Advanced Information Management (AIM): Advanced database technology for integrated applications

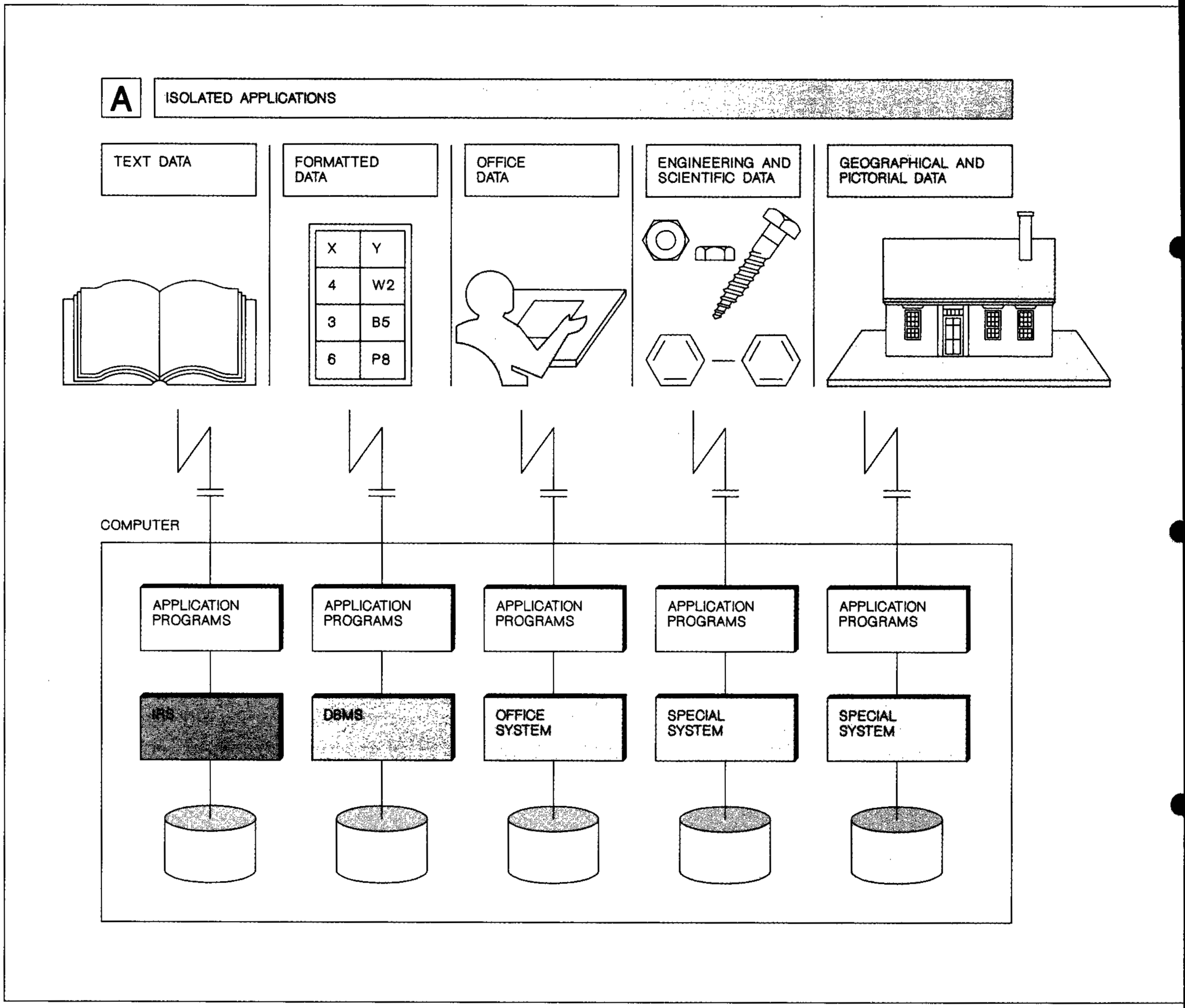
by P. Dadam
V. Linnemann

The Advanced Information Management (AIM) project is currently one of the main activities at the IBM Scientific Center in Heidelberg. The main purpose of the project is to understand the database requirements and respective solutions for advanced integrated applications such as computer-integrated manufacturing and computer-integrated office. These application areas require an advanced database technology which is able to manage a large variety of data of various types in a consistent and efficient way. The underlying database technology should support not only simple numbers and simple tables used in business administration, but also large complex structured objects, including text, image, and voice data, in a uniform way. This paper describes the background, goals, and accomplishments of the AIM project. It also provides an overview of the design goals, the implementation, and the underlying concepts of AIM-P, an experimental database management system under development in the AIM project.

Advanced application areas require an advanced database technology which is able to manage a large variety of data of various types. By representing these types of data as "naturally" as possible from an application point of view, complex mappings from the data representation used in the application program to the data representation offered at the database interface are avoided. This point is important if database technology is to become a productivity aid and not just an integration tool for application programming. Which representation is natural may be application-dependent. A system for computer-aided design (CAD) may use *object-oriented* data, e.g., a computer board x and its related (structured) objects, and a computer board y and its related objects, whereas a system for computer-aided manufacturing (CAM) may use *data-oriented* data, e.g., the type and number of chips used across all computer boards regardless of which objects these chips belong to. This means, in order to be adequate for computer-integrated manufacturing (CIM), database technology needs to support different *views* for one and the same type of data or object as well as to support a large variety of different data types in a uniform way.

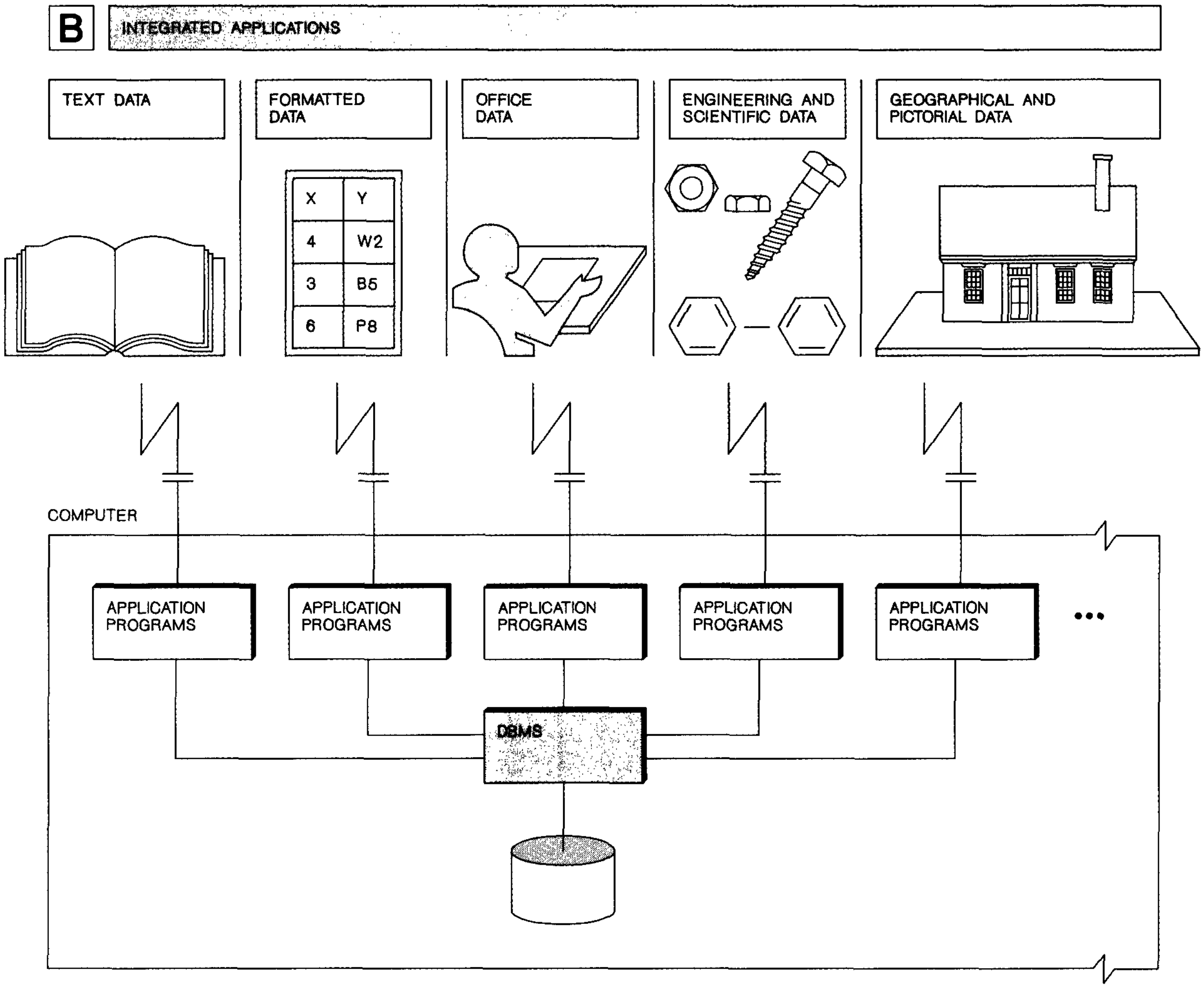
Today's database management systems have been designed with business administration applications in mind. They are not able to adequately support application examples such as those outlined above with respect to data model and efficiency aspects. As

Figure 1 Current and desired scenario. Note, one *type* of DBMS is meant in (B), not necessarily one *physical* database containing all the data.



a consequence, a large variety of special-purpose data management or file systems are currently in use for each major application area (Figure 1A). 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 data across the different application areas in a uniform and consistent way could improve this situation (Figure 1B). Very likely no single type of system will adequately support all application types; however, a goal would be to cover 80 to 90 percent of the main application types.

Currently, the area of extended database technology is quite heavily investigated. In the following, we review some of the work reported in the literature that has influenced this project.



The XSQL project[1] introduced the term *complex object* into the database world. Using special attributes (composed_of, component_of), hierarchical structures can be defined using a flat relational data model. At run time, these special attributes are used for collecting directly the tuples that make up a certain complex object to avoid unnecessary join operations. As part of the XSQL application program interface, a dedicated main memory data structure is used to pass the complex object's tuple structure and content to the application program.[2]

Postgres supports procedures consisting of Postquel statements, as well as procedures written in a conventional programming language such as LISP or C.[3–7] In this approach, attributes of database tables may be procedure-valued, i.e., an attribute value may be a procedure written in Postquel or C. Whenever the attribute is accessed, the corresponding procedure is called. Moreover, the concept of abstract data types is supported by Postgres, but only as a low-level representation of an unstructured storage area. Only the length of the area is given; there is no strong typing as far as the representation of an abstract data type is concerned. This method is also used for passing parameters from Postgres to functions written in LISP or C.[5]

PROBE[8,9] distinguishes between entities and functions. Access to the attribute values of an entity is only provided by invoking the corresponding function. Functions can be provided by the system or be user-defined.

The Starburst project[10,11] investigates how to design the database management system (DBMS) architecture so that storage alternatives for relations and foreign indexes can be supported.

GENESIS[12] and EXODUS[13] are, in essence, software engineering tools for configuring a DBMS according to a given specification. GENESIS relies on database components whose interfaces have been standardized in such a way that the components become exchangeable. One goal of EXODUS is to provide kernel DBMS facilities and software tools for the partial generation of application-specific DBMSs. Under the assumption that in the future there will exist large libraries of application-oriented data types and respective functions which can be optionally added to a database kernel (customization), tools such as GENESIS or EXODUS will be necessary to configure these systems.

The DASDBS project[14] provides a database kernel on top of which different application-oriented database interfaces can be provided. Support of nested relations, nested transactions, query optimization (supporting flat relational views on nested database relations), extensibility, and architectural aspects are treated in this project.

The PRIMA[15] project with its underlying data model is heavily influenced by the molecular objects approach.[16] It has an SQL-like data manipulation language which supports references to model recursive or arbitrary network-like data structures. Special emphasis is given to architectural issues and the processing of recursive queries.

More information can be found in the literature[17–19] on projects dealing with object-oriented database technology, as well as descriptions of projects[20–22] dealing with the foundations of (extended) relational technology.

This paper describes the Advanced Information Management (AIM) project which is currently one of the main activities at the IBM Scientific Center in Heidelberg. The main purpose of the AIM project is to understand how database technology can serve as a useful integration tool (see Figure 1B) for integrated applications such as CIM and computer-integrated office (CIO). This paper also describes the function and architecture of an experimental database management system based on the *extended* NF^2 (Non First Normal Form) *data model*, a relational data model.

The first section of this paper outlines the background and goals of the AIM project, followed by a section that describes the database language and the underlying data model. Additional sections: show how the database language can be extended by user-defined data types and functions; describe the application program interface (API) that allows the user to use the system from a programming language; and detail the system architecture. The paper concludes with a summary and an outlook for future work.

The Advanced Information Management project

The AIM project began in 1979, combining relational technology[23] with a new text indexing technique.[24–26] Looking at office-oriented applications, it was discovered that the pure relational data model, even when complemented with text search capabilities, was not suitable for modeling complex data objects such as books, office documents, and forms. On the other hand, relational database technology—with its flexibility for formulating database queries, structuring the results, defining alternative views over stored tables, and other features—clearly was the direction to follow. The desire to support structured objects in a relational way finally led (independent from other groups like VERSO[27]) to the idea of *nested relations.* They were called *NF^2 Relations* because the First Normal Form which requires that attribute values have to be atomic[28] had been dropped. Clearly, the most critical point in this case was whether this extended relational model could be put on a theoretical basis as equally sound as the original one. At first the project concentrated primarily on the theoretical issues of this data model, especially on its relationship to the relational design theory (functional and multivalued dependencies). This led to scientific contributions to the theory of nested relations.[29–32] In parallel to this more fundamental research work, conceptual work was begun that aimed at the development of an extended SQL-like database language able to deal with the extended relational data model at the user's level.[33–36]

In 1982 and 1983 the issue of integration of applications across formerly isolated application areas, as outlined earlier in the paper, and the understanding of the related database requirements and problems became important. It was therefore decided to redirect the research and development activities to look at database-related issues on a broader scope. The main objective was to understand the database requirements and how possible solutions for the related problems could be developed using an experimental type of DBMS, the Advanced Information Management Prototype, called *AIM-P.*

Complex objects should not be treated as special cases.

The key concept to be evaluated using the experimental database management system was the NF^2 data model because of its capability to support hierarchical structures and tables in a uniform, relational way. It also has a powerful query capability to treat the same type of data in both an object-oriented way and a data-oriented way. However, instead of using the pure NF^2 data model, an extended version supporting lists and sets in a more general way was used and is referred to as the *extended* NF^2 data model.[37] Other database-related aspects were studied in addition to the data model, by using this prototype model. The goals for the overall system design and the related research and development effort were characterized as follows.

Architecture. The DBMS architecture should support a workstation–server environment. That is, a central database server should maintain the shared data while the actual processing of database objects (data or object creation and manipulation) should be performed at workstations (user application front ends). Special attention should be given to provide adequate data exchange mechanisms between the server and the workstation in order to reduce the communication overhead, especially in cases where large complex objects are involved. In other words, the overall architecture should support efficient cooperative processing of complex objects in a workstation–server environment.

Database language and data model. The database server should provide a homogeneous view of all the data (from flat relations to complex objects) to serve as the integration tool. That is, complex objects should not be treated as special cases but should be an integral part of the data model. All, or nearly all, operations defined for 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. The interface that is offered to the user or application program at the workstation should be application-dependent.

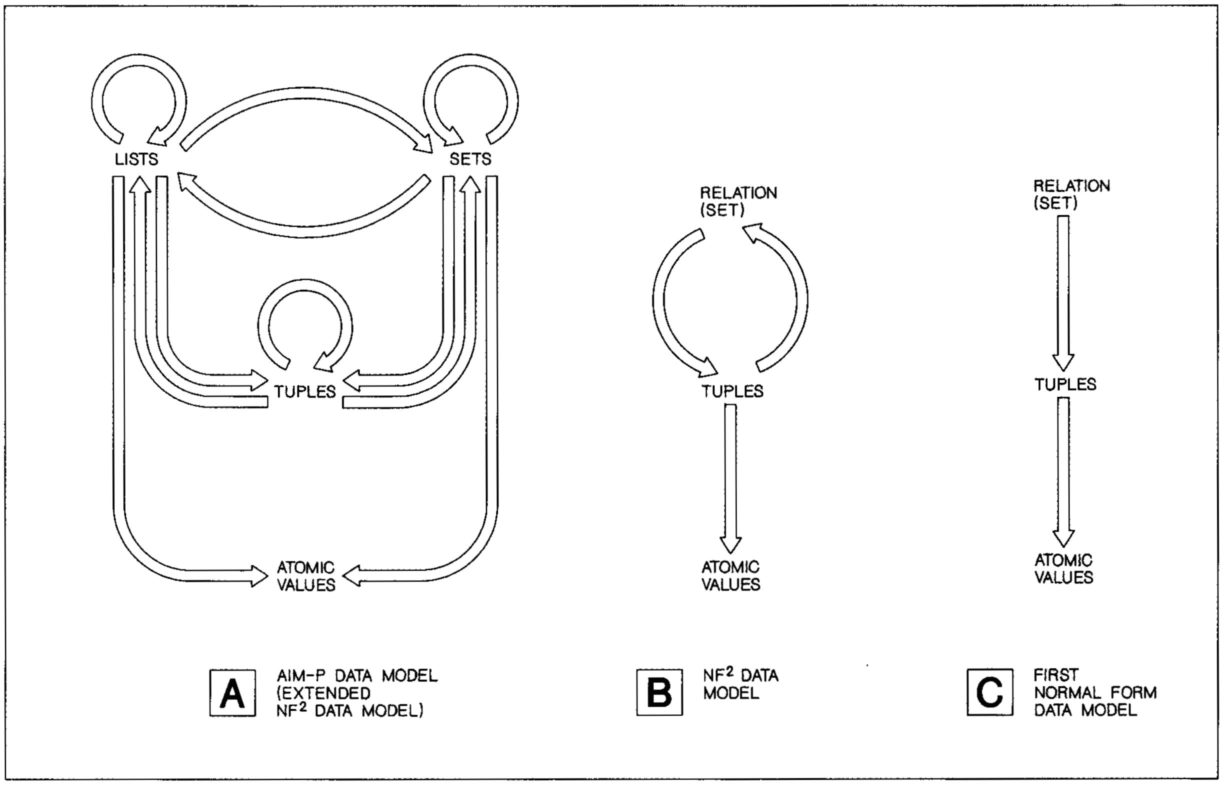
Extensibility. In the long term, database management systems should become more "customizable" according to the applications' (users') needs. Required functions, such as filter operations to select the correct objects from the database server, should become part of the query language of the server rather than being part of the application program of the workstation which performs the data selection only after all of the data have already been transmitted. As a first step in this direction, user-defined data types and functions should be supported by the database server.

Heidelberg Data Base Language (HDBL)

The following description concentrates on the AIM-P data model and the corresponding database language. Only a brief discussion of this Heidelberg Data Base Language (HDBL) is given here. A more comprehensive treatment can be found in the referenced literature.

Data model. The data model supported by AIM-P is an object-oriented generalization of NF^2 respectively nested relations. In the beginning, the AIM project concentrated on the pure NF^2 data model complemented by the concept of *ordered relations* and by a

Figure 2 Comparison of AIM-P data model, pure NF², and relational data model



list of atomic values as attribute values in order to adequately support numerical vectors, matrices, and similar constructs. Soon it was discovered that an orthogonal data model would be much more advantageous, not only from a user's point of view but also to ease query processing internally. That is, atomic types and constructor types should be combinable in an easy fashion, so that all resulting types which can occur based on legal queries are covered by the data model. SQL,[38] for example, is not orthogonal in that sense. The flat relational data model on which SQL is based only knows *sets of tuples.*

These considerations resulted in a data model design based on the concepts of *constructor types* (set, list, tuple) and *atomic types* (date, real, integer, Boolean, character, string [text], surrogate[39]). All constructor types can be combined with each other and with atomic types in an arbitrary way. Moreover, each of these constructs (constructor types and atomic types) can occur at every level of an object type. The attributes of a tuple-valued object, for example, can be either atomic, set-valued, list-valued, or again tuple-valued. Objects need not occur as elements of a table. A list of lists of real values (which is a two-dimensional matrix) can occur as an element in another list or set, as an attribute value within a tuple, or as a single standing object (having an object name). An object may be even as simple as a single integer—for example, the highest invoice number used so far. Figure 2A shows a graphical representation of this data model; both the Normal Form data model and the pure NF² data model are special cases of this more general data model. The data model is also called the *Extended NF² Data Model* because it was developed in an evolutionary manner from the NF² data model. Implementation issues (for example, storage structures for extended NF² data objects) are discussed in Reference 37.

Database language. AIM-P has an SQL-like language interface, HDBL, but in contrast to SQL, the user of HDBL is required to explicitly define the type of the result. In SQL the only result type supported is *set of* *tuples* and an expression like

```
SELECT dno
FROM   departments
```

would always lead to a result table having one column (unary relation). In HDBL, this would also be a legal result type, but a set of atomic values would be valid too. For this reason, HDBL uses the above-mentioned constructor types to explicitly describe the desired structure of the result elements (except that the source structure is to be directly used). It uses **tuple(...)** or **[...]** to define a tuple structure, **list(...)** or **<...>** to define a list structure, and **set(...)** or **{...}** for a set structure. Whether the elements of the result are unordered or ordered depends on the source data. If they are ordered, the result is ordered; otherwise it is unordered. If the result is computed using join operations, it is ordered if all involved tables are ordered (lists of tuples); otherwise it is partially ordered (join between sets and lists of tuples) or unordered (only sets of tuples involved).

In the following, HDBL is described by some examples. More comprehensive treatments of this language can be found in References 40 to 42; its relationship to the relational algebra and processing rules are described in References 43 and 44; and a description of the currently implemented language features can be found in Reference 45.

The basis for the subsequent examples is a table containing information about manufacturing departments, illustrated in Table 1. The corresponding CREATE statement is given in Figure 3. Like classical SQL, HDBL uses a SELECT-FROM-WHERE (SFW) construct to provide the facilities for expressing projections, restrictions, and joins. The SFW construct of HDBL is, however, far more powerful than that of the original SQL. The examples that follow show this. The first example shows how to retrieve the whole *manuf_depts* table.

```
SELECT x
FROM   x IN manuf_depts
```

The next example retrieves all numerical control (*nc*) machines.

```
SELECT nc
FROM   m    IN manuf_depts,
       cell IN m.manuf_cells,
       nc   IN cell.nc_mach
```

This example shows how subtables are retrieved: A variable *m* is defined which is bound to *manuf_depts*. The variable *cell* depends on *m*. The variable *nc*, in turn, depends hierarchically on *cell*. If all nc-machines are not of interest but only those with *qu* greater than 1, a corresponding predicate can be added to the query as follows:

```
SELECT nc
FROM   m    IN manuf_depts,
       cell IN m.manuf_cells,
       nc   IN cell.nc_mach
WHERE  nc.qu > 1
```

Table 1 The manuf_depts information in NF² representation

{ 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 3 **CREATE statement for manuf_depts of Table 1: {. . .}, < . . . >, [. . .] are set, list, and tuple constructors (alternatively, set (. . .), list (. . .), and tuple (. . .) could be used)**

```
CREATE manuf_depts
 { [    dno   : integer,
        dname : string(40),
        manuf_cells :
          { [ cid  : string(10),
              non_nc_mach :
                { [ qu   : integer,
                    type : string(40) ] },
              nc_mach :
                { [ qu   : integer,
                    type : string(40) ] }
            ] },
        staff :
          { [ eno      : integer,
              function : string(40) ] }
   ] }
END
```

The following example shows how SFW constructs can be nested. For each manufacturing department, only those manufacturing cells that have an *nc_mach* of type Flex 200 shall be retrieved:

```
SELECT [m.dno,
        manuf_cells:
         (SELECT [cell.CID, cell.nc_mach]
          FROM   cell IN m.manuf_cells
          WHERE  EXISTS (nc IN cell.nc_mach):
                 nc.TYPE = 'Flex 200')]
FROM    m IN manuf_depts
```

With the same subquery technique, nesting of tables can also be formulated. Assume two flat tables MDEPT (dno, dname) and *staff* (dno, eno, function). On the basis of these source tables, the *manuf_depts* table (Table 1) 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 of a nested table is formulated similar to a join. As an example, to unnest *manuf_depts* (Table 1) along the path from top to STAFF while retaining the *dno*, *dname*, *eno*, and *function* attributes, the following HDBL expression could be used:

```
SELECT [ x.dno, x.dname,
         y.eno, y.function ]
FROM   x IN manuf_depts, y IN x.staff
```

Clearly, HDBL can also be used to modify data. To delete manufacturing cell C11, for example, the following statement could be used:

```
DELETE mc
FROM   md IN manuf_depts,
       mc IN md.manuf_cells
WHERE  mc.cid = 'C11'
```

The quantity of *non_nc_mach* MLDX 300 within manufacturing cell C13 of department 15 can be incremented by 1 as follows:

```
ASSIGN nnc.qu+1
TO     nnc.qu
FROM   md  IN manuf_depts,
       mc  IN md.manuf_cells,
       nnc IN mc.non_nc_mach
WHERE  md.dno = 15 and
       mc.cid = 'C13' and
       nnc.type = 'MLDX 300'
```

An insertion of a new manufacturing department with no manufacturing cells and no staff could be performed as follows:

```
INSERT
   { [ dno:          33,
       dname:        'new_name',
       manuf_cells: { },
       staff:        { } ] }
INTO manuf_depts
```

A more complex table, the *robots* table, demonstrates some of the HDBL concepts that go beyond the pure NF^2 data model. The corresponding CREATE statements are shown in Figure 4 and Table 2. In addition to relation-valued attributes, the *robots* table shows *list valued* (axes, dh_matrix) and *tuple-valued* attributes (kinematics, joint_angle, dynamics). List valued means that the values occurring are *ordered*, for example, in the *axes* attribute. That is, there is a first axis, a second axis, etc. A tuple-valued attribute, such as *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. To retrieve all robots which have a Screw Driver in the set of *endeffectors* and which have at least 2 *arms*, each of which has 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')
```

User-defined data types and functions

Current query languages for relational databases usually provide only a fixed set of data types and operations. It is usually not possible to extend this set by user-defined data types or functions. This is a major drawback, especially in advanced applications such as engineering or office automation. In these areas, special data types and special functions are needed quite frequently, e.g., a data type for matrices and a function for matrix multiplication. Since matrices and matrix multiplication are not provided in conventional query languages, the user has to model matrices by low-level constructs, e.g., byte strings, and write a cumbersome application program in a conventional programming language to interpret and to manipulate these byte strings as matrices. Therefore, a mechanism is needed that allows the user to define his or her own data types and functions and to add them to the DBMS so that they can be used within the query language as a normal built-in function on basic data types.

This need has already been recognized in the Peterlee Relational Test Vehicle (PRTV[46]), which is known as one of the first running prototypes of a relational DBMS. The PRTV provides a simple mechanism for user extensions. The user can define his or her own procedures (written in PL/I) that can then be used in query statements and called by the DBMS at run time. Since PRTV tables are always in First Normal Form, complex (hierarchical) data structures as procedure input and output cannot be processed.

In this section, the AIM approach for user-defined data types and functions is introduced. It is based on HDBL and its underlying data model.

As opposed to most other projects on extensibility, AIM-P intentionally does not strictly enforce the abstract data type paradigm. That is, the structure on which the user-defined functions are operating may remain visible. By doing so, any instance of a user-defined data type may be queried and modified using normal HDBL expressions. Normal HDBL expressions and user-defined functions can be mixed. Clearly, the instances of user-defined data types can also be treated as *data capsules* which are accessible only by way of user-provided functions associated with that type. It was emphasized that no special knowledge about database internals is required. An example is given for a visible user-defined data type and related functions and then for an encapsulated type.

Assume, for example, that a user wants to see all robots having a *dh_matrix* whose value of the determinant is 1. Since computing the determinant of a matrix is a standard function in linear algebra, a corresponding function can usually be found in a library of mathematical functions. The connection between this function and HDBL is made by declaring, e.g., a type for 4 × 4 matrices of real values as follows:

```
DECLARE
   TYPE dhtype < 4 FIX < 4 FIX REAL >> END
```

In this example, *dhtype* is the name of a user-defined type. It can subsequently be used in other DECLARE TYPE statements or within CREATE statements to create new database objects. Now the user can define the interface of a user-defined function for computing the determinant as follows:

```
DECLARE
   FUNCTION determinant(matrix: dhtype): REAL
```

Figure 4 CREATE statement for robots of Table 2

```
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) ]} ]}
END
```

In order to make this function work, the user or system programmer has to program the function body. In programming the function body, it seems appropriate to use a general-purpose programming language. For AIM-P, Pascal has been selected because the system itself is implemented in this language. To allow users to implement their own functions for their own data types, previously declared using HDBL declare-type statements, the HDBL types (basic ones and user-defined ones) have to be mapped to Pascal data structures. As HDBL allows for user-defined types of nearly unlimited structural complexity, a Pascal representation as a byte string (character string) with a linearized representation of the HDBL data types would make function implementation rather complicated and error prone. Pascal type checking would be practically eliminated when pursuing this approach.

For AIM-P it was therefore decided to map the atomic HDBL data types as well as the HDBL constructor types (set, list, tuple) to corresponding predefined Pascal data types. This not only leads to more natural mappings, but also allows the utilization of Pascal's strong typing and type checking for the implementation of user-defined functions. In order to avoid mapping errors from HDBL-type representation to Pascal-type representation and vice versa, the mapping is not defined by the user but is provided by a *type compiler* which is part of AIM-P's catalog manager. At execution time, before calling a function, AIM-P will automatically map from the AIM-P internal representation to the Pascal representation and also will transform the result of the function back into AIM-P's internal representation. A comprehensive treatment of this subject, including implementation issues and run-time support, can be found in References 47 to 49.

How the Pascal representation looks can be influenced to some extent by specifying HDBL type-compiler directives (STANDARD, DENSE). Analogously, the Pascal function header is generated automatically. For our *dhtype* example, the Pascal representation using the DENSE directive would look like:[47]

```
TYPE dhtype$1 =
     RECORD
        ACT_ELEM: 0..4;
        VAL     : ARRAY [1..4] OF REAL
     END;
     dhtype   =
     RECORD
        ACT_ELEM: 0..4;
        VAL     : ARRAY [1..4] OF dhtype$1
     END;
```

The Pascal function header for the determinant function declared above looks like:

```
FUNCTION determinant(matrix: dhtype): REAL;
```

The determinant function can now be programmed in Pascal as follows (assume the existence of a library function *compute_determ* for computing the determinant):

Table 2 The robots table. The attribute axes contains a list (indicated by <. . . >) of tuples and is an ordered relation. Dh_matrix is also list valued, but the elements of this list are lists again forming 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.

{ 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	:	:	:	:	:	:	:

```
FUNCTION determinant(matrix: dhtype): REAL;
VAR local: ARRAY [1..4, 1..4] OF REAL;
VAR i,j: INTEGER;
BEGIN
   FOR i:=1 TO 4 DO
     FOR j:=1 TO 4 DO
       local[i,j]:= matrix.val[i].val[j];
   determinant := compute_determ(local,4)
END
```

After compiling the determinant function and linking it to the system, it can be used in arbitrary HDBL expressions wherever a REAL value is allowed as a result-type expression. For example, one can retrieve all *robots* having a *dh_matrix* whose determinant equals 1 by the following HDBL query:

```
SELECT r
FROM   r IN robots
WHERE  EXISTS(ar IN r.arms):
       EXISTS(ac IN ar.axes):
       determinant(ac.kinematics.dh_matrix) = 1
```

A user-defined function may even be as simple as a square root function that does not require any new type because only real values are involved. Therefore, a square root function can be declared as follows:

```
DECLARE FUNCTION square_root(r:REAL): REAL
```

The Pascal implementation is very simple:

```
FUNCTION square_root(r:REAL): REAL;
BEGIN
    square_root := sqrt(r)
END
```

Another example is the introduction of a data type COMPLEX for complex arithmetic. Assume that an abstract data type COMPLEX is desired where the user need not see the internals of a value of type COMPLEX but may use COMPLEX values only by functions. This can be done by declaring a type to be *encapsulated.* In this case, the system enforces the condition whereby instances of this type are accessible only by way of functions associated with that type. In this way, the representation can be changed without having to change the queries. A type COMPLEX can be declared as follows:

```
DECLARE TYPE COMPLEX
  [ re:REAL, im: REAL ]
ENC END
```

The keyword ENC means that COMPLEX is an encapsulated type. The corresponding Pascal type would be:

```
TYPE COMPLEX =
     RECORD
           re: REAL;
           im: REAL
     END
```

The complex arithmetic now is defined by functions, for example:

```
DECLARE
   FUNCTION compl_make(r1,r2:REAL): COMPLEX;
DECLARE
   FUNCTION compl_add(c1,c2: COMPLEX): COMPLEX;
DECLARE
   FUNCTION compl_negate(c:COMPLEX): COMPLEX;
```

The corresponding Pascal implementation is very simple:

```
FUNCTION compl_make(r1,r2: REAL): COMPLEX;
VAR result: COMPLEX;
BEGIN
    result.re  := r1;
    result.im  := r2;
    compl_make := result
END;
```

```
FUNCTION compl_add(c1,c2: COMPLEX): COMPLEX;
VAR result: COMPLEX;
BEGIN
    result.re := c1.re + c2.re;
    result.im := c1.im + c2.im;
    compl_add := result
END;
FUNCTION compl_negate(c:COMPLEX): COMPLEX;
VAR result: COMPLEX;
BEGIN
    result.re    := -c.re;
    result.im    := -c.im;
    compl_negate := result
END;
```

The data type COMPLEX can now be used within any CREATE statement for creating database objects. With the help of the functions, values of type COMPLEX can be used within HDBL queries.

All structure types which can be created using the type mechanism of HDBL are always also valid HDBL types. Consequently, user-defined types can also occur, e.g., on the left side of an assignment expression, on its right side, or they can be input for a subsequent step. The type and function mechanism has become an integral feature of the data model and language of AIM-P.

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 online interface accepts input of HDBL statements (data definition, query, data manipulation, type and function definition) and offers facilities for querying the catalogs (object catalog, type catalog, function catalog), for editing and retrieving stored queries and for browsing query results. It also supports user-provided display functions for user-defined data types.

The AIM-P application program interface (API) follows the same philosophy as System R[50] and SQL/DS.[38] 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.[51] The API language constructs can be roughly divided into two groups: declarative statements and operational statements. These language constructs will be summarized with the help of some selected examples. More detailed descriptions of this language can be found in References 52 and 53.

Figure 5 Examples of DECLARE statements

```
%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;
```

The declarative statements are used to describe database objects (DECLARE RESULT), the application program variables that take values of the database objects (DECLARE DATA), and cursors for navigating within 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 Table 1) *c_cursor* and *s_cursor* depend on *d_cursor*. That is, *c_cursor* can only operate on those manufacturing cells belonging to the manufacturing department on which *d_cursor* is currently positioned. Besides defining the scope of 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).

The operational statements are used to drive the API by way of an 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 (connect to the database), a BEGIN SESSION statement that provides a user identification and a password has to be issued. Transactions can subsequently be started with BEGIN TRANSACTION and closed using a COMMIT TRANSACTION or ABORT TRANSACTION statement. Finally, a session can be closed with an END SESSION.

The statement EVALUATE *manudept* triggers both the execution of the query shown in Figure 5 and the materialization of the result.[54] 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 of

Figure 6 MOVE and GET statements with object-oriented data transfer

```
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;
```

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 by the respective cursor will be transferred into individual application program variables (in Figure 6, 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 will be taken as a unit (tuple) and be assigned to a corresponding (type compatible) record variable. The specification BY n ($n > 1$) triggers the transfer of more than one instance (tuple) at a time. An optional variable of type AIM_RESULT_DESCR (see *staff_info* in Figure 5 and Figure 6) can be used to tell the user how many data instances (tuples) have actually been transferred into the application program.
- The option OBJECT_WISE specifies, in contrast to the keywords TUPLE_WISE and ATTR_WISE, that all the atomic and nonatomic data at the current cursor position will be transferred into the application program: A complete complex object is transferred at one 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 nonatomic (set-valued, list-valued) data. Appropriate type declarations for these program variables (e.g., CELLTUP in Figure 5) can be obtained from the type compiler of

AIM-P in conjunction with the support of user-defined data types as described in the previous section (see References 48 and 49 for details).

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[55]).

If a result has been declared FOR UPDATE as in Figure 5, it can be both read and modified. For that, the cursors must be positioned in the same way as for reading (GET, FETCH). UPDATE, INSERT, or DELETE statements (which are syntactically very similar to the GET statement) can be issued to modify the data.[52,53] 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 private workspace of the transaction. It is still necessary for the user to subsequently perform either a COMMIT or an ABORT.[56]

AIM-P has been built in a modular fashion.

AIM-P system architecture

The AIM-P architecture is constructed as a client-server architecture. That is, a database server runs at a host system and serves a collection of workstations. This architecture is motivated by a scenario where a collection of workstations[57] autonomously perform complex and time-consuming tasks such as design applications. It is assumed that these applications, written in a standard programming language, rely on the services of a central database server. By means of a high-level query and data manipulation language (such as HDBL, see earlier section), representations of objects that may be complex are requested from the server. After being processed at the workstation, changed data may be transferred back to the central site. Because of the potential complexity and volume of data, the workstation cooperatively supports the server in propagating changes back to the central data repository.

The subsystem running at the workstation consists of the Result Walk Manager which is a subcomponent of the Complex Object Manager (see Figure 7) of the AIM-P server, the On-line interface, which provides an interactive interface for querying the database and displaying the results, the API precompiler, and the API run-time system (see previous section). The On-line interface, API precompiler and API run-time system are usually only available at the workstation. The remainder of this section concentrates on the description of the server architecture.

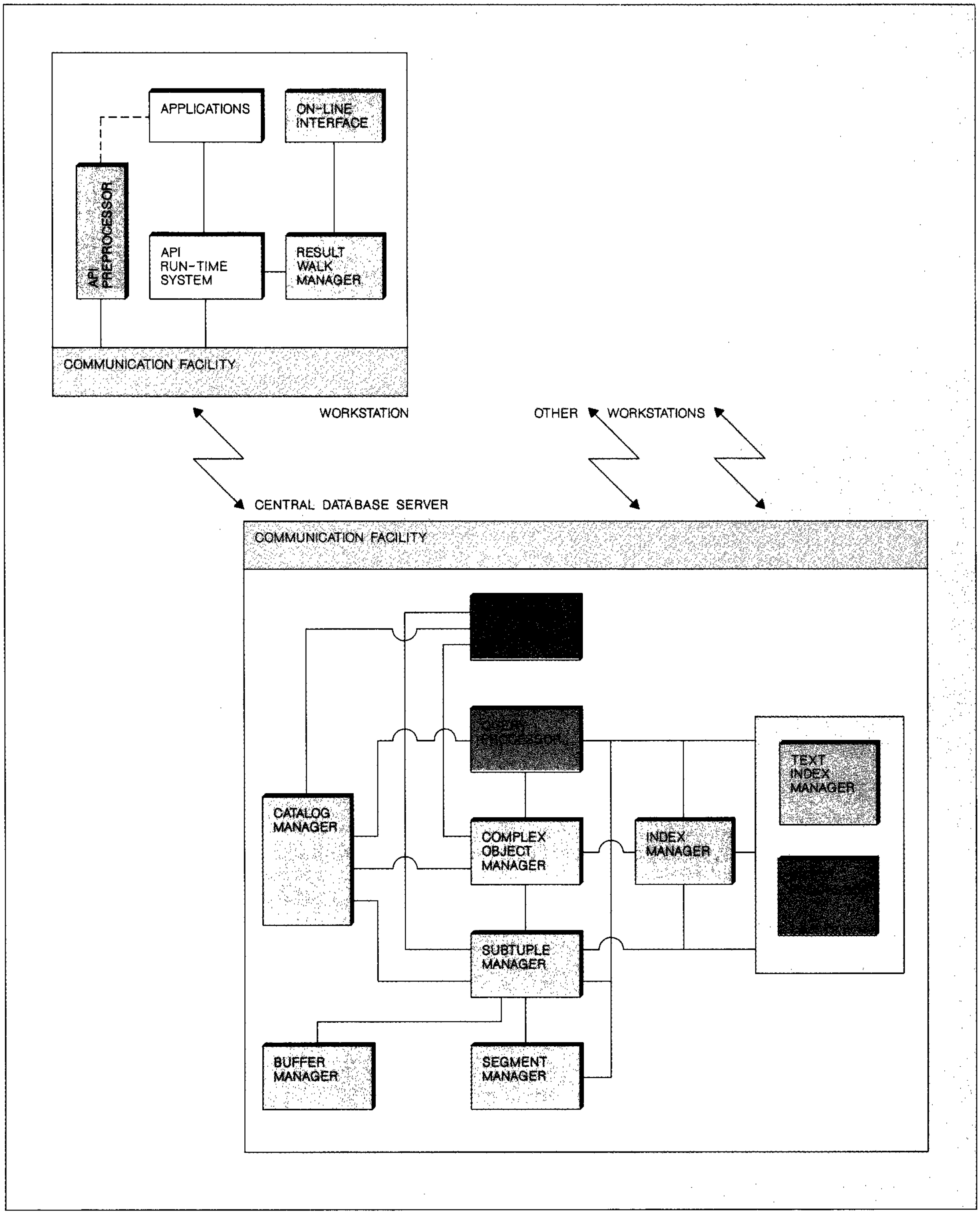
In order to allow system modifications because of changing requirements or to test new ideas, AIM-P has been built in a modular fashion (see Figure 7). The components Buffer Manager, Segment Manager, Catalog Manager, Supervisor, and Communication Facility perform the usual functions as in any DBMS. That is, they manage the I/O activity from external storage into the system buffer and vice versa, handle segment (file) creation, deletion, and free space bookkeeping, and also supervise the communication between the virtual DBMS machine and other virtual or real machines.

The Query Processor is responsible for parsing an HDBL query, performing query optimization (algebraic query optimization and access path selection), and executing the query (including data manipulation requests). This component is composed mainly of three major components: the Parser, the Query Tree Optimizer, and the Query Tree Evaluator (see Reference 58).

The Index Manager consists, in principle, of two parts: one handles normal DBMS indexes (such as B-trees), whereas the other was conceived to handle the text index.[24–26]

The Subtuple Manager is responsible for retrieving subtuples, or records (the basic logical access units in AIM-P), out of pages and for mapping subtuples to pages. Subtuples can be very small but can also span many pages, especially in cases of long fields. (Time version management is also done by this component.[43,59,60]) In addition, it has been designed to support the transaction concept. It offers Begin-of-Transaction (BOT), Commit, and Abort commands to start a new transaction, to make its updates

Figure 7 Architecture: A central database server is driven by a collection of workstations



permanent, or to drop them, respectively. For this reason, the Subtuple Manager maintains for each transaction an individual temporary workspace where all updates are performed. At commit time the content of this workspace is written to a transaction-oriented log file before the updates themselves are made available to other transactions.

The Complex Object Manager is the only component which can determine structure and implementation. Whereas the Subtuple Manager handles all subtuples in the same way, the Complex Object Manager distinguishes between structure subtuples and data subtuples. Structure subtuples, called minidirectories in AIM-P terminology, contain structural information (e.g., parent-child relationships), whereas data subtuples contain only data (except some management information such as subtuple length and null value information).

The Complex Object Manager cooperates very closely with the Subtuple Manager and with the Query Processor, or by relieving the latter from taking care of details about the physical aspects of data placement and retrieval. In cooperation with the Subtuple Manager and the Segment Manager, the Complex Object Manager places the data on external storage such that data belonging to one object (complex object, NF^2 tuple) is stored on contiguous pages, if possible. That is, it directs physical clustering. Moreover, the Complex Object Manager also masks the different representations of complex objects from the Query Processor. In the current implementation of AIM-P, the three different representations that may occur and need to be supported are database, object buffer, and external.

The *database representation* is used to represent database objects on external storage and in the DBMS system buffer. This structure has been designed to efficiently support access to any object especially for processing projection and selection operations.[37,43,61] The *object buffer representation* is used to represent the results of a query as well as temporary intermediate results. This representation is somewhat more compact than the database representation and is also used for the workstation–server cooperation.[62] The third representation is the *external representation* of HDBL types in Pascal structures. This representation is used for supporting user-defined data types and functions as described earlier. It is also used for passing query results to application programs if OBJECT_WISE transfer has been requested (see the section on on-line and application program interface).

For a more comprehensive discussion of the AIM-P architecture refer to References 43 and 44.

Summary, status, and outlook

The AIM project is a research and development effort of the Heidelberg Scientific Center to better understand the requirements of database technology for integrated applications. AIM-P, the research prototype under development at the Scientific Center, is based on the concept of nested relations. The experiences from applications or analytical studies with the Extended NF^2 Data Model, e.g., in the areas of real estate information systems,[36] chemical information systems,[40] geometric modeling,[63] and CAD/CAM[64–66] are very promising and give a high degree of confidence that this data model shows the correct direction for future database management systems.

AIM-P is based on the concept of nested relations.

The first version of AIM-P that was nearly complete became operational at the end of 1986. Since then, the system has been installed at various places within and outside of IBM for research and study purposes. The current implementation status (Release 2.0) supports the following:

- 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 yet supported.
- A large subset of HDBL is operational, but only rudimentary query optimization is currently performed. View support is still missing.
- Access to historical data in ASOF[67] fashion
- Access to HDBL facilities both in on-line mode and through the 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 or commit) in a single-user environment

In this brief description of the project, only a few aspects of the overall AIM project effort could be highlighted. System features such as time version support (which has been deeply integrated into AIM-P[43,59,60]), the integration of text search capabilities, and the important aspect of cooperative processing in a workstation–server environment[43,62] were not discussed in this paper. Also, joint research activities with partners at the University of Darmstadt in the area of workstation–server cooperation,[62,68,69] at the University of Hagen on engineering design version support,[70,71] and with our partners in the R^{2}D^2 project[72] at the University of Karlsruhe in the area of robotics and abstract data types[48,49,64,66,73–76] have significantly influenced and contributed to our work but were not covered in this paper.

Conceptual work has been started to improve the query processing of AIM-P 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[77,78] (that will provide the basis for the support of recursive queries[79–82]), and also on object-oriented concurrency control techniques.[83,84]

Our main target, however, remains the understanding of database requirements in advanced, integrated application areas. We therefore will increase the number of case studies performed in such areas using our prototype. Direction and emphasis of our future research and development work will, as in the past, be heavily influenced by the requirements and open problems discovered in these areas.

Acknowledgments

The development of AIM-P was (and is) a cooperative effort of IBM scientists, visiting scientists, and students. In addition to the authors, the AIM-P group presently consists of R. Erbe, J. Günauer, U. Herrmann (a doctoral student), U. Kessler (a visiting scientist), K. Küspert, V. Obermeit, P. Pistor, E. Roman, 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, 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 provided the stimulating factor and was an important part of the theoretical basis for the whole development effort. In addition, much 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, G. Saake, M. R. Scalas, H.-J. Schneider, J. Teuhola, R. Traunmüller, H. Wedekind, and L. Wegner. All of these contributions are gratefully acknowledged. The authors want to thank K. Küspert, P. Pistor, and E. Roman for carefully reading an earlier version of this paper and giving valuable suggestions which helped to improve the presentation. Thanks also to R. Erbe and G. Saake for their comments.

Cited references and notes

1. R. L. Haskin and R. A. Lorie, "On extending the functions of a relational database system," *Proceedings of the ACM SIGMOD Conference*, Orlando, FL (June 1982), pp. 207–212.
2. R. Lorie and W. Plouffe, "Complex objects and their use in design transactions," *Proceedings of the Engineering Design Applications Stream*, ACM-IEEE Data Base Week, San Jose, CA (May 1983), pp. 115–121.
3. L. A. Rowe and M. Stonebraker, "The Postgres data model," *Proceedings of VLDB*, Brighton, UK (September 1987), pp. 83–96.
4. M. Stonebraker, J. Anton, and E. Hanson, "Extending a database system with procedures," *ACM Transactions on Database Systems* **12,** No. 3, 350–376 (September 1987).
5. M. Stonebraker, "Inclusion of new types in relational data base systems," *Proceedings of the Second International Conference on Data Engineering*, Los Angeles (February 1986), pp. 262–269.
6. M. Stonebraker and L. A. Rowe, "The design of Postgres," *Proceedings of the ACM SIGMOD Conference*, Washington (1986), pp. 340–355.
7. M. Stonebraker et al., "Quel as a data type," *Proceedings of the ACM SIGMOD Conference*, Boston (June 1984), pp. 208–214.
8. D. Dayal, F. Manola, A. Buchman, U. Chakravarthy, D. Goldhirsch, S. Heiler, J. Orenstein, and A. Rosenthal, "Simplifying complex objects: The PROBE approach to modelling and querying them," *Informatik-Fachberichte* **136,** 17–37, Springer-Verlag, Berlin (1987).
9. D. Goldhirsch and J. A. Orenstein, "Extensibility in the PROBE database system," *Data Engineering* **10,** No. 2, 24–31 (June 1987).
10. P. Schwarz, W. Chang, J. C. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh, "Extensibility in the Starburst database system," *Proceedings of the 1986 IEEE International Workshop on Object Oriented Database Systems*, Pacific Grove, CA (1986), pp. 85–93.
11. B. Lindsay, J. McPherson, and H. Pirahesh, "A data management extension architecture," *Proceedings of the ACM SIGMOD Conference*, San Francisco (May 1987), pp. 220–227.
12. D. S. Batory et al., *GENESIS: A Reconfigurable Database Management System*, TR-86-07, Department of Computer Science, University of Texas at Austin (March 1986).
13. M. J. Carey, D. J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J. E. Richardson, and E. J. Shekita, "The architecture of the EXODUS extensible DBMS," *Proceedings of the 1986 IEEE International Workshop on Object Oriented Database Systems*, Pacific Grove, CA (1986), pp. 52–65.

14. H.-B. Paul, H.-J. Schek, M. H. Scholl, G. Weikum, and U. Deppisch, "Architecture and implementation of the Darmstadt database kernel system," *Proceedings of the ACM SIGMOD Conference*, San Francisco (May 1987), pp. 196–207.
15. T. Härder, K. Meyer-Wegener, B. Mitschang, and A. Sikeler, "PRIMA—A DBMS prototype supporting engineering applications," *Proceedings of VLDB 87*, Brighton, UK (1987), pp. 433–442.
16. D. S. Batory and A. P. Buchmann, "Molecular objects, abstract data types and data models: A framework," *Proceedings of VLDB 84*, Singapore (August 1984), pp. 172–184.
17. K. R. Dittrich, "Object oriented database systems: The notion and the issues," *Proceedings of the 1986 IEEE International Workshop on Object Oriented Database Systems*, Pacific Grove, CA (1986), pp. 2–6.
18. A. Albano, L. Cardelli, and R. Orsini, "Galileo: A strongly-typed, interactive conceptual language," *ACM Transactions on Database Systems* **10**, No. 2, 230–260 (June 1985).
19. J. Mylopoulos, Ph. A. Bernstein, and H. K. T. Wong, "A language facility for designing database-intensive applications," *ACM Transactions on Database Systems* **5**, No. 2, 185–207 (June 1980).
20. M. H. Scholl and H.-J. Schek, Editors, *Theory and Applications of Nested Relations and Complex Objects* (workshop material), International Workshop, Darmstadt, West Germany (April 1987).
21. S. Abiteboul, P. C. Fischer, H.-J. Schek, Editors, "Nested relations and complex objects," *Lecture Notes in Computer Science* **361,** Springer-Verlag, Berlin (1989).
22. G. Ozsoyoglu, Z. M. Ozsoyoglu, and V. Matos, "Extending relational algebra with set-valued attributes and aggregate functions," *ACM Transactions on Database Systems* **12,** No. 4, 566–592 (December 1987).
23. M. M. Astrahan et al., "System R: Relational approach to database management," *ACM Transactions on Database Systems* **1,** No. 2, 97–137 (June 1976).
24. H.-J. Schek, "The reference string indexing method," *Proceedings on Information Systems Methodology* (G. Bracchi, P. C. Lockemann, Editors), Venice, Italy, 1978, *Lecture Notes in Computer Science* **65,** Springer-Verlag, Berlin (1978), pp. 432–459.
25. D. Kropp, H.-J. Schek, and G. Walch, "Text field indexing," *Proceedings of the Meeting of the German Chapter of the ACM on Data Base Technology* (J. Niedereichholz, Editor), Bad Nauheim, West Germany, September 1979, Teubner-Verlag, Stuttgart (1979), pp. 101–115.
26. D. Kropp and G. Walch, "A graph-structured text-field index based on word fragments," *Information Processing and Management* **17(6),** 363–376 (1981).
27. F. Bancilhon, P. Richard, and M. Scholl, "On line processing of compacted relations," *Proceedings of VLDB 82*, Mexico (September 1982), pp. 263–269.
28. E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM* **13,** No. 6, 377–387 and 70–94 (June 1970).
29. G. Jaeschke, *An Algebra of Power Set Type Relations*, Technical Report TR 82.12.002, Heidelberg Scientific Center, IBM Corporation, Heidelberg, West Germany (December 1982).
30. G. Jaeschke and H.-J. Schek, "Remarks on the algebra of non first normal form relations," *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Data Base Systems*, Los Angeles (March 1982), pp. 124–138.
31. G. Jaeschke, *Nonrecursive Algebra for Relations with Relation Valued Attributes*, Technical Report TR 85.03.001, Heidelberg Scientific Center, IBM Corporation, Heidelberg, West Germany (March 1985).
32. G. Jaeschke, *Recursive Algebra for Relations with Relation Valued Attributes*, Technical Report TR 85.03.002, Heidelberg Scientific Center, IBM Corporation, Heidelberg, West Germany (March 1985).
33. H.-J. Schek and P. Pistor, "Data structures for an integrated database management and information retrieval system," *Proceedings of the VLDB Conference*, Mexico (September 1982).
34. B. Hansen, M. Hansen, and P. Pistor, *Formal Specification of the Syntax and Semantics of a High Level User Interface to an Extended* NF^2 *Data Model* (unpublished, 1982).
35. P. Pistor, B. Hansen, and M. Hansen, "An SQL-like query interface for the NF^2 model," in *Informatik-Fachberichte* **72,** Springer-Verlag (1983), pp. 134–147 (in German).
36. L. Gründig and P. Pistor, "Real estate information systems and their requirements for database interfaces," in *Informatik Fachberichte* **72,** Springer-Verlag (1983), pp. 61–75 (in German).
37. P. Dadam, K. Küspert, F. Andersen, H. Blanken, R. Erbe, J. Günauer, V. Lum, P. Pistor, and G. Walch, "A DBMS prototype to support extended NF^2 relations: An integrated view on flat tables and hierarchies," *Proceedings of the ACM SIGMOD Conference*, Washington (May 1986), pp. 356–367.
38. *SQL/Data System, Application Programming*, SH24-5018-2, IBM Corporation (August 1983); available through IBM branch offices.
39. Surrogates (references) are not supported in the current implementation.
40. P. Pistor and R. Traunmüller, *A Database Language for Sets, Lists, and Tables*, Technical Report TR 85.10.004, Heidelberg Scientific Center, IBM Corporation, Heidelberg, West Germany (October 1985).
41. P. Pistor and R. Traunmüller, "A database language for sets, lists, and tables," *Information Systems* **11(4),** 323–336 (1986).
42. P. Pistor and F. Andersen, "Designing a generalized NF^2 model with an SQL-type interface," *Proceedings of VLDB 86*, Kyoto, Japan (August 1986), pp. 278–288.
43. P. Pistor, "The advanced information management prototype: Architecture and language interface overview" (invited talk), *Proceedings of 3rd Journées Bases de Données Avancées*, Port-Camargue, France (May 1987), pp. 1–20.
44. P. Pistor and P. Dadam, "The advanced information management prototype," *Lecture Notes in Computer Science* **361,** Springer-Verlag, Berlin (1989), pp. 3–26.
45. F. Andersen, V. Linnemann, P. Pistor, and 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, IBM Corporation, Heidelberg, West Germany (February 1988).
46. S. J. P. Todd, "The Peterlee Relational Test Vehicle—A system overview," *IBM Systems Journal* **15,** No. 4, 285–308 (1976).
47. A. Kemper, K. Küspert, V. Linnemann, and M. Wallrath, *Pascal Structures for HDBL Types: Layout, Naming Conventions, Storage Allocation, and Usage in Functions*, Technical Note TN 87.05, Heidelberg Scientific Center, IBM Corporation, Heidelberg, West Germany (October 1987).
48. P. Dadam, K. Küspert, N. Südkamp, R. Erbe, V. Linnemann, P. Pistor, and G. Walch, "Managing complex objects in R^2D^2," in *HECTOR, Heterogeneous Computers Together, Volume II, Basic Projects*, Springer-Verlag, Berlin (1988), pp. 304–331.
49. V. Linnemann, K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Südkamp, G. Walch, and M. Wallrath, "Design and implementation of an extensible database management system supporting user defined data types and functions,"

Proceedings of VLDB 88, Los Angeles (August/September 1988), pp. 294–305.

50. D. D. Chamberlin et al., "Support for repetitive transactions and ad hoc queries in System R," *ACM Transactions on Database Systems* **6,** No. 1, 70–94 (March 1981).
51. Currently only Pascal is supported.
52. R. Erbe, N. Südkamp, and G. Walch, *Advanced Information Management Prototype, Application Program Interface User Manual*, Technical Note TN 88.03, Heidelberg Scientific Center, IBM Corporation, Heidelberg, West Germany (November 1988).
53. R. Erbe, N. Südkamp, and G. Walch, "An application program interface for a complex object database," *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, Jerusalem (June 1988).
54. The EVALUATE and OPEN CURSOR statements are not shown in Figure 6.
55. *PASCAL/VS Language Reference Manual, 3rd Edition,* Program No. 5796-PNQ (1985); available through IBM branch offices.
56. K. Küspert, U. Herrmann, R. Erbe, and P. Dadam, "The recovery manager of the advanced information management prototype," *Proceedings of Reliability '89*, Brighton, UK (June 1989), pp. 3B/4/1–11.
57. Within this context a workstation can be either a physically separate machine (e.g., a dedicated engineering workstation with graphics device) or just another virtual machine running on the same host computer as the database server.
58. V. Lum, P. Dadam, R. Erbe, J. Günauer, P. Pistor, G. Walch, H.-D. Werner, and J. Woodfill, "Design of an integrated DBMS to support advanced applications," *Proceedings of the International Conference* on Foundations of Data Organization (invited talk), Kyoto, Japan (May 1985), pp. 21–31.
59. P. Dadam, V. Lum, and H.-D. Werner, "Integration of time versions into a relational database system," *Proceedings VLDB 84*, Singapore (August 1984), pp. 509–522.
60. V. Lum, P. Dadam, R. Erbe, J. Günauer, P. Pistor, G. Walch, H.-D. Werner, J. Woodfill, "Designing DBMS support for the time dimension," *Proceedings of the SIGMOD 84 Conference*, Boston (June 18-21), pp. 115–130.
61. U. Deppisch, J. Günauer, and G. Walch, "Storage structures and addressing concepts for complex objects of the NF^2 relational model," *Proceedings of the GI Conference on Datenbanksysteme für Büro, Technik und Wissenschaft*, Karlsruhe, West Germany, March 1985, Springer-Verlag, Berlin (1985), pp. 441–459 (in German).
62. K. Küspert, P. Dadam, and J. Günauer, "Cooperative object buffer management in the advanced information management prototype," *Proceedings of VLDB 87*, Brighton, UK (September 1987), pp. 483–492.
63. A. Kemper and M. Wallrath, "An analysis of geometric modelling in database systems," *ACM Computing Surveys* **19**, No. 1, 47–91 (March 1987).
64. R. Dillmann and M. Huck, "R^2D^2: An integration tool for CIM," in *HECTOR, Heterogeneous Computers Together, Volume II, Basic Projects*, Springer-Verlag, Berlin (1988), pp. 355-372.
65. M. Mitchell, National Bureau of Standards, Automated Manufacturing Research Facility (AMRF), Gaithersburg, private communication, Heidelberg (January 1987).
66. P. Dadam, R. Dillmann, A. Kemper, and P. C. Lockemann, "Object oriented data management for robot programming," *Informatik Forschung und Entwicklung*, Springer-Verlag, Heidelberg **2,** 151–170 (1987), (in German).
67. This AIM-P feature has not been described in this paper. More on this topic can be found in References 43, 59, and 60.
68. U. Deppisch, J. Günauer, K. Küspert, V. Obermeit, and G. Walch, "Considerations about the cooperation between database server and workstations," *Proceedings of the 16th GI Jahrestagung*, Berlin, October 1986, *Informatik-Fachberichte* **126,** Springer-Verlag, Berlin (1986), pp. 565–580 (in German).
69. U. Deppisch and V. Obermeit, "Tight database cooperation in a server-workstation environment," *Proceedings of the 7th International Conference on Distributed Computing*, Berlin (September 1987), pp. 416–423.
70. P. Klahold, G. Schlageter, and W. Wilkes, "A general model for version management in databases," *Proceedings of VLDB 86*, Kyoto, Japan (August 1986), pp. 319–327.
71. W. Wilkes, *The Notion of Versions and Its Modelling in CAD/CAM Databases*, doctoral dissertation, University of Hagen, Department of Mathematics and Computer Science (September 1987), (in German).
72. R^2D^2 stands for Relational Robotics Database with Extensible Data Types and was 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.
73. P. Dadam, R. Dillmann, A. Kemper, and P. C. Lockemann, "Object-oriented databases for robot programming," in *HECTOR, Heterogeneous Computers Together, Volume II, Basic Projects*, Springer-Verlag, Berlin (1988), pp. 289–303.
74. A. Kemper, P. C. Lockemann, and M. Wallrath, "An object-oriented database system for engineering applications," *Proceedings of ACM-SIGMOD*, San Francisco (May 1987), pp. 299–311.
75. A. Kemper, M. Wallrath, and M. Dürr, "Object orientation in R^2D^2," in *HECTOR, Heterogeneous Computers Together, Volume II, Basic Projects*, Springer-Verlag, Berlin (1988), pp. 332–354.
76. A. Kemper, M. Wallrath, M. Dürr, K. Küspert, and V. Linnemann, *An Object Cache for Complex Object Engineering Databases*, Technical Report TR 89.03.005, Heidelberg Scientific Center, IBM Corporation, Heidelberg, West Germany (March 1989).
77. G. Saake, V. Linnemann, P. Pistor, and L. Wegner, "Sorting, grouping, and duplicate elimination in the advanced information management prototype," *Proceedings of VLDB 89*, Amsterdam, The Netherlands (August 1989).
78. K. Küspert, G. Saake, and L. Wegner, "Duplicate detection and deletion in the extended NF^2 data model," *Proceedings of the 3rd International Conference on Foundations of Data Organization and Algorithms* (FODO '89), Paris, June 1989, *Lecture Notes in Computer Science*, Vol. 367 (W. Litwin and H.-J. Schek, Editors), Springer-Verlag, Berlin (1989), pp. 83–100.
79. V. Linnemann, "Non first normal form relations and recursive queries: An SQL-based approach," *Proceedings of the 3rd IEEE International Conference on Data Engineering*, Los Angeles (February 1987), pp. 591–598.
80. V. Linnemann, *Optimization of Recursive Queries Over Nested Relations by a Differential Technique*, Technical Report TR 87.07.005, Heidelberg Scientific Center, IBM Corporation, Heidelberg, West Germany (July 1987).
81. V. Linnemann, *Functional Recursion and Complex Objects*, Technical Report TR 88.12.017, Heidelberg Scientific Center, IBM Corporation, Heidelberg, West Germany (December 1988).
82. V. Linnemann, "Functional recursion based on nested tables," in *Informatik-Fachberichte* **204**, 408–427 (in German).
83. U. Herrmann, P. Dadam, K. Küspert, and G. Schlageter, "Locking of disjoint, non-recursive complex objects by object and query specific lock graphs," in *Informatik-Fachberichte* **204,** 98–113 (in German).

84. U. Herrmann, P. Dadam, K. Küspert, E. Roman, and G. Schlageter, *A Lock Technique for Disjoint and Non-Disjoint Complex Objects*, Technical Report TR 89.01.003, Heidelberg Scientific Center, IBM Corporation, Heidelberg, West Germany (January 1989).

General references

T. Härder, Editor, *Proceedings Datenbanksysteme für Büro, Technik und Wissenschaft Informatik-Fachberichte* **204,** Zürich, Switzerland, March 1989, Springer-Verlag, Berlin (1989).

G. Krüger and G. Müller, Editors, *HECTOR, Heterogeneous Computers Together, Volume II, Basic Projects*, Springer-Verlag, Berlin (1988).

J. W. Schmidt, Editor, *Sprachen für Datenbanken, Informatik-Fachberichte* **72,** Springer-Verlag, Berlin (1983).

Peter Dadam *IBM Heidelberg Scientific Center, Tiergartenstrasse 15, D-6900 Heidelberg, Federal Republic of Germany.* Dr. Dadam is manager of the Advanced Information Management (AIM) project. He joined IBM in 1982 as a research staff member and became part of the initial design team of AIM-P where he did significant portions of the transaction management, concurrency control, recovery, time version support, and record management subsystems design. In 1985, he became the manager of the AIM project. He holds a German diploma degree (comparable to an M.S. degree) in industrial engineering from the University of Karlsruhe, West Germany, and a doctoral degree in computer science from the University of Hagen, West Germany. He has been chairman of the special interest group on databases of the German Informatics Society since 1987 and is an author or coauthor of more than 30 scientific publications.

Volker Linnemann *IBM Heidelberg Scientific Center, Tiergartenstrasse 15, D-6900 Heidelberg, Federal Republic of Germany.* Dr. Linnemann is a research staff member of the Advanced Information Management (AIM) project. He joined IBM in 1986 and became part of the query processor design team of AIM-P. His special interests include data modeling and recursive queries in the context of complex objects. From 1982 to 1986, he worked on the level of an assistant professor at the University of Frankfurt, West Germany, where he specialized in recursive queries in databases. He holds a German diploma degree (comparable to an M.S. degree) in computer science and a doctoral degree in computer science from the Technical University of Braunschweig, West Germany. He is an author or coauthor of more than 20 scientific publications.

Reprint Order No. G321-5381.