¡le

DATABASE DESIGN, Gio Wiederhold CHAPTER 8, file CHAP8.tex

 Failing macros recreated in addedmacros.tex 25Jun1997 Local macrodefinitions for table forms are LineExOne, LineExtwo

Chapter 8

Schemas

Schemas

The best laid schemes o' mice and men
Gang aft a-gley

Robert Burns
To a Mouse, 1787

The previous chapter described models of databases. The structure of the database was modeled by relations and connections. The relations were defined by their relation-schemas. In order to translate a model into an operational system, the model has to be described in a form which lends itself to implementation.

Such a description is called a *schema*, and the language used to describe it will be called the *schema language*. The schemas will have to include some practical detail which could be ignored in the models. Using the model concepts of domains, relations, and connections, the schema language has to be able to specify the types of the data elements, their organization into files, and the manner in which files are related.

A schema defines initially the structure of the database and makes the description available to the users of the database. If a database management system is used to manage the database, the schema will be used to automatically control the execution of the transaction programs which operate on the database.

An objective of a database system is to systematize the access to data elements. File systems have provided the means to fetch specific records according to defined keys or according to sequentiality. In order to fetch individual data elements the names of the data attributes, as known by the schema, are used to locate the data elements within the records. The data type information stored in the schema can be used by the processing transactions to direct the computation. Queries can use the connection specifications in the schema to locate successor data.

Predecessors of schemas are called *data dictionaries* and *database directories*. These systems collect information about the data and perhaps the database model, but do not make it directly available to a database management system.

In this chapter we present the construction and use of schemas to describe data elements and their relationships. Section 8-1 will look at the data elements to be described; Sec. 8-2 presents the schema for single files and gives some examples. The description of connecting structures is given in Sec. 8-3. In contrast with Chap. 7 we will present the material using mostly examples from available database management systems. This approach enhances the realism, but reduces the consistency of the material.

There is no universal schema language today. To apply the earlier concepts we have to use a variety of schema languages. Schema languages in the examples have been selected for specific features or because of widespread use. We do not go into detail to the extent that this chapter can replace system manuals, but relating the concepts developed here to actual examples should help greatly in the use of system documentation. For actual database specification a manual of the chosen language will be needed.

8-1 DEFINING THE ELEMENTS FOR A DATABASE

We need a description of the view requirements if we wish to design a database system. If an operational database is available, perhaps using a conventional file system, then we can begin by reviewing the usage of the file system by the programs that operate on the data to be integrated. A set of files used together may already contain multiple views, but if no conflicts have been experienced, the analysis can begin treating the existing database as a unit.

8-1-1 Analysis of Existing Data

Database problems are generally not recognizable until a certain volume of data has been collected. When a database system is to be installed, there are often already many programs which operate on the data to be incorporated in the database. The schema definition begins then with the collection of the information used to generate the processing programs that deal with the database. Information about the domains of the data elements and their dependencies can be used for the modeling processes described in Chap. 7.

Since the transactions will also be included in the eventual design, the processing steps applied to the information are also documented. For instance, results and intermediate derived data are defined in terms of their source variables. Many existing procedures must be studied in order to obtain a composite picture of the actions that are carried out. Documentation of data sources and destinations is collected. The frequency of use, the desired response time, and even the accuracy requirements are captured if feasible.

Similar information is obtained for new procedures and the data elements required for their operation. Here estimates may be needed, but documentation of assumptions made is at least as important to the design process as the documentation of established processes.

234pt Figure space

Figure file:

Figure 8-1 Payroll data element flow.

Figure 8-1 sketches an example of a very simple data flow. In practice, tabular descriptions for each variable are more convenient than a flowchart. The equivalent flow would be described as as follows:

8-1 Table of Data Elements

Variable	Source	Destination
`hours_worked`	time recorder	payroll program
`employee_salary`	supervisor	personnel file
`employee_department`	personnel office	personnel file
`employee_salary_item`	personnel file	payroll program
`employee_dept_item`	personnel file	payroll program
...	...	...

Elements may, of course, also have multiple destinations.

The analysis of current procedures has to be carried out with due regard to the actual information flow. The fact that certain reports produced contain a specific data element is not necessarily an indication that the element is being used, or if it is used, that it is the best means of presenting the desired information. On the other hand, there often are informal methods of data distribution which are not obvious during a system analysis which depends wholly on existing documents. Talking is an important, unformalized, and flexible form of communication.

For example, the fact that an engineering department manager warns an inventory clerk that he expects new employees so that necessary tools will be available when they arrive is an information function easily overlooked when automating a personnel system.

8-1-2 Characteristics of Data

We begin by describing the *characteristics* of data elements. The most important description of a data element are its name and its domain. These are familiar concepts to any programmer.

Programming languages have always provided facilities to name and characterize data elements. The domain of a data element, needed for computation, is mostly given as the *type* and perhaps the *length*. Some modern languages permit further specifications.

Other characteristics, for instance, the largest and smallest value expected, are defined implicitly or in the program documentation. For the database complete and explicit descriptions are advantageous.

We will present the quantitative characteristics of data elements – type or domain, length, conditionals, count – in this subsection. Qualitative specifications of data element characteristics are presented in Sec. 8-1-3. The number of attributes used to describe each type of data element is sufficiently large that we may keep the schema itself as a relation.

NAME Names have been used throughout to define attributes in records and relations. The name given to a data element by a programming language is restricted by fairly simple rules. These rules generally allow a short, variable–length string of alphabetic and numeric characters; the first character has to be alphabetic. This string is easily parsed or separated from other elements that make up the statements accepted by the schema language. Compatibility between the programming language and the schema language is desirable. If multiple languages are used, the sum of compatibilities can restrict names unduly.

Names used in files and databases are global; i.e., they are bound to their meaning over all programs while the files or database is open. The schema will be used by many programs over a long time. In programming languages a name can have a different definition as the process changes scope. In a database the scope of a name of a data element is affected only by structural scope, the name is defined within the database or relation that contains the data element.

To locate the values, the position or byte number within the record is kept associated with the names in the schema. If the field allocation is flexible, an approach as seen in a *pile file* may be used. A coded name is kept with each element in the record. The schema keeps the codes with the names to allow translation from name to code and vice versa.

Most schema language translators will assign positions only according to their own data allocation methods. Some languages permit the users to specify the position explicitly so that the schema can be applied to previously existing files.

TYPE It is common to associate a specific data type with each data element name. The type specification has several functions:

1 It limits the values to be associated with the data element names to a specific domain.

2 It simplifies processing by implying the category of legal operations to be used in the transformation of the data.

3 It provides specifications for encoding of the data values. (Encoding methods and associated semantics are summarized in Chap. 14.)

Most programming languages provide a small number of data types. Schema languages should handle at least the types that are used in the programmed transactions:

`CHARACTER:` variable or fixed-length string of arbitrary characters;
`DECIMAL:` fixed-length string of decimal digits;
`BINARY INTEGER:` fixed-length string of binary digits;
`DECIMAL FIXED POINT:` scaled decimal number;
`BINARY FIXED POINT:` scaled binary number;
`FLOATING POINT:` approximation to a real number; and `REFERENCE.`

The types defining computational values need little elaboration. Decimal fixed point is often scaled by two to represent dollars and cents. Binary fixed point is mainly used for representing real-time sensor acquired data in the range $\{0.00.. \ldots 1.00..\}$.

A reference variable in a program language, for instance, a record pointer in PASCAL or a base variable in PL/1, is however quite distinct from a reference in a database. One difference is their lifetime. References in databases have lifetimes that greatly exceed the execution time of a program. The other difference is the action taken when the referenced object is moved or deleted. Garbage collection methods are used in programs to retrieve or change invalid pointers; this approach is not feasible in a large database.

A *database reference* is a data element that refers to some object in the database. Three types of references are employed in database systems: pointers, symbolic references, and indirect pointers, as sketched in Fig. 8-2.

A *pointer reference* has the value of an address in the database space. If the number and location of all references to an object are known, specific pointers affected by the move of an object can be changed. In the general case pointers cannot be changed if the referenced data element is moved or deleted without a database reorganization. Until that time the old object spaces are marked with tombstones.

A *symbolic reference* contains the key of the referenced object. A procedure using key-based fetching has to be invoked in order to locate the referenced object in the database. The objects can be moved freely, as long as the access structures (indexes, etc.) are maintained. A symbolic reference will simply fail to find an object which has been deleted.

An *indirect reference* attempts to combine the benefits of symbolic references with the speed of pointer references. Here simple keys, perhaps organized as shown in Sec. 4-1, are used to access a table of pointer references. Now, when the object is moved, only the pointer value in the table has to be changed, and this takes care of all instances where this record has been referenced. If an object is deleted, the entry in the pointer table is set to a universal entry indicating `DELETED`. The look-up table is organized to be faster than a fetch using the symbolic reference key. If sufficient memory is available, the look-up may not require a file access.

81pt Figure space

Figure file:

Figure 8-2 Reference types.

DOMAIN The type of a variable has a strong connotation of its representation. We prefer to define a set of permissible values; this is closer to the concept of a *domain* as used in the database model. Statements to define a data domain are available in some modern programming languages as an adjunct to *type* declarations.

A `DEFINE` statement describes a new data type by associating a type name with a *domain* specification. The use of data elements outside of the domain during processing is detected and reported in a manner similar to the type conflicts occurring when, for instance, numbers and character strings are mixed up. Domain definitions are available in the language PASCAL. Example 8-2 shows such statements using a PL/1 style syntax.

8-2 Data Domain Definitions

```
DEFINE color AS("Red","Blue","Green","Mauve");
DEFINE year AS(FROM 1961 BY 1 TO 1976);
```
A declaration is used to create data elements of the type defined.
```
DECLARE (model) year;
DECLARE (body,fender,hood) color;
```
These elements can be used in statements for processing:
```
IF body = fender THEN CALL assemble_body;
ELSE GO TO get_next_fender;
```
The above statements will prevent the assembly of multicolored cars.
Such a rigorous type of definition will allow detection of erroneous statements such as:
```
body = "CHEVROLET";
body = model;
```
since in the context defined, `body` can only be set to a value of type `color`.

Domain definitions and their use to restrict assignment can play an important role in keeping improper elements out of a database. Furthermore, since the set of values is limited, an efficient encoding can be derived to cover the domain. We will try to incorporate such facilities in the schema languages for our database element definition even where the programming languages do not have such provisions.

LENGTH The length of a data element can be fixed or variable. If the length is fixed, it can be specified as a number of bits or characters with the element description. If the element length is variable, the marking method which is used to determine the length for a given instance of the element can be indicated. The available methods are those discussed for marking records (see Sec. 2-2-4).

The use of variable length data elements leads to a need for variable-length records. An example of a variable-length record specified in PL/1 is shown in Example 8-3. The variable (`address`) could also be decomposed into multiple lower–level elements.

8-3 Variable-Length Record

```
DECLARE 1  employer_record,   2   name CHAR VARYING(40),
2   sex CHAR(6),
2   birthdate INTEGER,
2   address CHAR VARYING(80);
```

As we have indicated earlier, the record written on the file is a direct image of the declared specification. The fact that the character strings are specified to be `VARYING` is ignored by most compilers, and the maximum specified is always allocated when the `employer_record` is written to a file. This practice leads to a considerable waste of space if long records occur infrequently. The record cannot be processed without the associated declaration.

The stored record can be composed as shown in Fig. 8-3.

90pt Figure space

Figure file:

Figure 8-3 PL/1 record with variable-length entries.

CONDITIONAL DATA ELEMENTS When a relation and its subrelations are implemented in one file we may have alternate data elements in different record subtypes. Some programming languages support this concept.

Records in PASCAL allow variants in their structure; a record can have alternate elements. Given a variable with two choices,

```
sex = (male,female);
```

then the record of Example 8-3 can be expanded as shown in Example 8-4.

8-4 Conditional Data Elements

```
RECORD    name       : PACKED ARRAY(1..†40) OF CHAR;
sex        : (Male,Female);
day        : 1..31;
month      : 1..12;
year       : 1800..2000;
address    : PACKED ARRAY(1..80) OF CHAR;
CASE sex OF
Male   : (years_military: 0..50);
Female : (pregnancies, children: 0..20)
END;
```

†PASCAL uses .. to indicate `TO` in the sense of up-to-and-including.

Here the record has two possible variants, and we should know the value of `sex` before processing the conditional elements of the record.

COUNT Within a single record it may be desirable to repeat a data element to account for multiple instances of one subsidiary item. This option enables us to implement simple nest concepts from the model within one record. An example was the list of children of an employee and their ages in Fig. 7-3. Identifying keys were used to distinguish the elements. To determine the number of elements in a specific record, mark techniques can also be used. If the count and size of such repeating elements is large, the manipulation of these records may become difficult. The general solution to this problem is the creation of a nest file for these elements. Nest structures are widely supported within database management systems and are described in Sec. 8-3.

8-1-3 Data Element Descriptions

In Sec. 8-1-2 we listed traditional and essential characteristics of the attributes describing the data elements. Schema entries which complement these essential but programming–
oriented characteristics will be listed in this section. These schema entries provide information beyond the programs' needs. Recording the characteristics shown here can greatly improve the automatic and manual aspects of the database interface presented to the users.

TITLE A descriptive sentence or title may be desired which allows a more detailed identification of the data elements than the program language name permits. Sometimes the title can also include abbreviations, synonyms, antonyms, or aliases. Specialist in a subject area often use abbreviations unique to their field of discourse when specifying a data element for entry or retrieval. When data are shared among a larger audience, a full identification or title is required.

For example, the variable `bp_dst` will have the title:

`"Diastolic bloodpressure, measured with patient sitting"`

UNIT SPECIFICATION A code or identification of the units of measurement for numeric data elements is often important. It might be essential to know whether a pressure is stated in pounds force per square inch, atmospheric fractions, millibars, millimeters or inches of mercury, kilograms per square centimeter, pascals, or football players per quarterback. The unit specification might be used only for reporting or verification of domain matching when files are joined, but could also be used to convert values automatically if the system has the necessary conversion factors available.

ESSENTIAL DATA An indication of whether this data element is essential or optional in the record may have to be maintained. We may wish to prevent the creation of records which have critical elements missing. An employee file needs to have a social security number and a departmental assignment for every employee. Other fields, such as a spouse name or a health certification, may be optional.

UNDEFINED VALUES If data can be missing or left undefined, it becomes necessary that programs which operate on the database recognize this fact. Undefined values are due to several causes.

`Missing data` Some data values may not have been available or known when the remainder of the record was entered.

`Not applicable` Some attributes may not be relevant in some records, especially when subrelations are combined into a file.

`Not collected` Some past data may be unavailable in areas where the scope of the database has expanded over time. In the design of a new system or the change of an existing one there is an opportunity to add new capabilities. This means that new data elements have to be described and their structural relationships charted. The new database will then have attributes for which no past values are available.

To manage these cases of undefined a notation in the database is needed. The schema must assign a value representation which does not conflict with legal values in the domain of the attribute. It is not clear if distinct values are needed for the three cases above, although the difference between "not applicable" and "missing" can lead to different actions.

Very few programming systems make appropriate provisions to handle any kind of undefined values in a manner distinct from zero values. The programmers are then forced to use codes as `-1, -0, 999999, 'undefined'`, and the like. These codes invariably find their way into computations and lead to major errors. Some statistical systems have provisions for missing data, and the database can provide the required input format automatically. For programming systems which do not handle undefined values the databases system should trigger an exception procedure when an undefined value is accessed, so that the processing programs will not compute nonsense.

Nonsense can be created for instance if a join operation creates result tuples for `null = null`. This type of problem can be avoided by never equating a `null` value with another `null` value, i.e., by treating each `null` value as distinct. In counting operations `null` values should be ignored, so that they do not distort computed averages or deviations. Not all problems are solved by these rules, and more sophisticated schemes have been proposed but have never been used in practice.

TRANSFORMATIONS There may be a need to transform data between the outside world and the database. Typical of such a transformation is the internal storage of dates in integer or Julian form (year.day) for computational convenience, while output of these dates is presented in `dd/mon/yy` form, so that the values make sense to humans. Using letters for the month removes the ambiguity of English and European notations.

Such a transformation may be carried out by encoding and decoding procedures appropriate for the domain of values. These routines will reside in a database library, and the schema can indicate which routine to use. If the transformation program can be handled similar to a domain definition, new and composite data types can be created and used.

The domain definitions in PASCAL are not adequate to permit

```
DECLARE birthdate DATE
```

but transformation programs can de defined which will take input as `12FEB82` and convert it to 29 993 (days since `1JAN00`) and vice versa. The input transformation can report errors, and interval computations can be performed on the internal value.

DATABASE SUBMODELS AND ACCESS PRIVILEGE The schema will also identify elements as belonging to specific database submodels, if this concept is supported by the database system and the database has been integrated. Within each submodel there will be elements for which a user has update responsibilities and privileges, and others which the user may only read but not modify. The schema is the obvious repository of access privilege information, that is, the specification of the users who have permission to access these data elements. Chapter 12 will deal with this aspect in detail.

FILE MANAGEMENT A schema may also be the repository for file management information. Data to control indexing, transposition, control of integrity, archiving, and erasure cycles can be placed in the schema. We will encounter such aspects incrementally as we proceed.

8-2 THE SCHEMA AND ITS USE

The collection of information that describes the database, when organized in a formal manner, is called the *schema*. Data element descriptions are an important part of the schema. Before discussing the definitions of the structural aspects of a database within the schema, we will illustrate its use. When the use of a schema is clear, it will be easier to deal with other requirements placed on the schema, since there is still no consensus on how schemas can satisfy those requirements.

A set of documents that is used by a programming staff to generate programs is an informal and often inconsistent form of a schema. Programmers will perform an analysis of the tasks and consider the available data elements. Subsequently they can code the required programs. Many statistical systems have provided directory facilities with their data files, so that the collected observations will always be properly identified and titled.

In the following sections we will discuss stronger, more automated approaches to the use of the schema information. The schema will be coded so that it can be read by the database system and used by generalized programs to control the flow of data to the files which contain the database. It will be stored within the system to be accessible when needed. The formal schema description also provides a means for the database users, database designers, and programming staff to communicate and to define their concepts.

The definition of the database using the schema precedes use of the database. In order to create a schema, we will need schema language services separate from the language services which are used when the database is manipulated. Often the processors for the schema language and the data-manipulation language are distinct. Sometimes they even have a completely different vocabulary and syntax.

Figure 8-4 places the idea of a schema in perspective. During computation the schema is used both to place incoming data properly into the files, and to locate requested data at a later time. The dictionaries in the schema aid the users in describing their requests, and perform a filtering function to improve data quality within the database. The database users do not modify the schema during database operations.

342pt Figure space

Figure file:

Figure 8-4 The place of a schema in input and output.

8-2-1 An Example of a Simple Schema-Driven System

Schemas may be employed with very simple file systems. The resulting database system will also be simple, and yet may provide a useful service. A schema for a database which uses only a single sequential file is shown here. This example shows operations using the TYMSHARE RETRIEVE system. The system operates interactively on a terminal. Figure 8-5 is an example of the initial definition of a schema. At this point no data exist in the data file.

171pt Figure space

Figure file:

Figure 8-5 Schema creation.

The schema, however, has been saved on a separate file so that it can be used by subsequent processes. The name of the schema file is obtained by suffixing the database name with some unusual characters; here “"STR.D"” is used. The system is now ready to receive data, and Fig. 8-6 demonstrates this phase.

The information from the schema is used to place each field properly into the file. These data may now be manipulated using computational instructions and selection statements. Table 8-1 lists the commands available to the user.

A complete session, which involves retrieval of the schema, addition of records, changing of data fields, selective computation of result fields, and a final printout, is shown in Fig. 8-7.

126pt Figure space

Figure file:

Figure 8-6 Entering data into the database.

8-1

RETRIEVE Commands

Database manipulation commands:

APPEND

```
CHANGE attribute [ FOR conditions ]
COUNT [ FOR conditions ]
DELETE [ FOR conditions ]
LIST [ FOR conditions ]
PRINT ( attributes ) [ FOR conditions ]
REPLACE attribute WITH expression [ FOR conditions ]
REPORT [ FOR conditions ]
SAVE file
SORT BY key attribute [ FOR conditions ]
SUM expression [ FOR conditions ]
```

Clauses as `[ FOR conditions ]` are optional.

In order to explain the nature of such a database system, a possible implementation of the `APPEND` command is shown in Example 8-5. The following steps take place:

1 The schema is fetched from its file

2 Storage is allocated for a data record

3 A header line is created from the schema and put on the terminal as a prompt

4 The data file is opened

5 The data lines provided by the user are obtained

6 The values are extracted and placed into the data record

7 Each record is appended to the data file

8 A blank input line causes the data file to be closed and a message to be placed on the terminal.

The entire schema is assumed to be small enough to fit in core. The actual code is considerably longer, since checks are provided throughout to prevent program failures if the schema and the data do not match.

The PL/1 function `SUBSTR(string, begin [,length])`, used extensively in this model program, takes a number of characters, up to the value `length`, out of a`string` of characters, beginning at position `begin`.

A system such as RETRIEVE is not adequate to support a general database. No means exist to cross-reference multiple files, and the data elements are limited to two simple types. The fact that the files are strictly sequential can cause long delays on large files. On the other hand, the facilities are easy to comprehend by the types of users the system designers envisaged, the possibilities for processing errors are reduced, and results of processing are well identified.

The facilities shown here are similar to those for manipulating a single file or relation of a larger system.

1pt Figure space

Figure 8-7 A database session.

Figure file:

1pt Figure-space

Figure 8-7 A database session (continued).

Figure file:

8-5 Example of Schema Usage: Implementation of APPEND

```
append: PROCEDURE(data_file_name);    /* Schema driven file append */
DECLARE data_file_name CHAR(*), schema_file_name CHAR(50);
DECLARE 1 schema(50),  /* No more than 50 attributes per record */
2 type CHAR(1),
2 width BINARY FIXED,
2 name CHAR(40) VARYING;
DECLARE headng CHAR(200) INITIAL(''),        /* empty */
blanks CHAR(200) INITIAL((200)' '),  /* full */
line CHAR(200) VARYING,
datarecord CHAR(*) CONTROLLED;
/* Fetch schema */
schema_file_name=data_file_name||'"STR.D"';
type,name=''; width=0;      /* Initialize all schema lines */
OPEN FILE(schema_file) INPUT TITLE(schema_file_name);
READ FILE(schema_file) INTO(schema);
CLOSE FILE(schema_file);
/* Set up data_record */
tot_width=SUM(width); ALLOCATE datarecord CHAR(tot_width);
/* Make a heading out of the names, extend each to the width      */
DO item=1 BY 1 WHILE(name(item) ¬= '');
headng=headng||SUBSTR((name(item)||blanks),1,width(item));
END;           PUT SKIP EDIT(headng)(A(200));
/* Prepare to write additional records onto the data file */
OPEN FILE(data_file) OUTPUT TITLE(data_file_name);
records=0;
/* Get a line from the user, if it's not empty, process it as the */
more: DO records = 0 BY 1;                     /* schema requires */
datarecord='';  PUT SKIP;
GET EDIT(line)(A(200));   IF line=''THEN GO TO done;
extract: DO item=1 BY 1 WHILE(name(item) ¬='');
commapos=INDEX(line,',');            /* locate a value */
IF commapos>0 THEN cp=commapos-1; ELSE cp=LENGTH(line);
/* Take out characters from line, adjust right or left, set width */
IF type(item) = 'N'/* and put away */
THEN datarecord=datarecord || SUBSTR(
(blanks||SUBSTR(line,1,cp)), 200-width, width );
ELSE datarecord=datarecord|| SUBSTR(
(SUBSTR(line,1,cp)||blanks), 1, width );
line=SUBSTR(line,cp+2); /* Chop processed input and   */
END extract;              /* Go to pick up next element */
WRITE FILE(data_file) FROM(datarecord);
END more;           /* Loop terminated by blank input line */
done: PUT EDIT(records,' RECORDS PROCESSED')(I(4),A(18));
CLOSE FILE(data_file);   RETURN;
END append;         /* End of transaction, return for next command */
```

8-3 DEFINING THE STRUCTURE OF A DATABASE

We have concentrated above on the description of data elements and then demonstrated a schema for a database with a minimal structure. The database management system only had to support the access of a record based on an attribute name and argument value.

An important aspect of a database is its structure. The relationships expressed by the structure allow the retrieval of related data elements and support the get-next type of operation. Structure can be found at three levels: within data elements, within records, and among records. Different database systems approach questions of structure in very diverse ways.

8-3-1 Structure within Data Elements

Some data elements are composed of smaller, simple elements. A few computer languages, notably COBOL and PL/1, have provisions to define such structures by assigning level numbers. An example of such a substructure can be shown by expanding the `address` shown in Example 8-3 into its component fields.

```
2 address,    3  street CHAR VARYING(80),
3  city    CHAR VARYING(80),
3  zip     DECIMAL(9);
```

Most database systems consider attributes only as simple elements. Any internal structure of a data field as `address` is the responsibility of the programs and their compilers. When modeling we also treat attributes only as having domains with simple values and ignore composite values. A technique to resolve these variables defines a domain having some transformation functions.

Several data elements may be grouped within a record, so that a single retrieval command may retrieve multiple elements. Such a group is typically composed of elements normally used together, perhaps belonging to one database submodel, or elements which are assigned to the same access privilege.

The smallest unit which a database can physically manipulate, be it a single element, a group of several data elements, or an entire record, will be called a *segment*. If the concept of database submodels is supported by a database management system, a segment should never contain attributes from more than one submodel. A read request will always obtain an entire segment.

A segment may have its own name in the schema or may be implied by referencing an attribute within it. If a segment is a single attribute or an entire record no segment name is expected. The attributes within a segment may be named in the schema, so that application programs can refer to individual data elements. We encountered single-attribute segments in MUMPS files. Those segments were named using a subscript notation.

8-3-2 Connections Placed within a Record

We have described the structure among elements within a record. The schema should state which data elements compose the ruling part and the dependent part. If a lexicon has been combined into the file, a secondary ruling part may appear in the record, and such information may be important to the processing programs.

How a data element is related to the file and hence to the entire database is critical but is difficult to describe in a manner which can be automatically interpreted by processing programs. The fact that an element is placed within a record implies a strong relationship with other elements in this record, but an analysis of a typical record (for example, Table 3-1) reveals a variety of relationships between the elements. The database model contains this information.

That an attribute belongs to the ruling part is often only implied in schema languages. An example is the specification of sequentiality and uniqueness (`SEQ,U`) in the key attribute description of a `Person` record segment defined using IBM's schema language DL/1. This field and key fields of any ancestor segments make up the ruling part.

```
FIELD NAME=(social_security_number,SEQ,U),BYTES=11, ...
```

Attributes in the dependent part may not be normalized in file structures. Denormalization may have been done explicitly, to achieve certain performance goals, or implicitly, when no normalized model was defined during the design process. We will show some more examples from DL/1.

Segments themselves may repeat within a record in order to implement a nest relationship within a record. The ownership connection is then implemented using physical sequentiality. This works only for one owner per owned record and does not support an *association* instance within a record. A `PARENT` specification in a segment relates nested segments to their owner, so that multiple-level nests can be described. This can make records very large indeed, at times spanning many blocks.

The DL/1 schema first specifies the relation and its file (`DBD` and `DATASET`), and then each segment in turn. Since in this example a direct file organization (`HDAM`)

8-6 DL/1 Record Specification

```
* filename
DBD    NAME=Employee,ACCESS=HDAM,RMNAME=hashprogram
* device specification
DATASET DD1=empty,DEVICE=2314,BLOCK=2000,SCAN=5
* Record name, length, position, and maximum frequency
SEGM   NAME=employee,BYTES=143,PARENT=0,FREQ=200
* attribute specifications
FIELD  NAME=name,BYTES=30,START=1,TYPE=0
FIELD  NAME=age,BYTES=4,START=31,TYPE=P
FIELD  NAME=sex,BYTES=1,START=140,TYPE=C
FIELD  NAME=years_military,BYTES=1,START=141,TYPE=P
FIELD  NAME=pregnancies,BYTES=1,START=142,TYPE=P
FIELD  NAME=children,BYTES=1,START=143,TYPE=P
* nest record specification
SEGM   NAME=skills,PARENT=employee,BYTES=26,FREQ=20
FIELD  NAME=(type,SEQ,U),BYTES=25,START=1,TYPE=C
FIELD  NAME=years_experience,BYTES=1,START=26,TYPE=P
* instruct assembler to generate the tables
DBDGEN
FINISH
```

is specified, there is no opportunity in the schema to specify the ruling part of the employee relation. The key used is a secret of the `hashprogram` which provides the key-to-address translation.

A frequency field in the schema (here `FREQ`) for a nest segment provides a count parameter to aid in the estimation of storage requirements.

The `TYPE` specification is limited to:

`0`	none
`C`	character
`P`	packed decimal

The DL/1 language is mainly concerned about segments, and only `FIELD`s containing keys for sorting or referencing have to be specified. The record in the example shows in fact a large gap between `age` and `sex` containing unnamed fields. Use of DL/1 will be investigated when the IMS database management system is described in Sec. 9-6. Schema tables set up by DL/1 can also be used by a number of other IBM and independent data manipulation and query systems.

8-3-3 Connections between Records

An ownership connection is best described by defining separate records for the nest or association records and providing references from owner to owned records. Pointer references can also implement reference and subset connections between records of entity, referenced entity, and lexicon relations.

When symbolic references are used, the database system need not be concerned with the relationship, since the reference attribute will contain a key value which can be used in a fetch operation. Constraints on deletion and insertion cannot be enforced if the relationship is not described.

When pointer references are to be used, the schema has to provide a description of the connection. The database system has to be able to place the reference value in the referencing attribute field. In DL/1 such specifications are explicit; an association, for instance, is denoted by specifying a pair of owners:

```
SEGM NAME=supply,,PARENT((supplier,SNGL)(parts,PHYSICAL))
```

The fact that the first `PARENT` is `SNGL` reflects only on the fact that there are only next and no prior pointers to the `supplier` owner. The indication that the other `PARENT` (`parts`) is `PHYSICAL` means that this owner is in the primary hierarchy, so that the owned record might also be found sequentially, selecting segments using the key attribute. Many other coupling arrangements are available in DL/1.

In the CII SOCRATE database management system, the specifications for nests and their linkages are implicit, so that the decision to use large records or multiple records for a nest implementation is independent of the schema description provided by the user. The term `SET` defines a nest with the tuple attributes delimited by a `BEGIN ... END` pair. Example 8-6 shows also the specification of the data element characteristics, with character type (`WORD`), domain definitions (`FROM ... TO ...` or as listed), unit (`IN ...`), and use of a conditional structure (`IF ... THEN store_one ELSE store_another`). The characteristic `LIKE` allows a domain specification to be copied.

In Example 8-7 we find a nest of skills within each record. If `skills` is extensive, it becomes desirable to use a referenced entity relation which allows the tuples to be shared. This alternative is shown in Example 8-8. The pointer reference to this relation (here `SET`) remains in a nest structure, so that again multiple `skills` can be entered for one `employee`. The `years_experience` can in this case only be implied from the requirements in the referenced `Skills` relation.

8-7 SOCRATE Record Description

```
BEGIN
SET  Employee
BEGIN
name  WORD
age FROM 16 TO 66 IN Years
address WORD
eye_color(Brown,Black,Blue)
hair_color(Black,Brown,Blond,Red,Gray)
sex(Male,Female)
IF sex = 'Male'
THEN  years_military FROM 0 TO 25 IN Years
ELSE
BEGIN
pregnancies FROM 0 TO 20
children    LIKE pregnancies
END
SET skills
BEGIN
type  WORD
years_experience FROM 0 TO 50 IN Years
END
END
END
```

8-8 Cross-Referenced Relations

```
BEGIN
SET Employee
BEGIN
name WORD
...
SET skills
BEGIN
skill REFERENCE Skill_description
END
END
SET Skill_description
BEGIN
skill_type WORD
years_of_school_req FROM 0 TO 20 IN Years
years_of_experience_req LIKE years_of_school_req
...
```

```
END
END
```

Sometimes references may be to other tuples of the same relation:

```
supervisor REFERENCE employee
```

An explicit reference may be bound by a join to be performed when the `CURRENT employee_record` is defined for insertion into the database.

```
back_up_person REFERENCE employee
HAVING skill = FIRST skill OF CURRENT employee
```

The schema implemented by SOCRATE describes a concept from a user's view rather than the structure of relations or files. We will discuss the subschema structure to support multiple user views in Sec. 8-4.

8-3-4 Derived Data

Processing derives new data, using source data found in the database. Processing implies having source data and an algorithm to process the data. We have described knowledge about the data in the model and encode this knowledge in the schema which implements the model. The schema makes the data description available to all users of the database.

If we have knowledge about processing which is useful to multiple users we should also store this information with the database schema. Knowledge about processing algorithms is encoded in the programs used to derive data. To make such algorithms available within the framework of a schema we associate programs with a new class of data elements: *derived data*. The use of derived data defined in the schema automatically invokes the processing programs and accesses the appropriate source data.

The effort to derive data can be such that it becomes desirable to store derived intermediate or final data also within the database, in addition to the algorithms or programs used to derive them. While the programs are still attached to the databases schema, actual stored derived data will appear in the database itself. If data are defined as derived they will not be available for conventional update.

A WARNING In some instances removal of computational responsibility from the users to the database-management system can increase the potential for serious errors. As an example, we take the automatic calculation of an `area` as the product of `width` and `length`. Some day a user will add an object to the database which is not rectangular and assume that the system-produced value for derived `area` is correct. A schema with a good descriptive capability, perhaps defining the computation in the title for the attribute, can help avoid such errors.

The user expects that derived data are as current as the source data available to the system. We distinguish two approaches to achieve this objective when supporting the generation of derived data:

On Entry Derived data is computed and the result stored whenever a source value is entered into the database, updated, or changed during processing.

On Access Derived data is computed from the current source elements only when a retrieval request to the result element is made.

Both alternatives are presented next, and then a mixed strategy will be suggested.

RESULTS DERIVED ON ENTRY With each source data element a process is identified which is automatically invoked when the data element is updated. The execution of the process causes the automatic updating of derived elements. Derived values kept in the file are actual results of entered data. All results derived on entry are always up to date relative to available source data elements.

8-9 Derivation on Entry of Actual Results

A part of a database is used for budgeting expenses on a yearly basis. The element for salary could be defined as

```
1 Employee;
...
2 salary, PICTURE IS 9999.99, ON MODIFY CALL pay_proc.
...
```

`pay_proc` is a procedure which is invoked whenever a `salary` is changed. This procedure may include statements as follows:

```
pay_proc: PROCEDURE;
raise = Employee.salary(NEW)-Employee.salary(OLD);
expense = raise*(12-current_month+1)
...
Department.salaries = Department.salaries + expense;
Department.profit = Department.profit - expense;
...
END pay_proc;
```

The summary fields in the `Department` relation reflect all changes introduced into the `Employee.salary` fields.

Use of this concept can lead to costly data entry. A chain of procedure calls may be generated during entry of a single value. The change of `salary` here leads to changes in the derived values in the `Department` record, and this in turn could affect a `Company` record.

The statement in Example 8-9, used to recompute the `Department.profit`, could also be generalized and be defined as a derived value to be invoked whenever a departmental budget amount is modified for whatever reason. Now no explicit profit calculation will be included in `pay_proc`, but when the `Department.salaries`, `Department.office_rent`, or a similar expense or income element changes, the procedure deriving `profit` will be executed.

These procedures assure that the derived results are immediately available when requested. An update to the file may, however, initiate a costly sequence of events, and a batch of updates (for a cost-of-living increase, for instance) may lead to a truly horrifying level of activity.

RESULTS DERIVED ON ACCESS The other method of accommodating changes in source data avoids the actual storage of derived data. Derived data to be computed on access are also described by means of a program which allows them to be automatically computed at the time of a retrieval request. Using the same situation shown above, the schema coding could read as shown in Example 8-10.

8-10 Derivation on Access of Potential Results

In the schema we find

```
DECLARE Department.profit  ...   GET_FROM(sum_budget_elements)
DECLARE Department.salaries  ... GET_FROM(sum_empl_salary)
```

and the processes would include two database procedures:

```
sum_budget_elements: PROCEDURE;
result = department_sales -
department_salaries -
department_overhead;
RETURN(result); END sum_budget_elements;
sum_empl_salary: PROCEDURE;
result = 0;
DO WHILE DEFINED(employee);
IF employee_department=department
THEN result = result + Employee.salary;
END;
RETURN(result);  END sum_empl_salary;
```

Here the chain of processes to be executed occurs at retrieval time. The results are never stored; the capability to generate them, however, is maintained. The cost of obtaining answers is high, and the same result will be computed many times if it requested frequently. If, for instance, `Product.profits` are compared with the overall `Department.profit`, the latter value will be recomputed from the source data once for each product comparision.

There are instances where an attribute value should certainly be defined to be derived on access; an `age` should always be derived from `birthdate` and today's date, a `shipping_cost` is best derived from `weight` and `shipping_rate`, and so forth.

ON-ENTRY VERSUS ON-ACCESS DERIVATION OF RESULTS Both of these methods lead to high costs when a usage pattern causes excessive recomputation. On-entry derivations require much effort when source-data are being entered, and on-access definitions cost when result-data are being generated. For each derived element the relative frequencies of access have to evaluted and the cost of the alternatives compared. On-entry derivation also increases storage and data transfer costs. Schema specifications for either choice are found in Sec. 8-3-5.

Where data do not need to be of ultimate currency, the problem of excessive cost can be resolved by not computing derived data at entry or at query time, but rather according to some regular processing cycle, using nightly or weekend processing resources. It is now well worthwhile to keep an additional data element with a *time-stamp* "`last_update_time`". The time-stamp is always set to the date and time of the last update of the derived data. One time-stamp may be used per record segment if the derivation process will update all derived elements in a segment as one transaction. This value can be used to check that expected derivations have indeed been recently computed.

TIME-TRIGGERED DERIVATION A combination of the two approaches is possible. We suggest that both dated *derived data* is kept as well as references to regeneration procedures to be used on access. A query can check if the derived data are adequately up to date, and if they are not, the query can trigger a regeneration.

Given this choice only time-critical queries have to cause a regeneration of the derived data element. The decision is made according to the time-stamp associated with the element and the need of this query, rather than according to some predefined schedule. There are several benefits:

No generation is triggered by data entry.
Casual queries can avoid any regeneration.
Multiple regeneration processes associated with repetitive accesses are avoided.

At query time the first access to a data segment with derived data will trigger a regeneration only if the `acceptable_time > last_update_time` for some data

360pt Figure space

Figure file:

Figure 8-8 Time-triggered data derivation.

segment. Since the regeneration sets the time-stamp to the current time, no segment will be regenerated more than once during one query transaction. A default decision can be to set the `acceptable_time` to the time the transaction began. Now the first reference will initiate a derivation on access, and any further refernces to the same element will use the stored value, as if the data were derived on entry.

The general flow of such an implementation is sketched in Fig. 8-8. A technique of this type is now employed within DEC PDP-10 programming systems: if the binary output code derived from some source code by a compiler is dated later than the last source code modification, a recompilation is triggered.

8-3-5 Summary: The DBTG Schemas

Figures 8-9 and 8-10 present the essentials of the specifications for a schema language developed by Data Description Language Committees of CODASYL (Conference on Data System Languages), an organization devoted to the development and standardization of business programming, which illustrates many of the points made above.

There are two distinct sections to the schema, one to contain the more conceptual definitions, as determined during establishment of the database model, using a Data Description Language (*DDL*), and the other, the *storage schema*, to contain the results of the mapping decisions to physical storage, which define the file structure to be eemployed, using a Data Storage Description Language (*DSDL*). The purpose of the separation is to permit changes of the storage structure to be made for efficiency reasons, without affecting the user's programs, which are solely derived from the database model. Any database model change has to be supported, of course, by a corresponding storage schema change. Changes of a user's external database submodel alone should not affect the model schema or the storage schema. In earlier versions of the DBTG proposals both schema sections were combined, and in the implementation a combination of the two sections is still permitted.

The specifications shown are abstracted from the third major revision, issued in 1978. Most existing systems are based on 1969 and 1974 specifications. Many of them have not implemented all the clauses and have changed some of them.

THE MODEL SCHEMA - DDL A schema contains all the definitions for the integrated database. A database may be divided into areas with distinct access and management responsibilities. The principal unit defined in the DDL schema is the *record-type*, roughly equivalent to one relation. `Keys` define ruling parts as needed to represent entity relations, nests, or associations.

Each record-type has one `RECORD-NAME` and within the record name the data elements can be specified. A data element is identified by a `data_name`; Data elements may be hierarchically arranged within a record by means of the `level_no`. An `OCCURS` clause with a `variable_count` can be given to indicate repetition of a data element within the record. Most of the options for data elements were discussed in Sec. 8-2.

The `SOURCE` and `RESULT` clauses specify derived attributes. Derivations are restricted to copying of values from the owner records and summarization of values in member records.

¡¡¡ This page to be inserted from file CHAP8FIG¿¿¿

310pt Figure space

Figure file:

Figure 8-9 DDL defined by the CODASYL Database Definition Language Committee. Clauses in [...] are optional, in { ... / ... } are alternatives.

Procedures, written by the user community, to carry out actions beyond the capability of the database system may be specified. They can be used to protect data items, or to carry out special conversions or derivations. We encountered such procedures as escape procedures in VSAM, Sec. 4-3-5. They will be `CALL`ed when the `SCHEMA`, or a specific `AREA, RECORD` or `data_name` is accessed. These routines can be designated to be entered `BEFORE, ON ERROR,` or `AFTER` access, and can also be limited to specific actions as `ALTER, COPY, DISPLAY, OPEN, CLOSE, GET, MODIFY, STORE.`

`ACCESS CONTROL LOCK`s are available at every level to force entry of a keyword or cause some verification procedure, as shown in Fig. 12-5. The specifications for the connections "`SET NAME IS link_set_name`" are strongly related to database manipulation and are covered in more detail in Figs. 9-13 and 9-14.

THE STORAGE SCHEMA - DSDL The storage (DSDL) schema as being defined by a CODASYL commitee has to provide the physical facilities for all the records and connections defined in the model DDL schema. Figure 8-10 summarizes the proposal for the DDLC storage schema.

Optionally more than one storage schema may be used to serve a model schema, and then each storage schema has to state which model records and link_sets it `REPRESENT`s. A specific record type used in the model schema can also be supported by multiple storage record types. The mapping rules from model record names to storage record names permit a conditional expression, so that, for instance, records having certain values, say `sales_location = "France"` may be selected for the record type `European_sales`. This record type can then be assigned a certain access protection distinct from, say, `Canadian_sales`. A subschema notion is hence possible, and if the storage schemas are distinct, a distributed system can be supported. The connections can not be mapped.

To support link_sets for connection either direct or indirect pointers, or indexes may be used. If no physical structure to provide link_sets is provided, the connections have to be established using sequential scanning.

Several alternatives and hybrids for placing storage records into the storage areas are available. `PLACEMENT` can be determined by a hashing function (`CALC`), so that all members of a nest occupy the same blocks if possible (`CLUSTERED VIA SET link_set`), perhaps sharing the pages `WITH` another nest, and optionally sharing the blocks with the owner record (`NEAR OWNER`) according to this ownership connection. Placement may also be sequential according to a key attribute. Indexes to the nests may be placed by `CALC` or `NEAR OWNER`. An area contains a specified, but extendible number of blocks (`pages`).

The representation of the data elements is also defined in the storage schema in the `FORMAT` clause. The format alternatives are intended to cover any representation chosen in the schema. If no format is specified, the implementor can choose any representation consistent with the `TYPE` given in the schema.

Derived data are also defined here, and an option is provided to permit storage of data derived on access. The actual derivation procedure is given in the `RESULT` clause in the main schema. Schema clauses related to processing and the statements used to manipulate the CODASYL database are shown in Sec. 9-5. In the next section we now will present some typical alternatives for schema implementation.

¡¡¡ This page to be inserted from file CHAP8FIG¿¿¿

310pt Figure space

Figure file:

Figure 8-10 The DBTG-DSDL storage schema specification proposal.

8-4 MANIPULATION OF THE SCHEMA

The data in the schema contains the information to be used to control a database. The information can be used to generate a database system and its processing functions, or its use can be deferred to the execution time of the processes that manipulate the database. This choice is related to the concept of a binding time introduced in Sec. 1-4.

The schema, as written by the organizer of the database, has to be translated for use by a database-management system. The binding choices are the familiar alternative of compiling versus interpretation.

In the environment of the database, compiling is equivalent to using all the information in the schema when the application programs are created; the schema can then be discarded. Interpreting, on the other hand, is equivalent to the use of a general program that, when called upon to carry out database manipulations, looks at the schema to find data items and determine their relationships.

8-4-1 Examples of Schema Translation

In order to clarify the alternatives, will sketch some examples of schema translation and schema use. In Sec. 8-4-2 we will summarize the relative advatages and disadvantages.

The schema in the RETRIEVE example (Fig. 8-5) used in Sec. 8-2-1 is simply stored in the form of an array on the schema file. The translation process consisted only of formatting the three fields of each schema entry. The schema subsequently was used interpretively. Figure 8-11 shows a similar process for a more complete schema language. Here, however, the source schema is analyzed and its elements are used to build tables and dictionaries for the interpreter shown in Fig. 8-14. The dictionaries are used for recognizing the terms which appear in the user's commands, and the tables direct a set of interpreting routines to the contents of the database.

190pt Figure space

Figure file:

Figure 8-11 SOCRATE interpretive sequence.

148pt Figure space

Figure file:

Figure 8-12 TOD schema compilation for execution interpretation.

250pt Figure space

Figure file:

Figure 8-13 IDS compilation sequence.

Figure 8-12 sketches the process in the Time-Oriented Database System (TOD). In order to support extensive data analysis in a research environment, a set of functions to manipulate the database is available in a TOD routine library. The users write programs using these functions. The database is kept in transposed form for rapid attribute access (see Sec. 4-1-4), but the users have a traditional tabular view.

The source schema is translated by a separate compiler into an encoded tabular form, essentially a relation, for interpretation. The library routines, invoked by the user's programs, fetch at execution time the required schema tables for request interpetation and data access.

Interpretation of the schema is avoided in the process used in many commercial database systems. We show the specifics of the Honeywell IDS system in Fig. 8-13. Here a user's COBOL program, with special database reference statements, is combined with the schema and processed by the IDS compiler. The result of the IDS translator is a clean COBOL program text which contains statement sequences and declarations to carry out the IDS functions as well as the user's own COBOL data processing specifications. This

360pt Figure space

Figure file:

Figure 8-14 IMS translation and operation sequence.

text is compiled by a COBOL compiler to produce assembly text, which in turn, is assembled into a binary program. Basic IDS file-manipulation routines, as well as COBOL subroutines, are added by the loader, and execution can commence without any further reference to the schema.

In the approach taken by the IBM IMS system, the schema and program components which become part of the final execution package are generated at different times, stored on separate libraries, and scheduled for execution by the scheduler component of the IMS system. The components and their integration are sketched in Fig. 8-14. The schema language, DL/1, concentrates on the structure of the file and leaves usage specification largely to the users' programs. No separate translator is provided for the schema language DL/1; all statements are assembly language macros. The assembler generates table entries from these macros, and these are linked by IMS to the user programs when IMS starts a user process.

Out of the DL/1 file, description subschemas, which omit unnecessary detail, can be extracted to be used by the programs. Some additional macro-statements in the schema provide a cross-referencing capability so that referenced relation types can be defined. We will discuss the use of DL/1 schemas in Sec. 9-6 when the IMS data-management system is discussed.

Many more implementation alternatives are possible. Which alternative is chosen will have a major effect on the structure permitted by the schema. The user and system developer perceive the data organization through the schema presented to them. The use of the database will depend on the concepts supported by the schema, since these concepts are needed for the user to manipulate the database. The debate which concepts are best will continue forever. Tabular models have served models of reasonable size well. When the models become large a schema which supports a hierarchical decomposition of the model space becomes desirable. An integrated database will probably require a schema which can support more than one hierachical view.

8-4-2 Evaluating the Alterntives for Translation of the Schema

The approach shown for IDS is typical of compilation methods, while the approach shown for SOCRATE is typical for interpretation. We will discuss the arguments for and against these two approaches, keeping in mind that compromises between the two methods exist as well.

COMPILATION The benefits of compiling the schema information are the following:

1 High process time efficiency due to use of actual machine code. It is generally conceded that interpretation of simple arithmetic statements is anywhere from 10 to 200 times slower than the execution of compiled code, and this experience may lead us to reject the interpretive approach.

2 Special programs can be written and compiled to deal with special situations. The entire processing repertory of the computer is available, including multiple compiler language and their libraries.

3 No explicit space is required at process time to hold the schema.

4 Facilities exist within programming languages to support this approach. One of these is the capability to include source modules from a schema file, and another is the tailoring of programs provided by the use of macro expansion capabilities.

The disadvantages of the compiled approach are the following:

1 All programs have to be regenerated when the schema changes, and new programs have to be written when new data and relationships have been specified.

2 It is difficult to control whether all programs use the schema completely and are not dependent on specific internal knowledge of the data organization. This can leead to unpleasant suprises when changes to the schema are made.

3 The apparent benefit of not requiring storage for the schema at processing time is largely illusory, since all the required information is stored somehow within the processing programs and might well appear multiple times in independent programs.

4 Most compilers and higher level languages do not have the capability of defining database structures as complex as we would like to be able to handle. Conditional, composite, and iterative fields, for example, are rarely available on the language level.

5 It can be difficult to adapt an existing compiler to generate code for some new hardware features which are being developed to aid databases.

INTERPRETATION The benefits of the interpretive methods are related, of course, to the disadvantages of compiling. We can emphasize the following:

1 Data-oriented changes need to be introduced in the schema only, and do not need programming effort.

2 Better control exists over the contents of, and access to, the database.

3 Programming can be carried out before the exact data specifications are known.

4 Synchronization of accesses to a shared file is easier to maintain in a multiprogrammed environment because more process information is available.

5 Greater independence between processes and files exists. This feature will protect the system from hardware changes.

6 It may be possible to implement frequently used and well-established functions in hardware or in micro-coded procedures.

6 The source-language statements may require less storage space than the machine instructions needed for file access.

Disadvantages of the interpretive method include the following:

1 The general interpreter has less flexibility when special considerations have to be taken care of. Recompiling of programs is not costly compared with the file reorganization effort involved when the schema is changed.

2 There is a high central processor cost to interpreting. Although, in terms of the total time to complete a job, this cost may be minimal because of the lengthy file accesses, less CPU time will be available to other processes.

3 Good general programs are difficult to construct and debug.

4 The single generalized interpreting program may occupy more space than the separate processing programs which are brought in on demand only. In a paging system, the demands on primary memory space for either approach may be substantially equal.

Compromises include compiled references to programs which can adapt to changes in the database schema. The actual choice of approach will be determined largely by available facilities and expertise, although the wrong choice can be costly.

8-4-3 Internal Representation of a Schema

The internal representation of a complex schema may itself require a linked structure to describe the structural dependencies of the data elements. In SOCRATE this is achieved through a number of interlinked tables as shown in Fig. 8-15, which is based on the schema in Example 8-6.

The dictionary table includes both attribute names and the data values permitted in domain controlled by lists of choices. The *next-pointers* provide rings within a nest and the *detail-pointers* provide access to nests when required.

306pt Figure space

Figure file:

Figure 8-15 SOCRATE internal schema representation.

8-5 SUBSCHEMAS

In a large database, the schema itself may be of massive proportions. Since not all the information in the schema is required by every user or program, the data in the schema may be categorized and selected according to several dimensions. These dimensions include:

Functional level of the schema user
Responsibility and ownership of data
Processing function
Host-language adaptations
Location of stored database fragments

All these dimensions can be used to divide schemas into subschemas which restrict the user or the database to a subset of data and function, depending on responsibility, *need-to-know*, and location.

The diversity and complexities of schemas are in part caused by the differences in objectives seen by schema designers.

206pt Figure space

Figure file:

Figure 8-16 Schemas and interfaces.

8-5-1 Subschemas Based on View and Function

As a part of preparations for a possible standardization effort, a committee of the American National Standards Institute (ANSI), X3-SPARC, defined in detail the schema function–user function correspondences and some of the many interfaces among people and system modules to be considered. The terminolgy used here and the interfaces shown in Fig. 8-16 are based on this work. The ANSI-SPARC descriptions are based on a three-part schema: the conceptual schema implements the database model, the internal schema implements the storage definitions, and external schemas implement the concept of a database submodel.

EXTERNAL SCHEMA A user who is interested in obtaining information from the database may need only an *external* description of the content of the database. The user can obtain a list of the entities, their attributes, and the type characteristics of these attributes from the system. The user can request the data in interesting combinations and specify data reduction processes.

The external schema for a user of the database may be closely related to the view model which was used to construct this portion of the database model. Many external schemas are composed of subsets of the database relations and subsets of the attributes of the selected relations. Where the database submodel differs structurally from the implemented database, substantial transformations may be required, including joins and sorts. Simple transformations include the specification of record segments. Substantial delays may be encountered if the transformations are major. A user who uses the database frequently will, of course, form concepts about the system by learning what the system does poorly and what it does well and willadapt to the database, ignoring the original view. External schemas may be changed to adapt to the needs of the user without impacting other schemas.

An example of simple subschema definition is shown in Example 8-11. A `VIEW` on the relations or `TABLE`s in IBM SQL/DS permits projection and rearranging of attribute columns, selection of rows, and specification of derived variables. The `VIEW`s in SQL/DS never create relations; all mapping is performed at the time when the view is used. The SQL language is summarized in Sec. 9-2.

8-11 An SQL External Subschema Definition

Given a relation `Children` with data in traditional measurements:

```
   Children: RELATION
     child_id, guardian, class, rank, age_c, height, weight, year_entered;
```

we define a view for the modern metric-thinking school nurse

```
CREATE VIEW School_nurse
( child,    age,   height_cm,     weight_kg      )
AS SELECT child_id, age_c, height * 2.54, weight * 0.4536
FROM Children;
```

A database will be accessed via multiple external schemas. One external schema contains the entries required for the data processing of one area of interest. External schemas may overlap to reflect overlapping data models. The use of multiple schemas also aids in the maintenance of the integrity of the database, since fewer of the relations are exposed to processes which may contain errors.

This concept of an external schema to constrain access provides control at little additional cost. If access programs are not constrained, a database system, in order to protect data from unauthorizee access, will have to verify of every record and field address submitted as being appropriate for the user. In SQL the original database `TABLE` creator can define `VIEWS` and assign them to specific users with certain privileges to provide access control.

Multiple active processes may use the same external schema. Distinct external schemas may overlap. If any overlapping schemas are used at the same time, it is desirable that the processes share the schema entries at some level. This provides a linkage which the system can use to avoid problems of access interference. Only privilege information, tied to the users rather than to the database elements, must remain separate.

CONCEPTUAL SCHEMA The external view is derived from the overall *conceptual* schema which represents the entire database model. Here all relationships are described. The conceptual model covers the information-processing needs of an enterprise or a large portion of the enterprise. As the real world changes, the conceptual schema will have to be adjusted. It is desirable that derived external schemas using conceptual elements which are changed can be restated to provide a relatively constant user interface. Only the external schema which required a conceptual change due to a related change in its view model will require modification.

If all users access the database through external schemas, the conceptual model may not be physically present during database processing; its main function is design and schema generation.

INTERNAL SCHEMA The operational management of the stored files requires further information to be placed in an *internal* schema. The internal schema defines where the database attribute values are placed and how they are accessed. The decisions encoded in the internal schema present the aggregate requirements of all the users. Here the notion of a *database administrator* appears. The load estimates or measurements, as defined in Sec. 5-4, are applied here and are balanced with response-time requirements for specific transactions. Figure 8-10 provided examples of the tools available to tune the performance of a database. The principal concept is always the addition of redundant access structures and the maintenance of locality. Control over actual and potential data derivations is part of the internal schema.

To provide input to performance control functions, the internal schema may also contain descriptive statistical data elements which are updated during operation. The TOD system, for instance, collects in its schema the access counts to attributes, to be used for periodic restructuring by the data administrator. If the database is distributed over multiple physical locations, corresponding multiple internal schemas can be used to implement the single conceptual schema.

8-5-2 Physical Organization of Schema Characteristics

A database schema may be organized by attribute or by characteristic: *names, domains, titles, control of access to data, etc.*

Certain database processes will not need all the characteristics that we have associated with each data element. Processing programs specifically may only need access to the type and length characteristics of the data elements. If data are only to be moved, only the length and count are needed. The descriptive information which contains titles, unit specifications, etc., can remain inaccessible. This physical organization reduces memory and paging requirements.

The TOD schema table, for instance, is transposed during translation to provide the schema tables in a compact form to the interpreter. This means that, for instance, all `TITLE` information is available as a single array to the output processor, and all `TYPE` information is placed into another array to be used by the computational processes.

8-5-3 Adaptation to a Host Language

Yet another aspect of a subschema is considered by CODASYL and SQL. The intention is that the schema and the schema description language can be utilized by programs written in a number of different computer languages. Variations of the subschema language are designed to fit the syntax, semantics, and power of the host language. Languages to be supported include COBOL, FORTRAN, and PL/1. Languages with flexible data structures, like LISP, are not easily supported by traditional schemas and have either used intermediate modules as interfaces, or treated databases as large virtual memory segments.

Even in procedural languages there are problems to be resolved due to representation and structure differences. An instance is the COBOL language capability to have a structure containing element types which `OCCUR` a variable number of times, as shown in Fig. 8-9, a notion not included in FORTRAN. In PL/1 a feature affecting schema power is the capability to handle variable-length character strings.

The problem is aggravated in a distributed system with heterogeneous processors, where, for instance, real values may have different representations. In a mixed language or computer environments users will have to agree to avoid incompatible data types.

8-5-4 Data Ownership

The database administrator has the responsibility for the successful operation of the database system. While the structural integrity and the adequacy of the interfaces between external, conceptual, and internal schemas require a central view, it may be necessary to delegate data content responsibility to those most closely associated with a particular database submodel. Other aspects of this function are discussed in Sec. 8-6-2.

The use of multiple external database schemas is related to the assignment of responsibilty for the maintenance of the stored data. Certain data elements may be collected and controlled for quality by one group. This group then may use an external schema to manage its data and assign permission for its use. A new concept pertaining to data emerges here, namely, *data ownership*. This is another candidate for an attribute to be specified in the schema language, or it may be a part of the access privilege specification. When attributes or relations are to be added, the maintenance responsibility over these data should be established. Private and public data may be distinguished, as well as source and derived data elements.

Feedback may be necessary to inform owners of the outside uses of the data, since not all functions of data elements and data relationships can be perceived from one external schema. The DBTG `AREA` specification allows the partitioning of the database into a number of areas. Areas can be individually protected. In several implementations this also means that the data can reside on physically distinct devices.

DERIVED DATA Ownership and data responsibility also extends to derived data. When `on-entry` and `on-access` database procedures are used, control can be well defined, but most derived data is placed into databases by explicit programming. These derived data, when stored within the database, become indistinguishable from data which entered the database from the outside. The responsibility for the validity of results derived by processing source input is assigned to the group which controls the processing programs. Errors in the source data may be the responsibility of another group. The chain of derived results can go through various levels and may have involved recursion, making checking and restoration difficult.

PROTECTION BY DATABASE PROCEDURES There may, for instance, be an AREA for medical data of employees which can only be updated and written by authorized staff, defined by a keyword variable check_MD.number. To disable access by nonauthorized staff to certain disease data, the database administrator can also specify a database procedure hush which disables retrieval of data based on values in the access path as shown in Example 8-12. Monitoring of access to data also may be necessary for confidential data items. The hush routine could achieve this by also reporting every approved use to a logging device.

8-12 Subschema Protection

```
AREA_NAME IS medical.
RECORD_NAME IS diseases WITHIN medical.
ON GET   CALL hush  USING diseases;
PRIVACY LOCK FOR STORE IS check_MD_number.
1    patient_name    PICTURE 20(X); OCCURS 1 TIMES; CHECK IS NULL.
1    social_sec      PICTURE 9(9); OCCURS 1 TIMES; CHECK IS NULL.
1    sick_call_no    PICTURE 99.
1    sick_call    OCCURS sick_call_no TIMES.
2    disease       PICTURE 50(X).
2    ICDA_code     PICTURE 9999V9.
2    days          PICTURE 999.
```

Once a database is operational any changes to the schema can cause many programs to fail. The use of a database may not have been controlled so that all users who are potentially affected by a contemplated change can be identified and warned. A check_program attached to the record type to be changed or to an entire AREA can be used to report on usage which may conflict with contemplated changes.

8-6 STRUCTURE, SCHEMA, AND USAGE

The approaches associated with the different structures and schema formulations also imply differences in the approaches to the envisaged use of the database. These differences will be accentuated in the implementation of database systems as described in Chap. 9. A categorization into three conceptual approaches follows:

1 A schema which concentrates on the naming and the semantics of source attributes and domains will be associated with a *relational* approach to database systems. The entire database is a potential resource, where linkages, derivations, and associations are established when needed. Potential relationships are found through matching domains in the schema.

2 A schema which organizes the data elements and segments into record-like structures is associated most strongly with traditional *data-processing* technology. Reports, tables, and the like will be easy to generate and to process. Structural binding between files increases the efficiency of the common processes. Potential relationships are specified in the schema; corresponding attribute values in the database relate actual tuples.

3 If the relationships among data elements are irregular and complex, and the data records are also irregular, it may not be possible to place relationship information into a schema. Now data segments found along network paths which

are dynamically selected by the user's programs and interaction. Examples of such approaches are found in some *artificial intelligence* systems. Such a system will support the *retrieval of information* in an intellectually complex mode. The volume of data is typically relatively small. The potential diversity of data and structure encountered may make traditional data processing difficult.

In practice the three categories are rarely pure. The DBTG approach, for instance, falls between **2** and **3**, and many statistics support systems are between categories **1** and **2**.

Important objectives of the use of databases are *data independence* and *control over the database*. The final sections of this chapter will comment on these issues.

8-6-1 Data Independence

An early impetus toward the use of schemas to describe content and structure of databases was the desire to move databases from one computer to another in order to support growth or to obtain the use of newer, more economical systems. The user of a large database is currently tied to the hardware that supports the files and programs for that database. Some of these anchors are due to low level constructs that are used in high-level programming. For instance, if pointer references have been used in one database system, it becomes very difficult to translate these references to another system directly, since we cannot expect that the new database structure will be mapped into the same relative positions on the files. More serious difficulties are caused when in one system, data is accessed via indexes and reside on sequential or random files, and on the other system the data is accessed through links as in a multiring structure.

The desired independence of applications from hardware and software systems has been called *data independence*. We can, however, distinguish three dependencies which have to be resolved:

Data Dependence Program instructions depend on the encoding of data elements. Issues are the size of elemental fields (i.e., 32- versus 36- or 60-bit words, and 6-, 7-, or 8-bit characters), floating-point representations, and pointer values.

Structure Dependence Program logic depends on facilities to segment records, manage nests, and provide and follow interfile linkages. Whether an employee's department is found by tracing through a ring, using a direct pointer, reading the file backward to obtain a master record, or reentering the file with a fetch argument can bind the employee-department relationship in specific ways, not foreseen in the original conceptual database model.

Program Dependence The dependency on assumptions made in the programs in regard to usage and data structure that are not reflected in the schemas. Aspects of data that are not essential to the information-storage and -manipulation problems but provide convenience and programming efficiency can bind data structures in unexpected ways. A programmer may *know* that an employee's personal data are in a record position which corresponds to the record position in the payroll file and make use of this knowledge, even though no direct relationship is defined.

A tool which helps to achieve data independence is the schema. In order to achieve structure independence the structure of the data has to be described in a standardized form. Programming independence will be achieved only if programs make full use of the capability for data and structural independence. It is not yet clear how much independence can be achieved while retaining maximal efficiency in large databases.

DATABASE TRANSLATION The development of automatic or semiautomatic tools to move a database with its programs from one system to another has demonstrated the need for the schematic description of the database. Database translation, if successful, provides an alternative to complete data and structure independence, since it should allow the translation of the database from an efficient version on one system into an efficient version in another situation. In general, the process is carried out in two stages. An intermediate stage is the formal representation of the database, which is not concerned with efficiency but rather with generality. The description of the intermediate stage is related to the conceptual schema obtained from a database-model construction. Previously existing constraints add detail and complications to the data translation process. We will not discuss the subject of database translation further, but it should be clear that many of the rigorous approaches to database management are closely related to the problem of database translation.

8-6-2 Database Control

In an organization where many users and programmers share the database and use the facilities of the schema, or of subschemas, some joint control of the data organization will be required. This may be implemented through an automatic protection system, which prevents schema changes that could cause problems to others. In general, a human arbiter will be called upon to resolve conflicts. Typical conflicts occur when an overall system improvement causes severe inconveniences for a few users, or when new requirements by some users affect many others to a slight extent. We will refer to this set of control decisions as *database management*, and we consider both automatic and manual actions under this heading.

In order to carry out the database management functions, additional information may be appended to the schema. Required are parameters that control system performance and the assignment of access privileges as mentioned previously. Maintaining additional copies of data, linkages, indexes, maintenance of actual results, and so forth, improves the retrieval performance of systems but leads to redundancy in stored data. In order to evaluate how much redundancy is cost-effective, the database manager will continuously monitor the operation of the database system at various levels.

The person who has the responsibility for the database, the *database administrator*, has extraordinary privileges. Because of these privileges, the individual should have insights which are beyond the scope of most users and programmers. This role as seen in one system is indicated in Fig. 8-13. Database systems intentionally screen unnecessary information between the levels and compartments of the database. The benefits of controlled communication between levels were discussed in Chap. 1. The user works with a simplified model of the system and is not able to improve or defeat the system. Because of this lack of knowledge, the procedures that have been developed can continue to work effectively as the database changes.

The database administrator, on the other hand, does have knowledge of the current internal structure. To safeguard the concept of a well-structured communication between levels, a database administrator may be denied the privilege to write data-manipulating programs.

As databases become bigger and more complex, the concept of centralized control by a single database administrator for the entire database may no longer be acceptable. The knowledge required to carry out the function, and the cost of failure to properly control a very large database may make the position untenable. Whether a committee can manage better is also questionable. An associated problem is the understandable reluctance of managers of enterprises to cede much of their power. These considerations may put a limit on the growth of databases and encourage distribution of database functions. Issues associated with very large databases will appear in Chap. 9, and their security problems will be touched on in Chaps. 11, 12, and 13. The role of the database adminstrator is further analyzed in Chap. 15.

BACKGROUND AND REFERENCES

The concepts which combined and evolved into the idea of a schema, as presented here, have had their origins from many relatively independent efforts. The users of statistical databases felt the need for tabular definition of the data elements they were using at a very early time, and many statistical systems support data dictionaries (Kidd[69], Ellis[72]). An outstanding example of such a system is OSIRIS (Rattenbury[74]).

Techniques to capture the required information are surveyed by Taggart[77]; a method is described by Teichroew[77]. Data dictionaries and their use are described by Uhrowczik[73] and Curtice[81]. Several systems available as independent software products are listed in Appendix B. The importance of data dictionaries to organize data, even if no DBMS is being used, has caused the National Bureau of Standards to issue proposals for standardization (NBS[80]).

COBOL and some of its predecessor commercial languages separate file-definition statements from procedural specifications (McGee[59]). An early database system using data descriptions to provide generalized retrieval capabilities was TDMS (Franks[66]). The term "schema" appeared about 1969. The SOCRATE system of CII has one of the nicest schema languages available.

Work by the CODASYL Development Committee produced a document on an information algebra (Bosak[62]) which considered a separation of the data and their description. CODASYL's System Committee began to look at database concepts in 1966, influenced by IDS (Bachman[66,73]) and the "Associative Programming Language" (Dodd[66]). This effort generated two analyses of existing database systems (CODASYL[69S,71A], Olle[71]), and the CODASYL Data Base Task Group produced proposals for a schema-oriented system facility (CODASYL[69R,71R] and CODASYL[74]). The proposals for the COBOL augmentation are now part of the COBOL specifications (CODASYL[73,75,78]). A number of CODASYL committees are active in the database area: DDLC and DSDL on the aggregate schema CODASYL[78L], DBLTG on the COBOL subschema, FDMLC on the FORTRAN subschema, and others on subjects as database translation, end-user interfaces, and database distribution (CODASYL[78D]). Palmer[75] gives a historical overview. A reference for CODASYL approaches is Olle[78].

The publication of the CODASYL proposal led to an evaluation of database requirements and standards. The SHARE-GUIDE group of IBM system users developed a set of database-system requirements, using mainly original terminology, which reflect on the need for a schema and the information it should contain and exclude. This work was reviewed by Everest in Codd[71].

These requirements led to the ANSI-SPARC report detailing three schemas levels (Steel[75]) which greatly influenced recent proposals. Papers in Jardine[77] review that work. Kent[80] proposes a further split of the conceptual schema. Issues of distribution have also been addressed (CODASYL[78D]). A method for representation of the external user view is considered by Navathe[78]; how to update the database through views is the concern of Dayal[78] and Keller[82].

A standardization effort for relational systems is underway (Brodie[81]) and was accompanied by a survey of many current systems. External schemas for relational system have tended to be simple since no connections are specified. The availability of IBM's SQL system (see Sec. 9-2) is bound to influence this area.

Most database systems developed commercially or in academia do contain schemas of various flavors, i.e., Karpinski[71], Weyl[75] or Wiederhold[75], Chamberlin[76]. Others are listed in Appendix B. Data-domain definition is handled well in PASCAL (Wirth[72]). Claybrook in Rustin[74] and Nance[75] presents facilities to specify the file organization.

Katz[79] proposes query compilation for schemas while the cost of interpretation is evaluated by Baroody[82]. Compilation while retaining flexibility is achieved in SYSTEM R (Chamberlin[81A]).

External schemas and derived data are unified by Adiba[81] and management of derived data is considered by Cammarata[81].

The subject of database independence and translation has been an area of interest for many years, some contributions are Sibley[73], McGee in Klimbie[75], Shu[75], Navathe[76], and Shneiderman[82]. The SIGFIDET Proceedings contain many papers devoted to this topic.

PERIODICALS The activity in the database area has led to the publication of several journals devoted to database subjects. The rapid development in this field makes bibliographies and descriptive books rapidly obsolete. We will use this space to indicate sources for current material to complement the background sections of this and the earlier chapters.

Major journals in the database area are *Informations Systems* (Pergamon Press, since 1974) and *Transactions on Database Systems* (ACM TODS, since 1975). Other database-oriented publications include the *Database Journal* (A. P. Publications Ltd., 322 St.John Street, London UK, since 1975) and *Management Datamatics*, published by the IFIP Administrative Data Processing Group (Nordhoff Pub., Leyden Netherlands, since 1971).

Newsletters in the area are *Database Engineering*, produced since 1977 by the IEEE Computer Society technical committee of that name and the *ACM-SIGMOD Record* of the ACM Special Interest Group (SIG) on Management of Data. From 1968 to 1976 it was named *FDT*, for File Definition and Translation. The annual conferences (ACM-SIGFIDET 1969 to 1974 and ACM-SIGMOD since 1975) sponsored by this interest group contain many important papers (i.e., Codd[71], Dean[72], Rustin[74], King[75], Rothnie[76], Lowenthal[78], Bernstein[79], Chen[80]).

A series of conferences on *Very Large Data Bases* (VLDB) cover all database topics. The first, in 1975 (Kerr[75], ed.), was published by the ACM. VLDB 2, in 1976, Lockemann[77] and Neuhold(eds.), was published by North-Holland in 1977. VLDB 3, Merten[77](ed.), VLDB 4, Bubenko[78] and Yao(eds.), VLDB 5, Furtado[79] and Morgan(eds.), VLDB 6, Lochovsky[80] and Taylor(eds.), and VLDB 7, Zaniolo[81] and Delobel(eds.) were published by the IEEE Computer Society. The Proceedings of VLDB 8, McLeod[82] and Villasenor(eds.), are published by the VLDB Endowment, Saratoga CA.

In Chap. 11 we present some journals which are not specific to databases, but carry much relevant material as well.

EXERCISES

1 Discuss, based on statements given in Secs. 1-3 to 1-6, how a schema solves or fails to solve some of the requirements for large database systems.

2 Use a schema language to describe the file examples given in Sec. 3-0.

3 Use a schema language to describe the MUMPS example from Sec. 4-5.

4 Computer languages have rules for data-value transformations when more than one data type appears in an expression. Find such a specification and discuss its advantages and disadvantages. Relate to databases.

5 How could one specify transformations between programmer-defined data types?

6 Provide domain names and domain sizes for the example in Sec. 4-5.

7 Which units in Sec. 8-1-3 can be automatically converted to each other? (Not a real database question, but a valid database problem.)

8 Categorize SOCRATE according to the definitions given in Sec. 8-6. Explain your choice.

9 Write a schema suitable for description of the schema file itself.