W. C. McGee

Data Base Technology

The evolution of data base technology over the past twenty-five years is surveyed, and major IBM contributions to this technology are identified and briefly described.

Introduction

Around 1964 a new term appeared in the computer literature to denote a new concept. The term was "data base," and it was coined by workers in military information systems to denote collections of data shared by end-users of time-sharing computer systems. The commercial data processing world at the time was in the throes of "integrated data processing," and quickly appropriated "data base" to denote the data collection which results from consolidating the data requirements of individual applications. Since that time, the term and the concept have become firmly entrenched in the computer world.

Today, computer applications in which many users at terminals concurrently access a (usually large) data base are called *data base applications*. A significant new kind of software, the *data base management system*, or DBMS, has evolved to facilitate the development of data base applications. The development of DBMS, in turn, has given rise to new languages, algorithms, and software techniques which together make up what might be called a *data base technology*.

Data base technology has been driven by, and to a large extent distinguished from other software technologies by, the following broad user requirements.

- *Data consolidation*

Early data processing applications used *master files* to maintain continuity between program runs. Master files "belonged to" applications, and the master files within an enterprise were often designed and maintained independently of one another. As a result, common data items often appeared in different master files, and the values of such items often did not agree. There was thus a requirement to consolidate the various master files into a single data base which could be centrally maintained and shared among various applications. Data consolidation was also required for the development of certain types of "management information" applications that were not feasible with fragmented master files.

- *Data independence*

Early applications were programmed in low-level languages, such as machine language and assembler language. Programmers were not highly productive with such languages, and their programs contained undesirable hardware dependencies. Further, the complexity of programming made data inaccessible to nonprogrammers. There was a requirement to raise the level of languages used to specify application procedures, and also to provide software for automatically transforming high-level specifications into equivalent low-level specifications. In the data base context, this property of languages has come to be known as *data independence*.

- *Data protection*

The consolidation of master files into data bases had the undesirable side effect of increasing the potential for data loss and unauthorized data use. The requirement for data consolidation thus carried with it a requirement for tools and techniques to control the use of data bases and to protect against their loss.

This paper surveys the development of data base technology over the past twenty-five years and identifies the major IBM contributions to this development. For

this purpose we organize the technology into three areas, roughly paralleling the three broad user requirements just cited:

1. The development of *data structuring methods* for the representation of consolidated data;
2. The development of *high-level data languages* for defining and manipulating consolidated data; and
3. The development of generalized *data protection facilities* for protecting and controlling the use of consolidated data.

Because of space limitations, coverage is limited to specific IBM activities that in the author's opinion have had the greatest impact on the technology. As a result, much important work has, unfortunately, had to be omitted. Also for space reasons, only brief descriptions are given of the activities which are included.

Data structuring methods

A data base management system is characterized by its *data structure class*, *i.e.*, the class of data structures which it makes available to users for the formulation of applications. Most DBMS distinguish between structure *instances* and structure *types*, the latter being abstractions of sets of structure instances.

A DBMS also provides an *implementation* of its data structure class, which is conceptually a mapping of the structures of the class into the structures of a lower-level class. The structures of the former class are often referred to as *logical* structures, while those of the latter are called *physical* structures.

The data structure classes of early systems were derived from punched card technology, and thus tended to be quite simple. A typical class was composed of *files* of *records* of a single type, with the record type being defined by an ordered set of fixed-length *fields*. Because of their regularity, such files are now referred to as *flat files*. Records were typically used to represent the entities of interest to applications (*e.g.*, students and courses), and fields were used to represent entity attributes (such as student name and course number). Files were typically implemented on sequential storage media, such as magnetic tape.

When data consolidation was first attempted, the limitations of early data structuring methods immediately became apparent. The main problem was the lack of an effective method for representing the entity associations that frequently appear when data are consolidated (*e.g.*, the one-many associations between courses and course offerings and the many-many associations between course offerings and students). The processing required to reflect such associations was not unlike punched card processing, involving many separate sorting and merging steps.

Early structuring methods had the additional problem of being hardware-oriented. As a result, the languages used to operate on structures were similarly oriented.

In response to these problems, data base technology has produced a variety of improved data structuring methods, many of which have been embodied in DBMS. While many specific data structure classes have been produced (essentially one class per system), these classes have tended to cluster into a small number of "families," the most important of which are the *hierarchic*, the *network*, the *relational*, and the *semantic* families. These families have evolved more or less in the order indicated, and all are represented in the data structure classes of present-day DBMS.

• *Hierarchic structures*

The hierarchic data structuring methods which began to appear in the early 1960s provided some relief for the entity association problem. These methods were developed primarily to accommodate the variability that frequently occurs in the records of a file. For example, in the popular two-level hierarchic method, a record was divided into a *header* segment and a variable number of *trailer* segments of one or more types. The header segment represented attributes common to all entities of a set, while the trailer segments were used for the variably occurring attributes. The method was also capable of representing one-many associations between two sets of entities, by representing one set as header segments and the other as trailers, and thus provided a primitive tool for data consolidation.

By the mid-1960s, the two-level hierarchic record had been generalized to n levels. For example, GIS [1, 2] provided up to fifteen levels, but with a single segment type only at each level. By the end of the 1960s, n-level hierarchies with multiple segment types at each level were found in such systems as TDMS [3], MARK IV [4], and IMS [5, 6]. Implementations of n-level hierarchic structures on sequential media tended to follow the segmented-record approach, with segments being recorded in "top down, left-right" sequence. These structures have also been implemented extensively on direct access storage devices, which afford numerous additional representation possibilities.

IMS was one of the first commercial systems to offer hierarchic data structuring and is often cited to illustrate

the hierarchic structuring concept. The IMS equivalent of a file is the *physical data base*, which consists of a set of hierarchically structured records of a single type. A record type is composed according to the following rules:

- The record type has a single type of *root* segment.
- The root segment type may have any number of *child* segment types.
- Each child of the root may also have any number of child segment types, and so on, up to a maximum of 15 segment types in any one hierarchical path and a maximum of 255 segment types in the complete data base record type.

Record occurrences are derived from the following rules:

- A record contains a single root segment.
- For one occurrence of any given segment type there may be any number of occurrences (possibly zero) of each of its children.
- No child segment occurrence can exist without its parent. This point is essentially a restatement of the hierarchic philosophy. It means, for example, that if a given segment occurrence is deleted, so are all its children.

An unusual feature of IMS is the multiple implementations which have been provided for its data structure class. For any given physical data base, the user may select an implementation that best matches the use to be made of that data base. For example, the Hierarchic Indexed Sequential Access Method (HISAM) implementation uses physical contiguity to represent hierarchic record structure, and thus provides efficient sequential access to the segments of a record. The Hierarchic Indexed Direct Access Method (HIDAM) implementation, on the other hand, uses pointers to represent hierarchic structure, thus providing for efficient segment insertion and deletion.

- *Network structures*

While hierarchic structures provided some relief for the entity association problem in the early 1960s, a more general solution had to await the introduction of the direct access storage device (DASD), which occurred on a large scale in the mid-1960s. DASD made possible a new family of data structuring methods, the *network* methods, and opened the door to the development of present-day DBMS.

The first network structuring method to be developed for commercial data processing had its origins in the bill-of-materials application, which requires the representation of many-many associations between a set of parts and itself; *e.g.*, a given part may simultaneously act as an assembly of other parts and as a component of other parts. To simplify the development of such applications, IBM developed in the mid-1960s an access method called the Bill-Of-Materials Processor (BOMP), and in the late 1960s, an enhanced version of BOMP known as the Data Base Organization and Maintenance Processor (DBOMP) [7]. The BOMP (and DBOMP) data structure class provides two types of files, *master files* and *chain files*, each file type containing records of a single fixed-format type, and a construct called a *chain*, consisting of a single master file record and a variable number of records from one chain file. A given chain file record can reside in multiple chains of different types, thus associating the master file records at the head of these chains. For the bill of materials application, two chain types—a "component" chain and a "where used" chain—are sufficient to represent many-many part associations.

Although developed for bill-of-materials applications, the BOMP data structure class has been used extensively in a variety of other applications. Essentially the same data structure class is provided in the TOTAL DBMS of CINCOM, perhaps the most widely used DBMS in the world today [8]. In TOTAL, two kinds of files are provided: *master files* (or *single-entry files*), corresponding to the master files of BOMP, and *variable-entry files*, corresponding to BOMP chain files. Provision for creating chains in TOTAL is similar to that in BOMP, although many of the restrictions in BOMP have been removed in TOTAL (*e.g.*, variable-entry files can have multiple record types). While the TOTAL system goes considerably beyond BOMP in terms of function provided, its BOMP heritage is still clearly discernible.

Another highly successful network structuring method is that developed by C. W. Bachman and associates at General Electric for the Integrated Data Store (IDS) System [9]. In IDS, a data base is composed of *records* and *record chains*. There is no concept of a file. The record chain is analogous to the BOMP chain, consisting of a single *owner* record and a variable number of *member* records. As in BOMP, a record can be a member of multiple chains of different types. Unlike BOMP, an owner record can, in turn, be a member of other chains. This generalization permits the construction of hierarchies of any depth, as well as networks of considerable complexity.

The IDS data structure class was used as the basis of a data base language developed by the Data Base Task Group of CODASYL in the late 1960s and early 1970s [10]. This language introduced some new terminology (*e.g.*, chains became *sets*) and generalized some features

STUDENT	NUMBER	NAME
	12345	ADAMS
	15418	BOSWELL
	31416	CHICHESTER

Figure 1 STUDENT table.

of the IDS class (*e.g.*, providing an ownerless set, yielding the equivalent of a file). The DBTG language has been incorporated into the *COBOL Journal of Development* [11] and has been implemented in a number of DBMS, including Cullinane's IDMS [12] and UNIVAC's DMS/1100 [13].

The IMS system provides a *logical relationship* facility, which yields many of the benefits of the DBTG data structure class. With this facility, a segment may be (in DBTG terms) a member of two sets: the set of *physical child* segments of a *physical parent* segment, all appearing in the same data base record, and the set of *logical child* segments of a *logical parent* segment, which may occur in different records in the same or different data bases. The logical relationship is thus a special case of the DBTG set construct, but is nevertheless capable of modeling most information situations of practical importance, such as many-many binary associations. The logical relationship is not, strictly speaking, a part of the IMS data structure class, since a mapping facility is used to shield the programmer from logical relationships and preserve his strictly hierarchical view of data. It is a significant contribution to the technology because it demonstrates that the entity association problem can be solved without exposing complex networks to the programmer.

- *Relational methods*

In the mid-1960s, a number of investigators began to grow dissatisfied with the hardware orientation of then extant data structuring methods, and in particular with the manner in which pointers and similar devices for implementing entity associations were being exposed to the users. These investigators sought a way of raising the perceived level of data structures, and at the same time bringing them closer to the way in which people look at information. Within IBM, Davies [14], Raver [15], Meltzer [16], and Engles [17] at different times and in different contexts described an *entity set* structuring method, wherein information is represented in a set of tables, with each table corresponding to a set of entities of a single type. (A similar construct was used in the MacAIMS system of MIT as a canonical form for representing associations among data items.) The rows of a table correspond to the entities in the set, and the columns correspond to the attributes which characterize the entity set type. The intersection of a row and a column contains the value of a particular attribute for a particular entity. For example, the STUDENT table in Fig. 1 describes a set of students having attributes NUMBER and NAME.

Tables can also be used to represent associations among entities. In this case, each row corresponds to an association, and the columns correspond to entity identifiers, *i.e.*, entity attributes which can be used to uniquely identify entities. Additional columns may be used to record attributes of the association itself (as opposed to attributes of the associated entities). For example, the ENROLL table of Fig. 2 describes a set of associations between course offerings (identified by COURSE and DATE) and the students (identified by STUNUM) enrolled in those offerings.

The key new concepts in the entity set method were the simplicity of the structures it provided and the use of entity identifiers (rather than pointers or hardware-dictated structures) for representing entity associations. These concepts represented a major step forward in meeting the general goal of data independence.

In the late 1960s, E. F. Codd [18] noted that an entity set could be viewed as a mathematical relation on a set of domains $D_1, D_2, \cdots, D_n$, where each domain corresponds to a different property of the entity set. Associations among entities could be similarly represented, with the domains in this case corresponding to entity identifiers. Codd defined a (data) *relation* to be a time-varying subset of the Cartesian product $D_1 \times D_2 \times \cdots \times D_n$, *i.e.*, a set of n-tuples (or simply *tuples*) of the form

$$\langle v_1, v_2, \cdots, v_n \rangle,$$

where v_i is an element selected from domain D_i. One or more domains whose values uniquely identify the tuples of a relation is called a *candidate key*.

Aside from the mathematical relation parallel, Codd's major contribution to data structures was the introduction of the notions of *normalization* and *normal forms*. Codd recognized that the domains on which a relation is constructed can in general be composed of elements of any kind; in particular, domains can be composed of other relations, thus leading to the "nesting" of relations of potentially any depth. Codd showed that there was no fundamental advantage to this nesting and that, in fact, it only tended to complicate the information modeling process. Instead, he proposed that relations be built exclusively on domains of elementary values—integers, char-

acter strings, etc. He called such relations *normalized relations* and the process of converting relations to normalized form, *normalization*. Virtually all work done since with relations has been with normalized relations.

Codd also perceived that the unconstrained construction of normalized relations could lead to semantic anomalies. For example, when a tuple represents an association between two or more entities and at the same time represents (parasitically) the attributes of the individual entities, values for the latter will in general be replicated throughout the relation, entailing duplicate updating. Similarly, when a tuple represents an entity, some of the attributes therein may be attributes of a second (masquerading) entity which is associated in some way with the first entity. When this occurs, entities of the second type cannot be represented (inserted, deleted, etc.) independently of entities of the first type.

To better explain these effects, Codd postulated levels of normalization called *normal forms*. An unconstrained normalized relation is in *first normal form* (1NF). A relation in 1NF in which all non-key domains are functionally dependent on (*i.e.*, have their values determined by) the entire key are in *second normal form* (2NF), which solves the problem of parasitic entity representation. A relation in 2NF in which all non-key domains are dependent *only* on the key is in *third normal form* (3NF), which solves the problem of masquerading entities.

To avoid update anomalies, Codd recommended that all information be represented in third normal form. While this conclusion may seem obvious today, it should be remembered that at the time the recommendation was made, the relationship between data structures and information was not well understood. Codd's work in effect paved the way for much of the work done on information modeling in the past ten years.

As part of the development of the relational method, Codd postulated a *relational algebra, i.e.*, a set of operations on relations which was closed in the sense of a traditional algebra, and thereby provided an important formal vehicle for carrying out a variety of research in data structures and systems [19]. In addition to the conventional set operations, the relational algebra provides such operations as *restriction*, to delete selected tuples of a relation; *projection*, to delete selected domains of a relation; and *join*, to join two relations into one.

Codd also proposed a *relational calculus* [19], whose distinguishing feature is the method used to designate sets of tuples. The method is patterned after the predicate calculus and makes use of free and bound variables and

ENROLL	COURSE	DATE	STUNUM	GRADE
	M23	F78	12345	A
	M23	F78	31416	F
	M23	W78	31416	A

Figure 2 ENROLL table.

the universal and existential quantifiers. For example, the set of names of students who received an 'A' in any offering of course M23 would be expressed as

$$\{x[\text{NAME}] \in \text{STUDENT}:$$
$$(\exists y \in \text{ENROLL})\ (y[\text{COURSE}] = \text{'M23'}\ \&$$
$$y[\text{GRADE}] = \text{'A'}\ \&$$
$$y[\text{STUNUM}] = x[\text{NUMBER}])\}$$

Codd recognized the existence of many possible manipulation languages for relations and proposed that the relational calculus be used as the standard against which these languages could be measured for completeness. In [19] he defined relational completeness: "a language is *relationally complete* if, given any finite collection of relations R_1, R_2, $\cdots$, R_n in simple normal form, the expressions of the language permit definition of any relation definable from R_1, R_2, $\cdots$, R_n by expressions of the relational calculus."

Codd characterized his methodology as a *data model*, and thereby provided a concise term for an important but previously unarticulated data base concept, namely, the *combination* of a class of data structures and the operations allowed on the structures of the class. (A similar concept, the *abstract data type* or *data abstraction*, has evolved elsewhere in software technology.) The term "model" has been applied retroactively to early data structuring methods, so that, for example, we now speak of "hierarchic models" and "network models," as well as the relational model. The term is now generally used to denote an abstract data structure class, although there is a growing realization that it should embrace operations as well as structures.

IBM investigators have made several refinements to Codd's original definitions of normal forms. Kent [20] simplified the definitions by removing references to prime attributes (an attribute in any candidate key). Boyce [21] noted that Codd's definition of 3NF still permitted undesirable functional dependencies among prime attributes and postulated a normal form which excluded these dependencies. Codd and Boyce later collaborated on the definition of the *Boyce-Codd normal form* (BCNF), a

System	*Developed by*	*Reference*
MacAIMS	MIT Project MAC	[28]
RDMS	General Motors	[29]
IS/I	IBM UK Scientific Centre	[30]
INGRES	U. California, Berkeley	[31]
ZETA	U. Toronto	[32]
System R	IBM Research, San Jose	[25, 26]
QBE	IBM Research, Yorktown	[33]
ORACLE	Relational Software Inc.	[34]
SQL/DS	IBM	[27]

Figure 3 Relational systems.

redefinition of 3NF which subsumed Boyce's normal form and made no reference to either keys or prime attributes [22].

Fagin [23] noted that relations in BCNF could still contain higher-order dependencies, which he called *multivalued dependencies*. He proposed a fourth normal form (4NF) to eliminate multivalued dependencies and provided algorithms for reducing relations to 4NF. In subsequent work [24], Fagin described the projection join normal form (PJ/NF), the ultimate normal form when only the projection and join operators are allowed.

By providing a common context for the formulation of data problems, the relational model has proved of great value as a vehicle for research and for communication among research workers. Areas in which the relational model has been used include data base system architecture, data base machines, concurrency theory, language completeness, view updating, query decomposition (especially in distributed systems), and data equivalence.

In addition, the relational model has been implemented in a number of DBMS. Two major implementations within IBM are System R [25, 26], an exploratory DBMS developed by the IBM Research Division in San Jose, and SQL/DS [27], a program product based on System R for use in the DOS/VSE operating system environment.

A partial list of relational systems appears in Fig. 3.

A question frequently asked about relational model implementations is: How efficiently do they represent the entity associations required for the consolidation of data into data bases? At the user level, a relation seems no different from a flat file, and if the latter was not adequate for data consolidation, how can we expect the former to be? The answer lies in the hardware improvements that have been made since flat file days (notably, DASD and faster CPUs with larger memories) and in a better understanding of the problems of implementing high-level data models. Thus, relational systems make extensive use of indexes and pointers in implementing relations and relational operations. Through the use of such devices, relational systems seem capable of achieving performance competitive with nonrelational systems, without compromising the simple view of data for which the model was conceived.

- *Semantic models*

During the evolution of the hierarchic, network, and relational methods, it gradually became apparent that building a data base was in fact equivalent to building a model of an enterprise and that data bases could be developed more or less independently of applications simply by studying the enterprise. This notion has been articulated in the widely referenced ANSI/SPARC data base system architecture [35], which provides the notion of a *conceptual schema* for the application-independent modeling of an enterprise and various *external schemata* derivable from the conceptual schema for expressing data requirements of specific applications.

Application-independent modeling has produced a spate of *semantic* data models and debate over which of these is best for modeling "reality." One of the most successful semantic models is the *entity-relationship model* [36], which provides data constructs at two levels: the *conceptual* level, whose constructs include entities, relationships (n-ary associations among entities), value sets, and attributes; and the *representation* level, in which conceptual constructs are mapped into tables. The latter are similar to relations in the relational model, with the important difference that the entity-relationship model provides distinct table types for representing entity sets and relationship sets. Such semantic interpretations of relations have existed for some time, but it took Chen's paper to give them wide circulation and to create a surge of interest in the entity-relationship model.

The data structure class of the IBM DB/DC Data Dictionary program product is an embodiment of the entity-relationship model [37]. The Dictionary provides *subjects*, which may have *attributes* and which may participate in many-many binary *relationships*, which may also have attributes. In the initial release of the Dictionary, subject and relationship types were fixed in the product design and reflected the entity types typically found in a computer installation about which the user wanted to record information: data bases, records, fields, programs, etc. Subsequently, the Dictionary has provided an extensibility facility, which allows the user to define arbitrary subject and relationship types. With this exten-

sion, the Dictionary has the modeling power of a generalized DBMS, making it one of the first systems to implement the entity-relationship model.

• *Data model implementation*

The success of a data model depends not only on the degree of its hardware independence, but also on the ability to translate operations on its constructs efficiently into equivalent operations on the underlying hardware. As one might expect, these goals often conflict with one another.

For performance reasons, most data model implementations make use of *indexes*, which are essentially sets of key value-data location pairs. Rather than develop indexing techniques from the ground up, many DBMS use existing indexed access methods as their implementation base. Two access methods which have been used extensively for this purpose are the IBM Indexed Sequential Access Method (ISAM) and the IBM Virtual Storage Access Method (VSAM). ISAM and VSAM are generalized *indexed sequential* access methods, meaning that they cater simultaneously to both random and sequential access to data.

ISAM was introduced in 1966 as a component of OS/360 and was the first indexed sequential access method to find widespread use in the data processing community. ISAM made practical the use of DASD for many users, especially those who could not devote the time and effort required to develop a viable indexed access method of their own. It has been widely referenced in the literature and in textbooks as the typical indexed sequential access method.

VSAM [38] was introduced in 1972. Its major contribution was the use of a record-splitting strategy to overcome the tendency in ISAM for long overflow chains to develop after many record insertions. In addition, VSAM has made innovative contributions in the areas of index compression and index replication. The VSAM index organization is known more generally as the B-tree organization, which was developed independently by Bayer and McCreight in the early 1970s [39].

Also for performance reasons, many data model implementations make use of *hashing*, *i.e.*, the calculation of data locations from key values. W. W. Peterson [40] was one of the first to apply hashing to DASD, and his work has been extensively referenced. V. Lum and his associates at IBM's Research Division (*e.g.*, [41, 42]) have conducted systematic investigations of hashing techniques and demonstrated the general utility of the division/remainder method, which is widely used today.

High-level data languages

The history of computer applications has been marked by a steady increase in the level of the languages used to implement applications. In data base technology, this trend is manifested in the development of high-level data definition languages and data manipulation languages.

A *data definition language* (DDL) provides the DBMS user with a way to declare the attributes of structure types within his data base, and thus enable the system to perform implicitly many operations (*e.g.*, name resolution, data type checking) that would otherwise have to be invoked explicitly. A DDL typically provides for the definition of both logical and physical data attributes, as well as the definition of different *views* of the (logical) data. The latter are useful in limiting or tailoring the way in which specific programs or end-users look at the data base.

A *data manipulation language* (DML) provides the user with a way to express operations on the data structure instances of a data base, using names previously established through data definition. Data manipulation facilities are of two general types: host-language and self-contained.

A *host-language* facility permits the manipulation of data bases through programs written in conventional procedural languages, such as COBOL or PL/I. It provides statements that the user may imbed in a program at the points where data base operations are to be performed. When such a statement is encountered, control is transferred to the data base system, which performs the operation and returns the results (data and return codes) to the program in pre-arranged main storage locations.

A *self-contained* facility permits the manipulation of the data base through a high-level, nonprocedural language, which is independent of any procedural language, *i.e.*, whose language is "self-contained." An important type of self-contained facility is the *query facility*, which enables "casual" users to access a data base without the mediation of a professional programmer. Other types of self-contained facility are available for performing generalizable operations on data base data, such as sorting, report generation, and data translation.

• *Host-language facilities*

Host-language facilities evolved from the need to standardize within an installation the way in which programmers code certain common data handling operations, such as buffering, error handling, and label processing. This need resulted in "I/O subroutine packages" which were invoked by all programs in the installation. Such

packages, in turn, were generalized over computers of a given type into "I/O systems" and "access methods" applicable to many installations. The introduction of DASD greatly extended the set of operations which could be usefully generalized. Functions typically included in DASD access methods are space allocation, formatting, key-to-address transformation, and indexing.

With the introduction of data base management systems, the access method interface was replaced by the *data base sublanguage*. The data manipulation facilities of a data base sublanguage tend to be more powerful than those of access methods, permitting, for example, the updating or deleting of multiple records with a single statement. Additionally, a data base sublanguage may include statements unique to the data base environment, such as locking and transaction control statements.

Because of main storage limitations, the units of data on which data base sublanguages operate are normally relatively small, the record being the typical unit. To access larger collections of data, the programmer must "navigate" through the data base. To assist him in this, the DBMS may provide objects called *cursors* or current position indicators, which the programmer can set to point to a particular item of data, and later use to refer to that item or to a related item.

A high-level data language which is proving to be of considerable importance to data base technology is the SQL data base sublanguage of system R [43, 44]. SQL is a relational language which had its origins in several relational languages developed by IBM's Research Division in the early 1970s, including:

- The ALPHA data base sublanguage [45], an adaptation of the relational calculus for use with conventional procedural languages. Continuing the example of the section "Relational Methods," the ALPHA sequence

```
RANGE STUDENT X
RANGE ENROLL Y SOME
GET W X.NAME:
  ∃Y((Y.COURSE = 'M23') &
    (Y.GRADE = 'A') &
    (Y.STUNUM = X.NUMBER))
```

  returns the names of 'A' students in M23 to the workspace relation W, where they can be operated on by statements of the host language.
- The GAMMA-0 language [46], a low-level relational language intended for implementing relational algebras and query languages.
- The SQUARE language [47, 48], a general purpose query language which attempted through graphic conventions to avoid some of the mathematical appearance of the relational calculus and at the same time remain relationally complete. The set of names of 'A' students in M23 would be expressed in SQUARE as:

```
    STUDENT                ENROLL              ('M23', 'A')
NAME        NUMBER STUNUM          COURSE,GRADE
```

- The SEQUEL language [49], a general purpose query language based on SQUARE but providing a string-type syntax with English keywords. For basic queries, SEQUEL borrowed the SELECT-FROM-WHERE construction of existing query languages such as GIS and then elaborated this structure in a consistent manner to achieve the completeness of the relational calculus, but with much improved readability. An important characteristic of SEQUEL is the ability to "nest" SELECT clauses, permitting complex queries to be articulated into intellectually manageable chunks without losing the important nonprocedural nature of the language.

  For example, the previous query would be rendered in SEQUEL as follows:

```
SELECT  NAME
FROM    STUDENT
WHERE   NUMBER   IN
        (SELECT  STUNUM
        FROM     ENROLL
        WHERE    COURSE = 'M23'
        AND      GRADE = 'A')
```

The SQL language of System R is an enhanced version of SEQUEL. In addition to SEQUEL's query facilities, SQL provides

- Data manipulation facilities that permit the insertion, deletion, and updating of individual tuples or sets of tuples.
- Data definition facilities for defining relations, views, and other data objects.
- Data control facilities for defining access authorities and for defining transactions, *i.e.*, units of recoverable processing.

The use of SQL from programs is facilitated by permitting language variables to appear in SQL statements and by providing a cursor facility for manipulating individual tuples. The statement

$LET cursor-name BE select-statement

associates the set of tuples designated by *select-statement* with the named cursor. A cursor contains a "current tuple" pointer, so that individual tuples can be designated simply through a cursor name. For example, the statement

$FETCH cursor-name

returns the tuple pointed to by the current tuple pointer of *cursor-name* and advances the pointer to the next tuple. To illustrate, the following pseudo-program provides processing of the names of 'A' students in M23:

```
initialize;
$LET P BE
     SELECT NAME INTO $STUNAME
     FROM STUDENT
     WHERE  NUMBER IN
            (SELECT STUNUM
            FROM    ENROLL
            WHERE   COURSE = 'M23'
            AND     GRADE = 'A');
$OPEN P;
do until the tuple set designated by P is exhausted;
  $FETCH P;
  process one name in variable STUNAME;
end;
$CLOSE P;
```

The use of SQL in generalized programs whose data requirements are not known until the program is invoked is facilitated by the PREPARE and EXECUTE statements. These statements may be used to construct string representations of SQL statements (*e.g.*, including data names supplied by the invoker) and then cause these representations to be executed exactly as if they had appeared in the program to begin with.

Whereas most relational DBMS use an interpretive approach to the execution of data sublanguage statements, System R uses a compiler approach. Programs are first processed by a *precompiler* [50], which generates a tailored *data access routine* for each SQL statement in the program and which replaces the SQL statement with a CALL to the access routine. When the program is executed, all the access routines are loaded to provide targets for the translated CALLs. This approach has two advantages:

1. Much of the work of parsing, name binding, access path selection, and authorization checking can be done once by the precompiler and thus be removed from the process of running the program.
2. The access routine, because it is tailored to one specific program, is much smaller and runs much more efficiently than a generalized SQL interpreter would.

The tailoring of System R access routines is done by an *optimizer* component [51–53], which attempts to minimize the "cost" of carrying out SQL statements. Cost is a weighted combination of CPU and DASD I/O activity, with the weighting adjustable for different system configurations. In computing cost, the optimizer makes use of such "statistics" as relation sizes and number of distinct key values within a relation.

Like the relational model on which it is based, SQL has been widely adopted as a research and educational vehicle and has been implemented in a number of DBMS products such as SQL/DS.

- *Self-contained facilities*

The nonprocedurality of data processing specifications that can be achieved with a host-language facility is effectively limited by the procedurality of the host language. This limitation was recognized as early as the mid-1950s, when another approach to application development was conceived. This approach took cognizance of the fact that most data processing logic can be articulated into executions of a small set of *generalized routines*, which can be particularized for specific applications with a fraction of the effort required to write an equivalent customized program. The processes which have been most frequently generalized for this purpose are report generation, file maintenance, and (more recently) data translation.

One of the earliest generalized file processing systems was developed at the Hanford Atomic Products Operation in the mid-1950s [54]. The work done there on generalized routines for sorting, report generation, and file maintenance was picked up by the SHARE organization around 1960 and distributed under the title "9PAC" [55]. This work, in turn, was extended in many directions over the next fifteen years, giving rise to numerous families of generalized systems [56].

The most pervasive application of the Hanford concept is found in the *report program generator*, a software package intended primarily for the production of reports from formatted files. Attributes of the source files and the desired reports are described by the user in a simple declarative language, and this description is then processed by a compiler to "generate" a program which, when run, produces the desired reports. A key concept of the report program generator is the use of a fixed structure for the generated program, consisting of input, calculation, and output phases. Such a structure limits the transformations that can be carried out with a single generated program, but has nevertheless proved remarkably versatile (report program generators are routinely used for file maintenance as well as report generation). Perhaps more importantly, the fixed structure of the generated program imposes a discipline on the user which enables him to produce a running program much more

quickly than he could with conventional languages. Report program generators are especially popular in smaller installations where conventional programming talent is scarce, and in some installations it is the only "programming language" used.

The original report program generator was the IBM Report Program Generator introduced in the early 1960s for the IBM 1401 computer [57]. It was patterned after the SHARE 9PAC system and proved to be a valuable tool in helping users to migrate from punched card equipment to electronic data processing. A report program generator for the System/360 series was introduced in 1964. A much enhanced version, RPG II, was introduced in 1969 for the IBM System/3 [58]. RPG II has been implemented on System/370 and many other machines, and today it is one of the most widely used computer programming languages.

While RPG was being developed in IBM's business sector, a closely related family of products, the formatted file systems, were being developed jointly by IBM's Federal Systems Division and various military and intelligence agencies of the federal government. A formatted file system typically provides a set of generalized programs which are sufficient to implement the bulk of the application at hand. The programs are separately invokable and are so designed that the output of one can be used as inputs to the others. File structures have limited complexity, typically providing a two-level hierarchic record with multiple segment types at the second level. The formatted file systems have been used extensively in intelligence and command-control applications, where information requirements are exceptionally volatile, and the time available to respond to new requirements precludes the use of conventional programming.

IBM has been a major contributor to a number of the formatted file systems, including:

- The Formatted File System for the Air Force Strategic Air Command, developed for the IBM 7090 around 1959 and used mainly for intelligence applications (this is believed to be the first formatted file system) [59];
- The Information Processing System (IPS) for the Navy, developed in the early 1960s for the IBM 7090 and CDC 1604 [60];
- The Formatted File System for the Naval Fleet Intelligence Center in Europe (FICEUR), developed for the IBM 1410 (believed to be the most widely used of the formatted file systems) [61];
- The National Military Command System Information Processing System (NIPS), developed for the IBM 1401 and later converted to the IBM System/360 [62].

The report program generators and the formatted file systems were the precursors of the contemporary DBMS query facility. A query processor is in effect a generalized routine which is particularized to a specific application (*i.e.*, the user's query) by the parameters (data names, Boolean predicates, etc.) appearing in the query. Query facilities are more advanced than most early generalized routines in that they provide online (as opposed to batch) access to data bases (as opposed to individual files). The basic concept is unchanged, however, and the lessons learned in implementing the generalized routines, and especially in reconciling ease of use with acceptable performance, have been directly applicable to query language processors.

Most query facilities use string-type languages, such as SQL. A significant departure from this practice is the Query-By-Example (QBE) language [63, 64], which is a graphical language intended for use from a display terminal. The QBE user is presented with an outline of the tables he wishes to query, and then he expresses his query by filling in the outline with the appropriate names and special characters. The basic idea is for the user to show the system an *example* of the information he wants to see and for the system to respond by showing the user all instances that conform to the example.

For example, to query the ENROLL table (Fig. 2), the system user would first call up the outline in Fig. 4(a). To see all students with an 'A' grade in any offering of course M23, the user would enter 'A' in the GRADE column and 'M23' in the COURSE column, and then in the STUNUM column enter an example of student number, underlined to indicate that it is an example only, and annotated with a P to indicate that it is values of this column that are to be printed or displayed [Fig. 4(b)]. The system responds by displaying the numbers of all qualifying students as in Fig. 4(c).

Queries involving two or more tables are expressed by using common values as examples of the attributes on which the tables are to be matched. For example, the *names* of all students in the previous query would be retrieved with the query shown in Fig. 4(d). The system responds by displaying the names, as in Fig. 4(e).

Through the use of various other graphic conventions, the QBE user is able to express quite sophisticated queries. Predicates may include Boolean expressions (*e.g.*, grade = 'A' or grade = 'B'), comparison of two variables (*e.g.*, grade better than a specific student's grade), and universal quantifiers (*e.g.*, all grades = 'A'). Both predicates and retrieved values can include aggregate operators, such as SUM, COUNT, and AVERAGE. The main goal

of the language, however, is to make the expression of simple queries very easy. Tests conducted by Thomas and Gould [65] suggest that QBE has indeed achieved this objective.

Data protection facilities

The consolidation of data accentuates the need to protect the data from loss or unauthorized use. This protection is in many cases secured (ironically) by re-introducing redundancy into the data, but in a controlled way.

This section surveys the facilities which data base technology has provided for the protection of data. For specific examples, we draw on IMS, which is widely regarded as the DBMS which pioneered data integrity technology, and on System R, which is believed to be the first relational DBMS to incorporate a full range of data protection facilities.

- *Concurrent access control*

Most DBMS permit a data base to be accessed concurrently by a number of users. If this access is not controlled, the consistency of the data can be compromised (*e.g.*, lost updates), or the logic of programs can be affected (*e.g.*, nonrepeatable read operations).

Concurrent access control generally takes the form of data locking, *i.e.*, giving a user exclusive access to some part of the data base for as long as necessary to avoid interference. Locking can, in general, lead to deadlock among users, necessitating some method of detecting and breaking deadlocks.

In early releases of IMS, concurrent access was controlled through program scheduling, *i.e.*, a program intending to update certain segment types would not be started until all programs updating these segment types had completed. Under this regime, the granule of sharing was effectively the segment type. The segment types to be updated by a program were effectively locked when the program was started and unlocked when it completed. Deadlock did not occur, since all resources required by a program were obtained at one time.

Around 1974, a *program isolation* facility was added to IMS which permitted programs updating the same segment type to run concurrently and which prevented interference by locking individual data base records as required. With program isolation, records are locked for a program upon updating any item within the record and unlocked when the program reaches a *synchpoint, i.e.*, a point at which the changes made by the program are committed to the data base. Deadlocks can occur and are resolved by selecting one of the deadlocked programs and restarting it at its most recent synchpoint (see next section).

In addition to the implicit protection provided by program isolation, IMS permits programs to explicitly lock and unlock segments and permits users to explicitly request exclusive use of segment types and data bases (for whatever reason) before a program is started.

ENROLL	COURSE	DATE	STUNUM	GRADE

(a)

ENROLL	COURSE	DATE	STUNUM	GRADE
	M23		P.12345	A

(b)

ENROLL	STUNUM
	12345 31416

(c)

ENROLL	COURSE	DATE	STUNUM	GRADE
	M23		12345	A

STUDENT	NUMBER	NAME
	12345	P.ADAMS

(d)

STUDENT	NAME
	ADAMS CHICHESTER

(e)

Figure 4 Query-by-example displays.

A significant new capability in IMS is the ability for programs running under different invocations of the system (*e.g.*, in different CPUs) to concurrently access a common set of data bases. Additional computer capacity may thus be applied to the processing of common data, and the systems sharing the data may be tailored to specific user needs while still retaining access to common data.

System R employs an implicit locking technique similar to program isolation and like IMS allows the user to explicitly lock data objects at several levels of granularity. A novel feature is the ability of the user to specify one of three *consistency levels* in reading data:

1. Read "dirty" data, *i.e.*, data subject to backout in the event that another program updating the data ends abnormally (see next section).
2. Read "clean" but possibly unstable data, *i.e.*, data not subject to backout, but subject to update by other users between successive reads by this user.
3. Read "clean," stable data, *i.e.*, data as it would be seen by this user if running alone.

The lower levels of consistency required less locking and produce less lock contention, and may thus be used, when the application permits, to improve system performance.

- *Recovery from abnormal program termination*

The data base updating performed by a program does not occur instantaneously (typically requiring several thousands of machine instruction executions); hence, there is nonzero probability that the program will fail to complete normally and as a result leave the data base in an inconsistent state (*e.g.*, crediting one bank account without a matching debit to another account). A program can fail to complete for a variety of reasons, including illegal instruction execution, termination by the system to break a loop or deadlock, and system failure.

IMS protects against data inconsistency due to abnormal program termination by recording all data base changes made by a program in a *dynamic log*. If the program reaches a synchpoint, its dynamic log entries are discarded, thereby committing its data changes. If the program ends abnormally before a synchpoint is reached, the system (after restart, if necessary) uses the dynamic log to back out all data base changes made by the program since its most recent synchpoint. If abnormal end is due to a program error, the system prevents the program from being rescheduled until an operator intervenes. Otherwise, the system automatically restarts the program.

IMS also protects against anomalous input and output behavior which can result from abnormal program termination. If a program ends abnormally, the system discards any output messages produced by the program since the most recent synchpoint and restores the program's input message to an input queue. The input message is discarded and the output messages are delivered to their destinations only when a synchpoint is reached.

In System R, recovery from system failure is facilitated through the use of a novel dual-copy recording technique and the use of "maps" or directories which point to physical records on DASD. Updated physical records, instead of being overwritten to their original locations, are written to available DASD space, and a "current map" is updated to point to them. At checkpoints, the current map becomes a "backup map," and a new current map is started. A log is also kept of all updates occurring between checkpoints. Following a system outage, the data base is restored to a consistent state by reinstating the backup map and using the log to re-do the updates of transactions which completed after the last checkpoint.

The processing which a program does between synchpoints has turned out to be a fundamentally important concept in data base technology. This unit has come to be known generally as a *transaction*, in recognition of the fact that it is typically done on behalf of one input message or "transaction." A transaction has been defined by Eswaran *et al.* at IBM's Research Division [66] as a unit of processing which transforms a consistent data state into a new consistent state. A transaction thus behaves externally as if it were atomic, even though internally it may extend over an arbitrarily long time interval. Eswaran *et al.* also introduced the concept of a *schedule* of interleaved actions of a set of concurrent transactions and showed that a schedule is consistent (*i.e.*, equivalent to the serial execution of the transactions) only if transactions can be divided into two phases: a *growing phase*, in which locks are acquired, and a *shrinking phase*, in which locks are released. Releasing locks at the end of a transaction thus proves to be a special case of a more general procedure for achieving schedule consistency.

The transaction concept has been implemented in System R. Programs may use the BEGIN TRANSACTION and END TRANSACTION statements to bracket processing which is to be considered atomic. A COMMIT statement permits the program to create intermediate points of consistency, analogous to IMS synchpoints, and a RE-

STORE TRANSACTION statement may be used to back out all data base changes to the most recent point of consistency.

• *Data recovery*

A data base may be damaged in a variety of ways, including write errors, physical damage to a volume, inadvertent erasing by an operator, and by an application program error. The effect of such a loss on the user's installation can be mitigated through the use of data base recovery facilities.

The basic approach to data base recovery in IMS is to make periodic copies of the data sets that underlie the data base and to record data base changes on the system log. In the event of failure in a data set, the latest copy can be updated with changes logged since the copy was made, thus restoring the data set to its condition at the point of failure.

A data base change is recorded in the system log in the form of two segment images: the segment as it appeared before the change and the segment as it appeared after the change. Additional information recorded includes the identity of the program that made the change, the date and time of entry, and the identity of the data base, data set, and record being modified.

The copying of data bases is done with an image copy utility program, which creates an image copy of the data set on disk or tape. Data bases are normally copied just after the data base has been initially loaded (to obviate reloading in the case of failure) and immediately after reorganization. (Copies made before a reorganization cannot be used in recovery.) Copies may also be made at intermediate points, as determined by the update activity against the data base. Copying may be done "on-line," *i.e.*, while the data base is being used by other programs.

When data base damage is discovered, the affected data sets may be recovered by running a recovery utility program. For each data set to be recovered, the utility allocates space for a new version of the data set, loads the latest image copy into this space, and then reads the system log in the forward (time ascending) sequence looking for changes that have been made to the data set after the image copy. For each such log entry, the "after" image is used to replace the corresponding data in the data set.

System R uses a similar approach to data base recovery. Provision is made for copying data base and log checkpoint information to tape, whence it may be recalled to reconstruct the data base in the event of damage to DASD contents.

• *Access authorization*

Consolidated data often constitute sensitive information which the user may not want divulged to other than authorized people, for reasons of national security, competitive advantage, or personal privacy. DBMS, therefore, provide mechanisms for limiting data access to properly authorized persons.

The basic technique used by IMS to control access to data is to control the use of programs which access data and the use of transactions and commands which invoke such programs. The system provides for the optional definition of security tables which are used to enforce control. These tables contain entries of the form ⟨r,u⟩, where r is a class of resources, u is a class of users, and the occurrence of an entry ⟨r,u⟩ signifies that user class u is authorized to use resource class r. For example, the entry ⟨UPDATE, LTERM1⟩ might signify that UPDATE transactions can be entered only through terminal LTERM1. "Users" who may be controlled in this manner include both terminals and programs.

IMS also provides for the use of individual user passwords in order to further control the use of a terminal. Through suitable definitions, passwords can be required at sign-on to IMS and at the entry of individual transactions and commands.

In System R, access control is provided through two mechanisms:

1. The *view* facility [67], which permits subsets of data to be defined through SQL SELECT statements, and thus restricts the user of the view to those subsets. SELECT statements may contain predicates of the form fieldname = USER, to restrict access to tuples containing the user's identification code.
2. The *grant* facility [68], which permits a system administrator to grant specific capabilities with respect to specific data objects to specific users. Grantable capabilities with respect to relations include the capability to read from the relation, to insert tuples, to delete tuples, to update specific fields, and to delete the relation. The holder of a capability may also be given authority to grant that capability to others, so that authorization tasks may be delegated to different individuals within an organization.

Conclusion

Data base technology has evolved in response to user needs to consolidate data in a secure, reliable way and to provide easy end-user access to these data. Although its accomplishments are impressive, it has yet to satisfactorily address a number of important requirements. Chief

among these is the need to distribute the data base over geographically separated computers. Such distribution is being motivated by a number of factors, such as the need to reduce response time for user access to the data base and the need to provide local or autonomous control over parts of the data base. Such distribution is, at the same time, being enabled by the continuing reduction in the cost of computer hardware, starting with mini-computers some ten years ago and continuing today with microprocessors, which promise to place substantial processing capability in the hands of individual users.

A number of challenging problems remain to be solved in distributing data bases. These include the maintenance of replicated data, which most distributed data schemes entail; the linking of different DBMS, having different data models and languages, into cooperative networks; and the provision of essentially continuous system availability, so that the end user comes to rely on his data base system in the same way that he relies on his telephone and utility services.

The solution of these problems promises to make the next twenty-five years of data base technology as eventful and stimulating as the past twenty-five years have been.

Acknowledgment

The author wishes to thank W. F. King for his help with early versions of this paper and for information on System R.

References

1. J. H. Bryant and P. Semple, Jr., "GIS and File Management," *Proceedings of the ACM National Conference*, Association for Computing Machinery, New York, 1966, pp. 97–107.
2. *Generalized Information System Virtual Storage (GIS/VS) General Information Manual*, Order No. GH20-9035, available through IBM branch offices.
3. E. W. Franks, "A Data Management System for Time-Shared File Processing Using a Cross-Index File and Self-Defining Entries," *Proc. Spring Joint Computer Conference* (AFIPS), AFIPS Press, Montvale, NJ, 1966, pp. 79–86.
4. J. A. Postley, "The MARK IV System," *Datamation* **14,** 28–30 (1968).
5. W. C. McGee, "The Information Management System IMS/VS," *IBM Syst. J.* **16,** 84–168 (1977).
6. *IMS/VS Version 1 General Information Manual*, Order No. GH20-1260, available through IBM branch offices.
7. *System/360 Data Base Organization and Maintenance Processor Application Development Manual*, Order No. GH20-0771, available through IBM branch offices.
8. D. Kroenke, *Database Processing*, Science Research Associates, Inc., 1977, pp. 280–293.
9. C. W. Bachman and S. B. Williams, "A General Purpose Programming System for Random Access Memories," *Proc. Fall Joint Computer Conference* (AFIPS) **26,** AFIPS Press, Montvale, NJ, 1964, pp. 411–422.
10. CODASYL Data Base Task Group, *April 1971 Report*, Association for Computing Machinery, New York.
11. CODASYL Programming Language Committee, *CODASYL COBOL Journal of Development*, Department of Supply and Services, Government of Canada, Technical Services Branch, Ottawa, Ontario, Canada.
12. R. F. Schubert, "Basic Concepts in Data Base Management Systems," *Datamation* **18,** 42–47 (1972).
13. E. J. Emerson, "DMS 1100 User Experience," *Database Management Systems*, D. A. Jardine, Ed., North-Holland Publishing Company, Amsterdam, 1974, pp. 35–46.
14. C. T. Davies, "A Logical Concept for Control and Management of Data," *Technical Report AR-0803-00*, IBM Laboratory, Poughkeepsie, New York, 1967.
15. N. Raver, "File Organization in Management Information Control Systems," Selected Papers from File 68: Occasional Publication No. 3, Swets and Zeitlinger, Amsterdam, 1968.
16. H. S. Meltzer, "Data Base Concepts and Architecture for Data Base Systems," IBM Report to SHARE Information Systems Research Project, August 20, 1969.
17. R. W. Engles, "A Tutorial on Data Base Organization," *Annual Review in Automatic Programming* **7,** Pergamon Press, Inc., Elmsford, NY, 1972, pp. 1–64.
18. E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun. ACM* **13,** 377–387 (1970).
19. E. F. Codd, "Relational Completeness of Data Base Sublanguages," *Courant Computer Science Symposia, Vol. 6: Data Base Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1971.
20. W. Kent, "A Primer of Normal Forms," *Technical Report TR.02.600*, IBM Laboratory, San Jose, CA, December 1973.
21. R. F. Boyce, "Fourth Normal Form and its Associated Decomposition Algorithm," *IBM Tech. Disclosure Bull.* **16,** 360–361 (1973).
22. E. F. Codd, "Recent Investigations in Relational Data Base Systems," *Proceedings IFIP Congress 74*, North-Holland Publishing Company, Amsterdam, 1974, pp. 1017–1021.
23. R. Fagin, "Multivalued Dependencies and a New Normal Form for Relational Data Bases," *ACM Trans. Database Syst.* **2,** 262–278 (1977).
24. R. Fagin, "Normal Forms and Relational Database Operators," *Proceedings of the 1979 ACM SIGMOD International Conference on the Management of Data*, Association for Computing Machinery, New York, 1979, pp. 153–160.
25. M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System R: Relational Approach to Database Management," *ACM Trans. Database Syst.* **1,** 97–137 (1976).
26. M. W. Blasgen *et al.*, "System R—An Architectural Overview," *IBM Syst. J.* **20,** 41–62 (1981).
27. *SQL/DS General Information Manual*, Order No. GH24-5012, available through IBM branch offices.
28. R. C. Goldstein and A. L. Strnad, "The MacAIMS Data Management System," *Proceedings 1970 ACM SIGFIDET Workshop on Data Description and Access*, Association for Computing Machinery, New York, 1970, pp. 201–229.
29. V. K. M. Whitney, "RDMS: A Relational Data Management System," *Proceedings Fourth International Symposium on Computer and Information Sciences (COINS IV)*, Plenum Press, New York, 1972.
30. M. G. Notley, "The Peterlee IS/I System," *Scientific Centre Report UK-SC 0018*, IBM Scientific Centre, Peterlee, United Kingdom, 1972.
31. G. Held, M. Stonebraker, and E. Wong, "INGRES—A Relational Data Base System," *Proceedings of the National Computer Conference*, AFIPS Press, Montvale, NJ, 1975, pp. 409–416.
32. B. Czarnik, S. Schuster, and D. Tsichritzis, "ZETA: A Relational Data Base Management System," *Proceedings of the ACM Pacific 75 Regional Conference*, Association for Computing Machinery, New York, 1975, pp. 21–25.

33. *Query-by-Example Program Description/Operations Manual*, Order No. SH20-2077, available through IBM branch offices.
34. H. M. Weiss, "The ORACLE Data Base Management System," *Mini-Micro Syst.* **13,** 111-114 (1980).
35. D. C. Tsichritzis and A. Klug, "The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management Systems," *Info. Syst.* **3,** 173-192 (1978).
36. P. Chen, "The Entity-Relationship Model—Toward A Unified View of Data," *ACM Trans. Database Syst.* **1,** 9-36 (1976).
37. *DB/DC Data Dictionary General Information Manual*, Order No. GH20-9104, available through IBM branch offices.
38. R. E. Wagner, "Indexing Design Considerations," *IBM Syst. J.* **12,** 351-367 (1973).
39. R. Bayer and E. M. McCreight, "Organization and Maintenance of Large Ordered Indexes," *Proceedings of the ACM SIGFIDET Workshop on Data Description and Access*, Association for Computing Machinery, New York, 1970, pp. 107-141.
40. W. W. Peterson, "Addressing for Random-Access Storage," *IBM J. Res. Develop.* **1,** 130-146 (1957).
41. V. Y. Lum, P. S. T. Yuen, and M. Dodd, "Key to Address Transform Techniques, A Fundamental Performance Study on Large Existing Formatted Files," *Commun. ACM* **14,** 228-239 (1971).
42. S. P. Ghosh and V. Y. Lum, "Analysis of Collision When Hashing by Division," *Info. Syst.* **1,** 15-22 (1975).
43. D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade, "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control," *IBM J. Res. Develop.* **20,** 560-575 (1976).
44. D. D. Chamberlin, "A Summary of User Experience with the SQL Data Sublanguage," *Proceedings of the International Conference on Data Bases*, British Computer Society and University of Aberdeen, Aberdeen, Scotland, 1980, pp. 181-203.
45. E. F. Codd, "A Data Base Sublanguage Founded on the Relational Calculus," *Proceedings of the ACM SIGFIDET Workshop on Data Description, Access, and Control*, Association for Computing Machinery, New York, 1971.
46. D. Bjorner, E. F. Codd, K. L. Deckert, and I. L. Traiger, "The GAMMA-0 *n*-ary Relational Data Base Interface Specification of Objects and Operations," *Research Report RJ1200*, IBM Research Division, San Jose, CA, 1973.
47. R. F. Boyce, D. D. Chamberlin, W. F. King, and M. M. Hammer, "Specifying Queries as Relational Expressions: SQUARE," *Data Base Management*, J. W. Klimbie and K. L. Koffeman, Eds., North-Holland Publishing Company, Amsterdam, 1974, pp. 169-177.
48. R. F. Boyce, D. D. Chamberlin, W. F. King, and M. M. Hammer, "Specifying Queries as Relational Expressions: the SQUARE Data Sublanguage," *Commun. ACM* **18,** 621-628 (1975).
49. D. D. Chamberlin and R. F. Boyce, "SEQUEL—A Structured English Query Language," *Proceedings of the ACM SIGFIDET Workshop on Data Description, Access, and Control*, Association for Computing Machinery, New York, 1974, pp. 249-264.
50. R. A. Lorie and B. W. Wade, "The Compilation of a High Level Data Language," *Research Report RJ2598*, IBM Research Division, San Jose, CA, 1979.
51. M. W. Blasgen and K. P. Eswaran, "Storage and Access in Relational Data Bases," *IBM Syst. J.* **16,** 363-377 (1977).
52. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of the ACM SIGMOD International Conference*, Association for Computing Machinery, New York, 1979, pp. 23-34.
53. Raymond A. Lorie and Jorgen F. Nilsson, "An Access Specification Language for a Relational Data Base System," *IBM J. Res. Develop.* **23,** 286-298 (1979).
54. W. C. McGee, "Generalization: Key to Successful Electronic Data Processing," *J. ACM* **6,** 1-23 (1959).
55. *SHARE 7090 9PAC, Part I: Introduction and General Principles*, Order No. J28-6166, IBM 7090 Programming Systems, Systems Reference Library, 1961.
56. J. P. Fry and E. H. Sibley, "Evolution of Data-Base Management Systems," *ACM Computing Surv.* **8,** 7-42 (1976).
57. H. Leslie, "The Report Program Generator," *Datamation* **13,** 26-28 (1967).
58. *Introduction to RPG II*, Order No. GC21-7514, available through IBM branch offices.
59. J. H. Bryant, "AIDS Experience in Managing Data-Base Operation," *Proceedings of the Symposium on Development and Management of a Computer-Centered Data Base*, System Development Corporation, Santa Monica, CA, 1964, pp. 36-42.
60. Naval Command Systems Support Activity, "User's Manual for NAVCOSSACT Information Processing System Phase I," *NAVCOSSACT Document No. 90S003A, CM-51*, IBM Federal Systems Division, Bethesda, MD, July 1963.
61. *Intelligence Data Processing System Formatted File System*, U.S. Navy Fleet Intelligence Center and IBM Federal Systems Division, Bethesda, MD, May 1963.
62. *NMCS Information Processing System 360 Formatted File System (NIPS FFS)*, National Military Command System Support Center, CSM VM 15-74, IBM Federal Systems Division, Bethesda, MD, October 1974. (Nine volumes.)
63. M. M. Zloof, "Query by Example," *Proceedings of the National Computer Conference* (AFIPS) **44,** AFIPS Press, Montvale, NJ, 1975, pp. 431-437.
64. M. M. Zloof, "Query-by-Example: A Data Base Language," *IBM Syst. J.* **16,** 324-343 (1977).
65. J. C. Thomas and J. P. Gould, "A Psychological Study of Query by Example," *Proceedings of the National Computer Conference* (AFIPS) **44,** AFIPS Press, Montvale, NJ, 1975, pp. 439-445.
66. K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "On the Notions of Consistency and Predicate Locks in a Data Base System," *Commun. ACM* **19,** 624-633 (1976).
67. D. D. Chamberlin, J. N. Gray, and I. L. Traiger, "Views, Authorization, and Locking in a Relational Data Base System," *Proceedings of the National Computer Conference* (AFIPS) **44,** AFIPS Press, Montvale, NJ, 1975, pp. 425-430.
68. P. Griffiths and B. W. Wade, "An Authorization Mechanism for a Relational Database System," *ACM Trans. Database Syst.* **1,** 242-255 (1976).

Received December 23, 1980; revised March 16, 1981

The author is located at IBM Data Processing Division laboratory, 555 Bailey Avenue, San Jose, California 95150.