, P.Dadam, J.Günauer: Cooperative Object Buffer Management in the Advanced Information Management Prototype. Proc. VLDB '87, Brighton, U.K., Sept. 1987, pp. 483-492
KLW87	A.Kemper, P.C.Lockemann, M.Wallrath: An Object-Oriented Database System for Engineering Applications. Proc. ACM-SIGMOD, San Francisco, May 1987, pp. 299-311
KM88	G.Krüger, G.Müller (eds.): HECTOR, Heterogenous Computer Together, Volume II, Basic Projects, Springer-Verlag, 1988
KSW79	Kropp, D., Schek, H.-J., Walch, G.: Text Field Indexing. Proc. Meeting of the German Chapter of the ACM on Data Base Technology (J. Niedereichholz, ed.), Bad Nauheim, West Germany, Sept. 1979, Teubner-Verlag, Stuttgart, pp. 101-115
KSW86	P.Klahold, G.Schlageter, W.Wilkes: A General Model for Version Management in Databases. Proc. VLDB 87, Kyoto, Japan, August 1986, pp. 319-327
KW81	D.Kropp, G.Walch: A Graph-Structured Text-Field Index Based on Word Fragments. Information Processing and Management, Vol. 17(6), 1981, pp. 363-376
KW87	A.Kemper, M.Wallrath: An Analysis of Geometric Modelling in Database Systems. ACM Computing Surveys, Vol. 19, No. 1, March 1987, pp. 47-91
KWD88	A.Kemper, M.Wallrath, M.Dürr: Object Orientation in R^2D^2. In /KM88/, pp. 332-354
LKDP88	V.Linnemann, K.Küspert, P.Dadam, P.Pistor, R.Erbe, A.Kemper, N.Südkamp, G.Walch, M.Wallrath: Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions. Proc. VLDB 88, Los Angeles, Aug./Sept. 1988 (also available as: Technical Report TR 87.12.011, Heidelberg Scientific Center, Dec. 1987)
Li87	V.Linnemann: Non First Normal Form Relations and Recursive Queries: An SQL-Based Approach. Proc. 3. IEEE Intern. Conf. on Data Engineering, Los Angeles, Feb. 1987, pp. 591-598
Li87a	V.Linnemann: Optimization of Recursive Queries Over Nested Relations by a Differential Technique. Technical Report TR 87.07.005, Heidelberg Scientific Center, July 1987
Lu84	V.Lum, P.Dadam, R.Erbe, J.Günauer, P.Pistor, G.Walch, H.-D.Werner, J.Woodfill: Designing DBMS Support for the Time Dimension. Proc. SIGMOD 84 Conf., Boston, Mass., June 18-21, pp. 115-130
Mi87	M.Mitchell, National Bureau of Standards, Automated Manufacturing Research Facility (AMRF), Gaithersburg (private communication, Heidelberg, Jan. 1987)
OOM87	G.Ozsoyoglu, Z.M.Ozsoyoglu, V.Matos: Extending Relational Algebra with Set-Valued Attributes and Aggregate Functions. ACM TODS, Vol. 12, No. 4, Dec. 1987, pp. 566-592
PA86	P.Pistor, F.Andersen: Designing a Generalized NF^2 Model with an SQL-Type Language Interface. Proc. VLDB 86, Kyoto, Aug. 1986, pp. 278-288.
PHH83	P. Pistor, B. Hansen, M. Hansen: Eine sequelartige Sprach-Schnittstelle für das NF2 Modell (an SQL-like query interface for the NF2 model). In /Sch83/, pp. 134-147 (in German)
Pi87	P.Pistor: The Advanced Information Management Prototype: Architecture and Language Interface Overview. Invited talk, Proc. 3. Journées Bases de Données Avancées, Port-Camargue, France, May 1987. pp. 1-20
PT85	P.Pistor, R.Traunmüller: A Database Language for Sets, Lists, and Tables. Technical Report TR 85.10.004, Heidelberg Scientific Center, Oct. 1985
PT86	P.Pistor, R.Traunmüller: A Database Language for Sets, Lists, and Tables. Information Systems, Vol. 11(4), 1986, pp. 323-336
Sch78	Schek, H.-J.: The Reference String Indexing Method. Proc. Information Systems Methodology (G. Bracchi, P.C. Lockemann, eds.), Venice, Italy, 1978, Lecture Notes in Computer Science 65, Springer-Verlag, pp. 432-459
Sch83	J.W. Schmidt (ed.): Sprachen für Datenbanken. Informatik Fachberichte 72, Springer-Verlag, 1983
SP82	H.-J.Schek, P.Pistor: Data Structures for an Integrated Database Management and Information Retrieval System, Proc. VLDB Conf. Mexico, Sept. 1982
SS87	M.H.Scholl, H.-J.Schek (eds.): Theory and Applications of Nested Relations and Complex Objects, Int. Workshop, Darmstadt, Germany, April 1987 (Workshop Material)
Wi87	W.Wilkes: Der Versionsbegriff und seine Modellierung in CAD/CAM Datenbanken (The Notion of Versions and its Modelling in CAD/CAM Databases. Doctoral dissertation, University of Hagen, Dept. of Mathematics and Computer Science, September 1987 (in German)

From simple to sophisticate languages for complex objects

Serge Abiteboul, Michel Scholl
I.N.R.I.A.

August 2, 1988

1 Introduction

The popularity of relational systems has raised a major demand from fields like computer aided design or office automation for such database technology. Unfortunately, current relational systems are not suited for these applications. One of the main limitations is the simplistic data structure of the relational model: the table. A major trend in the database field has been to incorporate more powerful data structures like "nested relations" or "complex objects" [NRCO].

We have been involved for several years in system design and theory for nested relations and complex objects. The purpose of the present paper is to briefly describe the evolution of our research and draw some general conclusions. We will more particularly discuss the following three points:

1. **The Verso system:** the Verso system [V] has been developed from 80 to 86. The first objective was to validate the notion of filtering on the fly based on automata techniques. The physical organization (in hierachical files) suggested the use of a logical organisation (V-relational model) based on nested relations. The manipulation language is an algebra [ABi].
2. **Power of languages:** in [ABe], various languages for complex objects are considered. The main result of that research is that an algebra, a calculus, and ruled-based language for complex objects are shown to be equivalent.
3. **Rule-based language:** we are currently working on a complex object language (called Col) [AG1,AH3]. The proposal shares with [B+,K2] the goal of extending Datalog to complex objects. The particularity is that it tries to integrate aspects from logic programming and functional programming.

It should be noted that database systems based on complex objects are still at the level of prototypes, and that the field has had no major impact yet on industry. However, the need for such systems is obvious. We believe that the main reason for this discrepancy is that languages for complex objects were to primitive. Tables were appropriate for business applications, so were first-order query languages. It was correct to distinguish the need of set/tuple-oriented data structures for the new database applications. However, there was also a need of more user-friendly extensible manipulation languages for these applications. This may be the reason for the time it takes to move the complex objects technology to industry. This does not mean that the concepts that were developed were wrong. It only indicates that more work is needed to improve interfaces to such systems.

In the development of the Verso system, two issues were greatly underestimated:

- **hardware development:** First, a hardware filter was considered. By the time it was designed and realized, the project was almost over, and the systems specifications were so different that the hardware filter was never connected to the system. Although much faster than the software filter (one order of magnitude), the hardware filter was certainly not cost effective.
- **software development:** In designing the Verso system, many tasks like concurrency control or recovery were not directly connected to the ideas that were to be validated. Also, the screen interface which was an essential component required a lot of effort.

The advent of packages like intelligent file servers certainly alleviate the pain of designing a full DBMS. Also, the availability of interface generators will greatly simplify the global task.

After gaining some experience with the development of a database system based on nested relations (1), we have investigated the interface to complex objects data bases. In particular, we have developed a graphical interface to the Verso system. We have studied various approaches such as algebraic, calculus and rule-based (2). The powers of these various approaches are compared in [ABe] (3). One conclusion of this work was that the rule-based paradigm was very convenient in a complex object context. "Convenience" is something that is hard to evaluate. The main advantage is in the use of constructed terms and the rewriting of terms. Recursion which is usually emphasized in languages like Datalog is of secondary importance. Indeed, recursion does not seem to be central to most applications.

The paper is organized as follows. The next three sections treat respectively points (1), (2) and (3). Some general conclusions together with directions of future research are given in the last section. Due to space limitations, our presentation will be informal and very general.

2 The Verso system

The major motivation behind the development of the Verso system was to increase the performance of relational DBMSs by using on-the-fly filtering.

To take full advantage of this filtering capability, it was decided to store data in hierarchical structures. This physical organization strongly suggested a logical data organization into nested relations called V-relations.

Nested relations have been widely accepted and studied independently by several researchers (see this issue) However, the Verso system was, to our knowledge, the first implemented system based on nested relations.

The query language is algebraic. The filter can be viewed as a finite state automaton (FSA) which scans sequentially one or two input buffers, and writes the result of the operation on an output buffer. It was shown in [BS,S] that this automaton-like device is sufficient to perform on the fly all algebraic operations except for restructuring which involves some sorting, and cannot be realized uniquely by the filter. The performance of the system thereby depends heavily on the performance of the filter.

2.1 Architecture

A version of the system, runs under the Unix operating system. Prototypes have already been experimented on a 68000 based multiprocessor machine, the SM90. Most of the code is written in Pascal.

The machine includes two processors (CPU and filter) which share RAM and a disk hosting the Unix system and the programs (Figure 1).

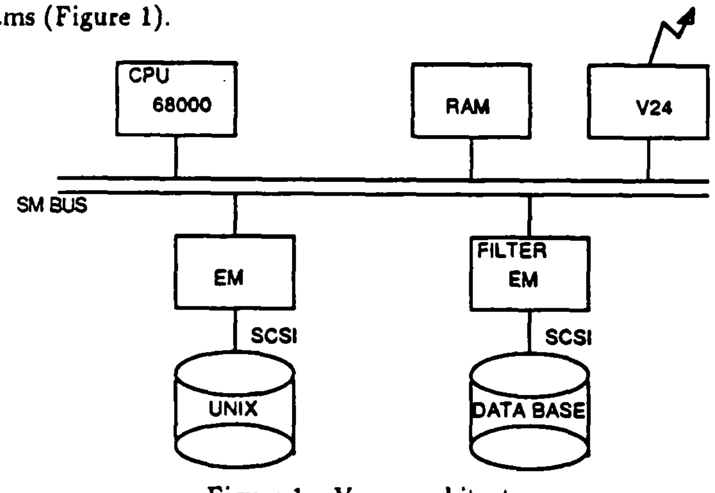


Figure 1 : Verso architecture

The CPU (Motorola, 68000 series) is in charge of the user interface, the high level DBMS layers (including transaction management and concurrency control) and the filter's control: it sends to the filter data transfer and filtering commands.

A specialized hardware processor was first designed to realize the FSA filter [V] and developed by the Inria SCD team . This hardware processor, connected to the mass storage as well as to the central bus was in charge of data transfer and data filtering. Later on, the hardware filter was abandoned and replaced by a standard disk exchange module including an Intel 8086 processor on which filtering is implemented by software.

The filter performs the algebraic operations on files which physically represent V-relations.

In the following section, we describe the Verso data model . A thorough presentation of the system may be found in [V]. Except for the use of a filter and for the model of V-relations, the Verso system is a quite standard system: the data is stored in relations contained in databases; concurrency is offered via the concept of transaction, and managed using two phase locking; mechanisms for handling crash recovery are provided, as well as simultaneous access from separate sites.

2.2 The Model

In this section, we briefly describe the Verso data model. We first describe the data structure called V-relation. We then present the Verso algebra. A presentation of the model, together with some basic results on V-relations can be found in [ABi].

In the Verso data model, the data is organized in V-relations. An example of V-relation is given in Figure 2. The first line of the figure represents the structure or *format* of the V-relation.

A simple algebra can be defined for V-relations. As mentioned above, all algebraic operations but restructuring can be computed by the filter. This operation involves some sorting. Thus, the complexity of main memory computation is restricted to a unique module, namely the sorter.

MOVIE	(THEATER	(TIME)*)*	(ACTOR)*
straw dogs	rex	18	hoffman
		20	georges
		22	vaughan
	chinese	19	
		20:30	
		23	
metropolis	studio	16	helm
		18	abel
			frolich
pierrot le fou	studio	20	belmondo
		22	carina
karate III	studio	16	

Figure 2: example of V-relation

The algebra consists of unary and binary operations. The unary operations are projection, selection, renaming, and restructuring. The binary ones are join, union, and difference. Examples of unary operations are now given. The queries are expressed here in natural language.

Example 2.1 The following projection- selection can be performed on the MOVIE database: "who are the actors playing in a movie shown at the Rex between 7:30 and 8:30 featuring J.P. Belmondo and A. Karina, The language Col is based on a clausal logic. The database consists of facts concerning both the predicates, and the data functions. New facts can be deduced using rules. The use of monovalued data functions in Col yields consistency problems which are studied in [AH3] where a compile-time sufficient criterium for consistency is exhibited.

A Col program consists of facts and rules. Its semantics is given in terms of minimal models as it is standard for instance in Datalog, or in Datalog with negation. Unfortunately, because of sets and data functions, some programs may have more than one minimal model. A stratification in the spirit of the stratification introduced by [ABW,vG,N] is used. It is shown in [AG1] that for stratified programs, a canonical model of a given program can be computed using a sequence of fixpoints of operators.

We are currently implementing the Col language in *CAML* [CH] a functional language of the *ML* family. The goal is to validate the various features of Col. Furthermore, the goal is to obtain an "extensible" language. More precisely, we want to be able in the same session to define an aggregate function in *CAML* and use it in Col. We are planning to consider a data intensive application such as geometrical databases which requires both complex data and computational aspects.

Other issues will also be addressed such as:

1. query optimization,
2. updates and their semantics, and
3. the introduction of some inheritence mechanisms.

5 Conclusion

We presented our past and present work on complex objects. With respect to more classical relational DBMS designs, Verso's major novel features are the following:

- Data is organized in nested relations. This is a first step toward the use of more complex structures, while keeping the advantages of the relational model (e.g. an algebraic language).
- It includes a filter in charge of all algebraic operations except for restructuring. This automaton-like filtering mechanism is well adapted to processing of both unary, and binary algebraic operations. Furthermore, the filter is also used for providing fast updates.
- Although by using dedicated hardware for filtering, one should gain one order of magnitude on response time, standard microcomputers have a performance that increases rapidly with time. For that reason, we believe that the use of "off-the-shelf" components for filtering should be preferred to a time-consuming and costly design of dedicated hardware.

The evolution clearly indicates two tendencies that can also be noticed in the field at large:

- the data structures are more general: restriction on the use of set and tuple constructors of the nested model are removed, heterogeneous sets are introduced, and
- the languages become more elaborate: rule-based, recursion, language extensibility.

The problem of performance clearly remains a key issue. The influence of the logic programming field, and the object-oriented paradigm should also become more and more important in the world of complex objects. The challenge is clearly to incorporate the nice features from these two disciplines.

Complex objects is a field where it is quite hard to evaluate improvements. There is a real lack of benchmarks:

- First, for performance evaluation. The Verso system was tested against benchmarks for pure relations, and
- Next, for "user friendliness". There is no way to compare language proposals.

Acknowledgements: This paper is the description of the work of a group. The following people participated is the design and development of the Verso system, F. Bancilhon, N. Bidoit, V. Delebarre, S. Gamerman, J.M. Laubin, M. Mainguenaud, T. Mostardi, P. Pauthe, D. Plateau, P. Richard, and A. Verroust. D.Tusera and D.Fortin realized the hardware filter.

Some of the research described here also involved C. Beeri from Hebrew University, and R. Hull from USC. Finally, the following people are involved in the design of Col: S. Grumbach, A. Voisard and E. Waller.

6 References

[ABe] Abiteboul, S., C. Beeri, On the power of languages for the manipulation of complex objects, abstract dans les proc. International Workshop on Theory and Applications of Nested Relations and Complex Objects, Darmstadt RFA, (1987) Inria internal report (1988).

[ABGG] Abiteboul, S., C. Beeri, M. Gyssens, D. van Gucht, Completeness of Languages for Complex Objects, submitted to publication (1988).

[ABi] Abiteboul, S., N. Bidoit, Non first normal form relations: An Algebra Allowing Data Restructuring, *Journal of Computer and System Sciences* (Dec. 86), extended abstract in proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems. (1984)

[ABW] Apt, K., H. Blair, A. Walker, Toward a theory of declarative knowledge, Foundations of Deductive Databases and Logic Programming, (J. Minker ed.) (1988).

[AH1] Abiteboul, S., R. Hull, IFO: A formal semantic database model, extended abstract in proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems (1984), to appear in *ACM Transactions on Database Systems*

[AH2] Abiteboul, S., R. Hull, Restructuring of semantic database objects, proc. Int. Conf. on Theory of Databases, Rome, Sept. (1986) to appear in *Theoretical Computer Science*

[AH3] Abiteboul, S., R. Hull, Data Functions, Datalog, and Negation, Proc. of the ACM SIGMOD Conference on the Management of Data, (1988)

[AG1] Abiteboul, S., S. Grumbach, COL: a logic-based language for complex objects, proc. Intern. Conf. on Extending Data Base Technology (1988)

[AG2] Abiteboul, S., S. Grumbach, Manipulation d'objets complexes, in *Technique et Science de l'informatique* (1987).

[AU] Aho, A.V., J.D. Ullman, Universality of data retrieval languages, proc. Symp. on Principles of Programming Languages (1979)

[BS] Bancilhon, F., M. Scholl, Design of a Backend Processor for a Database Machine, proc. of ACM-SIGMOD, Santa Monica (1980)

[BRS] Bancilhon, F., P. Richard, M. Scholl, Verso: A Relational Back End Data Base Machine, proc. of the Int. Workshop on Database Machines, San Diego, (1982)

[B+] Beeri, C., S. Naqvi, R. Ramakrishnan, O. Schmueli, S. Tsur, Sets and Negation in a Logic and Database Language (LDL1), proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems (1987)

[BF] Buneman, P., R.E. Frankel, FQL - a Functional Query Language (preliminary report) proc. ACM SIGMOD conf. on Management of Data (1979).

[CH] Cousineau, G., G. Huet, The CAML Primer, Inria report (1988).

[D] Dadam, P., History and Status of the Advanced Information Management Prototype (extended abstract), proc. International Workshop on Theory and Applications of Nested Relations and Complex Objects, Darmstadt RFA, (1987)

[FT] Fischer, P., S. Thomas, Operators for Non-First-Normal-Form Relations, proc. of the 7th Int. Computer Software Applications Conference, Chicago (1983)

[GS] Gamerman, S., M. Scholl, Hardware versus Software Data Filtering: The Verso Experience, proc. of the 4th International Workshop on Database Machines, Grand Bahama Island (1985)

[H] Hull, R., A Survey of Theoretical Research on Typed Complex Database Objects, U.S.C. Computer Science Technical Report (1986)

[HK] Hull, R., R. King, Semantic database modeling: Survey, applications, and research issues. U.S.C. Computer Science Technical Report (1986)

[J] Jacobs, B.E., On database logic, *Journal of the ACM* 29(2) (1982)

[JS] Jaeschke, B., H.J. Schek, Remarks on the algebra of non first normal form relations, proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, Los Angeles (1982)
[Kl] Klug, A., Equivalence of relational algebra and calculus query languages having aggregate functions. *Journal of the ACM* 29(3) (1982)
[K1] Kuper, G.M., M.Y. Vardi, A new approach to database logic, proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems (1984)
[K2] Kuper, G.M., Logic Programming with Sets, proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems (1987)
[L] Lloyd, J., Foundations of logic programming, *Springer Verlag* (1984)
[M] Makinouchi, A., A consideration on Normal Form of Not-Necessarily-Normalized Relation in the Relational Data Model, proc. of the Third Int. Conf. on Very Large Data Bases, Tokyo (1977)
[N] Naqvi, S., a logic for negation in database systems, Proc. Workshop on Logic Databases (1986).
[NRCO] Proceedings of International Workshop on Nested Relations and Complex Objects, Darmstadt (1987). Selected papers to appear in Springer-Verlag.
[P] Pauthe, P., EVER, un e'diteur de V-relations, The'se de Doctorat, Universite' d'Orsay (1985)
[RKS] Roth, M.A., H.F. Korth, A. Silberschatz, Extended Algebra and Calculus for non 1NF Relational Databases, Technical Report, Department of Computer Science, University of Texas at Austin (1985)
[SP] Scheck H-J., P. Pistor, Data Structures for an Integrated Database Management and Information Retrieval System, proc. of conf. on Very Large Data Bases, Mexico (1982)
[SS] Scheck H-J., M.H. Scholl, The Relational Model with Relation-Valued Attributes, *Information Systems* (1986)
[S] Scholl, M., Architecture pour le Filtrage dans les Bases de Donnees Relationelles, These d'etat, INPG, Grenoble (1985)
[TF] Thomas S.J. , P.C. Fischer, Nested Relational Structures, in *The Theory of Databases,* JAI press, P. Kanellakis ed.(1983)
[U] Ullman, J., Principles of Database Systems, Computer Science Press (1983).
[V], Verso, J., VERSO: a Database Machine based on non 1NF Relations, Rapport de Recherche Inria 523 (1986)
[vG] van Gelder, A., Negation as failure using tight derivations for general logic programs, Proc. IEEE symposium on Logic Programming (1986).

The Data Model and Query Language of LauRel

Per-Ake Larson

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2J 3G1

ABSTRACT

LauRel is a prototype database system being developed at the University of Waterloo. LauRel, and its query language SQL/W, are based on an extended relational model. This paper gives an overview of the language facilities supporting set valued attributes (nested relations) and reference attributes (pointers).

1. Introduction

The data model and query language supported by a database system determine the functionality of the system and thereby its application domain. In the area of administrative data processing, the relational model and SQL have rapidly gained widespread acceptance. The relational model and SQL are simple, yet powerful, at least compared with older approaches. However, it has become apparent that the relational model has serious shortcomings, even for traditional database applications. The basic problem is its limited modelling capability; being forced to model everything as (flat) relations containing only atomic attributes often leads to awkward database designs and unnecessarily complicated queries. Needless to say, this is not a new observation and several proposals for extending the relational model have been reported in the literature.

LauRel is a prototype database system being developed at the University of Waterloo. The system will support an extended relational model and an SQL-based language called SQL/W. It will also serve as a testbed for research on parallelism in database systems and is being designed to run on a shared-memory multiprocessor system. This paper gives a brief overview of the data model and the query language of LauRel, focusing on the support for set valued attributes (nested relations) and reference attributes (pointers, foreign keys). The design has not yet been completely finalized and may still be modified. Several other additions to the data model are being considered.

The design of LauRel and SQL/W has been influenced by, and borrows freely from other proposed data models and query languages. Readers familiar with the literature will recognize the result as a synthesis and extension of ideas from many sources, including GEM [TZ, Za], NF^2 data models [RKB, Dad, SS], EXODUS [CDV], Iris [Fi], POSTGRES [RS], and DAPLEX [Sh]. Although the concepts used are not new by themselves, the overall design is unique and, in our opinion, represents an improvement over previous proposals.

This work is supported by ITRC: Information Technology Research Centre, Ontario and the National Science and Engineering Research Council of Canada.
Electronic mail: palarson@waterloo.edu

2. Data Definition

In addition to atomic (single valued) attributes, SQL/W supports set valued attributes (nested relations). The atomic attribute types are `string`, `numeric`, `float`, `date`, `time`, `money`, and `ref`. Strings can be of fixed or variable length. Attributes of type `date` contain three subfields: `year`, `month`, `day`. Attributes of type `time` also contain three subfields: `hour`, `min`, `sec`. Each relation must have a primary key consisting of one or more atomic attributes. `ref` is short for "reference to" and denotes a pointer to a tuple in the referenced relation. Logically, a reference attribute is treated as consisting of the primary key of the referenced tuple. However, it may be implemented differently and stored, for example, as an internal tuple identifier (TID).

The following simplified college database will be used as a running example. It contains information about professors, courses, course offerings, students and course enrollment.

```
create database COLLEGE
       owner palarson ;

create relation PROFS of COLLEGE
(  profid         numeric(5), key ;
   name           string ;
   datejoined     date ;
   rank           string(2), values( 'AP', 'AS', 'P') ;
   phone          numeric(7) ;
   salary         money(6), range(30000..100000) ;
) ;

create relation COURSES of COLLEGE
(  courseid       string(6) , key ;
   name           string ;
   prereq         setof ref COURSES ;
   offerings      setof
   (  term           string(3), key ;
      instructor     ref PROFS ;
      takenby        setof ref ENROLLMENT ;
   )
) ;

create relation STUDENTS of COLLEGE
(  studentid      numeric(8), key ;
   name           string ;
   credits        setof ref ENROLLMENT ;
) ;

create relation ENROLLMENT of COLLEGE
(  course         ref COURSES.offerings ;
   student        ref STUDENTS ;
   status         string(3), values( 'Crd', 'Inc');
   grade          numeric(3) ;
   dategranted    date ;
) enrkey = key( course, student) ;
```

Relations are grouped together into databases. Each database has an owner who has all privileges required to create and destroy relations, access and update any relation, and grant various privileges to other users.

Most of the attribute declarations are self-explanatory but a few comments are in order. PROFS is a traditional flat relation. A range or a set of permissible values may be specified for an atomic attribute, as done for `rank` and `salary`. The relation COURSES contains two nested relations: `prereq` and offerings. However, they are of different types: prereq is a set of pointers, while each tuple in offerings consists of several attributes, one of which is a set of pointers. The declaration ref COURSES.offerings in ENROLLMENT means a pointer to a tuple of type offerings within a tuple of type COURSES. In other words, references to tuples of a relation within another relation (to any depth) are allowed. This is one of the reasons why subrelations must have a key. (A subrelation is a relation nested within another relation.) Unless otherwise specified, keys for subrelations are taken to be locally, but not globally, unique. For example, there cannot be two offerings of the same course having the same value for term. However, two different courses may well have two offerings with the same value for term.

Disclaimer: The above organization of the database may not be the best. It was chosen purely to illustrate the main features of the data model and the query language.

3. Query Facilities

This section illustrates the main features of SQL/W by means of a series of examples. The language is not a strict extension of SQL. Support for set valued attributes and reference attributes are the main additions, but, at the same time, the language has been generalized and simplified by removing many unnecessary restrictions. Orthogonality was an important design goal.

3.1. Basic Queries

This section illustrates the basic form of a query and also introduces an alternative, more succinct, form.

Q1: For every CS course with at least one prerequisite course, find the course id, name and all offerings of the course.

```
select courseid, name, offerings( term )
from COURSES
where courseid like 'CS%' and exists( prereq )
```

The projection list in the select clause shows the attributes and the nesting structure of the result. In addition to attributes at the top level (courseid, name, offerings), we can also specify what attributes of a subrelation to keep.

From a query point of view, a set of references behaves as a subrelation consisting of the tuples referred to, i.e. pointers are automatically dereferenced. This is illustrated by the following query.

Q2: For each CS course, find the course id and name of the course, the course id and name of all its prerequisite courses, and the term of all offerings of the course.

```
select courseid, name, prereq( courseid, name), offerings( term )
from COURSES
where courseid like 'CS%'
```

Even though prereq is set of pointers, it can be used in the same way as offerings which contains actual tuples.

Q3: Same as Q2, except that we want to retain all attributes of COURSES (including all attributes of the subrelation offerings).

```
select *
from COURSES
where courseid like 'CS%'
```

An asterisk in the projection list is a shorthand for "all attributes of the operand relation(s)". For a reference attribute, the key of the referenced tuple will be returned. The phrase select * from can be omitted. Hence, the above query can be written simply as

```
COURSES where courseid like 'CS%'
```

There is an alternative form of a query which omits the key words select and from. In this case, the projection list is placed after the name of the operand relation. If there is more than one operand relation, the list must be enclosed in parentheses. Hence, the following expression is equivalent to the query above under Q2.

```
COURSES( courseid, name, prereq( courseid, name), offerings( term ) )
where courseid like 'CS%'
```

As we shall see, this alternative form is most useful for nested queries, that is, queries within query expressions. Note that the expression `prereq(courseid, name )` is, in fact, a query and could have been written as `select courseid, name from prereq`. However, the alternative form of the query is much more succinct.

3.2. Implicit Joins

In the same way as in GEM and EXODUS, joins through reference attributes are specified implicitly by path expressions using the dot notation. This is consistent with the view that reference attributes are automatically dereferenced and that a dot followed by an attribute name represents attribute (field) selection. The dot notation is commonly used for field selection in programming languages. A sequence of (relation or attribute) names separated by dots is called a path expression.

Q4: Find the student name, course id and term for all incomplete course credits in courses taught by John Doe during the fall term of 1987.

```
select student.name, course
from ENROLLMENT
where status = 'Inc' and course.term = 'F87'
and course.instructor.name = 'John Doe'
```

We can now state the rule for dereferencing of pointers: a pointer is dereferenced whenever further field selection is done. Further field selection can be occur in two circumstances: in a path expression or in the select clause or where-clause of a query having a set of reference attributes as an operand. The path expression `student.name` represents the attribute `name` of the STUDENTS tuple pointed to by `student`. `student` is of type `ref` STUDENTS. `.name` dereferences the pointer, resulting in a tuple of type STUDENTS, from which the attribute `name` is selected. The attribute `course` is of type `ref` COURSES`.offerings` and no further field selection is done. Hence, it represents the (global) key of a COURSES`.offerings` tuple, i.e. the attributes `courseid` and `offerings.term`.

Q5: Find the name and salary of all profs who joined in the same year as John Doe and who earn more than him.

```
select P2.name, P2.salary
from P1 = PROFS, P2 = PROFS
where P1.name = 'John Doe'
and P1.datejoined.year = P2.datejoined.year
and P1.salary < P2.salary
```

SQL/W supports explicit joins in the normal way, as illustrated by the example above. In this query, it is necessary to introduce explicitly declared range variables. P1 and P2 range independently over tuples in PROFS. If range variables are not declared, each relation in the from-clause is given an implicit range variable with the same name as the name of the relation.

Implicit joins are always outer joins, that is, if the value of a reference attribute in a path expression is null, the value of the whole path expression is null. For queries containing explicit joins, an outer join can be specified through a preserve clause, as proposed in [Dat]. The preserve clause is placed immediately after the where-clause. It simply consists of the keyword `preserve` followed by a list of (implicit or explicit) range variables.

3.3. Nested Queries and the With-Clause

A query expression embedded within another query is called a nested query. SQL/W allows a query expression wherever a relation name or an attribute name is allowed. This feature makes it possible to manipulate relations nested within other relations and also to let the output of a query be the input to another query.

Q6: For each student, find all courses in which the student has received a grade of 90 or more. Report only those students who have at least one such course.

```
select name, credits( course.courseid, grade) where grade > 90
from STUDENTS
where exists( credits where grade > 90 )
```

For each tuple in STUDENTS, the where-clause `where exists(...)...` is first evaluated. Each such tuple contains an instance of the nested relation `credits`, which (logically) represents a set of ENROLL-MENT tuples. To determine whether a STUDENTS tuple qualifies, the nested query `credits where grade > 90` is first evaluated, after which the function `exists` is applied to the result. If the STUDENTS tuple qualifies, an output tuple is constructed by evaluating the expressions in the select clause (using the complete STUDENTS tuple as input). The value of the attribute `name` is first copied to the output. Then the query `credits(...) where grade > 90` is evaluated, returning a set of tuples (consisting of two attributes: `course.courseid, grade`), which are copied to the output.

Embedding query expressions within queries may result in complex expressions which are difficult to grasp. A query expression defines a function which returns a set of tuples. What is needed is the ability to separate the *invocation* of a function from its *definition*. This is standard practice in programming languages. To this end, a new clause, the with-clause, has been added to SQL/W. A with-clause defines a nested query (or scalar expression) and assigns it a name. An occurrence of this name elsewhere in the query invokes the function. Function calls have copy semantics: the result of a function call is the same as would be obtained by replacing the function call by the definition of the function. Using this feature, the query above can be written as

```
select name, topcredits
from STUDENTS
where exists( topcredits )
with topcredits = ( credits( course.courseid, grade )
                    where grade > 90 )
```

A nested query does not necessarily operate on a relation nested within one of the operand relations of the outer query. It may also operate on top-level relations.

Q7: For each assistant prof, find his/her name and all courses he/she taught during the winter term of 1987.

```
select name, taught
from PROFS
where rank = 'AP'
with taught = select courseid, name
              from COURSES
              where exists( offerings where term='W87'
                            and PROFS.profid = instructor.profid )
```

This is an appropriate point to discuss scoping rules. Let Q_n be a query occurring in the select clause or where-clause of a query Q_{n-1}, which, in turn, occurs within a query Q_{n-2}, and so on until we reach the outermost query Q_1. Q_n may then operate on any collection of top-level relations and set-valued attributes of the operand relations of $Q_{n-1}, Q_{n-2}, \cdots, Q_1$. The select clause and where-clause of Q_n may refer to any attribute of the operand relations of $Q_n, Q_{n-1}, \cdots, Q_1$. Attribute names are resolved bottom up, i.e., given a name, we first attempt to find it as an attribute of the operand relations of Q_n, then Q_{n-1}, and so on. These rules are applicable to functions defined in a with-clause because of the copy semantics of function calls. They apply to the left-most attribute of a path expression, i.e., the path defined by a path expression must begin from an attribute within the scope of the query.

Applying this rule to the example query above, we find that the where-clause of the query `offerings where ...` may refer to the following attributes:

offerings:	term, instructor, takenby
COURSES:	courseid, name, prereq
PROFS:	profid, name, datejoined, rank, phone, salary

Any other attribute reachable from PROFS, COURSES, or offerings must be specified through a path expression beginning with one of the attributes within the scope of the query.

3.4. Set Operators and Functions

SQL/W supports the following comparisons between sets: `=`, `<>`, `contains`, `subset of`. A set (of tuples) can be defined by a set constant, a set-valued attribute, or a query. To test whether an element is a member of a set, we can use either `element of` or `in`. As already seen in several queries, `exists` tests whether a set is empty. The normal set operators are also supported: `union`, `intersect`, `minus`. As the language does not have any facilities for handling sets of sets, all sets of sets are converted to sets (without duplicate elimination). The following queries illustrate some of these features.

Q8: Find every course that has both CS240 and CS340 as prerequisites.

```
select courseid, name
from COURSES
where prereq( courseid ) contains ( 'CS240', 'CS340' )
```

The query `prereq( courseid )` represents a set of course id's which is then compared with the set constant `( 'CS240', 'CS340' )`.

Q9: For each course, find the total number of students who have taken the course.

```
select courseid, count( offerings.takenby( student ) )
from COURSES
```

The expression `offerings.takenby( student )` represents a set of sets of student id's, which is automatically converted to a set of student id's. For example, the following set of sets `( ( 100, 120, 099), ( 234 ), (200, 120, 333) )` would be converted to the set `( 100, 120, 099, 234, 200, 120, 333 )`.

The normal aggregate functions `count`, `min`, `max`, `sum`, and `avg` are all supported. `count` can be applied to any set. The other functions can only be applied to a set containing a single attribute, which, in the case of `sum` and `avg`, must be numeric.

Q10: For every student, find the student id, name, the number of completed courses and their average.

```
select studentid, name, count( completed ), avg( completed(grade) )
from STUDENTS
with completed = credits where status = 'Crd'
```

3.5. Nesting and Unnesting

Any language that operates on nested structures needs facilities for altering the nesting of a structure. In SQL/NF [RKB], this is achieved by providing two functions: nest and unnest. Nest creates a new nested relation. Unnest expands or unrolls a relation nested within another relation. In SQL/W, nesting is provided through a slight extension of the group-by clause and unnesting is done implicitly according to the way attributes are specified in the select clause. As in standard SQL, group-by can also be combined with aggregate functions. The basic idea is best explained by a few examples.

Q11: For each year, count and list all professors hired that year.

```
select datejoined.year, count( set ), hired = set( profid, name )
from PROFS
group by datejoined.year
```

The group-by clause is applied after the where-clause. It partitions the tuples of the operand relation(s) into groups according to the attributes specified in the group-by clause. For each group, one tuple is formed. The tuple contains all the attributes mentioned in the qroup-by clause and a nested relation called `set`, consisting of all attributes not specified in the group-by clause. (The default name `set` may be changed.) All the normal operations can then be applied to the relation `set` in the select clause. Above, the function `count` is applied to `set` to count the number of tuples. A new nested relation is also formed by projecting `set` onto `profid` and `name` and the new relation is assigned the name `hired`.

Q12: List the course id, name and instructor of all course offerings in 1987.

```
select courseid, name, offerings.term, offerings.instructor.name
from COURSES( courseid, name, offerings where term like '_87')
```

The query in the from-clause is first computed resulting in a relation consisting of two atomic attributes and a nested relation containing course offerings for 1987. Then the outer query is evaluated. As written, the select clause of the outer query specifies that the result be a relation consisting of four atomic attributes, two of which are from the nested relation `offerings`. The unnesting required to bring these attributes to the top level is performed, after which the select clause can be evaluated. In other words, the unnesting required is inferred from the nesting structure of the desired result as specified in the select clause. If we had specified `... offerings( term, instructor.name ) ...` no unnesting would have been required. The following query combines unnesting, group-by, and aggregate functions.

Q13: For each instructor, find the total number of course offerings taught by him/her.

```
select teacher, count( set )
from COURSES( teacher = offerings.instructor.name, courseid )
group by teacher
```

4. Update Facilities

The standard commands, `insert`, `update`, `delete` must be extended to handle set valued attributes. The following examples illustrate the main features.

U1: Add a new course PM350, Group Theory to the COURSES relation. The course has two prerequisites: MA131 and PM250. There are no offerings of the course.

```
insert into COURSES
values < 'PM350', 'Group Theory', ( <'MA131'>, <'PM250'> ), none >
```

This example shows how tuple constants and set constants are constructed. An empty set is indicated by the keyword `none`.

U2: Add PM330 as a prerequisite for PM350.

```
insert into COURSES.prereq
values < 'PM330' >
where courseid = 'PM350'
```

The target of an update may be a subrelation. This requires the addition of a where-clause to the insert command so that the correct instance(s) of the subrelation can be selected.

U3: Change the name of the course CO230 to Graph Theory and add MA130 as a prerequisite.

```
update COURSES
  set name = 'Graph Theory'
  insert into prereq values < 'MA130' >
where courseid = 'CO230'
```

An update command may contain nested insert, update, and delete commands to modify set valued attributes of the target relation. This ability is not needed, however, for insert or delete commands.

U4: Make CS340 have the same prerequisites as CS345 and delete all 1987 offerings of CS340.

```
update COURSES
  set prereq = ( COURSES( prereq ) where courseid = 'CS345' )
  (delete from offerings where term like '_87')
```

```
    where courseid = 'CS340'
```

This example shows that queries can be also be embedded in update commands. Queries can be used to provide values in an update expression, as shown above, or values for an expression in a where-clause.

5. Further Extensions

Several other attribute types are being considered for inclusion in the data model: conditional attributes, embedded attributes, and arrays. Conditional attributes are attributes (or attribute groups) which occur in a tuple depending on the value of other attributes. They can be viewed as a generalization of the idea of variant records. The problem is how to incorporate facilities in the query and update language for processing sets of mixed type. Embedded attributes are attributes embedded in an attribute of type `string`, i.e., they provide the ability to name and treat as a normal attribute parts of a string. The issue is how much flexibility to provide in specifying the location of an embedded attribute.

References

[CDV] M.J. Carey, D.J. DeWitt and S.L. Vandenberg, A Data Model and Query Language for EXODUS, Proc. ACM-SIGMOD Conference, Chicago, IL, 1988, 413-423.

[Dad] P. Dadam et al., A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View of Flat Tables and Hierarchies, Proc. ACM-SIGMOD Conference, 1986, 356-367.

[Dat] C.J. Date, The Outer Join, Proc. 2nd International Conference on Databases (ICOD-2), Wiley Heyden Ltd., 1983, 76-106.

[Fi] D. Fishman et al., Iris: An Object-Oriented Database Management System, ACM Trans. on Office Information Systems, 5, 1(1987).

[RKB] M.A. Roth, H.F. Korth and D.S. Batory, SQL/NF: A Query Language for ¬NF Relational Databases, Tech. Report TR-85-19, Dep. of Computer Science, University of Texas at Austin, 1985

[RS] L. Rowe and M. Stonebraker, The POSTGRES Data Model, Proc. 13th Conference on Very Large Data Bases, 1987, 83-96.

[SS] H.-J. Schek and M. Scholl, The Relational Model with Relation-Valued Attributes, Information Systems, 11, 2(1986), 137-147.

[Sh] D.W. Shipman, The Functional Data Model and the Data Language DAPLEX, ACM Trans. on Database Systems, 6, 1(1981), 140-173.

[TZ] S. Tsur and C. Zaniolo, An Implementation of GEM - Supporting a Semantic Data Model on a Relational Back-End, Proc. ACM-SIGMOD Conference, 1984, 286-295.

[Za] C. Zaniolo, The Database Language GEM, Proc. ACM-SIGMOD Conference, 1983, 207-218.

Storage Structures for Nested Relations

Aladdin Hafez and Gultekin Ozsoyoglu

Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio 44106

ABSTRACT

This paper summarizes the work done on the storage models for nested relations. In doing so, we also discuss the storage models for complex objects.

1. Introduction

Storage structures for nested relations [AbiB84, DKAB86, StoR86, DesV88] and complex objects [VaKC86, KiCB87, KCJB87, CopK85] have been investigated by many researchers. In this paper, we first briefly summarize the recent research on nested relation storage models, as well as the research on complex object storage models (as they apply to nested relation storage models). We do not discuss those complex object storage models that are implemented over existing relational storage subsystems, namely the models of Kim et al [KiCB87], and Lorie et al [LKMPM85]. We then summarize the partial normalized storage model of nested relations [HafO88] which uses the workload information of the database system under consideration to obtain a "better" storage model (i.e., one with a lower query cost) for a given nested relation. Based on the normalized storage model, the nested relation scheme is graphically represented as a tree called the scheme tree [OzsY87]. By using the workload information, and by performing a series of merges on the nodes of the scheme tree, a near-optimum scheme tree is produced to represent the partial normalized storage model.

It is shown [HafO88] that the approach which uses the greedy method, locates the optimum scheme tree in most of the cases. In a few cases, when the approach locates a "near" optimum scheme tree, the relative difference between the costs of the produced scheme tree and the optimum scheme tree is shown to be very small.

2. A Brief Survey of Storage Models for Nested Relations

As a running example in this paper, we use the nested relation scheme R where R=(a, A, B), A=(b, C), B=(c, d), C=(e, f). R, the outermost relation, is called the *external relation*. A, B, and C are *internal relations* (i.e., relations inside the external relation). *Atomic attributes* (i.e., attributes whose values are single values) are denoted by lower-case letters. Fig 1. illustrates the scheme tree of R.

System 2000 and ADABAS [Olle71] are two hierarchical database systems used to implement nested relations. The segments of hierarchical records are inserted into one file. All hierarchical relationships are expressed by a second file. In OASIS [Wied83], an instance of a tuple, together with its descendants, is placed into a single compact variable-length record.

The research in this paper is supported by the National Science Foundation grant DCR-8605554.

In IMS [McGe77], a relation and all its internal relations appear in a single file. Each top-level entry in a file contains the atomic attributes of a tuple in the external relation. The four file options for IMS are HSAM, HISAM, HDAM and HIDAM.

Recently, some new database management systems supporting nested relations have been designed and are being implemented for non-traditional applications areas. EXODUS [CDRS86] and POSTGRES [StoR86] are two examples. In EXODUS, the basic unit of stored data is the storage object (an entire tuple). Storage objects can grow and shrink in size without putting any restrictions on where we should delete or insert. Accordingly, the system supports insertion and deletion of new portions of a storage object anywhere within the object. In POSTGRES, a new datatype, called POSTQUEL, has been defined to support nested relations. A field of type POSTQUEL contains a sequence of commands to retrieve data from other relations that represent the subobjects. All relations are stored as heaps within an optional collection of secondary indices.

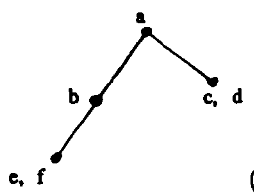


Fig. 1. Scheme tree for R

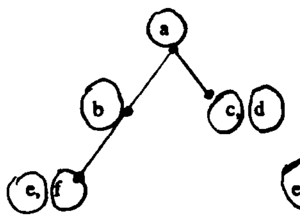


Fig.2. DSM Representation of R

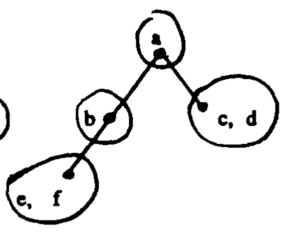


Fig. 3. NSM Representation of R

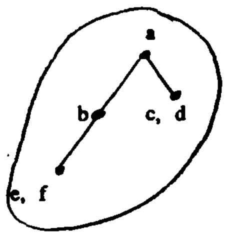


Fig. 4. FSM Representation of R

In the literature, storage models for nested relations are in general classified into four storage models, namely, the decomposed storage model, the normalized storage model, the flattened storage model, and the partial decomposed storage model. There are very few performance evaluation studies of these models [VaKC86, ASGS86, KiCB87]. In addition, typically, the research involving these models have been done under the context of complex object storage models, and have not specifically considered the performance of a rich set of relational algebra expressions. Therefore, our discussion of the performances of the storage models is quite preliminary.

The *Decomposed Storage Model* (DSM) [CopK85, KCJB87] utilizes a transposed storage. Each atomic attribute of a relation with a surrogate for record identity forms a binary relation. Each binary relation is stored into a separate file.

Example 1. (DSM)
There are 6 binary relations and thus 6 files in the DSM representation of R. The binary relations are:

$$R_a=(S_R,a),\ A_b=(S_A,b),\ B_c=(S_B,c),\ B_d=(S_B,d),\ C_e=(S_C,e) \text{ and } C_f=(S_C,f),$$

where S_Φ is used as an identifier for relation Φ. A surrogate is a unique value, possibly invisible to the user. Surrogates in the context of nested relations provide uniformity and provision for capturing joins [Vald87], and eliminate the need for user-defined, multi-attribute keys for external/internal relations.

Fig. 2 illustrates the DSM, where each circle represents a distinct file.

The DSM performs well for selections and projections on a few attributes, and, in general has an advantage over other storage models when a few attributes are involved in the query. The performance of the DSM is poor when a large number of attributes are involved [CopK85, VaKC86]. The DSM is proposed to be accompanied by join indices [ValB86, Vald87] for faster joins.

The *Normalized Storage Model* (NSM) [StoR86, VaKC86] decomposes a nested relation in such a way that the atomic attributes of each external/internal relation tuple form a record of a file, and inner relations at a given level are related to each other by using surrogates. Join indices may be used in order to retrieve an internal relation.

Example 2. (NSM)
There are 4 files in the NSM representation of R. Each file contains the atomic attributes of an internal/external relation along with its surrogate attribute. The records of each file have the format

$$R=(S_R,a),\ A=(S_A,b),\ B=(S_B,c,d) \text{ and } C=(S_C,e,f)$$

Fig. 3 illustrates the NSM, where each circle represents a distinct file.

Projection on a large number of attributes at the same level is very well supported by the NSM. On the other hand, projection on a small number of attributes at different levels degrades the performance of the system. Also, the existence of a long attribute in the file hurts the performance of all operations involving the other attributes [VaKC86].

The *Flattened Storage Model* (FSM) [DKAB86, VaKC86] is originally called the direct storage model. A nested relation is stored directly into a single file. Each record of a file represents an entire external relation tuple. The records of a file are clustered on atomic attributes of the external relation tuples. Access to nested relation tuples based on attributes other than those of the external relation tuples are done either by sequential scans or by using additional data structures such as secondary indices or elaborate hashing schemes [DesV 88] or array-pointers [DKAB86]. The basic disadvantage of additional data structures is their maintenance and access costs.

Example 3. (FSM)
The whole nested relation instance for R is stored into one file whose record format is

$$(S_R,a,\{S_A,b,\{S_C,e,f\}\},\{S_B,c,d\})$$

which is illustrated in fig. 4.

The main advantages of the FSM are: (a) retrieval of entire external relation tuples is efficient, (b) compared to storing the nested relation into several files and joining them when there are queries involving several files, there are no such joins, and (c) there is a one-to-one correspondance between a conceptual nested relation tuple and the internal file record. The basic disadvantage of the FSM is its performance degradation for large tuples that do not fit into a page. In this case, retrieving tuples of inner relations is not efficient [VaKC86].

Yet another approach is to disregard the structure of tuples and store clusters of all atomic values in a single file [DesV88]. However, in such a case, in addition to the need for additional data structures, simple operations such as sorting or tuple deletions may become inefficient.

The DSM and the NSM may be viewed as special cases of another storage model. The *Partial Decomposed Storage Model* (P-DSM) [HofS75, NCWJ84, VaCK86] is a mix between the DSM and the NSM. The atomic attributes of an internal/external relation are partitioned (also called "vertical partitioning" [HamN78, RoTK82]) such that those atomic attributes that are frequently accessed together are stored in the same file. Each file contains a set of atomic attributes and an identifier of their conceptual relation.

Example 4. (P-DSM)
The P-DSM model for R can take several possible forms. One of those forms produces 5 files with the record formats

$$R=(S_R,a),\ A=(S_A,b),\ B_c=(S_B,c),\ B_d=(S_B,d) \text{ and } C=(S_C,e,f).$$

The P-DSM is a compromise between the DSM and the NSM, and supports the selection operation and the projection operation on those attribute se·s accessed frequently. The efficiency of the P-DSM depends on the effectiveness of the vertical partitioning. Thus, when the workload characterization (and , hence, the joint utilization of attributes) changes, P-DSM performs poorly.

3. New Storage Models for Nested Relations

In [HafO88], we have introduced a new type of storage model, the *Partial Normalized Storage Model* (P-NSM). The NSM and the FSM may be viewed as two special cases of the P-NSM. In the P-NSM, the nested relation is vertically partitioned such that those internal relations which are frequently accessed together are stored in the same file. Each file contains the atomic attributes of an internal/external relation and some of its descendants.

Example 5. (P-NSM)
One possible P-NSM model for R may have 3 files with formats

$$R=(S_R,a,\{S_A,b\}),B=(S_B,c,d) \text{ and } C=(S_C,e,f).$$

which is illustrated in fig. 5.

The spectrum of different storage models is illustrated in Fig. 6.

By storing into the same file the internal relations which are accessed together frequently, the number of I/O operations required to respond to queries can be reduced. In general, the storage model of a nested relation can be classified as supporting three different query types: queries manipulating

(a) entire external relation tuples,
(b) tuples of an internal relation at a given level, or
(c) specific individual components of (external/internal) relation tuples and their atomic attributes at different nesting levels.

If the query type (a) is the dominant query type then clearly the FSM is the best choice to implement

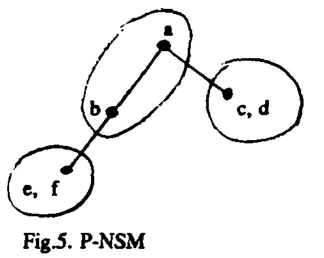


Fig.5. P-NSM Representation of R

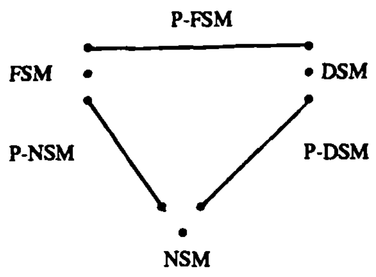


Fig.6. The Spectrum of Different Storage Models for Nested Relations

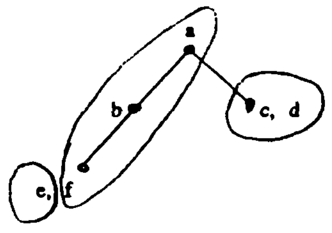


Fig.7. P-FSM Representation of R

the nested relation. For the query type (b), the NSM is the obvious choice to implement the nested relation. Query type (c) constitutes the general case for nested relation queries. The P-NSM provides good support for those nested database systems which have the query type (c) as the dominant query type. By using the appropriate methodology and the workload information of the system, the NSM may be turned into a P-NSM with a better query processing performance. Below we describe such a methodology.

The NSM is graphically represented as the scheme tree. Each node in the scheme tree represents a file containing the atomic attributes of an internal/external relation. Figure 1 describes the scheme tree of R. We are specifically concerned with two parameters of the workload information of a database system. *Query count* is the first parameter. Each node and each edge in the scheme tree has its own query count (frequency) denoting the number of queries that manipulate the information in that node and in the nodes adjacent to that edge, respectively. In [HafO88], we give the ASSIGN-F algorithm which, given a set of queries, assigns frequencies to the nodes and edges in the scheme tree in a consistent manner. The second parameter is the *query cost.* The query cost of each node is a function of the size of the file represented by the node, and the type of the query used. Each node may have a different query cost such as constant, linear, logarithmic, etc., depending on its file organization and the specific query type. Queries are classified according to their disk processing costs. Each group of queries contains those queries that have the same disk processing cost.

We compute the *scheme tree cost* by using the query costs and the frequencies of the nodes and the edges of the scheme tree. Depending on the query types, the cost of the scheme tree in the NSM can be changed by reducing the depth of the scheme tree through the *merge operation* of two or more nodes (which produces a P-NSM representation). Clearly, the file (node) obtained after merging two files (nodes) with different file organization types will need to choose between the two file organizations. Then, the scheme tree with the lowest cost can be found by enumerating all the possible trees derivable from the scheme tree by the merge operation.

[HafO88] gives the GREEDY-MERGE algorithm which takes the original NSM scheme tree, the query types, the associated query costs for each node, and frequencies for each node and for each edge in the scheme tree, and uses the greedy method to convert the NSM scheme tree into a P-NSM scheme tree with a lower scheme tree cost. The GREEDY-MERGE algorithm utilizes a level order traversal starting at the lowest level. At each step, a subset of nodes (at the present level) are checked for merging them into their father node. After examining the nodes at a certain level, the algorithm moves up to the next higher level and repeats the merge checks at the new level. For all query cost types, except the logarithmic query cost, the GREEDY-MERGE algorithm finds the scheme tree with the lowest cost, without having to enumerate all the possible trees derivable from the scheme tree. For the logarithmic query cost type, the GREEDY-MERGE algorithm locates the optimum scheme tree in most of the cases. In a few cases, when the approach locates a non-optimum scheme tree, the percentage of error is very small.

In [HafO88], our concern is to find the best P-NSM (i.e., the one with the lowest cost) among all the P-NSM models derived from the NSM. That means the GREEDY-MERGE algorithm covers only one variation in the spectrum of storage models in figure 6 (i.e., from NSM to FSM). To cover the whole spectrum of storage models (i.e., from DSM to FSM), we now describe yet another storage model, the *partial flattened storage model* (P-FSM). The P-FSM is a hybrid between the DSM, the NSM and the FSM, where the nested relation is vertically partitioned based on the atomic attributes and the internal relations affinities. In other words, each file contains a possible subset of atomic attributes of an internal/external relation and/or parts of its descendants. This is in contrast with incorporating all atomic attributes of a relation into a P-NSM file. The P-FSM is illustrated in Fig. 7. All the storage models of nested relations can be considered as special cases of the P-FSM.

So far we have discussed the effect of using abstract types of queries (having constant, linear, logarithmic, etc., file processing costs) in obtaining near-optimum P-NSM or P-FSM storage models. In [HafO88a] we discuss the implementations of specific nested relational algebra operators with explicit costs and specific file organization techniques. Clearly, any such discussion must precede with specific ways of establishing connections between a nested tuple and its subtuples, which is briefly discussed below.

4. Establishing Connection Between Different Subrelations

Clearly, storing surrogate attribute values in inner relation tuples is sufficient to maintain unambiguously the structures of nested relation tuples. Valduriez et al [VaKC86] discusses two additional techniques, namely *Binary Join Indices* and *Hierarchical Join Indices*, to establish the connection between two files representing the two external/internal relations R_i and R_j, where R_i is the father and R_j is the son. In binary join indices, for each pair of father-son files, there is a binary index, which is itself a relation with two surrogate attributes. The number of entries in the binary index equals the number of tuples in the son relation. Binary join indices have been shown to be very useful in optimizing joins [Vald 87]. However, the performance may slow down in some relational algebra operators such as nest, unnest, projection or (range) selection.

In the hierarchical join indices, the structure of the whole relation is captured by using the surrogates of the external/internal relations in the nested relation. For example, the hierarchical join index of a tuple of R in our running example is $\{S_R\ \{S_A\ \{S_C\}\ \}\ \{S_B\}\ \}$. The hierarchical join indices is similar to the FSM, and can be clustered on the root surrogate (S_R, in our example). The tradeoffs between binary join indices and hierarchical join indices are similar to the tradeoffs between the DSM and the FSM. Also, even when hierarchical join indices are used, the retrieval of subtuples may necessitate the maintenance of additional binary join indices.

Yet another approach is to separate structural information from data completely. In [DKAB86], each external relation tuple is stored into one record, and, for each tuple, there is an entry in a directory, called the *mini-directory* (MD). The mini-directory is used to handle the allocation of each tuple.

[DesV88] generalizes the join indices of [Vald87] and defines a single tree structure in which, given a single atomic value, one can locate all external and internal tuples having that value. To achieve this goal, a hierarchical tuple identification scheme is devised. The details of this approach are not yet available.

5. References

[AbiB84] Abiteboul, S. and Bidoit, N., "Non First Normal Relations to Represent Hierarchically Organized Data", Proc., *the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Apr. 1984.

[ASGS86] Abiteboul, S., Scholl, M., Gardarin, G., and Simon, E., "Towards DBMSs for Supporting New Applications", Proc., *VLDB Conf.*, Aug. 1986.

[CDRS86] Carey, M.J., DeWitt, D.J., Richardson, J.E. and Shekita, E., "Object and File Management in the EXODUS Extensible Database System", Proc., *VLDB Conf.*, Aug. 1986.

[CopK85] Copeland, G. and Khoshafian, S., "A Decomposed Storage Model", *ACM SIGMOD Int. Conf. on Management of Data*, May 1985.

[DesV88] Deshpande, A. and Van Gucht, D., "An Implementation for Nested Relational Databases", Proc., *VLDB Conf.*, Aug. 1988.

[DKAB86] Dadam, P., Kuespert, 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", Proc. *ACM SIGMOD Int. Conf. on the Management of Data*, May 1986.

[HafO88] Hafez, A. and Ozsoyoglu, G.,"The Partial Normalized Storage Model of Nested Relations", Proc., *VLDB Conf.*, Aug. 1988.

[HafO88a] Hafez, A. and Ozsoyoglu, G.,"On Storage Models for Nested Relations", Technical Report, Department of Computer Science, Case Western Reserve University, September 1988.

[HamN78] Hammer, M. and Niamir, B., "A Heuristic Approach to Attribute Partitioning", Proc. *ACM SIGMOD Int. Conf. on Management of Data*, May 1979.

[HofS75] Hoffer, J.A. and Severance, D.G.,"The Use of Cluster Analysis in Physical Database Design", Proc., *VLDB Conf.*, 1975.

[KCJB87] Khoshafian, S., Copeland, G., Jagodits, T., Boral, H. and Valduriez, P., "A Query Processing Strategy for the Decomposed Storage Model", *3rd Int. Conf. on Data Engineering*, Feb. 1987.

[KiCB87] Kim, W., Chou, H. and Banerjee, J., "Operations and Implementation of Complex Objects", *IEEE Trans. on Software Engineering*, July 1988.

[LKMPM85] Lorie, R., Kim, W., McNabb, D., Plouffe, W., and Meier, A., "Supporting Complex Objects in a Relational System for Engineering Databases", in *Query Processing in Database Systems*, W. Kim, D. Reiner, and D. Batory, Ed.s, Springer-Verlag, 1985.

[McGe77] McGee, W.C., "The Information Management System IMS/VS Part 1: General Structure and Operation", *IBM Syst. J., Vol. 16, No.* 2, 1977.

[NCWJ84] Navathe, S., Ceri, S.,Wiederhold, G. and Jinglie, D., "Vertical Partitioning Algorithms for Database Design", *ACM Trans. on Database Systems, Vol.8, No.4,* Dec. 1984.

[Olle71] Olle, T.W., "Introduction to 'Feature Analysis of Generalized Data Base Management Systems'", *CACM, Vol.14, No.5,* May 1971.

[OzsY87] Ozsoyoglu, Z.M. and Yuan, L-Y., "A New Normal Form for Nested Relations", ACM TODS, March 1987.

[RoTK82] Rotem, D., Tompa, F. and Kirkpatrick, D., "Foundations for Multiple Design by application Partitioning", Proc., *ACM Symposium on Principles of Database Systems,* Mar. 1982.

[StoR86] Stonebraker, M. and Rowe, L.A., "The Design of POSTGRES", Proc., *ACM SIGMOD Int. Conf. on Management of Data,* May 1986.

[VaKC86] Valduriez, P., Khoshafian, S. and Copeland, G., "Implementation Techniques of Complex Objects", *Int. Conf. on VLDB,* Aug. 1986.

[ValB86] Valduriez, P. and Boral, H., "Evaluation of Recursive Queries Using Join Indices", *Proc. of First Int. Conf. on Expert Systems,* Apr. 1986.

[Vald87] Valduriez, P., "Join Indices", ACM Trans. on Database Systems, Vol.12, No. 2, June 1987.

[Wied83] Wiederhold, G., *Database Design* (2nd ed.), McGraw-Hill, 1983.

Algebras for Nested Relations

Mark A. Roth and James E. Kirkpatrick

Department of Electrical and Computer Engineering
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433-6583

Abstract

This paper summarizes work that has been done to date in the area of relational algebras for nested relations. We emphasize the form and nature of the algebra operators and provide pointers to more formal papers. We pay particular attention to the join operator and methods for joining nested relations in various forms.

1 Introduction

In this paper, we summarize the development of relational algebras for nested relations, paying particular attention to the join operator since this is where the most interesting results occur. We trace the development of algebra operators for nested relations created using single attribute, single level nest and unnest through the multi-attribute, multi-level algebra operators. A normal form for nested relations is introduced which permits discussion of a set of "reasonable" algebra operators for nested relations. We then concentrate on the natural join operator and discuss the unique characteristics of joining relations in nested normal form [OY87a]. Finally, we discuss ideas for recursive forms of the algebra [DL87,SS86] which more closely follow the structure of nested relations. We do not, however, specifically discuss algebras extended with the powerset operator [GVG88].

2 The Development of Operators for Nested Relations

2.1 Nest, Unnest, and Associated Relational Algebra Operators

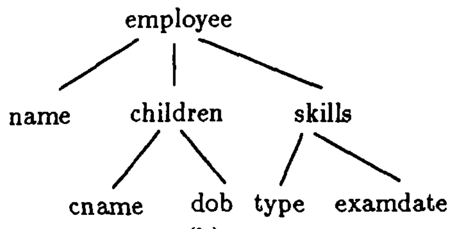
In 1977, Makinouchi [Mak77] first pointed out the need to look further at nested relations. In particular, further research was required in determining how one might create nested relations from traditional (flat) relations and what operations should be defined for nested relations. In 1982, Jaeschke and Schek [JS82] introduced the "nest" operator, ν, which formed single-level, single-attribute nested relations from flat relations and the "unnest" operator, μ, which is the inverse of nest. The nest operator requires the projection of the attributes *other* than the *one* attribute which is being nested. This projection forms the partitions (or distinct combinations) over which the nested attribute is then grouped. For each of these partitions, the "value" of the nested attribute is the *set* of all instances of that attribute which were associated with that partition in the original flat relation. Unnesting allows one to "flatten" a nested relation formed via the nest operator. This unnesting is performed by combining each element of a nested attribute with repeated occurrences of the associated partition of unnested attributes.

In addition to the one-attribute nest and unnest operators, [JS82] briefly presents the possibility of applying the normal relational algebra operators to nested relations. In particular, the point is made that union ($\cup$), difference ($-$), projection (π), and the cartesian product ($\times$) can be defined for all relations; so they also apply on nested relations. In addition, by allowing the comparison of set-valued attributes via the set comparison symbols $\subset, \subseteq, \supset, \supseteq$, and $=$, [JS82] extends selection (σ) to nested relations.

In [Tho83] and [FT83] Fischer and Thomas expand the work of [JS82] by generalizing the nest and unnest operators to allow for multiple attributes and multiple levels; extending the definitions of the relational algebra operators in view of the enlarged "scope" of nested relations; and providing results concerning properties associated with the interaction of nest, unnest, and the relational algebra operators.

$$\begin{aligned} \text{employee} &= (\text{name}, \text{children}, \text{skills}) \\ \text{children} &= (\text{cname}, \text{dob}) \\ \text{skills} &= (\text{type}, \text{examdate}) \end{aligned}$$

(a)



(b)

Figure 1: Structure Tree

2.2 A Nested Relational Model

We will now introduce and adopt the notation of [Tho83] and [Rot86] which is based on the concepts of "database logic" as introduced by Jacobs [Jac82].

A *relation scheme*, S, is a collection of rules (defining relations) of the form $R_j = (R_{j1}, R_{j2}, \ldots, R_{jn})$. The elements R_{ji} are attributes (of the relation defined by the rule R_j). R_j is a *higher-order attribute* if it appears on the left hand side of some rule, otherwise it is a *zero-order attribute*. The elements on the right hand side of a rule R_j form a set denoted E_{R_j}. As is true of any set, these attribute names are unique. In addition, no two distinct rules can have the same attribute on the left hand side.

For every database or relation scheme S, we can define a unique directed graph, the *structure tree* of S (T_S) (as opposed to the scheme tree of [OY87a]) which contains exactly those nodes corresponding to every attribute R_j (zero or higher-order) in S. In addition, T_S contains a directed edge (R_i, R_j) from R_i to R_j iff $R_j \in E_{R_i}$ (i.e., there is an edge from the higher-order attribute defining the relation to each of the internal attributes within that relation).

For example, consider the database scheme shown in Figure 1a, and its associated structure tree given in Figure 1b. Here, an employee is defined to have a name, a set of children, and a set of skills. Each child is defined by a name and a date of birth, while each of the employees skills are represented by the type of skill and the date that particular skill was verified via an examination. Note that if the attribute "name" had appeared in both the employee and children rules, we would not be able to consider "employee" a relation scheme.

Given this starting point, we can define flat database schemes as a collection of rules of the form $R_j = (R_{j1}, R_{j2}, \ldots, R_{jn})$ in which each R_{ji} is of zero-order. Similarly, we define nested schemes as those rules in which the individual R_{ji}'s may be of either zero or higher order. Thus we see that the term "nested relation" describes the situation which occurs when a higher-order attribute appears on the right hand side of some rule and thereby "nests" a relation *within* another relation. In this paper, we also require that the rules remain non-recursive.

2.3 Multi-level, Multi-attribute Nest and Unnest

The nest (ν) operator, as defined by [JS82], partitions the tuples on the basis of the attributes *not* being nested and collects sets of nested attributes associated with each of these partitions. The first difference between this operator and the "extended" version presented in [Tho83] is that the extended definition allows each element of a set associated with a given partition to in turn be *a set of attributes* instead of merely a single attribute. This allows for multiple levels of nesting. In addition, the extended definition allows for nesting along more than one attribute at a time.

In a similar manner, the multi-attribute version of the *unnest* (μ) operator is a straightforward extension of that defined earlier in [JS82]. In particular, each element of a nested attribute is repeatedly combined with its associated partition of remaining attributes so as to "flatten" the overall structure. Again, the key to the "extension" is that each element of the nested attribute may be composed of several attributes rather than one, as was the case earlier. See Figure 2 for an example.

It should be stressed that the purpose of extending nest and unnest as shown above is to allow for the creation of more general nested relations from a given flat (or normalized) relation, and the manipulation of these nested structures in a more general way.

A	C	D	E
a_1	c_1	d_1	e_1
a_1	c_1	d_2	e_1
a_1	c_1	d_2	e_2
a_2	c_2	d_1	e_1
a_2	c_2	d_1	e_2

(a) $r_1 = \mu_B(r_2) = \mu_B(\nu_{B=(C,D)}(r_1))$

	B		
A	C	D	E
a_1	c_1	d_1	e_1
	c_1	d_2	
a_1	c_1	d_2	e_2
a_2	c_2	d_1	e_1
a_2	c_2	d_1	e_2

(b) $r_2 = \nu_{B=(C,D)}(r_1)$

Figure 2: Multi-attribute nest and unnest

A	X
a_1	$\{x_1, x_2, x_3\}$
a_2	$\{x_4\}$

r_1

X	B
$\{x_2, x_3, x_4\}$	b_2
$\{x_5, x_6\}$	b_1

r_2

A	X	B
a_1	$\{x_2, x_3\}$	b_2
a_2	$\{x_4\}$	b_2

$r_1 \overline{\times} r_2$

Figure 3: Intersection Join

When performing such manipulations, one should be aware of the following basic properties concerning the interaction of nest and unnest operators [Tho83]:

1. As illustrated in Figure 2, the results of a particular nest operation can be reversed by a subsequent unnest operation. In fact, this is a property which holds in general.

2. An interesting counter-intuitive property is that unnest operations can *not* always be undone by a subsequent nest operation.

3. The order of unnesting is not important.

4. Unlike unnest, the order in which nest operations are applied *is* important.

As a result of the third of these properties, it is apparent that a nested relation can be "completely unnested" (transformed into a flat relation via a series of unnest operations) in different ways. The results of each of these series of unnests will always be the same. Since this might be a desirable process, [Tho83] also introduces the aggregate operation μ^*. The application of μ^* results in the *complete unnesting* of the relation on which it operates. This, of course, will always yield a flat relation.

Having defined the more powerful versions of μ and ν, Fischer and Thomas [FT83,Tho83] also define ways in which the traditional relational algebra operators might be extended to apply to the more general nested relations now achievable. This extension of the relational algebra operators is similar to that presented in [JS82]. The primary difference is that a comparison of tuples no longer involves a relatively simple set comparison of a single nested attribute from each relation, each nested to only a single level. Now, we must compare complex structures in which several attributes can be nested to various levels.

[JS82] define two methods of performing a natural join ($\bowtie$) of nested relations. The first of these methods can be considered an "extension" of the "normal" natural join where set equality is used when set-valued attributes are compared. The second method of joining two nested relations is the "intersection join" ($\overline{\times}$). With this operator we take the intersection of the common nested sets when combining tuples in the join. If the intersections of all common nested sets are not empty and the tuples join on their atomic attributes, then the tuples contribute to the result. Figure 3 shows the results of applying the intersection join operator to two sample nested relations. In particular, note that $r_1 \bowtie r_2 \neq r_1 \overline{\times} r_2$ since $r_1 \bowtie r_2 = \emptyset$. Extensions of these operators to multi-attribute, multi-level nested structures is straightforward [FT83,Tho83].

2.4 Partitioned Normal Form and Reasonable Extensions of the Algebra Operators

In the previous sections, the purpose of each operator extension was to obtain an operator which worked "similarly" to the standard operator of the same name, but which operated on increasingly complex structures. The resulting operators were then shown to possess or, in the case of the join operator, not possess various properties.

An alternative method for defining relational algebra operators might be to first define a desirable property (or set of properties) and *then* define operators in such a manner as to ensure that these properties hold with respect to the new operators. In [Rot86], Roth defines "reasonableness" as such a desirable property and then presents "reasonable" extensions of the relational algebra operators as presented in [Tho83]. Similar definitions of several of these operators are also presented in terms of the *Verso* model in [AB84,Bid87].

2.4.1 Partitioned Normal Form

As shown in section 2.3, there exist nested relations for which there is *not* a nest operation (or sequence of nest operations) which will "undo" (reverse the effects of) a valid sequence of unnest operations. The *PNF* class of relations is a class for which there is always a sequence of nest operations which will be an inverse for any sequence of valid unnest operations. A relation is in PNF if the zero-order attributes form a key for the relation, and considering each set-valued attribute, the (nested) relation within that attribute is in PNF. The class of PNF relations is a natural one, corresponding to "real world" nested relations (see also [OY87b,VF86]).

Two critical properties are:

1. The class of PNF relations is closed under unnesting, and
2. The nesting of a PNF relation is in PNF iff the relation when projected onto the attributes not being nested is in PNF.

These properties ensure that unnesting a relation which is in PNF will always result in a relation which is in PNF, and they give necessary and sufficient conditions under which nesting operations applied to PNF relations will return PNF relations.

2.4.2 Reasonableness

The reasonableness property requires that two sub-properties hold. The first of these sub-properties, *faithfulness*, requires an operator to act on flat relations in the same manner as the "standard" operator from which it is extended. The second sub-property, *preciseness*, requires the normalized (completely unnested) result of applying an operator to a nested relation be the same as if the standard operator had been applied to the normalized versions of the nested relations.

Formally, faithfulness and precision are defined, for binary operators, as follows:

Let *Rel* be the set of all flat relations and let Rel^* be the set of all nested relations that have at least one higher-order attribute in the scheme.

- Let γ be an operator on *Rel* and let γ' be an operator on $Rel^* \cup Rel$. Then γ' is *faithful* to γ if $r \gamma q = r \gamma' q$ for every $r, q \in Rel$ for which $r \gamma q$ is defined.
- Let γ be an operator on *Rel* and let γ' be an operator on Rel^*. Then γ' is a *precise* generalization of γ relative to unnesting if $\mu^*(r \gamma' q) = \mu^*(r) \gamma \mu^*(q)$ for every $r, q \in Rel^*$ for which $r \gamma' q$ is defined.

Having defined what is meant by a *reasonable* operator, we now examine definitions of extended intersection ($\cap^e$) and extended natural join ($\bowtie^e$). Formal proofs of the reasonableness of these operators for PNF relations are given in [Rot86].

2.4.3 Extended Intersection

As is true of the standard intersection of flat relations and each of its extensions which we have examined, the extended intersection of two relations is defined only if their schemes are equal. Also, as one would expect, the scheme of the result is the same as the scheme of the relations being operated upon. At the instance level, tuples contribute to the extended intersection if they are the same with respect to their zero-order attributes, and if the *extended* intersections of their higher-order attributes are non-empty. The resulting tuple contains the extended intersections for each higher-order attribute.

s_1

A	B	X	
		C	D
a_1	b_1	c_1	d_1
		c_2	d_2
		c_1	d_3
a_2	b_1	c_3	d_1
		c_2	d_2
		c_1	d_1
a_2	b_2	c_1	d_2
		c_3	d_2

s_2

E	B	X	
		C	D
e_1	b_1	c_1	d_1
		c_1	d_3
		c_3	d_4
e_3	b_2	c_3	d_2
e_4	b_1	c_3	d_1
		c_4	d_2

$s_1 \bowtie^e s_2$

A	E	B	X	
			C	D
a_1	e_1	b_1	c_1	d_1
			c_1	d_3
a_2	e_1	b_1	c_1	d_1
a_2	e_4	b_1	c_3	d_1
a_2	e_3	b_2	c_3	d_2

Figure 4: Extended natural join.

2.4.4 Extended Natural Join

In defining the extended natural join, Roth remarks that "join operations are difficult to define in the [nested] world due to the possibility of different nesting depths for the attributes."[Rot86, pg 89] As an example, consider the situation in which r_1 is a relation on $R_1 = (A, X), X = (B, C)$; while r_2 is a relation on $R_2 = (B, D)$. In this case, a natural join of the relations would degenerate into a cartesian product of r_1 and r_2 since there are no common attributes on which to join (i.e., $E_{R_1} \cap E_{R_2} = \emptyset$). However, if r_1 and r_2 were completely unnested, then B *would be* a common attribute and a join on B would take place.

As we can see from the discussion of reasonableness above, unless a join operator prevents the above situation from occurring, it would *not* meet the requirements for being reasonable. This problem can be solved by limiting the relations which can be operated on by the extended natural join to those whose only common attributes are elements of the top level scheme, or are descendants of higher-order attributes which are common at the top-level scheme.

Two tuples participate in an extended natural join if the values of their common zero-order attributes are equal and the common higher-order attributes have a non-empty extended intersection. Figure 4 shows an example. The first tuple from relation s_1 and s_2 join to create the first tuple in $s_1 \bowtie^e s_2$ since the common zero-order attribute, B, is equal in both tuples, and the higher order attribute, X, has a non-empty intersection between the two nested relations. The resulting tuple contains the zero-order attributes and their associated values from s_1 and s_2, and X which contains the intersection of the nested X relations from s_1 and s_2.

3 Reasonable Natural Joins of Arbitrary Nested Relations

The results which we have discussed thus far provide us with two methods of performing a natural join over *arbitrary* nested relations — the equality and intersection joins as defined in [FT83,Tho83]. Unfortunately, as proven in these papers, some desirable properties do not hold in general for these operators. In particular, [Tho83] proves that the precision property does *not* hold for either form of the join. Therefore, we see that neither are reasonable.

We were able to improve this situation by defining extended natural join ($\bowtie^e$) which is reasonable. However, in doing so we restrict the types of relations to which the extended natural join can be applied. We will now look at how the extended join can be used to allow for the *reasonable* natural join of *arbitrary* nested relations.

The method which is presented below is, in fact, a further extension of $\bowtie^e$ in which the relations being joined are "reshaped" (if necessary) in such a manner that the resulting relations are "joinable."

A first attempt at such a reshaping might well be the brute force method of completely unnesting the given relations and then applying $\bowtie^e$. In this manner, the operation reduces to a standard natural join of flat relations. This certainly meets the "reasonableness" criteria, and in addition, the procedure can be applied to arbitrary nested relations. However, this simplistic approach is really just "sidestepping" the problem. The results are unsatisfying in that all information embedded in (and thereby the advantages of) the original nested relations is lost.

We might attempt to regain the original nested structure, but we must be careful in the subsequent applications of ν. Recall that reasonableness is applied to relations in PNF and that the closure of PNF with respect to nesting limits the nest operations which can be applied to a relation in PNF while ensuring that the result will be in PNF.

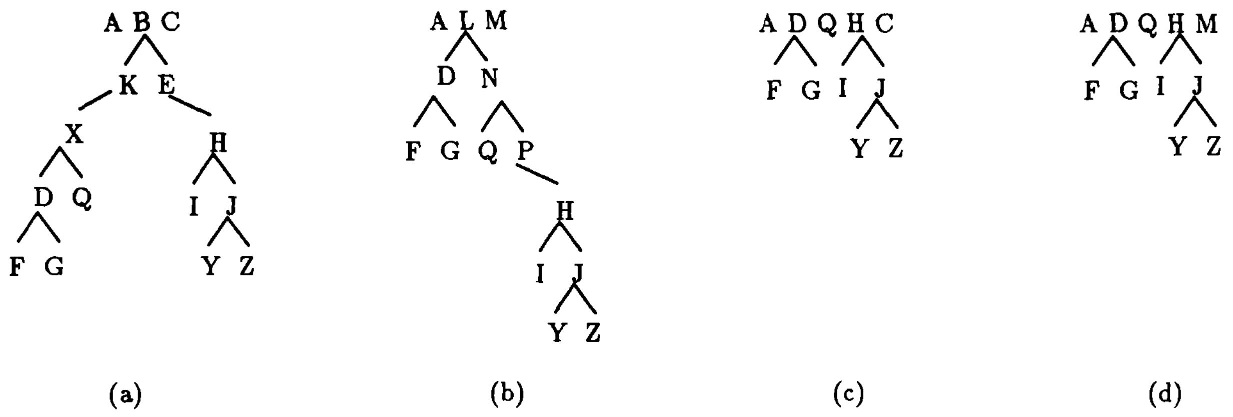


Figure 5: Complex Nested Relations

Given the fact that the relations to be joined can have vastly different structures, the form which the resulting structure should take on is far from clear.

One method of using $\bowtie^e$ to achieve a reasonable natural join of arbitrary nested relations involves unnesting each relation being joined *just to the point* at which $\bowtie^e$ is applicable. The extended natural join, and its accompanying proven properties are then available for use. The first benefit of this approach is that the manipulation of each relation involves only unnest operations, under which PNF relations are closed. In addition, the actual join is performed using $\bowtie^e$, which which we know to be reasonable in and of itself. Finally, the resulting structure will already be nested to the proper level (i.e., those sub-trees, and only those sub-trees, which are present in the structure trees of both original relations will be present in the structure tree of the result).

Unfortunately, arbitrary nested relations can be much more complicated than those we have seen so far. Consider the examples shown in Figures 5a and b. These relations have common attributes (both zero and higher order) which are neither elements of the top level scheme nor descendants of higher-order attributes which are common at the top level scheme. Thus, they are not joinable using the basic $\bowtie^e$. Note that the common attributes are D, H, and Q, none of which occur at the same level in R_1 and R_2. The (joinable) schemes which result from the proper unnesting of R_1 and R_2 are shown in Figures 5c and d. An algorithm which is proposed as a method of reshaping arbitrary nested relations R_1 and R_2 as described above is given in [Kir88]. The basic idea is to search for common higher-order attributes which are not at the top-level of the nested relation. The path to these common attributes represent the schemes which must be unnested so that the common attribute becomes a top-level attribute. Thus in Figure 5, the common higher-order attributes are D and H, and so in tree (a), we must unnest B, K, and X (the path to D), and also E in the path to H. Tree (b) requires unnesting of L, N, and P.

So far, we have been concerned with how we might join two relations which have some *previously defined* nested structure. We have not, as yet, examined the concept of "useful" or "meaningful" nesting strategies. The next section will present such a viewpoint and, in addition, will allow us to examine the "joinability" of relations whose structures *we* have determined *a priori*.

4 Nested Normal Form and Joinability

When we consider the process of designing "good" database schemes in the non-nested world, we quickly encounter the idea of ***normalization***. The basic idea behind this thoroughly researched process, is to eliminate "problems" which can easily occur if *ad hoc* decisions are made in decomposing the relation schemes.

The results obtained to date [OY87a,RK87] depend heavily on an interesting relationship between the meaning of ***multivalued dependencies*** (or MVDs) which hold on flat relations and the effect of performing a nest operation on the same relation. In [Ull88], Ullman gives a useful description of the meaning of an MVD:

> Suppose we are given a relation scheme R, and X and Y are subsets of R. Intuitively, we say that $X \rightarrow\rightarrow Y$ ... if given values for the attributes of X there is a set of zero or more associated values for the attributes of Y, and this set of Y-values is not connected in any way to values of the attributes in R−X−Y.

The idea of associating a set of values from one or more attributes with a single value of another attribute is, in fact, the basic function of the nest operator. [JS82] that when the MVD $X \rightarrow\rightarrow Y \mid Z$ holds on a flat relation, then nesting the attributes in any order will generate the same result. This is *not* true in general. The underlying principle here is that it is vital to consider the effect of MVDs on the structure of nested relations when we "normalize" them.

In [OY87a], *scheme trees* are used to represent nested relations. A scheme tree is a tree whose vertices are pair-wise disjoint sets of attributes. The edges of these trees represent the MVDs which hold between the attributes of the relation. Note that a scheme tree is similar to a structure tree except the intermediate names for the nested relations are not represented.

As it turns out, some scheme trees are "better" than others in that they represent structures which further reduce redundancy while representing the MVDs implied by the given MVDs. A relation is in *nested normal form* (NNF) if the scheme tree associated with the relation is normal. A scheme tree is normal if (1) the MVDs represented by the tree are implied by the given MVDs, (2) their are no undesirable dependencies (partial and transitive dependencies), and (3) each node of the scheme tree is a fundamental key for that tree. The last condition ensures that an arbitrary structuring is not used for the nested relation.

An NNF decomposition procedure is given by [OY87a] which begins with a flat relation containing all attributes of interest and nests this relation to best meet criteria (1) and (3) of NNF. Then, if partial or transitive dependencies exist, new scheme trees are created and attributes are eliminated from the original scheme tree to eliminate the undesirable dependency.

We can now summarize the algebra operators which are used to join nested relations.

1. Arbitrary joins between any two nested relations can be performed via the equality and intersection join operators defined in [FT83]. Unfortunately, as Fischer and Thomas have shown, none of the properties which might be considered "desirable" hold, in general, under these operators.

2. Arbitrary joins between any two nested relations can be performed in a *reasonable* manner [Rot86,Kir88]. However, this may require restructuring of the relations involved. With respect to the desired goal of *maintaining* a specific structure (i.e., NNF), this approach appears to have limited significance.

3. The NNF relations can be rejoined in *exactly* the reverse order from which they were formed. This simply takes us back to a state *comparable* to that which existed after initially forming a non-decomposable tree. This proposal will generate some (useful) side effects as we will see below. However, in light of the next item, this becomes a trivial case.

4. Given a set of relations in NNF, arbitrary joins can be performed between any two of these relations *without* restructuring either relation.

5. In "rejoining" NNF relations which were split during the normalization process, we will re-introduce partial dependencies into the result; however, we will also *eliminate* all transitive dependencies which might have held earlier.

5 Recursive Algebras

All of the operators we have seen so far operate on entire tuples of a relation. The only way to manipulate interior nested relations is to unnest the outer relation to the point where the operators can be applied and then renesting to the desired structure. Proposals for using nested expressions in a recursively defined algebra appear in [DL87,Jae84,SS86]. We summarize here the algebra of [SS86] and an extension of it in [DL87].

Schek and Scholl [SS86] allow algebra expressions to replace attributes in selection predicates and projection lists. Since attribute values may be relations whose attribute values again may be relations, etc., a *nested* application of the basic algebra operations leads to a high level of expressiveness. We may eliminate unwanted tuples from nested relations by the recursive use of a selection or we may eliminate unwanted attributes from nested relations by the recursive use of a projection. Interior rearrangement via nest or unnest can be easily specified as well as the use of binary operators such as join and Cartesian product. To illustrate, we show the following queries against the example relation used in [SS86]. The scheme is DEPT = (D, DN, AE, TE), AE = (AN, AJD), TE = (TN, TJD, C), C = (CN, Y). AE and TE contain information about administrative and technical employees, respectively. AJD and TJD are job descriptions and C contains information about courses taken by employees.

The Expressiveness of Query Languages for Nested Relations

Marc Gyssens[*]

Dept. of Math. and Comp. Science
University of Antwerp (UIA)
B-2610 Antwerpen, Belgium

Dirk Van Gucht

Computer Science Dept.
Indiana University
Bloomington, IN 47405, USA

In this paper, we review the expressiveness of query languages for nested relations. We discuss the naturalness of the nested algebra, which is emphasized by its BP-completeness. At the same time, however, the nested algebra is only a poor extension of the flat algebra with respect to expressive power. Indeed, the restriction of the nested algebra to flat relations is equivalent to the flat algebra. Finally, we propose the powerset algebra, or one of the many equivalent formalisms, as a valid alternative to the nested algebra, overcoming most of its limitations.

1. Introduction

In the last years, much attention has been paid to structured relations. In order to model some database applications more naturally, Makinouchi [21] proposed to generalize the relational model by removing Codd's first normal form assumption [9], thus allowing relations with set-valued attributes. Subsequently, a generalization of the relational algebra to relations with set-valued attributes was introduced by Jaeschke and Schek [16]. More specifically, they presented the nest and the unnest operator as tools to restructure such relations, though only over single attributes. Thomas and Fisher [32] generalized this model by allowing multi-level relational structures of arbitrary (but fixed) depth. Consequently, several query languages were introduced for this model or slight generalizations of it. We mention SQL-like query languages [19,25,26,28], graphical-oriented query languages [14] and datalog-like languages [4,5,6,17]. Calculus-like query languages were considered and discussed in [1,15,22,24,29,30]. Our primary concern in this article is with the expressiveness of these languages: how can we measure their expressive power and what criterions should a query language for nested relations at least have to meet to be considered sufficiently rich?

For the classical flat case, Codd "solved" these questions in [10], by simply proposing the expressive power of the flat relational calculus (or, equivalently, the flat relational algebra) as a standard. Unfortunately, this approach had two major drawbacks: first, it is not very convincing to use a measure of expressiveness based on particular query language, and, second, it has been shown [2] that very natural queries, such as the transitive closure of a binary relation, cannot be expressed

[*] Senior research assistant of the Belgian National Fund of Scientific Research

in the flat relational algebra. Therefore, Bancilhon [3] and Paredaens [23] independently searched for a language-independent measure, which has come to be known as *BP-completeness*, after [7]. In essence, a language is BP-complete if whenever two relation instances satisfy certain structural properties, there exist a query transforming one into the other. Bancilhon and Paredaens showed that the flat relational algebra and calculus were BP-complete, thus formally confirming their naturalness as query languages. Nevertheless, BP-completeness also has its shortcomings, since it is a criterion defined on instance-level. E.g. for each binary flat relation instance, there exists a flat relational algebra expression transforming this instance into its transitive closure; however there does not exist a single expression that will do the transformation for *all* instances, as noticed earlier. Chandra and Harel [7] therefore extended the fundamental ideas behind the notion of BP-completeness to the query-level. The condition they proposed became known as *CH-completeness*. Unfortunately, CH-completeness is generally believed to be too strong a criterion; indeed, CH-complete languages have the computable power of Turing Machines. So, in general, BP-completeness may be too weak as a measure for completeness, whereas CH-completeness is obviously too strong.

Here, we focus mainly on the *nested algebra* as proposed in [32], and some of its extensions. We describe recent results concerning their expressive power. In Section 3, we review a result of [34], establishing the BP-completeness of the nested algebra. This result yields confidence in the naturalness of the nested algebra as a query language for nested relations. There is however another major corollary to this result. In the criterion for BP-completeness, the "structure" of a instance is always described by a flat relation, even if that instance is deeply nested. Therefore, it seems that, in the context of the nested algebra, the "information" contained in a nested relation can be encoded in a flat relation, a feeling which is confirmed by results in [11,34]. This observation also led to another question, discussed in Section 4: is the restriction of the nested algebra to flat relations more powerful than the ordinary flat algebra? Or, in other words, is it possible to write queries in the nested algebra with flat operands as input and with a flat output which do not have an equivalent in the flat algebra? This question was answered in the negative in [24], implying that the nested algebra is hardly an extension of the flat algebra, as far as only expressiveness is concerned.

There is however one major difference between the flat and the nested algebra, which makes the latter a less desirable query language for nested relations than the former for flat relations. In contrast with the flat relational algebra, all "natural" calculi introduced for nested relations are strictly more powerful than the nested algebra. In order to restore the equivalence, an operator should be added to the nested algebra. As is shown by Abiteboul and Beeri [1], the powerset operator, introduced by Kuper and Vardi in [18], is a possible choice for such an operator; the algebra obtained by adding the powerset operator to the nested algebra is called the *powerset algebra*. In Section 5, we discuss the powerset algebra and emphasize its equivalence to several other natural extensions of the nested algebra, as established in [1,12,13]. Among these extensions are closures under least fixpoint operations as well as programming constructs. (Similar extensions for the flat algebra were considered by Chandra and Harel in [7,8].) The equivalence of all these algebras and calculi underlines the naturalness of the powerset algebra as a measure for expressive power for nested relational query languages, although it may be still too powerful from a computational point of view.

2. Nested relations and nested algebra

In this section, we briefly review our formalism for nested relations as well as for the operators of the nested algebra, defined in [32]. Basically, we assume that we start with an infinitely enumerable set U of *atomic attributes* and an infinitely enumerable set V of *atomic values*. From these sets, we build arbitrary attributes, nested relation schemes, nested relation instances and nested relations in a recursive manner. An arbitrary attribute is either an atomic attribute or a set of attributes in which no atomic attribute is repeated. E.g. if $A, B, C, D \in U$, then $\{A, \{B, \{C, D\}\}\}$ is an attribute, but $\{A, \{A, B\}\}$ is not. This condition is obviously included to avoid ambiguity. Non-atomic attributes are also called *composed attributes*. Since composed attributes are sets of attributes they can alternatively be interpreted as *nested relation schemes*. Given a nested relation scheme Ω (i.e. a composed attribute), a *nested relation instance* over Ω is a set of tuples over Ω. A tuple t over Ω is a mapping defined on Ω; if $A \in \Omega$ is an atomic attribute, i.e. an element of U, then $t(A)$ is an atomic value, i.e. an element of V; if $X \in \Omega$ is a composed attribute, then $t(X)$ is a relation instance over X, interpreted as a relation scheme. Finally, a *nested relation* is a pair (Ω, ω) in which Ω is a nested relation scheme and ω a nested relation instance over Ω. Interested readers can find formal details of these definitions in [11,12,13]. Here, we limit ourselves to illustrating them with an example. Therefore, consider a relation representing persons, their jobs and the cities in which these jobs are executed.

{PERSON	{{JOB }	CITY }}
Jeff Willows	professor president	Austin
	consultant	Dallas
Mary Higgins	teacher	Houston

In this relation, there are two levels of nesting: jobs are grouped by the city in which they are executed and sets of jobs and city are grouped by the person having these jobs. In the table above, each box must be interpreted as a nested relation instance the tuples of which are the rows in that box and the scheme of which is the set of attributes embraced by the brackets respectively corresponding to the left and right edge of the box.

To conclude this section, we recall that the *nested algebra* of [32] consists of the following operators: *union* ($\cup$), *difference* ($-$), *cartesian product* ($\times$), *projection* ($\pi_{\Omega'}$), *renaming* ($\rho_{Y \leftarrow X}$), *selection* ($\sigma_{X=Y}$), *nest* (ν_Z) and *unnest* ($\mu_{\Omega'}$). In the above notations, Ω' is a subset of the scheme of the nested relation under consideration, whereas X, Y and Z are attributes belonging to it; X and Y can be either atomic or composed, but must be "compatible" in order to have the operations make sense; Z on the other hand must always be a composed attribute. All these operations, except for nest and unnest, are straightforwardly borrowed from the flat algebra and hence need little explanation. In particular, note that union and difference are only defined on relations having the same scheme, whereas the cartesian product requires relations with schemes that do not share any atomic attribute in order to be consistent with our definitions. This, however, is not a real restriction, since we have a renaming operator. Finally, nest and unnest are two restructuring operators. For details about their precise definition, we refer to [27], in this issue. By executing nested algebra operators consecutively, we can construct nested algebra expressions. In the sequel, $E(x_1, \ldots, x_n)$ will denote an expression which takes n relations as operands.

3. A completeness result for the nested algebra

As explained in the introduction, Bancilhon [3] and Paredaens [23] proposed *BP-completeness* as a language-independent notion of completeness for a query language. In essence, a query language for flat relations, be it specified as an algebra or as a logic, is BP-complete if one can show that:

- For every relation r and every query $E(x)$ of the language, $E(r)$ remains invariant under the permutations that leave r invariant. In other words, if ψ is a permutation on the set of values occurring in r, then $\psi(r) = r$ implies $\psi(E(r)) = E(r)$;
- For every relation r and every relation r', if r' remains invariant for each permutation which leaves r invariant, then there exists a query $E(x)$ in the language such that $E(r) = r'$.

The basic idea behind BP-completeness is that a query language should deal with values as essentially uninterpreted objects, only the equality or inequality of which is relevant. In this philosophy, constants or a partial ordering among values can only be considered by explicitly introducing them through special relations.

In [34], it was observed that the definition of BP-completeness can also be applied to nested relations; subsequently, it was shown that the nested algebra is BP-complete. Afterwards, a generalization of this result to a somewhat richer formalism for nested relations was shown in [11]. These results emphasize to some extent the naturalness of the nested algebra as a query language for nested relations.

It is interesting to observe that the notion of BP-completeness can be rephrased using the notion of the *cogroup*. The cogroup CG(r) of a flat relation r is simply the set of all permutations on the values occuring in that relation, which leave the relation invariant. In particular, the cogroup of the relation in Section 2 consists of two permutations: the identity and the mapping which only interchanges the values ***professor*** and ***president***. Obviously, a query language is BP-complete if and only if for every relation r and every relation r' such that the values of r' also occur in r, the existence of a query $E(x)$ in the language is equivalent to the containment CG$(r) \subseteq$ CG(r') of their cogroups. Although at first, this may seem a rather trivial rephrasing of the original definition, it is important to notice that the cogroup of a relation can be interpreted as a flat relation the attributes of which correspond to the values of the original relation; each tuple of the cogroup relation then corresponds to a particular permutation. So, even for deeply nested relations, the cogroup can be seen as a flat relation. BP-completeness of the nested algebra does hence imply that the "basic information" of any nested relation, with respect to the operators of the nested algebra, can be "encoded" into a flat relation.

There is also a more intuitive, but at the same time, more complicated way to see that a nested relation can be encoded into a flat relation. In general, an unnest performed on a nested relation cannot be undone by the corresponding nest. E.g. if Mary Higgins in the relation in Section 2 would be replaced by Jeff Willows, then unnesting over $\{\{JOB\}, CITY\}$, followed by the corresponding nest, would yield a one-tuple relation, different from the original way. We can however "store" the way in which the data are grouped by first "copying" the column we we want to unnest. This technique is called *tagging*. E.g. by tagging the relation of Section 2 over $\{\{JOB\}, CITY\}$, we obtain:

{PERSON	{JOB }	CITY	{{J2 }	C2 }}
Jeff Willows	professor president	Austin	professor president	Austin
			consultant	Dallas
Jeff Willows	consultant	Dallas	professor president	Austin
			consultant	Dallas
Mary Higgins	teacher	Houston	teacher	Houston

We invite the reader to check that there exists a nested algebra expression involving projection, renaming, cartesian product, selection and unnest, that transforms the relation of Section 2 into the relation represented above. If we would repeat this tagging procedure once more over the attribute $\{JOB\}$, we would end up with a flat relation, provided we were allowed to disregard the internal representation of the copied values. This technique of tagging was used in [34] where the BP-completeness of the nested algebra was reduced to BP-completeness of the flat algebra. In [11], the technique of tagging was formally incorporated in the nested relational model.

4. Comparing flat and nested algebra

From the observations made in the previous section, we may deduce that, as long as only expressiveness is under consideration, there is a strong relationship between the flat and the nested algebra. Hence it is legitimate to wonder whether there exists a difference between the flat algebra and the restriction of the nested algebra to flat relations. In [24], it was shown that there is no difference at all! Hence, if $E(x)$ is an expression in the nested algebra, whose operands and corresponding results are flat relations (the intermediate computations, however, may yield nested relations), then there exists an expression $E'(x)$ in the flat relational algebra such that $E(r) = E'(r)$ for all flat relations r for which $E(r)$ makes sense.

This result confirms our feeling that the nested algebra is hardly an extension of the flat algebra, as far as only expressiveness is concerned. Nevertheless, we need to relax our position a little. Indeed, in order to show the result in [24] mentioned above, it was necessary to first translate the given nested algebra expression into a calculus-like expression and then translate this calculus-like expression into a flat calculus expression. The well-known equivalence between the flat algebra and calculus then yielded the result. At this time, it seems very hard to find an algorithm that gives a direct translation of a nested algebra expression with flat operands and results into a flat algebra expression. Also, Vardi conjectured there might exist a fundamental difference between the length of the nested algebra expression and the corresponding flat algebra expression. As an example we invite the reader to find an equivalent flat algebra expression for the simple nested algebra expression $\mu_{\{B\}}\,\mu_{\{A\}}\,\sigma_{\{A\}=\{B\}}\,\nu_{\{B\}}\,\nu_{\{A\}}$, the operands and results of which are both flat relations over the scheme $\{A, B\}$. Even for such a simple expression, a corresponding flat algebra expression is long and complicated!

5. Extending nested algebra by the powerset operator

There is another major difference between the nested and the flat algebra which was not yet discussed in the previous sections. It is a well-known fact that the flat relational algebra and (domain or tuple) calculus are equivalent. However, if one tries to define a reasonable calculus for nested relations, one obtains a query language being strictly more powerful than the nested algebra. Let us try to explain why this must be the case. If one examines the proof of the equivalence between the flat algebra and calculus, and more in particular, the part in which the calculus is reduced to the algebra (e.g. [20,33]), one sees that one needs a one tuple relation DOM(A), which consists of all values occuring in the operand relation. In the flat algebra, this relation can be easily obtained as the union of appropriate renamings of all the one-attribute projections of that relation. Let us now consider the nested case. There, for e.g. the composed attribute $\{A, B\}$, DOM($\{A, B\}$) should consist of all possible relations over the scheme $\{A, B\}$, i.e. of all subrelations of DOM(A) $\times$ DOM(B). By a combinatorial argument [12], it can be shown there is no expression in the nested algebra that computes DOM($\{A, B\}$). Indeed, for any nested algebra expression, the size of the resulting relation is polynomial in the size of the operand relation; it goes without saying that the size of DOM($\{A, B\}$) is exponential in the size of the original relation. Notice also that it is possible to express the transitive closure of a binary flat relation in most nested calculi.

One way to solve this dilemma could be trying to restrict the nested calculus. However, such a restriction [24,29,30] requires very pathological conditions, that one really does not want to deal with. On the other hand, it is conceptually very easy to extend the nested algebra by introducing the *powerset operator*. If (Ω, ω) is a nested relation with n tuples, then its powerset $\Pi(\Omega, \omega)$ is a relation with a one-attribute scheme $\{\Omega\}$ the instance of which consists of all 2^n subrelations of ω. E.g. the powerset operator applied to the relation in Section 2 yields a four tuple relation over the one-attribute scheme $\{\{PERSON, \{\{JOB\}, CITY\}\}\}$. The *powerset algebra* thus obtained is equivalent to most nested calculi, as shown in [1].

Unfortunately, using the powerset operator to express queries is not very attractive from a computational point of view. One really does not want to first generate all possible solutions to the query in order to be able to select from these the right one. There are however other conceivable extensions to the nested algebra. One such way is introducing the *least fixpoint operator*. Actually, the least fixpoint operator does work on queries rather than relations; it transforms queries satisfying certain monotonicity conditions [31] into other ones. E.g. the transitive closure of a binary flat relation can be obtained by applying the least fixpoint operator to some flat algebra expression. One could now consider the closure of the nested algebra under the least fixpoint operator. In [1,12] it is shown that the query language thus obtained is equivalent to the powerset algebra with respect to expressive power. Still another possibility to enrich the nested algebra is considering programming constructs. In [13], while-loops and for-loops were considered, thereby borrowing an idea of Chandra and Harel [7,8] for the flat algebra. The test in the while-loops involved checking whether a given nested algebra expression returned an empty relation; in the for-loops, the number of times the expression under consideration had to be iterated was determined by the size of an input relation. Provided one allows a minor extension of the powerset algebra in order to be able to deal with undefinedness, which can occur in case of neverending while-loops, we were able to show in [13] that both the closure of the nested algebra under while-loops and for-loops were equivalent to the powerset algebra. In particular, the second result emphasizes the strength of the powerset algebra: although it is not possible to do general counting in the powerset algebra (the powerset algebra is not complete and hence does not possess the computing power of general Turing Machines), some more restricted form of counting is possible. Maybe the powerset algebra is even too strong for several applications.

A good argument in support of measuring the completeness of a flat relational query language by comparing it to the flat tuple calculus, as proposed by Codd [10], is its equivalence to several other reasonable simple formalisms, such as the flat algebra and the flat domain calculus. In the same spirit, we might see the powerset algebra as a standard of expressiveness for query languages for nested relations. In contrast to the flat calculus or algebra, however, we do not know of a language-independent criterion supporting our views for the powerset algebra. Finding such a condition is a problem which is wide open.

References

[1] S. Abiteboul, C. Beeri, "On the Power of Languages for the Manipulation of Complex Objects", *Techn. Report*, March 1988.

[2] A.V. Aho, J.D. Ullman, "Universality of Data Retrieval Languages", *Proc. 6th POPL*, San-Antonio, Texas, Jan. 1979, pp. 110–117.

[3] F. Bancilhon, "On the Completeness of Query Languages for Relational Data Bases", *Proceedings 7th Symposium on Mathematical Foundations of Computer Science*, Zakopane, Poland, in *Lecture Notes of Computer Science 64*, Springer Verlag, 1978, pp. 112-123.

[4] F. Bancilhon, "A Logic Programming/Object Oriented Cocktail", *SIGMOD Record 15:3*, September 1986, pp. 11–20.

[5] F. Bancilhon, S. Khoshafian, "A Calculus for Complex Objects", *Proceedings 5th PODS*, Cambridge, Mass., 1986, pp. 53–59.

[6] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, S. Tsur, "Sets and Negation in a Logic Database Language (LDL1)", *Proceedings 6th PODS*, San Diego, 1987, pp. 21–37.

[7] A.K. Chandra, D. Harel, "Computable Queries for Relational Data Bases", *Journal of Computer and System Sciences 21*, 1980, pp. 156–178.

[8] A.K. Chandra, D. Harel, "Structure and Complexity of Relational Queries", *Journal of Computer and System Sciences 25*, 1985, pp. 99–128.

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

[10] E.F. Codd, "Relational Completeness of Database Sublanguages", *Database Systems*, R. Rustin, eds., Prentice-Hall, Englewood Cliffs, 1972, pp. 65–98.

[11] M. Gyssens, "The extended nested relational algebra", *Techn. Report 87-11*, University of Antwerp, 1987.

[12] M. Gyssens, D. Van Gucht, "The Powerset Operator as an Algebraic Tool for Understanding Least Fixpoint Semantics in the Context of Nested Relations", *Techn. Report no. 233*, Indiana University, Bloomington, October 1987.

[13] M. Gyssens, D. Van Gucht, "The Powerset Algebra as a Result of Adding Programming Constructs to the Nested Relational Algebra", *Proceedings SIGMOD Conference on Management of Data*, Chicago, 1988, pp. 225–232.

[14] G. Houben, J. Paredaens, "The R^2-Algebra: An Extension of an Algebra for Nested Relations", *Techn. Report CSN 87/20*, Techn. University, Eindhoven, 1987.

[15] R. Hull, J. Su, "On the Expressive Power of Database Queries with Intermediate Types", *Proceedings 7th PODS*, Austin, 1988, pp. 39–51.

[16] G. Jaeschke, H.J. Schek, "Remarks on the Algebra on Non First Normal Form Relations", *Proceedings 1st PODS*, Los Angeles, 1982, pp. 124–138.

[17] G.M. Kuper, "Logic Programming With Sets", *Proceedings 6th PODS*, San Diego, 1987, pp. 11–20.

[18] G.M. Kuper, M.Y. Vardi, "A New Approach to Database Logic", *Proceedings 3rd PODS*, Waterloo, 1984, pp. 86–96.

[19] V. Linnemann, "Non First Normal Form Relations and Recursive Queries: An SQL-Based Approach", *Proceedings 3rd IEEE Conference on Database Engineering*, Los Angeles, 1987, pp. 591–598.

[20] D. Maier, "The Theory of Relational Databases", Computer Science Press, 1983.

[21] A. Makinouchi, "A Consideration of Normal Form of Not-Necessarily-Normalized Relations in the Relational Data Model", *Proceedings 3rd VLDB*, Tokyo, 1977, pp. 447–453.

[22] G. Özsoyoğlu, Z.M. Özsoyoğlu, V. Matos, "Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions", *ACM Transactions on Database Systems 12:4*, December 1987, pp. 566–592.

[23] Paredaens J., "On the Expressive Power of the Relational Algebra", *Information Processing Letters 7:2*, February 1978, pp. 107–111.

[24] J. Paredaens, D. Van Gucht, "Possibilities and Limitations of Using Flat Operators in Nested Algebra Expressions", *Proceedings 7th PODS*, Austin, 1988, pp. 29–38.

[25] P. Pistor, F. Andersen, "Designing a Generalized NF^2 Model with an SQL-Type Language Interface", *Proceedings 12th VLDB*, Kyoto, 1986, pp. 278–288.

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

[27] M.A. Roth, J.E. Kirkpatrick, "Algebras for Nested Relations", *Special Issue of Database Engineering on Nested Relations (11:3)*, September 1988, this issue.

[28] M.A. Roth, H.F. Korth, D.S. Batory, "SQL/NF: A Query Language for $\neg$1NF Relational Databases", *Information Systems 12:1*, 1987, pp. 99–114.

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

[30] M.A. Roth, H.F. Korth, A. Silberschatz, "Extended Algebra and Calculus for Not1NF Relational Databases", *Techn. Report TR-85-19*, University of Texas, Austin, 1985.

[31] A. Tarski, "A Lattice-Theoretical Fixpoint Theorem and its Applications", *Pacific Journal of Mathematics 5*, 1955, pp. 285–309.

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

[33] D. Ullman, "Principles of Database Systems", 2nd edition, Computer Science Press, 1982.

[34] D. Van Gucht, "On the Expressive Power of the Extended Relational Algebra for the Unnormalized Relational Model", *Proceedings 6th PODS*, San Diego, CA, March 1987, pp. 302–312.

Nested Relations, a Step Forward or Backward?*

H.-J. Schek
Technical University of Darmstadt
Computer Science Department
Alexanderstrasse 24
D-6100 Darmstadt, West-Germany
e-mail: schek at ddadvs1.bitnet

Abstract

When the first investigation on nested relations appeared, opponents of this direction pointed out that nested relations do not support $n : m$ relationships properly, nor do they consider shared subobjects, nor do they allow recursive data structures, and so they are a step backward. In this paper, however, we will demonstrate by an example that the operations on nested relations, e.g. the nested relational algebra, can be applied without change on recursively defined nested relational schemes. Such schemes are essential within semantic data models, as e.g. KL-ONE, and contain all the missing features mentioned above. Therefore we consider nested relations a step forward rather than backward.

1 Introduction

Since the first investigations on nested relations have been published in the years 1982 and 1983[2,6,15,5], almost immediately controversial discussion started. Opponents of this direction - somewhat simplified - said that there is no support for shared subobjects, nor for $n : m$ relationships, nor for recursive data structures. Others insisted in the statement that the simplicity of Codd's model is lost, and some even lamented that this is a step back to the old world of hierarchical data bases.

In fact, considering the latter aspect first, it is true that types of hierarchically organized data are obtained if a tuple constructor and a set constructor repeatedly are allowed for the construction of complex object types. This is presented nicely and in a compact way in [1]. In semantic data modeling these constructors realize aggregation and association and so, they are desired by a broad database community. People arguing with "back to IMS" forget that the real question in this context is, what kind of operations is allowed on these types. The nested relational algebra extends the high-level set-oriented operations on flat relations to any set-of-tuple type which is encountered in a hierarchical data structure. This is the important difference to the old world of hierarchical data models.

With regard to the second concern let us consider the question of $n : m$ relationships and recursive data in more detail. In fact, if we take over a static schema definition for nested relations as we are used to in the flat model, we were unable to model symmetrically $n : m$ relationships, apart from introducing references symmetrically. Therefore this objection against the nested relational model is more serious. It is even substantiated by the HDBL language of the AIM-P project [12]. The user of this model has to distinguish whether an object explicitly appears as an attribute in a relation or whether only references ("surrogates") appear there.

*This is a written version of the author's presentation for the panel discussion at the SIGMOD Conference, Chicago, 1988.

For the latter case the "materialize" operation has to be applied. There seems to be no way out if the static schema definition for nested relations is kept and it looks like we were unable to invalidate the previous objections.

Fortunately there is no reason why we should keep a static nested relational schema definition for this layer. In fact, in the DASDBS [11] object layer the starting point is not a static nested relational schema but rather a recursive relational schema definition. This is in the spirit of definition of a KL-ONE oriented semantic data model [4] which allows us to deal with $n : m$ relationships and shared objects in a symmetric way. Nested relations then appear quite naturally as results of operations on this net. As the schema of the nested relation is derived from the KL-ONE net and from the operation we call these "dynamically nested relations".

In the following we will introduce and explain our example in section 2. We then show in section 3 how nested algebra or nested SQL can be applied on recursive nested relations, i.e. on a KL-ONE oriented net. In section 4 we introduce a user-friendly version of nested SQL and show how we are able to extend this language for recursive queries.

2 Recursive Relational Schema Definitions, KL-ONE and Nested Relational Views

As a running example throughout this paper we take the following type definitions in a hypothetical - say PASCAL++ - language:

```
type project = record
                 pno : integer;
                 pn : name;
                 members : set of employee;
                 produces : set of part;
              end;

type employee = record
                  eno : integer;
                  ename : name;
                  assignmts : set of projects inverse members;
                  education : set of course;
                  manages : set of employee;
                end;

type part = record
              pname : name;
              produced_by : set of project inverse produces;
            end;

type course = record
                cname : name;
                  ⋮
              end;
type projects = set of project;
type employees = set of employee;
type parts = set of part;
```

Apparently this is not Standard Pascal because - as mentioned - set of "structured" type is not allowed (the first +). More importantly, recursively defined types occur (the second +). Recursion not only occurs because of the manages component in employee which is defined by the employee type again in the manages component, but, more generally, due to the description of one object by means of others and vice versa. For example project is described by its members which are employees and employees are described by their project assignments which is the

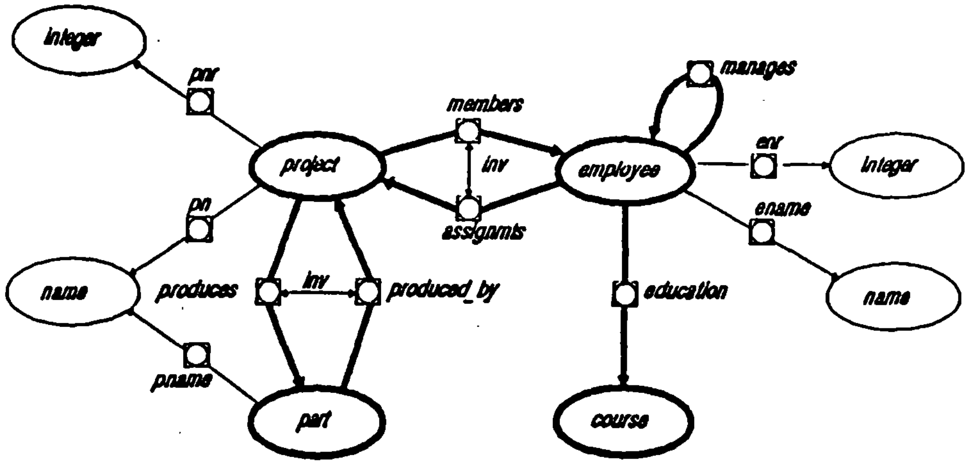


Figure 1: KL-ONE

"inverse" relationship to members in project. We may also regard the definitions above as the definition of the schemes of *recursive* nested relations. This is because the component of a tuple (i.e. record type) is either atomic or is a set of tuples. Note, however, that we no longer exclude recursive schema definitions as in [16].

2.1 Relationship to KL-ONE

Those who are familiar with Brachman's KL-ONE semantic network for knowledge representation [4] will have recognized that the view we have taken is the same as KL-ONE. In fact the previous type definitions in PASCAL++ are not hypothetical. They directly correspond to the definition of KL-ONE "generic concepts". Project in this terminology is a generic concept with "roles" *pno, pn, members*, and *produces*. The roles *pno* and *pn* have to be filled with at most one value from some primitives while members is filled with a set of values from a generic concept *employee* and *produces* is filled with a set of *parts*. So everything in our previous example can be expressed in KL-ONE's terminology definition (called T-Box there). A corresponding graphical representation is shown in Figure 1.

2.2 Examples of Non-Recursive Nested Relational Views Taken From the KL-ONE Net

The symmetric view taken in KL-ONE, i.e the recursive type definitions, are the reason why we are not able to assign a static nested relational schema being equivalent to the type definitions. Nevertheless we see that, basically, there are tuples (projects, employees, parts) with relation valued attributes (components). The interesting fact is that many different nested relational schemata are contained in the previous definitions. We could get

(1) *projects (pno, pn, members (eno, ename))*

or, symmetrically we may see

(2) *employees (eno, ename, assignmts (pno, pname))*

In these examples we see either projects as the root and the member with name of their employees as children, or, vice versa, we see employees at the root and their projects as children. More interestingly, we could also see

```
(3) parts (pname,
        produced-by (pno,
                members (eno),
                produces (pname))),
```

This gives us a nested relation where for each part we find in which projects the part is produced (in the relation-valued attribute produced-by) and for each such project we see the employees (by their number *eno*, in *members*) in this project and all parts which are produced there (by their name *pname* in *produces*).
In order to see who manages whom we could look at

(4) *employees(eno, manages (eno))*

The principle we applied in all these examples is straightforward: We decided which object type should be the root and selected a subset of its components. For each selected component which is a set type we selected again which components of the children we wanted to see and so on. More generally we could think of a (data definition) language which would allow to define non-recursive nested relational views after having specified objects in our PASCAL++ type definition.

3 Nested Algebra or Nested SQL for KL-ONE

The exciting fact now is that such a language already exists [8] : We can apply the nested relational algebra [16] or nested SQL [12,13,7] without any change. We are able to express all previous examples by regular nested algebra expressions or by nested SQL in a very simple way as shown by the following examples

```
(S1)  select pno, pn,                          (A1)  π[pno, pn,
         (select eno, ename from members)               π[eno, ename](members)](projects)
      from projects
```

We omitted renaming here, i.e. we called the resulting objects "projects" with their attributes *pno*, *pn*, and *members*. The symmetric case, also without renaming, is

```
(S2)  select eno, ename,                       (A2)  π[eno, ename,
         (select pno, pn from assignmts)                π[pno, pn](assignmts)](employees)
      from employees
```

Notice the wellknown role of the select clause as type constructor (or the π in the algebra). As in the usual SQL the result of the select clause is a set of tuples with components as designated by the select list. The following examples are expressed as easily as the previous ones

```
(S3)  select pname,                            (A3)  π[pname,
         (select pno,                                   π[pno,
            (select eno from members),                     π[eno](members),
            (select pname from produces),                  π[pname](produces)]
         from produced_by)                              (produced_by)]
      from parts                                     (parts)
```

Also a recursive type is simply handled

```
(S4)  select eno, (select eno from manages)    (A4)  π[eno, π[eno](manages)](employees)
      from employees
```

The previous examples have shown how the schema of a nested relation can be defined dynamically. At the same time the instances are defined. Up to now we have applied projection which selects a subset of attribute values in each tuple as usually. But we may apply selection too. We may restrict the instances by some predicates as usually. If, for example, we were interested in all projects but wished only to see the member with name Smith we would write

```
(S1') select pno, pn,
          (select eno from members
          where ename='Smith')
      from projects
```

Note that an empty set would be returned if a project has no Smith among its members. In order to see people reporting directly to John we would write

```
(S4') select eno,
          (select eno from manages)
      from employees
      where ename='John'
```

While this looks elegant there are still two concerns which we must look at. The first concern is user friendliness related also to simplicity, the second is the subject of recursive queries.

4 A User-Friendly Version of Nested SQL

Fortunately there is a proposal which may be an answer to both concerns at the same time. The idea is to apply simple, flat SQL-like statements and to connect them via names [9]. Let us first look into the aspect of user-friendliness taking some of our previous examples and rewriting them

```
(F1) begin select
          select pno, pn, M from projects
          M: select eno, ename from members
     end select
```

```
(F3) begin select
          select pname, P from parts
          P: select pno, X, Y from produced_by
          X: select eno from members
          Y: select pname from produces
     end select
```

As it can be seen, we introduce names in the select clauses as placeholders and define them later step by step. In the previous example we know that we want to see *pname* together with something we abbreviate *P* for every *part*. *P* is defined later as a set of tuples calculated from *produced-by* which is a set-valued component of *part*, the element type of *parts*. Within *P*, in turn, we select *pno* and two things called *X* and *Y* defined later for every element in *produced-by* which as we know is of type *project*. The value *pno* is a component of a basic type in *project* whereas *X* or *Y* are computed from the set-valued components *members* or *produces* of *project* respectively. The simple rule is that we specify by select what we want to see if a set-valued component (object or subobject) is encountered. In other words, a select * as a default for selecting everything must be applied more carefully.

This "trick" with names allows us to handle the recursively defined types in a way as simple as the nonrecursive ones. The manages example is one:

```
(F4') begin select
          select eno, Down from employees
              where ename='John'
          Down: select eno from manages
      end select
```

In a similar way we could get people reporting directly to John and those at the next lower level under John

```
(F4*) begin select
          select eno, Down1 from employees where ename='John'
          Down1: select eno, Down2 from manages
          Down2: select eno from manages
      end select
```

Also aggregate functions can be used easily as in

```
begin select
      select pname, P from parts
      P: select pno, C from produced_by
      C: select count(*) from members
end select
```

which gives us for every part the *pname* and the set of tuples consisting of the project number *pno* and the number of employees *C* working in this project for all projects producing this part.

5 Recursive Queries

While the previous examples still can be handled with standard operations of the nested relational model we have to extend the model for real recursive queries, e.g. if we want to see the complete hierarchy under John. The difference is that in all previous examples we can determine the type of the result, i.e. the schema of the resulting nested relation by parsing the select statements without execution. In other words, given the recursive type definition in PASCAL++ (i.e. the KL-ONE net definition) and given the query specification, the type of the result can be determined at compile time. For example in F4" the type is

Q(eno, Down1(eno, Down2(eno)))

just by inspecting the select lists. This is no longer possible for John's complete hierarchy as we don't know before how many levels we will obtain in a computation. However, it is simple to express these kinds of queries. The utilization of names within SQL expressions becomes crucial now and is not only "syntactical sugar" [9]. Look at the hierarchy below John:

```
(F4*)begin select
          R: select eno, Down from employees where ename='John';
          Down: select eno, Down from manages
     end select
```

As in the previous examples the result *R* is the set of "John" tuples consisting of *eno* and *Down*, defined in the next statement. This defines *Down* recursively: The type of the result is recursive

```
type R = set of record
                eno: integer;
                Down: R
             end
```

or equivalent to a recursive nested relation

R(eno, Down(R)).

Its instances may have an arbitrary depth. As soon as the manages component is empty, the result is empty, the recursion terminates and the empty set is returned to the next higher layer indicating that we arrived at an employee who is not manager.

6 Conclusion

We have shown that the operations of the nested relational model can be applied without change to a semantic data model such as KL-ONE. The key observation was that we can give up a static (nested) relational schema definition and replace it by recursive nested relational schemes, i.e. by KL-ONE oriented "concepts". The result of a nested algebra expression is a non-recursive nested relation which is defined dynamically by the nested algebra expression. We have sketched a user-friendly version of nested SQL which allows to specify a nested relation query stepwise. The extension of this language allowed to formulate recursive queries whose result is no longer a static nested relation but again a recursive nested relation. The latter aspect of recursive queries as well as the aspect of updates deserves further investigations.

We did not mention the advantages of nested relations over flat relations with respect to more natural normal forms [10] and its ability for a theoretical foundation on complex object languages [3,19,14]. We see also advantages of this model in describing storage structures formally and so in broadening the scope of algebraic optimization [18,17]. If we take all these nice results together we should conclude that nested relations are a step forward.

7 Acknowledgement

I would like to thank Bin Jiang for many discussions and for his help in preparing this presentation.

References

[1] F. Bancilhon and S. Khoshafian. A calculus for complex objects. In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 53–59, ACM, New York, Cambridge, 1986.

[2] F. Bancilhon, P. Richard, and M. Scholl. On line processing of compacted relations. In *Proc. Int. Conf. on Very Large Databases*, pages 263–269, Mexico, 1982.

[3] C. Beeri and S. Abiteboul. An algebra and a calculus for complex objects. In M. H. Scholl and H.-J. Schek, editors, *Handout Int. Workshop on Theory and Applications of Nested Relations and Complex Objects*, Darmstadt, April 1987. (Position Paper. Proceedings to be published with full papers).

[4] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[5] P. C. Fischer and S. J. Thomas. Operators for non-first-normal-form relations. In *Proc. IEEE Computer Software and Applications Conf.*, pages 464–475, 1983.

[6] G. Jaeschke and H.-J. Schek. Remarks on the algebra of non-first-normal-form relations. In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 124–138, ACM, New York, Los Angeles, 1982.

[7] P.-A. Larson. The data model and query language of LauRel. Unpublished manuscript, University Waterloo, 1988.

[8] G. Lausen and H.-J. Schek. Semantic specification of complex objects. In *Proc. IEEE CS Symp. on Office Automation*, Gaitherburg, 1987.

[9] R. Lorie and H.-J. Schek. On dynamically defined complex objects and SQL. Accepted for the 2nd International Workshop on Object-Oriented Database Systems, Sept. 1988. Bad Münster, Germany.

[10] Z. M. Ozsoyoglu and L. Y. Yuan. A normal form for nested relations. In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 251–260, ACM, New York, Portland, March 1985.

[11] H.-B. Paul, H.-J. Schek, M. H. Scholl, G. Weikum, and U. Deppisch. Architecture and implementation of the Darmstadt database kernel system. In *Proc. ACM SIGMOD Conf. on Management of Data*, ACM, New York, San Francisco, 1987.

[12] P. Pistor and F. Andersen. Designing a generalized NF^2 model with an SQL-type language interface. In *Proc. Int. Conf. on Very Large Databases*, pages 278–285, Kyoto, August 1986.

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

[14] M. A. Roth, H. F. Korth, and A. Silberschatz. *Extended Algebra and Calculus for ¬1NF Relational Databases.* Technical Report TR-84-36, University of Texas at Austin, Austin, TX, 1984. (revised version, January 1986).

[15] H.-J. Schek and P. Pistor. Data structures for an integrated database management and information retrieval system. In *Proc. Int. Conf. on Very Large Databases*, pages 197–207, Mexico, 1982.

[16] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, June 1986.

[17] M. H. Scholl. Theoretical foundation of algebraic optimization utilizing unnormalized relations. In *ICDT '86: Int. Conf. on Database Theory, Rome*, pages 380–396, LNCS 243, Springer, Berlin, Heidelberg, 1986.

[18] M. H. Scholl, H.-B. Paul, and H.-J. Schek. Supporting flat relations by a nested relational kernel. In *Proc. Int. Conf. on Very Large Databases*, pages 137–146, Brighton, September 1987.

[19] D. van Gucht. On the expressive power of the extended relational algebra for the unnormalized relational model. In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 302–312, ACM, New York, San Diego, March 1987.

ER APPROACH

7th International Conference on Entity-Relationship Approach

November 16-18, 1988 **Rome, Italy** **Hotel Ambasciatori**

Sponsored by: In Cooperation with:

ER Institute

ACM SIGBDP (Approval Pending)

IEEE Computer Sociey

DAMA Data Administration Management Association

afcet

AICA

The Entity-Relationship Approach is the basis for many Database Design and System Development Methodologies. ER Conferences bring together practitioners and researchers to share new developments and issues related to the use of the ER Approach.

3-DAY CONFERENCE FEATURES:

* Internationally Known Experts in Database and Information Management from both Academic and Practitioner Communities
* Tutorials, Technical Presentations, Panel Discussions, and Vendor Demonstrations
* Published Conference Proceedings and Tutorial Materials
* Traditional Italian Banquet
* Lunches and Continuous Coffee Service Provided Each Day of the Conference

PROGRAM HIGHLIGHTS:

Papers Presented on Major Themes:

Database Development and Management

* database design
* database management systems
* languages for data description and manipulation
* data models and modeling
* database dynamics
* database constraints

Application Systems

* CAD/CAM and engineering databases
* knowledge-based systems
* object-oriented systems
* multi-media databases
* user interfaces
* business systems

Managing Organizational Information Resources

* data planning
* information architectures
* translating data plans into application systems
* data dictionaries
* information centers
* end user computing

Tutorials Include:

* Klaus Dittrich
  Universitat Karlsruhe (West Germany)
  Object Oriented Databases
* Stefano Ceri
  Politecnico di Milano (Italy)
  Distributed Database
* Potevolpe
  Italsiel (Italy)
  Software Maintenance
* Sham Navathe
  University of Florida (USA)
  Conceptual and Logical Database Design

Conference Location:

The Conference will be held at the Hotel Ambasciatori, a First Class Hotel located at Via Veneto 70, Rome, Italy 00817 (396-47493). Via Veneto is among the most touristic streets in Rome. More moderately priced hotels, convenient to the conference include: Hotel Savoy, Via Ludovisi 15 (396-4744141), Sicilia Hotel, Via Sicilia 24 (396-493841). Less expensive hotels include: Hotel Veneto, Via Piemonte 63 (396-789251), Dan Hotel, Via Boncompagni 37 (396-483520), Hotel Alexandra, Via Veneto 18 (396-481943). Make arrangements directly with the hotel of your choice.

CONFERENCE FEES:

	Prior to Sept. 25	After Sept. 25
__ Regular	$375.00	$425.00
__ Discount	$325.00	$375.00
__ Student	$100.00	$150.00

* Discount applies to Cooperating and Affiliated Organizations
* Student Fee Excludes Luncheons and Banquet

U.S. REGISTRATION:

Professor Salvatore T. March
Department of Management Sciences
University of Minnesota
271 19th Avenue South
Minneapolis, MN 55455 USA
(612) 624-2017
march@umnacvx.bitnet

NAME ____________________ Discount Organization ____________________

ADDRESS ____

CITY ____________________ STATE ______ ZIP ________ PHONE ____________

Make Checks Payable to: Seventh International ER Conference (US Dollars Only)