Design of a prototype ANSI/SPARC three-schema data base system*

by ERIC K. CLEMONS
University of Pennsylvania
Philadelphia, Pennsylvania

INTRODUCTION TO THE THREE-SCHEMA ARCHITECTURE

This paper has three objectives. First, it describes briefly how an ANSI/SPARC three-schema data base system prototype could be constructed, using wherever possible available data models, system software, and research results. Second, it lists data models selected for each of the three levels; it also explains these selections. We find no existing proposal for an external schema facility to be adequate; therefore, our third objective is to develop specifications for the external level. A user interface based upon simple hierarchical user records is proposed, the necessary theory is described and a mapping language for the definition of external schemata is proposed.

The first three sections treat the first and second objective, describing and justifying design decisions for each level. The next three sections introduce specifications for the external level. The final section presents conclusions.

Need for multiple data models

Data structures employed in an integrated data base management system must address three goals: enterprise support, user support, and machine access for retrieval and storage. Enterprise support requires logical completeness: if data have been gathered and maintained at considerable cost, then it is essential that it be possible to use this data to respond to any logically meaningful query. User support requires logical simplicity: regardless of the complexity of the structures needed to support the enterprise's data processing, it is essential that the structures with which an individual user must interact be both simple and well suited to his programming needs. And, for machine access to the stored data, data description must be provided at a level low enough to permit efficient operation by the physical devices.

Unfortunately, these requirements usually appear to be incompatible. A structure that is logically complete enough for the enterprise is not sufficiently simple for convenient use by most programmers. Likewise, as is shown by example in the following section, a structure that is well designed for one application may not be suitable for another.

A promising mechanism for resolving these difficulties is the three schema model offered by ANSI/SPARC. Rather than attempt to define a single class of data structures of universal applicability, ANSI/SPARC proposes three levels of structures, one each for the enterprise, the users and the machine itself.

Such a model requires not only the ability to declare data structures of different classes, but to define maps between these structures. In this paper we propose choices for data models appropriate to each of the three levels. In particular, we develop an original model for the user level and a language for mapping to it from the enterprise level.

ANSI/SPARC Architecture

In the proposed ANSI/SPARC architecture there are three separate but related levels of data base schema: conceptual schema, external schema, and internal schema. The conceptual schema must be complete; it supports the enterprise and its view of the data required for its operations. The external level includes many external schemata; each external schema supports one or more applications programmers and provides a set of data structures required by and designed for their applications. The internal schema is needed for data access at the device level.

The following terms will prove useful. We define the stored data base as the actual data described in the internal schema, and a user data base as the collection of user records described by the user's external schema. Mappings between the stored data base and a user data base are, at least in theory, composed of maps from the internal to the conceptual level and from the conceptual to the external level. An external schema is often referred to as a user view of a data base or as a user view.

* Research supported in part by National Science Foundation Grant MCS77-18106.
Design of a prototype

The following design is proposed for a three-schema data base prototype. Its construction can be facilitated by exploiting existing software. It supports necessary functions for both machine efficiency and applications programmer effectiveness. And the data models used at all three levels are well understood.

For the internal level we propose that a CODASYL data base management system be employed. The conceptual level will be a relational system. And the external level will be based upon a Virtual Information Object (VIO) interface, introduced in an earlier work and summarized in a later section. These choices are defended in the third section. The remainder of the paper describes the proposed external schema facility.

The three schema prototype will have the structure shown in Figure 1. We shall examine in detail only Interface One.

AN ANS/SPARC EXAMPLE

We consider a simple university data base including six entity types and the relationships among them: Departments offer courses, employ both students and faculty and have students taking a major concentration in the department's courses. Faculty members teach course sections and advise students. Students enroll in course sections, and for each course taken by each student there is a course grade. A conceptual schema can be presented in several different ways; the data structure diagram in Figure 2 is but one way of displaying this university data base.

Several interesting external schemata can be defined on this conceptual schema; we offer three:

1. A grade report for each student, listing all courses and grades for courses taken.
2. A course roster, listing for each course section the faculty member who taught it and a list of students in the section and their grades.
3. A departmental roster, listing all student majors of each department and their grade-point average.

These data structures are shown in Figure 3.

We note that the structures of Figures 3a and 3b appear to be incompatible; for the first we want the data base organized as a hierarchy with courses listed for each student, while for the second an organization with the students listed for each course seems most appropriate. Also, the grade-point average data in Figure 3c, while derivable from data in Figure 2, is not explicitly stored in the university data base. These observations can be summarized:

1. None of the figures correspond exactly to the data structure of Figure 2 nor to a subset of this figure.
2. Each is a legitimate user view for a data processing application.
3. All are to be supported by the conceptual schema for the academic data base.
4. Each is to be supported by an appropriate external schema.

SELECTION OF DATA MODELS—INTERNAL, CONCEPTUAL AND EXTERNAL SCHEMATA

CODASYL—Choice for internal level

For a variety of reasons relating to efficiency of data access, we select the CODASYL model as the basis for the internal level.

Requirements at this level are control of record placement and definition of access paths for efficient selection of single records, subset queries and complex queries requiring data base navigation. The CODASYL model permits placement of records in areas, and supports indexed and hashed single record access, multi-list processing using pointer chains, inverted access using pointer arrays, and navigation using access through sets.

But we must reject the CODASYL model as too complex for the external level. Likewise, we must reject it as too rigid and inflexible for the conceptual level; while extensions to the functions to be supported by a data base can generally be encompassed by changes to the CODASYL schema, these changes are likely to result in dramatic restructuring of the schema, rather than mere extensions to it. For example, additional functions require replacing a hierarchical relationship between two record types with a confluence among three.
The relational model is flexible, in that changes in use of the data base will rarely require changes in the structure of the conceptual schema. It is mathematically rigorous, in that its operators are defined in terms of the predicate calculus. And it is complete, in the sense that any retrieval request expressible in the first order predicate calculus may be performed.

But we must reject the relational model as too tedious for the external level: overcoming normalization to recreate records with structured or repeating data items requires lengthy and unnatural queries. Likewise, we reject the relational model for the internal level, because we require more explicit control of record placement and of the specification of efficient access paths than the relational model provides.

There exist other possibilities for the conceptual level, e.g., Chen's Entity Relationship Model or recent work by Bachman or Gerritsen and Lee. We prefer the relational model because it is more proven and more robust. The relational model has been subjected to considerable analysis and several implementations are currently operational. The relational model's syntactic simplicity makes it unlikely that changes to the perceived relationships among entities will require changes to the schema.

Choice for external—No model available

There is no model available that fully meets our requirements for an external schema facility. Requirements are summarized and a proposed model is presented in the following section.

DESIGN OF AN EXTERNAL SCHEMA FACILITY

The external schema provides the basis for convenient use of the data base by individual applications programmers. It supports the definition of user views, that is, virtual data bases constructed from the common stored data base. These virtual data bases may differ dramatically in format and content from the stored data base; ideally this would permit a close or exact match between the cognitive structures employed while analyzing the problem and the data structures employed while writing the program.

Requirements

The external schema facility must support restructuring of data and definition of data items in user records that are not necessarily explicitly present in the stored data base or described in the conceptual schema. Data structures in user views need not support multiple and diverse users; a user view need not be flexible or complete in the sense of a schema designed to support the data processing of the enterprise. Rather, we want the simplest possible use: to obtain the description of a single entity, the user requests a single data access. Thus, to obtain grade-point averages for students majoring in decision sciences, the user requests a single data access. Thus, to obtain grade-point averages for students majoring in decision sciences, we should be able to write a query of the form:

```
SELECT GRADE-POINT-AVE
FROM STUDENT
WHERE MAJOR = 'DEC SCI'
```

It should not be necessary to traverse a complex data base like the one depicted in Figure 2; it should not be necessary for the user to specify the full details of obtaining a student record, testing the major, then obtaining all grade records for this student for the course grade.

These requirements argue against supporting networks in the external schema. Networks are useful principally because data bases must support multiple users employing different access paths through the data; confluencies, for example, usually appear because different users require different and inconsistent hierarchies, not because any single application requires a network. Networks offer more power than a single user needs, but at a cost: they are too complex to be truly convenient. Similarly, these requirements argue against flat systems like the relational model: because of the simplicity of the supported structures these systems do not permit data structures that closely match the cognitive structures with which the programmer solves his problem, and they are too simple to be convenient. We believe that an external schema facility must support the greatest possible variety of virtual hierarchies.

We define a virtual information object (VIO) as the program structure corresponding to the programmer's cognitive structure employed during problem analysis and solution. VIOs are constructed as required from the stored data base to meet the needs of individual applications programmers. A VIO instance serves as a user record, and a VIO decla-

<table>
<thead>
<tr>
<th>STUDENT-NAME</th>
<th>COURSE</th>
<th>GRADE</th>
<th>COURSE</th>
<th>GRADE</th>
<th>COURSE</th>
<th>GRADE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) A student Grade Report based upon the data base shown in Figure 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FACULTY-NAME</td>
<td>COURSE</td>
<td>STUDENT-NAME</td>
<td>GRADE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STUDENT-NAME</td>
<td>GRADE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STUDENT-NAME</td>
<td>GRADE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STUDENT-NAME</td>
<td>GRADE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) A Course Roster based upon the data base shown in Figure 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEPT-NAME</td>
<td>STUDENT-NAME</td>
<td>GRADE-POINT-AVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STUDENT-NAME</td>
<td>GRADE-POINT-AVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STUDENT-NAME</td>
<td>GRADE-POINT-AVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c) A Departmental Roster based upon the data base shown in Figure 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3—Three user views defined upon the university data base of Figure 2.
ration provides both the schema definition for a record type in the user view and the definition of the map needed to construct records of this type from the stored data base.

Classification of hierarchies

Hierarchies are either basic or recursive. A recursive hierarchy is one where subordinate entries are of the same type as their parent node and have all the same properties; thus they can have subordinate entries also of the same type. Such hierarchies can reach arbitrary depths and degrees of complexity. Discussion of recursive hierarchies is deferred until a later section.

In this section we introduce three dichotomies that permit the classification of all basic hierarchies:

1. Extent—"Single" or "all"
2. Entries—"Simple" or "grouped"
3. Content—"Complete" or "summary"

When entries are "simple," descendant nodes correspond to single entities (e.g., single treatments for a patient), and are constructed from single instances of entity descriptions (e.g., a single treatment record). When entries are "grouped" descendant nodes correspond to groups of entities (e.g., groups of all medical treatments or all surgical treatments for a patient), and are constructed from several instances of entity descriptions (e.g., the collection treatment records of the appropriate type).

When extent is "all," then all of a node's descendants are present in a single, wide, tree-structured record; when extent is "single" each node has one descendant and thus relationships between a node and its descendants are represented by a forest of narrow trees.

Combination of options for entries and extent provides power and flexibility in the definition of hierarchies. Grouped entries and single extent would provide a set of records, one for each type of treatment received by a patient. Grouped entries, all extent would provide a single record including data on all treatments, while simple entries and all extent would provide data on all treatments but without aggregation by treatment type.

Finally, "summary" content indicates that only summary information such as counts, totals or averages are to be included in the hierarchies while "complete" content indicates that both the summaries and the complete data from which they are calculated are to be included.

These three dichotomies can be combined to form seven classes of hierarchies** as shown in Figure 4. By forming hierarchies of greater depth, structures of arbitrary complexity can be formed.

** Generally, three dichotomies would yield eight classifications. Since the combination single, simple, summary is not useful, it is not included. Thus there are only seven types of hierarchies.

LANGUAGE FOR DECLARATION OF USER VIEWS

Information required by the external schema facility

The external schema facility itself requires information, specifying how data from the physical data base are to be accessed and combined to produce records in the user view. This information is of three types:

1. Access information
2. Restructuring information
3. Data item declaration

Access information specifies which data are to be accessed. It names the relations in the conceptual schema that contain the desired information and gives conditions that determine which tuples are actually to be used.
Restructuring information controls the combination of data from normalized relations to produce the desired general hierarchical records. These hierarchies are of the forms presented in the fourth section; restructuring information must therefore be specified in terms of the three dichotomies that were introduced.

Data item declaration specifies what information is actually to be included in user records; this information may of course be different from the data accessed from the stored data base, as an average of course grades differs from the extensive list of grades and courses taken. Data items may be of three types: real elementary, virtual elementary or structured. A real elementary item is one present in the conceptual schema. A virtual elementary item is unstructured (i.e., a field or COBOL elementary item) and computed from items in the conceptual schema. A structured item itself contains one or more data items; since these in turn may possess structure, hierarchies of arbitrary complexity may be declared.

Introduction to the language

The language we developed for the declaration of user views is based upon SEQUEL. Access information is specified using the form:

FROM relation-name
WHERE condition

We have found that the condition that qualifies the tuples to be accessed is almost always of the same form: use in the descendant items those tuples whose keys are related "in the obvious way" to the keys of the parent items. As an example, in a personnel data base, keys of children contain the keys of a parent; in a customer-order data base, keys identifying details include the keys of orders to which they belong. We call this form of qualification implication; e.g., access those children implied by employee tuple, access those details implied by the order tuple. We represent implication by the symbol "\(\leftarrow\)"

Restructuring information is specified by placing an option list within parentheses. The default options are "single" extent, "simple" entries and "complete" content, and only non-standard options need be specified.

There are different formats for declaring each of the types of data items. Virtual elementary items have the form:

item-name: defining-expression

Real elementary items may be declared with either the form:

item-name: conceptual-schema-item-name

or, if the data item is to have the same name in both the conceptual schema and the user record, the equivalent form:

SELECT item-name

may be used. Finally, to declare structured data items, the form is:

STRUCTURE structure-name: (option-list) item declaration

END structure-name.

Comments on this language

We offer below an example of a declaration of a map to construct a user record. We present the language statements needed to map from the conceptual schema of the university data base presented in Figure 2 to a departmental roster. In this external schema for each department there is a record containing the department name and a list of all students and their grade-point averages.

STRUCTURE ROSTER:
SELECT DEPT-NAME FROM DEPARTMENT;
STRUCTURE STUDENT-ENTRY: (All)
SELECT STUDENT-NAME FROM STUDENT \(\leftarrow\) DEPARTMENT;
STRUCTURE COMPUTE-AVERAGE: (All, SUMMARY)
SELECT GRADE FROM GRADE-REC \(\leftarrow\) STUDENT;
GRADE-POINT-AVE: (SUMMARY) AVERAGE (GRADE);
END COMPUTE-AVERAGE.
END STUDENT-ENTRY.
END ROSTER.

That is, undeniably, an ugly language for schema declaration. We offer the following observations in its defense:

1. It is not solely a DDL, as the CODASYL subschema facility is. Its mapping function subsumes much that is DML and thus eases the writing of applications programs.
2. It is not solely a subschema facility. Its DML function subsumes many of the data access and restructuring tasks that would otherwise be performed by applications programmers to construct item descriptions from several data base sources.
3. The declaration of a single map, while prepared only once, will be of use in several programs associated with an application.

Thus, while this map may not be easy to code, it permits the easy retrieval of names and averages for students majoring in Decision Sciences:

SELECT STUDENT-NAME, GRADE-POINT-AVE FROM ROSTER
WHERE DEPT-NAME = 'DEC SCI'

Likewise, it is now easy to perform several other tasks, e.g.,
comparing averages in the Finance and Marketing Departments:

```sql
SELECT AVERAGE (GRADE-POINT-AVE) FROM ROSTER
WHERE DEPT-NAME = 'FIN'
SELECT AVERAGE (GPA) FROM ROSTER
WHERE NAME = 'MARKET'
```

In the next section we offer a more advanced example of VIO declaration. Additional details of the language, its use and its implementation, can be found in an earlier work.8

A more advanced example of VIO declaration

The following example illustrates use of grouping and multiple levels of summary in VIO declaration. We examine a hospital data base, in which records are maintained on patients, physicians and treatments. In particular, patient records include patient identification number, patient name and identification number of attending physician; treatment records include patient identifier, identifier of the administering physician and associated data.

We wish to construct a patient history that lists all treatments of a single patient, grouped by treatment type and within type grouped by physician. We want only summary information: Counts of the number of treatments by physician within type, by physician, and total, as well as counts of the number of types of treatment and the number of physicians administering treatments to this patient.

An illustrative collection of conceptual level records and an associated VIO instance are depicted in Figure 5. The necessary VIO declaration is shown in Figure 6. We note that in the construction of this user record the interface performed both DML and DDL functions—data are accessed from several data base sources, grouping and ordering is performed and the necessary summaries are prepared. While the associated VIO declaration is somewhat lengthy, it is now possible for simple data processing tasks to be accomplished by the retrieval of the appropriate virtual records.

RECURSIVE HIERARCHIES

Finally, the external schema facility is extended to include declaration and processing of recursive hierarchical structures. Recursive structures are those where lower levels in the hierarchy are of the same type as higher levels, and may themselves have similar lower levels. Recursive hierarchies do arise (e.g., organizational chains of command, parts explosion diagrams) and may readily be encoded in a single relation. Unfortunately, even relationally-complete languages do not support retrieval from such structures.8 For example, the ORG relation represents the relationship between a department and its subordinate departments:

```sql
ORG: (DEPT#, SUB#)
```

The query:

```sql
SELECT SUB# FROM ORG
WHERE DEPT# = 1
```
retrieves all sub-departments of Department 1.
SELECT SUB# FROM ORG
WHERE DEPT# =
SELECT SUB# FROM ORG
WHERE DEPT# = 1

retrieves all sub-departments of sub-departments of Department 1. But there is no single query which will traverse the entire organizational structure, retrieving all subordinate departments regardless of their depth in the tree.

Three concepts are essential in an external schema facility that is to treat recursive structures:

- **Definition—Node specification** is the declaration of the individual nodes of a recursive structure, their format and data content.
- **Definition—Iteration control** is the control of which nodes are to be included in a recursive structure.
- **Definition—Tree traversal** is the processing of the nodes of a recursive structure.

Node specification has much in common with VIO declaration. The structure of the node may be specified using the three dichotomies introduced in the fourth section, data may be accessed from desired tuples of any combination of relations, and the data included in the node may be real or computed virtual items as appropriate. Thus, considerable generality in the format of individual nodes is provided.

Iteration control is used to prune the depth and breadth of the tree. It determines which nodes are to be included, based upon distance from the root, content or contents of subordinate nodes. By combining node specification, which determines the contents of nodes, with iteration control, which determines the shape of the tree, the necessary generality in definition of recursive hierarchies is attained.

Tree traversal is used to process recursive hierarchies. It is used for such things as determining manpower totals over an organization, or cost of components using a parts explosion diagram. A more detailed treatment of recursive structure definition and processing is not possible here, but is available elsewhere.⁸

CONCLUSIONS

Summary of the proposed prototype

For the conceptual schema we want generality, completeness and the ability to support unanticipated users of the database. For this we employ the relational model.

For the internal schema we want control over storage, record placement and use of indices and access paths. Wherever possible, we want efficient operation; in particular, for common and anticipated requests, we require this efficiency. For this we employ the CODASYL network model.

For the external schema we want user orientation, ease of applications programming, close relationships between the cognitive structures of the programmer and the data structures of the program. For this we employ a hierarchical external schema facility, the details of which have been outlined previously.

Features that must be provided by an external schema facility

A facility to treat adequately the declaration of external schemata must provide the following features:

1. **A set of hierarchies**—These hierarchies are virtual; that is, they are declared in the external schema but need not be stored in the data base in terms of set memberships, repeating groups or related records. Hence the set of hierarchies provided may be redundant or inconsistent; we feel this last point eliminates the need for the support of more general networks.⁶,⁷

2. **Full support of the classes of hierarchies presented in the fourth section**— Provision must be made for "simple" or "grouped" entries, "single" or "all" extent, "complete" or "summary" content. Additional features have also proved useful; e.g., sorting entries, limiting the number of entries included.

3. **Recursive hierarchies**—Provision must be made for node specification, iteration control, and tree traversal. We feel that node specification is best managed using the features of VIO declaration and that iteration control should be a simple extension to qualification.

Status of implementation

We have completed both an analysis of the requirements for a hierarchical external schema facility and the design of a VIO-based interface to support this facility. A language processed by the interface, used for mapping between conceptual and external levels, has been presented. Two experimental tools have been constructed: a VIO-to-relational processor and a VIO-to-CODASYL processor. No serious conceptual difficulties have been encountered. Implementation of the three-schema prototype described here requires only acquisition of a relational-to-CODASYL mapping facility.

REFERENCES