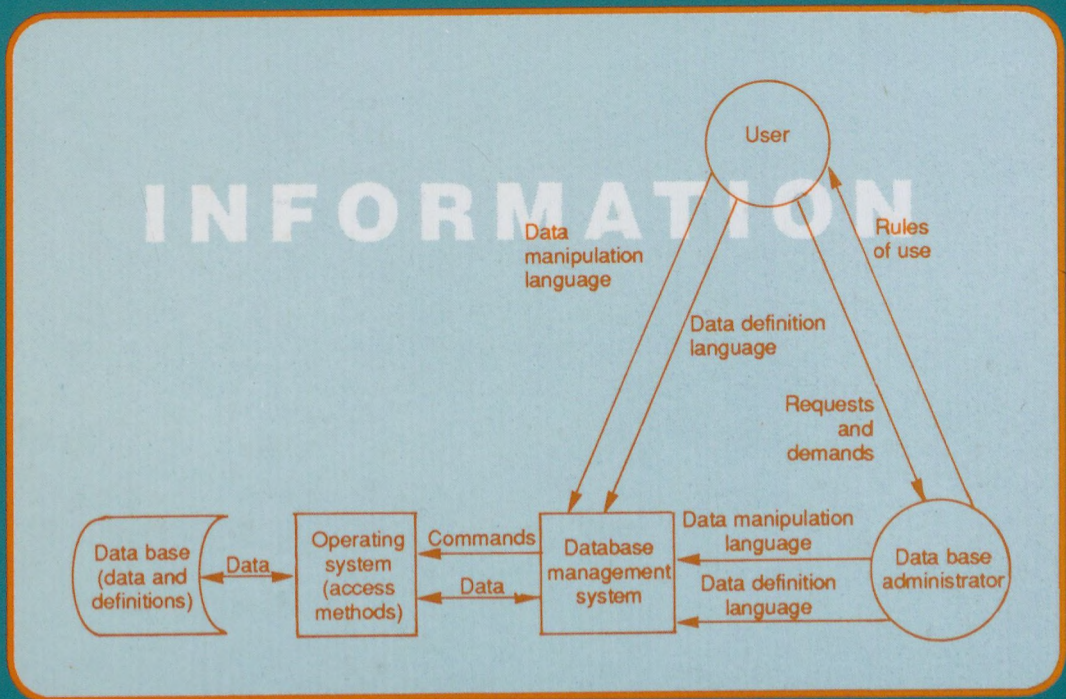


PÁL QUITTNER

A PRACTICAL APPROACH TO DATABASE SYSTEMS



AKADÉMIAI KIADÓ, BUDAPEST

A PRACTICAL APPROACH TO DATABASE SYSTEMS

A PRACTICAL APPROACH TO DATABASE SYSTEMS



NTAK

0 6467 65780 7



AKADÉMIAI KIADÓ, BUDAPEST

PÁL QUITTNER

A PRACTICAL APPROACH TO DATABASE SYSTEMS

INTRODUCTION

1. Information Systems and Data Bases

1. Information Systems	17
1.1. Information Infrastructure	17
1.2. The Goal of Information Systems	18
1.3. Components of the Information System	19
1.3.1. Data	19
1.3.2. Hardware	20
1.3.3. Software	20
1.3.4. Users	24
1.3.5. Information Center	26

2. Data Bases

2.1. Effective Service of Diverse Demands	29
2.2. Data Independence	30
2.3. The Three Level Data Base Architecture	31
2.3.1. The External Level	33
2.3.2. The Conceptual Level	33
2.3.3. The Internal Level	33
2.4. The Components of the Data Base Management	34
2.4.1. Data Definition Language	34
2.4.2. Data Manipulation Language	35
2.4.3. Data Control Language	36
2.4.4. Reporting Language	37
2.4.5. Security Language	37
2.4.6. Utility Language	38
2.4.7. Data Base Administration	38
2.4.8. Data Base Design	39
2.4.9. Data Base Maintenance	40
2.4.10. Data Base Performance	41
2.4.11. Data Base Security	42
2.4.12. Data Base Backup and Recovery	43
2.4.13. Data Base Migration	44
2.4.14. Data Base Interfacing	45
2.4.15. Data Base Integration	46

MTAK



AKADÉMIAI KIADÓ, BUDAPEST

696659

This book is the English version of "Adatbázis-kezelés a gyakorlatban"
published by Akadémiai Kiadó, Budapest, 1993

Translated by I. Horváth and B. Rosenberg

A PRACTICAL APPROACH TO DATABASE SYSTEMS

HUNGARIAN
ACADEMIC
LIBRARY

ISBN 963 05 6636 2

© P. Quittner, 1994

© English translation I. Horváth, B. Rosenberg, 1994

All rights reserved. No part of this book may be reproduced by any means,
or transmitted, or translated into machine language without
the written permission of the publisher.

Printed in Hungary

M. TUD. AKADÉMIA KÖNYVTÁRA

Könyvteltár 1478.../19 94... sz.

A PRACTICAL APPROACH TO DATABASE SYSTEMS

INTRODUCTION

I. Information Systems and Data Bases

1. Information Systems	17
1.1. Information Infrastructure	17
1.2. The Goal of Information Systems	18
1.3. Components of the Information System	19
1.3.1. Data	19
1.3.2. Hardware	20
1.3.3. Software	20
1.3.4. Users	24
1.3.5. Information Center	26
2. Data Bases	29
2.1. Effective Service of Different Demands	30
2.2. Data Independence	30
2.3. The Three Level Data Base Architecture	31
2.3.1. The External Level	33
2.3.2. The Conceptual Level	33
2.3.3. The Internal Level	33
2.4. The Components of the Data Base Management System	34
2.4.1. Data Definition Language	34
2.4.2. Data Manipulation Language	35
2.4.3. Interface to Programming Languages	36
2.4.4. Report Generator	37
2.4.5. Screen Formatter	37
2.4.6. Utility Programs	38
2.4.7. Data Dictionary (Catalogue)	38
2.4.8. Data Security	39
2.5. Benefits of the Integrated Data Base	39
2.6. Data Base Administrator	40
2.7. Some Disadvantages	44
2.8. Data Base Management Systems of Personal Computers	45
2.9. Distributed Data Bases	46

2.9.1. Distributed Systems	46
2.9.2. Features of Distributed Data Bases	51
2.9.3. Advantages and Problems	52
2.9.4. Data Distribution	54
2.9.5. Ensuring Consistency	54
II. Data models	
3. General Description	59
3.1. Classifying Data Models	59
3.2. Main Guidelines of Model Development	60
3.3. Types of Relationships	62
4. Physical Storage Structure and Access Path	65
4.1. General Considerations	65
4.2. Magnetic Disks	67
4.2.1. Physical Features	67
4.2.2. Access Time	68
4.3. Page Structure	69
4.4. Access Methods	73
4.4.1. Addressing	73
4.4.2. Basic Access Paths	73
4.5. Serial Access	74
4.6. Sequential Organization and Access	75
4.6.1. List Structure	76
4.6.2. Comparison of List and Physically Serial Structures	80
4.7. Indexing	81
4.7.1. Example of Indices	81
4.7.2. Indices	83
4.7.3. Duplicate Indices	84
4.7.4. Index Table Search	88
4.7.5. B^+ Tree	93
4.7.6. Multiple Key Search	96
4.8. Hashing	98
4.9. Data Condensing and Encoding	99
4.9.1. Data Condensing Methods	99
4.9.2. Encoding Algorithms	101
4.10. Sorting	101
4.10.1. Automatic Sorting	102
4.10.2. Sorting Methods	103
5. The Hierarchical Model	109
5.1. Hierarchical Systems	109
5.2. Tree Structure	109
5.3. Realization	114
5.3.1. Segments	115
5.3.2. Description of the Hierarchical Data Base	115
5.3.3. Hierarchical Data Base Operations	116
5.4. An Example of the Hierarchical Data Base	117
6. The Network Model	123
6.1. Network Data Structure	123
6.2. Connecting Records	125

6.3. The DBTG Proposal	127
6.4. Schema Definition	128
6.5. The Set Conception	131
6.6. Subschema Description	134
6.7. Data Manipulation Language	135
6.8. Efficiency	140
 7. The Relational Model	 143
7.1. Introduction	143
7.2. Features of Relations	143
7.3. Tables, Views	147
7.4. Keys	148
7.4.1. Data Base Integrity	148
7.4.2. Primary Key	150
7.4.3. Foreign Keys	151
7.4.4. Indices	156
7.5. Relational Operations	156
7.5.1. The Purpose of Operations	156
7.5.2. Relational Algebra and Calculus	157
7.5.3. Relational Operations	157
7.6. Definitions of the Most Important Operations	158
7.6.1. Renaming	158
7.6.2. Restriction	158
7.6.3. Projection	159
7.6.4. Cross Product	160
7.6.5. Union	161
7.6.6. Intersection and Difference	163
7.6.7. Join	164
7.6.8. Join with Default Value (Outer Join)	166
7.6.9. Extend	166
7.6.10. Group Operation Forming (Summarize)	167
7.6.11. Assignment	168
7.7. Examples of Relational Operations	168
7.7.1. Example of Restriction and Projection	168
7.7.2. Example of Join, Restriction, and Projection	168
7.7.3. Grouping and Group Function Example	170
7.8. Optimizing the Access Path	172
7.9. Normalization	175
7.9.1. Concept of Normalization	175
7.9.2. Dependencies	176
7.9.3. First Normal Form (1NF)	178
7.9.4. Second Normal Form (2NF)	180
7.9.5. Third Normal Form (3NF)	184
7.9.6. Post-Normalization Problems, Higher Normalization Forms	188
7.9.7. Defining Views from the Normalized Relations	190
7.10. Catalogue / Data Dictionary	192
7.10.1. The Purpose of the Catalogue	192
7.10.2. Catalogue Tables	193
7.10.3. Object-Oriented Catalogues	193
7.10.4. Catalogue about Access Authorities	194
7.10.5. Dependency Relations	195

7.10.6. General System Information	196
7.10.7. Catalogue in Distributed Data Bases	196
7.11. Summary of the Relational Mode	198
8. Text retrieval systems	201

III. The SQL language

9. General Introduction to the SQL Language	209
9.1. SQL as a Standard	210
9.2. Syntax Notation	210
9.2.1. Syntax Notation	211
9.2.2. Interpretation of the Diagram	213
9.3. Identification of SQL Commands	215
10. Data Definition Commands	215
10.1. General Characteristics	216
10.2. Table Creation (CREATE TABLE Command)	220
10.3. View Creation (CREATE VIEW Command)	220
10.3.1. View Definition	221
10.3.2. Examples of View Creation	222
10.3.3. Restrictions	223
10.4. Index Creation (CREATE INDEX Command)	225
10.5. Dropping of Table, View, and Index (DROP Command)	226
10.6. Table Modification (ALTER TABLE Command)	227
11. Data Manipulation Commands	227
11.1. Introduction	229
11.2. Selection (SELECT Command)	229
11.2.1. The Operation of the Full Select and Subselect	229
11.2.2. The Basic Form of the Select Command (Subselect)	233
11.2.3. The Full Select	234
11.2.4. Examples of Selections	236
11.2.5. SQL Functions	238
11.2.6. Further Selection Conditions	242
11.2.7. Remarks to the Selection Conditions	243
11.3. Alteration Commands	243
11.3.1. The Common Features of the Alteration Commands	244
11.3.2. Deletion (DELETE Command)	246
11.3.3. Data Entry (INSERT Command)	247
11.3.4. Modification (UPDATE Command)	249
12. Control Commands	249
12.1. Introduction	249
12.2. Granting Authority and Privileges (GRANT Command)	251
12.3. Revoking Authority (REVOKE Command)	251
12.4. Temporary Locking of Objects	251
12.4.1. Lock Handling	252
12.4.2. Deadlock	253
12.5. Alteration Confirmation (COMMIT Command)	254
12.6. Restoration of the Pre-Command State (ROLLBACK Command)	254

13. Further SQL Commands	255
13.1. Introduction	255
13.2. Operator Commands	255
13.3. The Storage and Execution of Stored Commands	256
13.4. Changing the Format of the Results (FORMAT Command)	257
13.5. Customizing the Operating Environment (SET Command)	258
13.6. Optimal Access Path	258
13.6.1. Description of the Generated Access Paths (EXPLAIN Command)	259
13.6.2. Some Remarks about Efficiency	260
13.6.3. The Automatic Use of Indices	261
13.7. Other Commands	262
14. Embedding SQL Commands in Programs	265
14.1. The Creation of SQL Programs	265
14.2. SQL Command Types	266
14.3. Recognizing SQL Commands	268
14.4. Host Variables	269
14.5. Communication Area (SQLCA)	271
14.6. Handling Particular Events (WHENEVER Command)	272
14.7. Static SQL Commands	274
14.7.1. General Features	274
14.7.2. Selecting a Single Row (SELECT INTO Command)	275
14.7.3. Cursor Use	276
14.7.4. Modification based on Cursor Position	278
14.8. Dynamic SQL Commands in the Program	279
14.8.1. The Features of Dynamic Commands	279
14.8.2. Immediate Execution of Dynamic Commands (EXECUTE IMMEDIATE Command)	280
14.8.3. Preparation and Execution of Dynamic Commands (PREPARE and EXECUTE Command)	281
14.9. Authority Control, Access Path	285

Appendix

A) For Further Reading	287
B) Index	289

Introduction

The widespread use of computers has already had a significant influence on our everyday activity. Their application has become common in every field of economy, science and technical domain. The information revolution has changed our life just like the industrial revolution did two centuries ago.

The impact of computer data bases and information systems (in everyday language called data banks) on the spreading of human knowledge and information can be compared to that of printing. As carefully written, rare codices with hand drawings, attainable only by a few people, were replaced by printed books and newspapers which can be cheaply produced in large quantities and within a short time, and are bought widely, so the computer information systems have replaced the cardboard-folios selected by hand. (It is a special point of interest that computing technique has completely altered printing technology as well in the last decade.)

The proliferation of PC's and the connection of computers with different sizes and capacities to telecommunication networks makes it possible to store and process all the data necessary for even housekeeping and also to build standardized information systems for worldwide bank networks, furthermore to provide data and information quickly and in a reliable way for those concerned and authorized, regardless on which point of the globe they are.

Some type of data base management system has been written and sold for nearly every serious computer. Their prices vary between less than one hundred dollars covering programs for simple data management on PCs and several hundred thousand dollars providing software suitable for managing complex information systems, performing real data base management functions and which can be run on different mainframes and under various operating systems. In many places, office prestige requires that a firm must have its own 'data base', even when it is no more than a simple record-keeping file system. Whereas at other institutions it is hardly understandable why the building of a complex database satisfying many demands simultaneously takes such a long time and why are such expensive softwares necessary for their operation.

This book presents the characteristics of modern data bases and data base management systems. How data must be stored, what relationships can be created between them, and what software methods are available to store data, keep

them up-to-date, and retrieve them quickly, and at the same time ensure their security, accuracy, integrity, and unanimity.

The theme is discussed above all from a practical point of view, from the perspectives of the designers, operators, and users of the data base. The description of the data bases and the examination of the efficiency of various access methods tries to be lucid in order to foster a clear understanding of the methods and their application in practice. The implementation of the relational model is discussed in detail. Several practical problems are pointed out and methods for their solution are proposed. Many of these are often absent or are mentioned at most in passing in theoretical books dealing with this subject. Yet at the same time in everyday life the user often finds himself face to face with these problems.

Consequently, in contrast with manuals used mainly as reference books, quotation of literature in the text is missing. Methods expounded in the book are selected and summarized based on the practical experience of the author and an analytical summary of the literature.

Naturally, it is not presumed that by studying one book the reader can learn all significant aspects of this complex subject to the necessary degree. Thus, at the end of the book some publications are listed which include further useful information for the reader who wants to dig deeper into the entire topic or particular details of it.

The data bases of the Nineties, can be divided into three main groups:

- data base implemented on a PC, satisfying information requirements of one person or a group. Only *one user* can access it at a time;
- data base installed on a mainframe or a PC network, with an adequate access protocol. It is *centralized* and integrated, and able to satisfy requests of individuals and groups simultaneously and independently of each other;
- integrated, but *distributed* database based on a network of mainframes and/or PCs. The locally required data are at the place of utilization, where they can be accessed from any part of the network. The storage of commonly-used data and the maintenance of the data dictionary indicating in which data base the data can be found is centralized, or at least under central supervision.

Examples and methods dealt with in the book concern mainly databases belonging to the second category. This can be partly explained by the development of these processes in the past decade; what is more, they can be regarded already as standard to some extent. A further reason is that the solutions given here can be made use of with the other two types as well. In single-user systems the tasks are easier since the data structure and data relationships are generally simpler and the problems related to simultaneous access can be ignored. With distributed data bases, it must be kept in mind that a data communication network is also utilized for locating and gathering information. However, the network is kept hidden from data base users by the data communication software. For that reason, these differences regarding the data base management do not influence the substance of our tasks.

The main stress is laid on the design, creation and operation of data bases in which:

- there are many (hundreds) records of different types,

- numerous kinds of relationships exist between the different types of records,
- the total number of records can range from millions to hundreds of millions,
- thousands of operations (retrieval, modification) are executed daily on the data base,
- data can be accessed simultaneously and used interactively from hundreds of terminals, and
- data protection, security, and recovery are ensured by the system.

Naturally, users having fewer demands and working with databases of simpler structure and smaller in size than the above, will also find the book useful. They might encounter certain problems in a less pronounced way or later. However, it makes no difference what kind of machine or software is used, the general rule is that below a certain data and/or transaction volume, efficiency need not be a concern. Above a given threshold it is advisable to start thinking about it; while at a certain defined level it must unfailingly be taken into account if an operation-capable system is to be created.

Whether these levels are at a thousand or a hundred thousand records, at one or a hundred transactions per minute will depend on the hardware available, the operating system used, and not in the least measure on the database management software and the efficiency of the database management system.

INFORMATION SYSTEMS AND DATA BASES

1.1. Information Infrastructure

Information systems are used not only to store necessary data. They also have to guarantee that the data will be available to the user at the right time, in a suitable form, and at the required number or speed, as well as ensure that newly-obtained data will be entered into and spread by the system for the immediate use of those for whom the information is relevant. For this reason, an information infrastructure became necessary. Its basic features were developed in the 1960s. The infrastructure consisted of the following parts:

1. Data processing, storage and distribution

This has been an old idea in the machine age. Typical early applications were payroll accounting, airline seat reservation and check processing. Since the 1960s, it has become a necessary information for management decision-making.

2. Information networks

It is used to send information to a different, distant location. At first, the phone and telex, and later the fax, operated independently of the computer. (At best, they transmitted data supplied by computers which were used by other computers.) Later these pieces of equipment were integrated with the computer, and today they enable direct man-machine or machine-machine connection, in principle from any distance, however great. For the machines of a common network and for the users connected to them, it does not matter where data originates, is stored, or is sent.

3. Central and peripheral Automated Office Equipment

Mail systems, electronic readers, typewriters, word processors, and copy machines belong to this category. Most of these machines work independently (just as most use of the telephone is independent, strictly speaking, from the information system). However, a significant part of their use is part of a narrowly or widely defined information system, too. The word processor, a computer used as a typewriter, can function in other ways, e.g. query a data base or send messages to other users via an electronic mail system.

Information Systems

1.1. Information Infrastructure

Information systems are used not only to store necessary data. They also have to guarantee that the data will be available to the user at the right time, in a suitable form, and in the required number of copies, as well as ensure that newly-obtained data will be entered into and stored by the system for the immediate use of those for whom the information is relevant. For this reason, an information infrastructure became necessary. Its basic features were developed in the 1960's. The infrastructure consisted of the following parts:

1. *Data Processing, Management Information*

This job was handled by the computer center. Typical early applications were payroll, accounting, airplane seat reservation, and check processing. Since the Seventies they have included supplying information for management decision-making.

2. *Telecommunication*

It is used to send information to a different, distant location. At first, the phone and telex, and later the fax operated independently of the computer. (At best, they transmitted data supplied by computers which were used by other computers.) Later these pieces of equipment were integrated with the computer, and today they enable direct man-machine or machine-machine connection, in principle from any distance, however great. For the machines of a common network and for the users connected to them, it does not matter where data originates, is stored, or is sent.

3. *Central and Individual Automated Office Equipment*

Mail sorters, microfilm readers, typewriters, word processors, and copy machines belong to this category. Most of these machines work independently (just as most use of the telephone is independent, strictly speaking, from the information system). However, a significant part of their use is part of a narrowly or widely-defined information system, too. The word processor, a computer used as a typewriter, can function in other ways, e.g., query a data base or send messages to other users via an electronic mail system.

Nowadays, the independence of the above-mentioned machines continues, but within the information system their use has become so interwoven that together they form the basis for the information infrastructure.

Modern information systems combine traditional transaction-oriented data processing, decision support, telecommunication, and office automation. Of course, this does not mean that the same software can be used to write a text and update data in the central data base, or that software which is processing data can also direct communication between shared data bases. The above-mentioned tasks of a unified information system can be solved by different specialists using specialized equipment and methods suitable for solving particular problems.

This book deals with the first point of the infrastructure: production of information, data needed for management decision-making, data bases, and data base management systems. Other subjects will be touched upon only insofar as needed for understanding the aforementioned topics.

1.2. The Goal of Information Systems

The 'information systems' of the 1960's were mainly enormous 'paper mills' handling transaction-oriented processing of files. They produced accounts and lists on a large scale. One portion of the output was successfully used, for example, to prepare payrolls, invoices, accounts, or shipping orders. But the other part typically went into the waste paper basket — the only question being at which management level did it happen. According to some malicious observers, the paper shredder should have been placed next to the computer printer. Of course, this is an overstatement. Yet it is a fact that a significant share of the computer systems of those days poured out far more data than practically could be used. The efficiency of the system was measured by the throughput of the computer, the number of transactions processed daily, or the response time which is the actual time needed from start to finish to achieve the result. Programmers were evaluated based upon the number of correct program instructions they wrote daily. This way of thinking changed in the 1970's when management information systems became commonplace. Their goals were to supply information to management for decision-making and to call attention to exceptional or extraordinary cases, i.e., situations which need special intervention. At that time the fashionable slogan was: the right information to the right person at the right time.

This assessment is still valid, but due to a higher appraisal of human effort, it must be interpreted in a wider manner. It can be said that the *goal of the modern information system is to apply information technology in order to increase the productivity of the people in an organization.* In this book, when referring to information technology, the application of computers, software, computer-readable information, and telecommunication methods is meant.

The above-mentioned definition includes the concept that he who possesses current information at the moment of decision-making is more likely to make good decisions or at least better ones (assuming that he has all the other knowl-

edge needed for the decision and the power and the means to execute it). It is also understood in the definition that the computer provides all the necessary details. For example, a delivery list contains exactly what to pack, where to, when, what delivery method, and what route, as well.

1.3. Components of the Information System

Every information system consists of four main components:

- data,
- hardware,
- software, and
- users.

The relationship between them can be summarized thus: *the information system stores data on the hardware, from which the users can get information with the help of the hardware and software.*

1.3.1. Data

The words 'information' and 'data' are often used as synonyms. It often does not make any difference, but in precise computer terminology, it is necessary to distinguish between them.

Information means a clear communication concerning some event which in general contains some new knowledge important to the user. However, in most computer systems information cannot be retrieved directly. Indeed, most information is not available in the desired form. A management information system needs information in different levels of detail and aggregation. The higher the management level needs, the greater demand for an overview of the whole system, and less weight should be given by them to details. Besides these general conditions, informational demands change too, such that it is almost impossible to store answers suitable for every imaginable query. For this very reason, before transmitting the information in final form, the data stored in the computer must be processed to the required form, i.e., related details are drawn together and operations performed on them, etc. In such cases, the actual meaning of the information or what kind of events it describes is irrelevant. Rather the character strings and numeric data including the information are processed.

Data is the concrete form of the information in the computer system. In the processing phase, data, not information, is dealt with. For example, in the case of aggregating data based upon county-level statistics to country level, the same operations with the same type of data are made, regardless of whence they came. As for computer processing, the only important thing is to unambiguously identify the origin of the data, their proper place, and affiliation. For this processing, the meaning of the information need not be understood, rather to know the data which includes the information is enough, for example, that the value of something is 893.

Of course, the same data can be interpreted in various ways, giving different information. The 07/04/1992 date type data means July 4, the national holiday, for an American. However, for an Englishman, it is only the 7th of April. For the standard Hungarian date control procedure this date is invalid. (There is no 19th month and no 92nd day based upon the year, month, day order in use in Hungary.) Similarly, in a mainframe computer using EBCDIC code a byte containing 11010010 could mean -2 or 'K', depending upon whether it represents a signed decimal number or an alphabetic character. For that very reason, to get information from the data, in addition to its value, it must be known not only what it means, but in what form it appears.

These issues and how data base management systems solve them are covered in Chapter 10 dealing with the definition of the data base.

1.3.2. Hardware

Every system is based upon a specific hardware configuration. A part of this is the computer (in case of shared data bases, computers) in which the data are stored. In addition, there are terminals (work stations) and the telecommunications network to transmit data between users and computers. In this book, hardware will be touched upon briefly. Only the structure and operation of magnetic disks used for physical data storage, and the buffer technique which stores data temporarily in a suitable memory location will be reviewed in Sections 4.2 and 4.3. The efficiency of these components can be significantly improved by skillful data organization, access paths, and buffering methods.

1.3.3. Software

The connection between the data physically stored on the computer and the user is realized by software. It consists of three parts:

- operating system,
- data base management system, and
- application software.

The *operating system* controls and monitors the execution of all the programs running on the computer, and controls the peripherals (disk drives, terminals, data communication lines, etc.). For this reason, the *data base management system runs under the supervision of the operating system and communicates with the peripherals via the operating system* to retrieve data from or write data to them. This separation has the advantage that the data base management system need not include the capability of dealing individually with the special features of particular peripherals. It is sufficient to translate the peripherals' input-output commands to the level of operating system commands. This reduces significantly the programming work needed to develop a data base management system. At the same time, it permits wide-ranging application of the data base management system because at the level of the operating system the commands can be the same for different devices, central processing units, or operating systems.

The *application software* determines which information should be assembled from which data. In Part Two, this topic will be reviewed briefly and rather theoretically by examining data models. It will be discussed in more detail when the data manipulation language is reviewed in Chapter 11.

In Fig. 1.1, the operating principle of a data base management system is shown by entering data.

1. The application software asks for data from the data base. (Several application programs can run simultaneously on the computer, and any one of them can turn to the database management system. Their requests are usually put into a queue.)
2. The database management system interprets and analyzes the command. It establishes whether it is possible to execute it. (Is it syntactically correct, do the requested objects exist and are they accessible, and has the user the right to perform the requested operation(s)?)
- 3a. If the result of the analysis is positive, then it transforms the commands related to the data base into commands which can be interpreted by the operating system and passes them towards the operating system. One data base operation usually generates sequences of commands.
- 3b. If the request cannot be fulfilled for any reason, then the database management system sends a message to the user's work and/or communication area, usually marked with the reason for the refusal.
4. The operating system accesses the external storage device. (If unsuccessful, it issues an error message.)

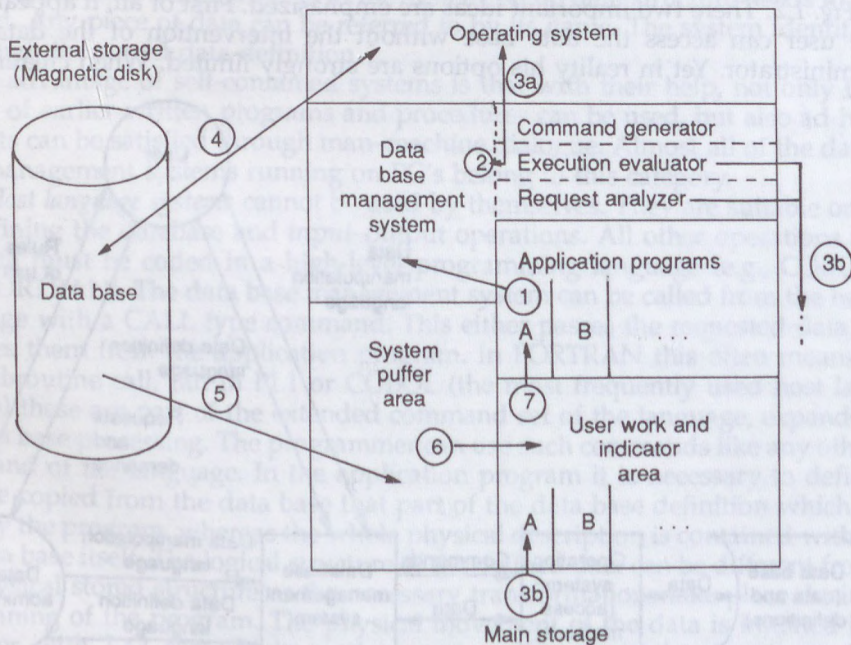


Fig. 1.1. Operating principle of the database management software

5. It brings the requested data into the system buffer area.
6. It passes the data to the application program. It reports the success of the operation, the reason for any (partial) failure, or the observation of any special event.
7. The application program takes over the data (if received) and evaluates the return codes.

Essentially the same process occurs when data are written, existing data are changed or deleted, or any kind of data manipulation occurs.

At Fig. 1.1., for simplicity, it has been assumed that the data manipulation commands of the application programs are in the memory. In practice, this is often not the case. A lot of database management systems work interactively. The user gives commands from a remote terminal while the work is progressing and requests that partial or full results be displayed on his terminal. In this case, the link between the terminal and the application program is handled by the data communication software running either as a separate program, a part of the operating system, or in direct cooperation with the database management system.

The data base can be accessed by the user only through the database management system as this ensures that the latter's data protection measures cannot be bypassed. The rules that have to be observed at a given installation were fixed by the data base administrator when he created the data base or modified its definition. (The role of the data base administrator is described in detail in Section 2.6. For the time being, consider him as responsible for the whole data base and its operation.) The passing of the data to the application program can be seen in Fig. 1.2. There two important ideas are emphasized. First of all, it appears that the user can access the data base without the intervention of the data base administrator. Yet in reality his options are strongly limited. When creating the

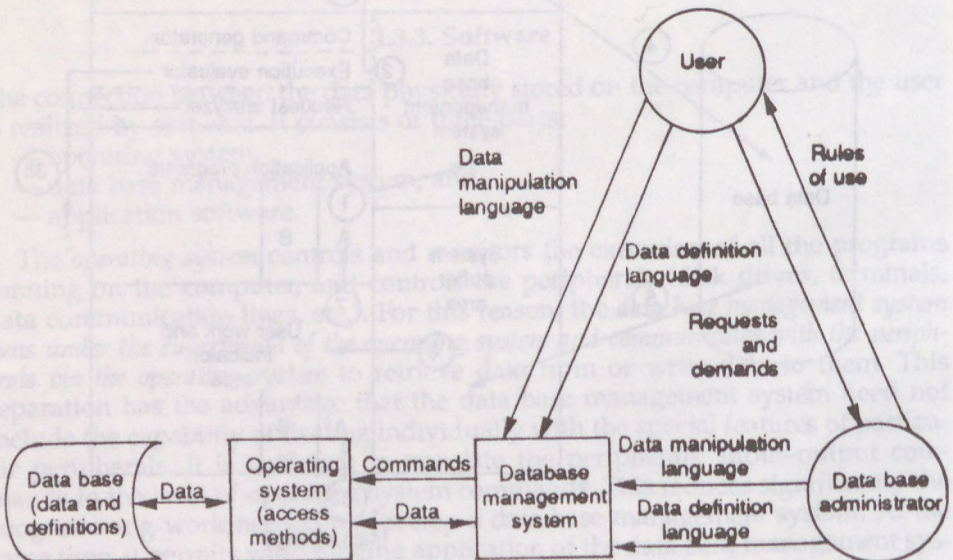


Fig 1.2. Data base access

data base, the data base administrator uses the data definition language (Point 2.4.1.) to design unambiguously those control procedures which hinder illegal 'intervention' or hacking. Naturally, if a user is allowed, for example, to modify data, then the data base administrator typically cannot check whether the new data contain valid values which are meaningful. (However, certain revisions can be controlled, for example, when data must fall between defined limits.) However, he can block access to or modifying of data which are off-limits to the user.

The other significant feature is that *most database management systems rely strongly on the operating system* of the computer. So the physical movement of data between the main memory and external storage is handled by the input/output routines of the operating system. Real access paths are limited to the available access options of the operating system.

Users' groups may have various types of requests. Self-contained and host language systems satisfy these needs in different ways.

Some products operate in both modes. It depends upon the installation and invocation as to which becomes operative.

a) In *self-contained systems*, the data base management system contains within itself the generally-defined routines. The system has its own and easily-learnable language. The solution of the problem has to be described in this language. Such languages are, of course, less rich than high-level programming languages. In addition, they generate programs which require more machine-time. However, their use is more convenient for users who are not programmers. The user can handle the data only at the logical level. The whole description of the data base, including the data field names, is in the system, so no data structure needs to be defined. Any piece of data can be referred to by its name. The system identifies it through the stored data definition.

The advantage of self-contained systems is that with their help, not only the results of earlier written programs and procedures can be used, but also ad hoc requests can be satisfied through man-machine dialogue. Almost all of the database management systems running on PC's belong to this category.

b) *Host language systems* cannot be used by themselves. They are suitable only for defining the database and input-output operations. All other operations on the data must be coded in a high-level programming language (e.g., COBOL, PL1, FORTRAN). The data base management system can be called from the host language with a CALL type command. This either passes the requested data to or takes them from the application program. In FORTRAN this often means a real subroutine call, but in PL1 or COBOL (the most frequently used host languages) these are part of the extended command set of the language, expanded for data base processing. The programmer can use such commands like any other command of the language. In the application program it is necessary to define or have copied from the data base that part of the data base definition which is used by the program, whereas the whole physical description is contained within the data base itself. The logical structure of the defined data can be different from the physical stored structure as the necessary transformations take place during the running of the program. The physical movement of the data is initiated by the user with I-O commands in the program. However, they are executed independently of him by the operating and the database management systems.

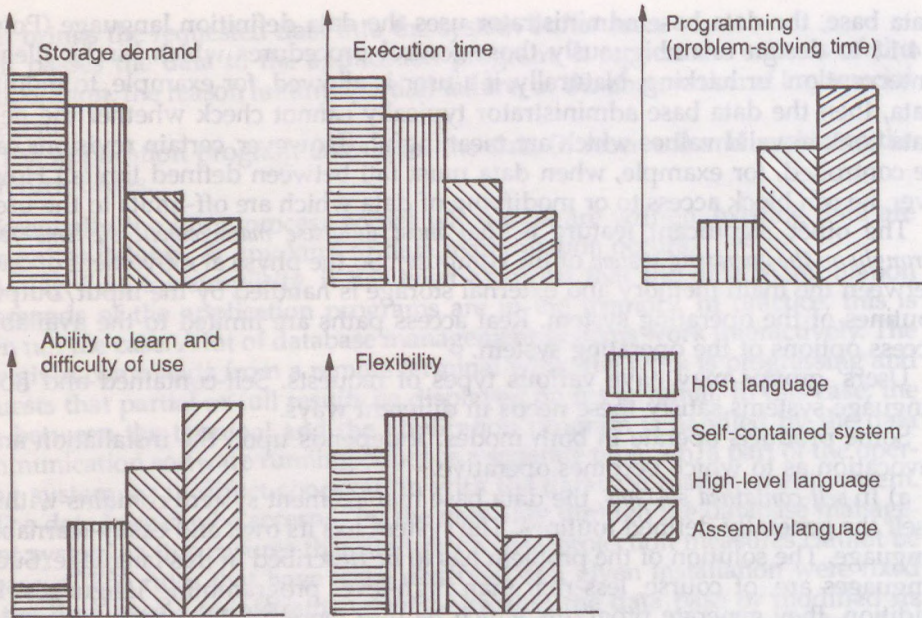


Fig 1.3. Qualitative comparison, based on different aspects of self-contained and host language database management systems with high-level and assembly-level program languages (qualitative description)

In Fig. 1.3., host language and self-contained systems are compared with assembly and high-level programming languages according to different aspects. (The figure serves only for qualitative information, e.g., fast, medium, slow.)

1.3.4. Users

A data base is too serious and important a matter to be left fully to the users, programmers, and systems analysts. The elaboration of the necessary conditions for effective processing, the systematic saving of data, back tracking and reloading in case of error, the planning and modifying of the data model, and its reorganization on demand cannot be entrusted solely to the whim of the users. This job is handled by the data base administrator. He guarantees that the data base and its related features can be used, if possible, easily, by others, too. The responsibilities of the data base administrator will be discussed in greater detail in Section 2.6.

Based on the purpose of the use, two major user groups can be distinguished:

- administrators, and
- decision-makers.

The following subgroups can be identified inside both groups, depending upon the method of use:

- application programmers,

- non-programmer users, and
- menu-oriented users.

The differences between the groups and subgroups are not sharp. The same person, depending upon the task, uses the data base at one time in one way, but at another time in an alternate way.

Members of the first group, *administrators*, typically deal with *transactions on a large scale*. The importance of any one transaction is trifling. (Of course, for example, the remittance of a large amount is an exception.) The transaction is entered based upon a *well-defined procedure*. The efficiency of the system can be measured, more or less, by the number of transactions that can be processed in a given time, compared to the computer resources used for that.

The characteristic of the second group, *decision-makers*, is that they deal with far fewer transactions. These can be executed and evaluated in different ways. As a result of the information obtained from the data base, decisions can be made. Their efficacy stems from the degree to which these decisions advance the operation and development of the organization. Certainly this can be defined far less exactly than for the first group.

A typical example of these two processes is when administrative processing involves several years of historical data, such as a loan applicant's financial transactions in his bank account, his prior credit applications and loan repayments, and other financial information. The bank officer authorized to make credit decisions analyses the above information and takes into account the balance sheet of the bank. Based on these factors, the loan officer decides whether the applicant should be granted a loan and, if so, under what conditions.

By looking at the application method of the data base, as earlier mentioned, application programmers, non-programmers who describe independent tasks, and menu-oriented users can be distinguished.

Application programmers develop programs for specific applications based upon their own specifications or those requested by the end-user. Programmers use the database management system, first of all, to search for the proper data or to write them to a storage device. Usually they work with a large volume of data which is basically of the same type. The data can be defined by grouping the well-encircled part of the data base into records according to the nature of the task. In programming what counts is not the speed with which programs are developed, but rather the effectiveness of their running. Thus the data base can be manipulated only by a skilled programmer. The layman user can make contact with the system only via earlier-written programs; his activity is limited to the options provided by these programs. His role is essentially the same as that of the parameter users described in the following paragraph.

The majority of users are not programmers. They possess no or minimal programming knowledge. Those users whose requests are ad hoc and their form or content needs to be described according to the current situation, and therefore cannot be programmed in advance, are *non-programmer* users. In contrast, those users whose information requests are limited to well-bounded subjects and questions which can be programmed in advance are *menu-oriented* users. The former need for the handling of their work a programming language suitable for generalized data-handling, easily learnable by an amateur, which can describe, first of

all, simple logical relationships and basic arithmetic rules. With the help of this language, users can define by themselves the required data relationships and operations to be performed on the data. All these are included usually in a self-contained data base management system. Menu-oriented users do not need even this relatively simple syntactic language to prepare their questions. By giving the parameters of some standard, predefined questions, they can obtain the requested information (for example, bank query system, seat reservation system).

With the growth of computer technology and the development of user-friendly software, the number of computer users who are specialists in their own application field has increased. They can describe logically their task and information requirements. They also possess sufficient computer knowledge to formulate them with the help of the database management software into a form directly understandable by the machine. This dramatically enlarges the number of practical uses and potential applications of the database management software.

1.3.5. Information Center

Due to the high start-up and operational costs and the importance of the information and results, tasks such as data collection, storage, and organization, as well as information services are dealt with by an independent, specialized department¹ of the organization. In the 'Classic Age' of data processing, this unit was called the *Computing Center*. Nowadays, when much more stress is placed upon the value of the information supplied (for the organization it is far more valuable than the hardware and software with which the system works), this unit's name reflects this change as well. For simplicity, from the variety of choices the unit handling the computing tasks and information services of the organization will be called the *Information Center*. For reasons of expediency, the director of this unit is directly subordinate to one of the chief executives of the organization.

The Information Center consists of the main departments described below. The size and the work force of these departments can vary markedly, but all of them are necessary for optimal operation.

The Computing Center, today it should rather be called the Information Services Department, has the largest staff and is the most important one for handling daily tasks. Typical data processing tasks are carried out here, including the development of traditional computer programs handling the needs of the entire company. The Computing Center supplies technical services, too, among other things, data base supervision, system software support, connecting computers, and, in general, telecommunication services. Naturally, in large organizations each department can have its own computing center, too. These centers can also provide information. They can also develop and implement computer programs by following the directives of the central computing center.

¹ While the size of the organization determines whether a given function is handled by senior management, a department, a section, or a group, for simplicity, the word 'department' will be used to describe such parts of the organization.

The structure of the Computing Center has changed little in the last 10–15 years. It consists mainly of computer operation, application software development, (also maintenance), a systems software unit responsible for technical matters related to the system software, and a separate unit for telecommunication. The difference is that while formerly these units accounted for the whole Information Services Department, now they have been consolidated into one section of the Information Center.

Daily tasks are currently carried out by a relatively large number of employees at the Computing Center. However, for the long run, a department, generally with a small staff, is indispensable for directing the planning of information resources and their use. They design the organization-wide information systems and assist each department of the organization in adjusting its information plan so that it harmonizes with the globally-established schema. Their main design and coordination tasks are: networks, office automation, end-user applications, data base technology, automation and standardization of system-development procedures, and, of course, the long-range planning of application systems which serve the projected needs of the organization.

A third department is responsible for advising computer users throughout the organization. One section provides advice about general software. This can be the software of the central machine, but it may also be found on local PC systems. A second section offers advice about the development of application tasks. This can be implemented on a centralized basis. (For example, a locally-developed application is transformed into a form that everyone can use.) It can be very effective when the advisors of this section work together with the developers in the end-user departments to solve their problems. A third, independent section instructs end-users how to use existing applications and newly-adopted information technology methods.

The types of data bases can be distinguished. The fact-type database includes data in a well-defined form. Between the entities of this data base there are clear relationships. The data base system produces the information based on the value of the formatted data and on their relationships.

The document-type data base, otherwise called text retrieval system, includes non-formatted information about documents or written texts. The system supplies those records in which a given text pattern occurs in a defined context and form inside the text or the record.

In practice, the term data base without any adjective is used for fact-type data bases. Document-type data bases are indicated by adding this adjective to them. In this book, the former will be reviewed in detail, and only the main features of the latter will be discussed separately in Chapter 8.

It should be pointed out that the difference between the two is not very sharp. Formatted data can be found in text retrieval systems while it is not all fact-type database systems non-formatted. Text retrieval is possible to some degree.

The fact database system must ensure:

- effective service of different requests,
- data independence,
- representation of complex relationships between the data,
- avoidance of redundancy and checking for this.

Data Bases

In everyday language, data base means the collection of data used in a data processing environment. However, this definition is too general. According to it, the combination of the files of a traditional data processing system could be considered as a data base and the programs handling these files could be considered as a database management system. However, a real data base means much more than this. Moreover, the data base includes also their description and the relationships between the different data types. (For example, besides the data of the individual students, marks, subjects and teachers, it includes also the relationships which can help to establish which students attend which classes, who are their teachers, and what subjects they study.) In addition, there must exist a software which manages the data base. This database management system produces the information gathered from the data.

In a wider sense, two types of data bases can be distinguished. The *fact-type* data base includes data in a well-defined form. Between the entities of this data base there are different relationships. The data base system produces the information based on the value of the formatted data and on their relationships.

The *document-type* data base, otherwise called *text retrieval system*, includes non-formatted information about documents of written texts. The system supplies those records in which a given text pattern occurs in a defined context and form inside the text of the record.

In practice, the term data base without any adjective is used for fact-type data bases. Document-type data bases are indicated by adding this adjective to them. In this book, the former will be reviewed in detail, and only the main features of the latter will be discussed separately in Chapter 8.

It should be pointed out that the difference between the two is not very sharp. Formatted data can be found in most text retrieval systems while in almost all fact-type database systems non-formatted text retrieval is possible to some degree.

The (fact) database system must ensure:

- effective service of different requests,
- data independence,
- representation of complex relationships between the data,
- avoidance of redundancy and checking for this,

- ease of use,
- data protection, i.e., non-authorized users cannot gain access,
- data integrity, i.e., to the extent possible, prevent authorized users from damaging the data base,
- recoverability, i.e., in case of any errors, the original condition can be restored,
- in a multi-user environment, concurrent access and security control,
- in shared data bases, in addition to the above, data distribution, the ability to locate them from other locations, multi-access control, and data flow optimization.

These tasks will be analyzed in detail later on.

2.1. Effective Service of Different Demands

Any kind of information service makes sense only when the information obtained is

- exact,
- sufficiently detailed,
- understandable,
- available in time,
- easily accessible,
- not too expensive, and
- those who get it, use it.

To achieve this, it must be ensured that:

- from the data stored in the data base, the information required by authorized users is available to them within the desired time frame,
- the data can be grouped without redundancy (or with the minimal redundancy which was retained consciously) in such a way that their selection and grouping occurs effectively, according to various aspects, and
- the user needs to become acquainted only with the logical structure of the data. He need not be aware of the physical storage structure.

2.2. Data Independence

Since the same data can be accessed by several users and programs, data and programs must be rendered independent as much as possible. If, for example, a record in a program is expanded by a new field due to a change in processing needs, then this change should make necessary only the revision of this single program.

All programs which use the same records or part of them should remain invariant. To achieve this, the physical description of the data must be separated from the data structure used by the program. Professionals dealing with data bases call this *data independence*. In reality, there are two types, physical and logical data independence.

A system is *physically data independent* if the programs using its data or the user's ad hoc information requirements are independent in practice from the storage structure and access method.

Logical data independence means, on the other hand, that changes made to the logical structure of the data base do not influence significantly the programs' use of the data base.

Actual data independence means, based on the above-mentioned interpretation, the collective realization of physical and logical data independence. Accordingly, a system is data independent if the application software working under the database management system, the programs, and the ad hoc requests that use the data stored in the data base are in practice independent from the storage and data access path, as well as the logical structure of the data base.

Naturally, complete data independence cannot be achieved (for this reason, the mitigating phrase 'in practice' is used in the definitions), because, for example, the access method chosen influences the efficiency of the system to a large degree, as will be shown in detail in Chapter 4. (E.g., instead of a pointer chain, an index table is used for retrieval.) At the user level, these changes do not mean that he has to write new programs or formulate his questions in a different way. Rather, the consequence is that he will get the results and the answer sooner, cheaper, or, maybe, due to data base reorganization to satisfy the needs of other users more effectively, he will get them slower and more expensively.

2.3. The Three-Level Database Architecture

The information system describes in a model form the relationships of objects of the real world and the relationship of their characteristics and connections. The data model does not deal with actual data, rather it works with data types and relationships. The first step of problem-solving is always the development of the most suitable data model. This model not only reflects truly the world that it seeks to describe, but on this represented data structure the changes to the reality can be followed also by using available computer tools.

To ensure data independence, the data model is divided into three levels (see Fig. 2.1.):

- *Inner or physical level.* This describes the physical storage structure and the physical access path of the data.
- *External level.* This describes how the data base is seen by individual users.
- *Conceptual level.* This describes the effective appearance of the data base combined into one logical unit. The different projections of this are seen by the users at the external level, and this is mapped unambiguously into the storage and access structure at the inner level.

Data independence is complete if the three levels are totally independent of each other. The entities, their attributes, and their relationships represented in the three-level model will be introduced by a simple example.

The conceptual level includes the following data records (entities and their attributes):

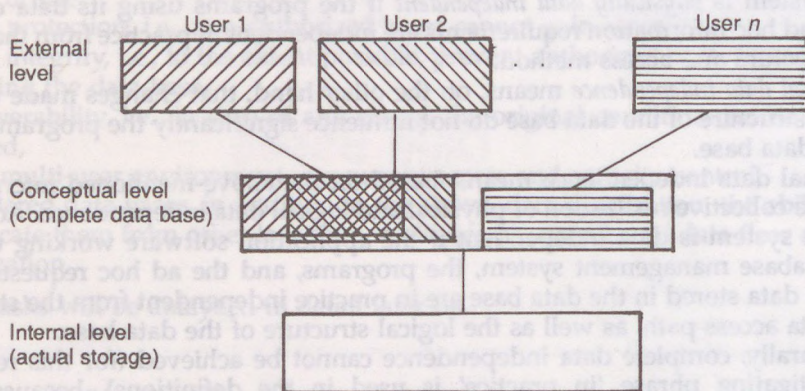


Fig 2.1 Three levels of the database architecture

- Student** (name, 20 characters; the class he attends (the code of the class), 4 characters; his/her place of birth, 20 characters, and his/her date of birth, 8 characters);
- Class** (the code of the class, 4 characters; the name of the homeroom teacher, 20 characters; schedule, 7*12*8 characters).

In addition to this, two restrictions apply at the conceptual level. The first is that only such class code values can be entered in a student record which are included among the class records. The second is that no two identical class codes can occur among the class records. This means that the database management system automatically blocks those modifications or data inputs where these conditions are not satisfied.

The external level for user-1 is:

- Student** (name, 15 characters; class code 4 characters)
(in this case the student's name may be truncated).

The external level for user-2 is:

- Student** (name, 20 characters; homeroom teacher name, 20 characters).

Here the name of the homeroom teacher is supplied automatically by the database management system, because the conceptual level ensures this relationship via the class code.¹

The inner level describes on which disk and within it at which cylinders and tracks (sectors) are stored the records which include student data. For example, they are stored in order by class and inside of this by name order. The record length is 52 characters, of which the first 20 are the name of the student, the following 4 are the class code, the next 20 are the place of birth, and the last

¹ This representation of the relationship, for example, is equivalent to that of the method of the relational model which will be detailed in Chapter 7.

8 are the date in year, month, and day order. Any student record can be retrieved based on the indices built on the student's name.

At the external and conceptual levels, the same data model is used. The internal data description is independent from the data model.

2.3.1. The External Level

The external level is the one at which the users see the data base. From the objects stored at the conceptual level and from their features and relationships, the users define the record types and their attributes at the external level with the help of the data definition language. The records at the external level may be equal to the records at the conceptual level and the internal level too. However, they can include a part of them (in the user-1 example), or it can happen that they are composed from several records through well-defined selection criteria (user-2).

2.3.2. The Conceptual Level

The conceptual level describes how data would look if everyone could view every aspect of them. This description was formerly called the data base schema (and the external levels which compose a part of this used to be called the sub-schemas). This includes the logical definition of all the record types at the conceptual level. (What attributes do they have and in what form are their values stored?) However, for the sake of data independence, it includes neither a reference to the location of these data nor how to access them exactly.

Who and for what purpose can access a given record type, and within a record type, different fields, is defined at the conceptual level. Thus this is where the security of the data base is defined.

The relationships that can exist between the record types are also defined at the conceptual level; an example of this in the student-class case was shown earlier. The description method depends upon the data model, so it will be discussed in more detail at that part of the book.

2.3.3. The Internal Level

It describes the storage structure and the access methods.

The storage structure defines the exact location where records of a given record type are stored, i.e., at which particular disk, cylinder, track, and block (page), as well as the field order within each record. It also specifies what conversion must be performed on the physically stored data, so that they will be available in suitable form for the users.

The access method defines how and in what order the records are stored, and their access.

Typical access paths are:

- in physical order,

- in logical order, according to the values of a field, or
- based directly upon indices.

As for the internal level, in practice, its determination significantly influences the effectiveness of the data base, so it will be covered in further detail in Chapter 4.

2.4. The Components of the Database Management System

Every database management system includes a control program, which coordinates the function of the individual components.

Its parts are the following:

- *Database Communication Controller*. This allows the applications to connect to the data base.
- *Task Converter*. This evaluates and analyzes the commands given in the language of the database management system. If they are executable, it selects the method of execution expected to be the most efficient.
- *Database Storage Subsystem*. It transforms the method defined for the access of the objects into commands which can be interpreted by the operating system. It starts and evaluates the execution of these commands.

The following briefly-mentioned components are all essential for a well-operating mainframe or PC-based large database management system. Despite this, not all systems provide every one of the described features. It can happen that the desired function can be implemented only with additional components or by the installation of a special application software supplied by outside vendors.

It is expected that a modern database management software provides the following capabilities at an adequate level:

- it can be used by those who have only limited computer knowledge,
- it includes a user-friendly query language and report generator,
- it ensures the automatic documentation of the most important features,
- it provides easy screen handling,
- it can be used interactively, and
- (high-level) programming commands can be generated easily from database management commands.

2.4.1. Data Definition Language

The formal and practical formulation of the data model, which is suitable for further processing, is called the schema. The data definition language (DDL) fosters its accurate description. In most database management systems subschema descriptions are attached to the schema, showing how the data base can be seen by the application programs. Each data base has only one schema, which precisely defines the logical data and storage structure, as well as the logical relationships between the data elements. Part of the physical structure and relationships might be described also in the schema. At the same time, however, many subschemas can be built on one schema in which data can be renamed, and their

type and grouping can differ from the structure defined in the schema. The relationships defined in the subschema are transformed by the database management system into a form which is in accordance with that of the schema. The system completes the necessary conversions, too. The transformation process, the part of the data base which was not included in the subschema, and the physical structure of the data base will all be hidden from the user. So the data definition language has to define the data at three different levels:

- a) *At the user level.* This means the subschema of the actual data base, one of the external levels of the data model. It can be given in a data definition language serving only for this purpose, but very often it can be written in the language of the application software.
- b) *At the logical data relationship level.* This includes the definition of the whole logical schema. This is the definition of the conceptual model.
- c) *At the physical storage level of the data.* This defines the physical accessibility, the storage structure, and the physical connections among the data (pointers, indices, hashing routines). The last two functions are usually achieved by an independent language.

The data base has to include not only the data, but their relationships, too. For example, in a personnel register the departments and employees are both data entities, but at the same time, there is a relationship between them, i.e., employees are members of various departments, and these departments have employees and supervisors. These relationships are as much a part of the data base as the data, although they can be described in a different way (e.g., with pointers). The data definition language has to be suitable for describing these relationships as well.

While the definition of the records in the different data models is basically similar at all three levels of the model, the representation of the relationships is strongly model-dependent, as will be seen in Part II of the book.

2.4.2. Data Manipulation Language

For processing purposes, a data manipulation language (DML) is needed *with whose help the user can carry out operations on the data base defined by the data definition language*. The operations relate to entering data into the data base, the deletion, modification and retrieval of the data. They include procedures:

- to open and close the data base (or the part defined in the subschema),
- to search for data according to a key value or relative address (next, previous, first, last),
- to retrieve a record once found,
- to restore the modified record,
- to insert a record based upon a record key or an address,
- to delete completely or partially a record, and
- to modify some record fields.

If these changes affect fields to which certain access structures (e.g., indices) are connected at the physical level, then the database management system automatically handles their maintenance.

The data manipulation language can be a self-contained system, but it can also be included in a high-level programming language, (Host Language System) as has been mentioned in Point 1.3.3.

The older database management systems' languages were process-oriented. The user had to formulate how and in which way he wanted to access the data in the data base. To accomplish this, it was necessary to know well the structure of the data base, and how to advance towards the desired goal, based upon the relationships of the records. It used to be said that the user navigates in the data base. On the one hand, a person who knew well the various routes could navigate very effectively. Whereas the hapless soul who did not have a sufficiently clear idea may not have sailed 'round the world' two times, but he often examined needlessly many thousands of records before he arrived in the 'Promised Land'.

The ships which once plowed the oceans have been displaced by transcontinental airplanes. The saying, "*navigare necesse est*," has gotten a new meaning in database technology. Today's modern relational database management systems have automated the search for the route in the data base. The user need declare only his goal. The optimal path for this is defined by the system itself.

Of course, to realize this a lot of current information about the system itself has to be stored in it. This will then be used for access optimization. Besides this, by solving special tasks, expert users can gain knowledge that the system cannot recognize automatically. It can happen when in a given situation a generally not too selective condition becomes highly selective. For example, from a general hospital register only those individuals are important who possess a very rare blood group. In that case, it would be more expedient if data retrieval were based first upon this blood group, because only a few records would have to be examined later whether they satisfy the other conditions as well. But when dealing with common blood groups, this method would result in many thousands of records being needlessly retrieved and evaluated. That is why the automatic access path generator of the system would not select this method. However, in a specific case this can be faster to a large degree.

This situation resembles the automatic transmission. It is simpler, more convenient, and, in general, more economical than the manual stick shift for a driver who is crawling through town during rush hour. But on an empty highway or a racetrack, a professional driver can best exploit a vehicle's capabilities if he himself can decide when to change the gear.

2.4.3. Interface to Programming Languages

When the commands of a database management system are issued in a self-contained language in an interactive mode, this component enables the user to issue these commands from his/her terminal and follow the results on the terminal. If the database management system is built into a programming language, then it interprets the data definition and data manipulation commands. The programmers write these data definition and manipulation commands — in conformity with the syntax of the host programming and database management lan-

guages — into their program. The language preprocessor transforms these commands into a form suitable for the given host programming language.

The advantages of this preprocessing, otherwise called precompiling, are the following:

- programs can be written in the language which best suits the task, which is used generally at the installation, or which is known best by the programmer (the first is advisable in theory; the other two are followed in practice), and
- the programmer has to learn in addition only the data manipulation language (or at most the data definition language as well) in order to use the data base.

Mainframe database management systems are available both in interactive versions and with host-language capable interfaces. With this interface, they can be built into COBOL, PL/I, Assembler, and even many of them into FORTRAN and C, as well as other languages.

For most PC systems, there is no such interface for any host language.

2.4.4. Report Generator

If the result of a query is more than a few records, then it is expedient to display it in a well-designed report form. This means the attractive arrangement of the results, header, page break, and group changes, the formation of subtotals/totals for numeric data, and even the presentation of graphs.

The report generator commands are usually given in a special language close to English. Their complexity and efficiency can vary to a large degree, depending on the system. There are generators principally designed for programmers and their capabilities are almost the same as those of a programming language. On the other hand, there are generators which can produce basically only simple formats and they offer help to non-programmers.

No matter how it is viewed, a good report generator — if its use has already been studied — in any case, makes it easier to survey the results and saves time for programmers and end-users alike.

2.4.5. Screen Formatter

A user-friendly environment includes the ability to easily enter and manipulate data from the terminal. The design of the screen needed by an application is fairly complicated. Counting by rows and columns, the location of all messages and fields has to be defined, and they need to be adjusted for corresponding display attributes.

This task is facilitated by screen designers. The user can 'draw' on the screen what, where, and in what form he would like to see or enter data. The designer translates this into screen format generating commands. Almost all PC database management systems include screen designers.

2.4.6. Utility Programs

They handle in a standardized manner the solution of regularly-occurring tasks.

Their most frequent uses are:

- loading formatted data when creating a new data base, importing a large amount of data, or restoring saved data,
- formatted saving of the data base for security reasons or transmitting it to another system,
- reorganizing the data of the data base (saving and reloading by a given sequence) for achieving optimal efficiency,
- large-scale deletion of unnecessary data,
- large-scale, non-formatted data unloading for security reasons, and data restoring in case of error, and
- creating a data base dump so that data can be viewed in their real physical storage form, rather than in their logical structure.

2.4.7. Data Dictionary (Catalogue)

It enables data base administrators, programmers, system analysts, and end-users to document the data base and its applications uniformly, in a method accessible to all interested persons. The data dictionary decreases significantly the paper-work associated with the data base and the application systems.

In its simplest form, the data dictionary, often called system catalogue or just catalogue, includes data definitions, the relationships among record types or different objects, the access and operation rights concerning the specified objects, report formats, and views defined at the external level.

More developed versions make possible the generation of new objects based upon existing records, and the examination of the effect of different changes in the system and their contradiction-free execution. In case of logical contradiction or potential error, respectively, they call attention to the consequences, and, perhaps, block the execution of the change.

A typical example of the latter is when at a conceptual level a record type is to be deleted, and this record type is at the same time identical to an existing record type at the external level or is a part of it. It depends upon the system whether this deletion is executed automatically, or the system warns about this dependency, and, only after confirmation of the request, executes it. Or it is not at all willing to delete as long as prior references exist. As will be seen in the section discussing the catalogue, all three methods have their pros and cons, too. Naturally, in case of automatic deletion, the ability to find out with the help of the data dictionary what other objects would be affected by the potential deletion is essential.

Many data dictionaries include information about files which are not part of the data base, as well as translated programs which work with the data base.

2.4.8. Data Security

Data security, integrity and inviolability deserve increased attention because many programs use the same data simultaneously or successively. Strict rules are required concerning who is authorized to read, insert, and modify certain data types, and the system has to check this automatically. The data base has to be protected from unauthorized access, which can be intentional or accidental (e.g., as a consequence of a defective program). Besides this, it must be ensured that modifications will be consistently executed at all levels, the different programs work with consistent data, and in case of an error, the data base can be restored to its pre-error condition. Because more and more institutions' entire information systems are based solely on data stored by computer and programs handling these data, the system must be protected against breakdown or destruction caused by willful or accidental human intervention, or natural calamity.

2.5. Benefits of the Integrated Data Base

To summarize the above-mentioned points, the following definition can be given for the data base and database management systems: *The integrated data base is a collection of interconnected data which can be used by different users in different arrangements. Data storage is planned and controlled centrally with minimal redundancy. Data protection, the insertion of new data, and the modification of existing ones are controlled centrally, too. The database management system allows access to the data base by authorized users in a way that permits application programs to be independent from the data storage method.*

The benefits of the integrated data base can be summarized as follows:

- the institution using the data base can control all the data needed for its operation in a centralized way;
- manifold storage of the data can be quashed. Thus accidental contradiction among the actual values of the same data can be eliminated;
- protection of data from unauthorized access can be implemented better;
- the logical contradictions among different data can be sifted out better;
- various questions can be formulated differently, the relationships between data can be evaluated better, and
- from the perspective of the whole institution, processing can be more effective.

Since the data base should satisfy all of the above demands, its definition, creation, maintenance, control of its access path, and periodic reorganization according to changing demands must be carried out centrally. The *data base administrator* is responsible for this activity. His tasks will be discussed in greater detail in the next section.

Until the end of the Seventies the benefits of logically and physically-integrated data bases, i.e., the data were stored in one place physically, too, were obvious. However, at the beginning of the Eighties some shortcomings of physical centralization came to the fore. The most basic one stemmed from the fact that the use of certain data groups was necessary only at one place, but the data had to advance through the entire data communication network. Thus, besides

the need for much more expensive computers which were larger and capable of storing everything from everywhere in an accessible way, the size and the expense of the data transmission network increased significantly. *Distributed data bases* arose to eliminate this. These make possible the logically-unified handling of data bases which exist physically in different places and the centrally-controlled flow of data between particular data bases. The most important features of distributed data bases will be discussed in more detail in Section 2.9.

2.6. Database Administrator

The installation of database technology at an institute or company triggers a change in management methods, too. The integrated information system modifies people's tasks, style of work, and what is more important than these, their way of thinking. If these changes do not occur, then the firm cannot exploit the advantages of the database technology, the application of a database management system. Indeed, the firm might obtain more defective information at a slower rate and at greater expense.

To ensure that the data base would be designed in a correct manner and properly used, a new function had to be created, the range of duties assigned to the data base administrator. Today at large institutions this function is handled by a group, rather than a person.

The tasks of the data base administrator of the Seventies and Eighties, the wizard of information science, are today more and more divided into two parts.

The data base administrator handles above all the direction and control of technical tasks connected with the computer hardware and software.

Other kinds of skills and expertise are needed for communicating with users who submit the essential information requests, reconciling these competing demands, and deciding their priority. Previously, this job was carried out by the data base administrator, too. Today this person is called instead the *supervisor of the information resources*. He knows the organization excellently, yet also possesses a thorough computing knowledge. So this supervisor can make himself understood to the data base administrator, who is a first class computer specialist in the area of database technology. As the latter knows well enough the organization's tasks and goals, he can clearly understand and successfully implement the requests of his colleague, the resource supervisor. Their synergy redounds to the benefit of the entire firm.

Despite this division of duties, later on the data base administrator's job will be discussed in a unified manner. The nature of the task will determine which part of the double ego is being referred to. As for satisfying the needs enumerated at the beginning of Chapter 2 and Section 2.1., they have to work jointly starting from different directions and only the last point falls into the sphere of the information resources supervisor's authority, according to which information is worth something only when it is used properly.

The consistent development of database management systems does not stimulate automatically an improvement in users' data processing jobs. A good database management system is not sufficient in itself; it must be used properly with

a fittingly developed data base. If the data base is created simply by the union of already existing files, then the programmers will use similarly the customary field names for the duplicated data. So essentially nothing is changed other than that instead of the former, simpler methods, a significantly more complicated (and more expensive!) method is used, whose hardware configuration and run time are larger. Completely redundant-free data storage can increase the access time to a large degree, but this same undesirable result can also be achieved by a skillfully-designed redundancy, where the data are not referred to by suitable names. In practice, such mistakes can occur due to the following reasons:

- Open logical overlapping. The same data type with the same naming is used in many files.
- Hidden logical overlapping. The same data type, albeit with different naming, is present in many files.
- Apparent overlapping. The same naming in different data files, but refers to different data types.
- Physical overlapping. The same data types are stored manifoldly.
- Absence of logical overlapping. The correct logical relationship among the different data files cannot be achieved. (E.g., all the data are given about specific goods and warehouses, but it is not possible to establish in which warehouse particular goods can be found.)
- Dissimilar coding. For the identification of the same object, different code systems are used.

The formation of suitable conditions needed for effective processing, the consistent execution of changes, the regular saving of data bases' content, and the regulating of access rights cannot be entrusted to the users' whims. These tasks must be centrally directed. This job is handled by the data base administrator or an otherwise titled person responsible for the data base.

The tasks of the data base administrator are the following:

- The organization of the data base. This usually involves the designing of the data model, the definition of the data base, the naming of the defined data, and, if possible, the selection of the search strategy, the definition of the access rights, and the loading of the data base.
- Assisting users of the data base.
- Checking the validity of the data entered into the data base, or rather helping users to do this.
- The periodic saving of the data base's data, so that in case of accidental errors, the data base can be restored to its original error-free condition by the data base administrator.
- The constant supervision of the data base's use, and the reorganization of the data base in case of a significant decrease in efficiency.
- The modification of the data base's structure, based on the demands of the users.

The job of the data base administrator is even more complicated, because creating the data base and maintaining it occurs amidst the advocacy of interests which often conflict with each other, yet have to be taken into account. Moreover, the ranking of the viewpoints is often not unambiguous and in the course of time it can change.

Table 2.1.

The most important goals which should be achieved by creating and using the data base are as follows:

Goal		The Available Method to Reach the Goal
Primary Goals	Satisfaction of different requests by many users	Physical and logical data independence, data manipulation language(s)
	Compatibility with former programs, data, and extension potential for projected demands	Physical and logical data independence, data definition language, creation of schemas and subschemas
	The minimization of the modification costs	Physical and logical data independence, data definition and data manipulation languages, creation of schemas and subschemas
	The minimization of processing expenses	Controlled minimal redundancy, effective search and access paths, data standardization
	Precision, contradiction-freeness, security, and secrecy of data	Centralized control of users, minimal redundancy, automatic protection against unauthorized users, ability to centrally restore the data base
Further Important Goals	Reducing new data collection	Insuring the multiple use of existing data, data definition language, creation of new subschemas
	Ease of use	Ability to build into high-level programming languages, simple query languages, remote processing ability
	Simple modification feature	Physical and logical data independence, data definition language, creation of new subschemas
	Availability according to demands	Efficient search and access paths, on-line access
	Central control feature	Programs which check automatically access, modification, and efficiency
	Quick elimination of errors	Programs which log automatically the changes, and programs which restore and restart the system after a failure

In Table 2.1. the most important goals are summarized. It should be stressed once again that the ranking is only relative, and it can change, first of all, according to the demands of the given information system. As for the practical application of the data base, the data base administrator's most difficult job is to survey the information demands and plan accordingly the information structure, as well as arrange it into a physical structure which the database management system can handle.

The first step in the planning phase is to create the different levels of the data model. To accomplish this, the data important to the user and their relationships must be defined precisely. In general, it is a truly difficult task, because for this not only is it necessary to know the current structure and information flow of the organization or institute which uses the data base, but also their projected expansion and change.

By the end of the conceptual data model planning procedure, the data base administrator must define and name all the elements which will go into the data base, as well as their attributes and relationships. He has to distinguish the attributes which have values that identify certain elements (keys) from those which characterize other attributes.

The second step is the planning of the physical data base, the creation of the records, their placement, and the definition of the search and access paths. These conditions, which are described by the database definition language, are transformed into tables and routines by the database management system. During actual use it takes the necessary information from these tables and routines.

Some database management systems create automatically the storage structure, based on the logical description. These are basically easier to use, but a system that is built in this manner often does not work efficiently enough. For this reason, most systems permit a certain leeway to the data base administrator. The data base administrator can decide (when the system offers this choice) which keys can be accessed by indices and which by hashing, which hashing algorithm should be used (see Section 4.8.), on what key should the sequentiality be realized physically, on what with pointers, is the pointer chain uni- or bidirectional, and what should be the hierarchical route (see Section 4.6. and Point 5.3.2.), etc.

In order to choose the most effective method available from the database management system, the one which considers all users' weighted requests, the data base administrator must know well not only the database management system and the data manipulation language, but users' demands too. Nevertheless, regardless of how multifaceted a large database management system may be, sometimes it cannot handle certain logical relationships. For example, in simple hierarchical systems (see Chapter 5), a M:N relationship cannot be implemented. Many relational systems do not support foreign keys for cross-referencing tables (see Point 7.4.3.).

In this case, it is also the task of the data base administrator to supplement the system with additional routines that enable the description of these relationships or modify the logical information system so that it conforms to the possibilities available in the database management system.

After the data base administrator creates the physical and conceptual levels of the data base, he constructs also the external levels of the data model which suit the individual users' demands.

2.7. Some Disadvantages

In the foregoing, solely the advantages of using data bases and database management systems were described. However, by this time it should be obvious that the fundamental truth that 'there are no free lunches' discovered originally by economists, is valid in every field of life. The advantages of using a data base have to be paid for in various ways. However, it goes without saying that in most cases the advantages outweigh the disadvantages. So when all is said and done it is worth deciding in favor of database management systems after all.

Here follow the most important arguments against such a system, or, at least, those not unambiguously in support of it.

The first and, perhaps, the most important one is that there is no organization which stores all of its data in a unified, integrated data base, or in its more modern, distributed form. On the basis of Murphy's Law, if there ever was a data base that contained all the data of a company (even the smallest firm), the relationships among them, and the access authorities, then its handling would be so complicated that no one would use it. To say nothing of the 'trifle' that planning and creating such a data base, as well as loading it with current data and regularly updating it, would consume so much money, time, and human labor, that nothing would remain from these resources for the basic operations of the company. Thus the company that based all its activities upon only one single integrated data base would become bankrupt within nine months. (The author's complement to the previously-cited Murphy's Law).

There will always be some — perhaps only local — applications which work with their own data, their own data bases, and what is more, because of the simplicity of the application and the speed of its development, with their own simple files. Most database management systems cannot handle 'ordinary' files such as these, unless the data are first loaded into the data base. With this, the classical problem of 'it is twice stored, it needs to be maintained twice, and how to ensure the consistency of the two types of data files' arises at the data base level. There are only a few systems, such as, for example, the SQL/400 relational database management system, which runs on the AS 400 of IBM, which are entirely transparent to the data structure. It can work on traditional files with the DML commands, just as on the database tables, and work can be done without restriction on the objects of the data base using also the file handling commands.

A further serious disadvantage is that, in comparison to handling files, a database management system is expensive, has a voracious appetite for computer resources, is often more complicated, and takes more time to learn.

It always costs a lot of money and time to switch over to database technology from file use. The new technology must be learned. To achieve this, it is usually necessary to call upon outside experts for help, which is expensive if useful advice is desired.

These problems are enumerated here not to dissuade anybody from using database technology. Rather the reason is to clear up some misconceptions and unrealistic expectations about the omnipotence of this method.

Expenses increase further when old data are transferred into the data base. Due to the better checking and stricter demands of the new system, at this time many inconsistencies are discovered among the data which were previously considered to be free of errors and contradictions. Their elimination naturally makes the information better and more exact, but due to the nature of the task, it can generally be handled only individually via actual examination and modification of the data. Thus this is a very time-consuming process that requires a lot of human effort as well; it is therefore expensive.

A contributing factor is that existing programs must be rewritten so that they can use the data base. The old and new systems have to be operated in parallel until the new programs operate reliably.

Database management systems are written, above all, for the purpose of being used with various data structures and in many kinds of applications. So they provide something for everyone. Yet it follows from their general-purpose nature that an application system which is made specifically to solve one task and which works with traditional files tailored to the special needs of the task is more effective with it. This counterargument will be stressed if the application involves a large number of transactions that are not too complex, and it is designed to solve a task related to the main function of the organization.

The relationship between the data and the application is always realized through several software layers when working with a database management system. It is manifested by a larger storage requirement and mainly a longer run time. This problem is even more significant for large data volumes and/or a great number of transactions where hardware is exposed to heavy burdens anyway.

An additional problem arises in practice since most firms use commercially-available software products that were prepared to solve specific application problems. It is not sure that such softwares can cooperate with the central database management system. Frequently the two softwares are not compatible with each other, even at the data level; neither of them can read the data produced by the other. If an operating database system is in place, then among the many possible options for solving the new task, the firm is often forced to decide in favor of a software that is more expensive or has inferior capabilities because only the latter can cooperate with the current database management system.

2.8. Data Base Management Systems of Personal Computers

The previous considerations are valid above all in integrated data bases with several users. These are installed on one or more large mainframe computers and in PC networks with a server machine, respectively.

The database management systems of individually-operated PC's are slightly different from their richer cousins on larger machines. The main differences are the following:

- PC-based database management systems can handle only less complex data structures. They are based exclusively on the relational data model, but implement only a part of the functions proposed there.
- Their data protection, access regulation, and control are weaker.
- Making security copies and restoring the original condition, in case of a failure, are not or only partly automated, although losing two orders of \$ 1,000 because of a computer system failure can cause (relatively) the same damage for a small company as losing several hundred similar records in the large system of a big firm.
- Since only one user works with the machine at any one time, there is no protection (record locks) against the problems which stem from concurrent access. That is why it is an arduous task to insert later such systems into a network. However, life and business pressures force this many times. In systems that are subsequently tinkered with in this way, record lock handling is much more clumsy, resulting in far more problems and frustration for the user than in systems on large computers which were planned from the start with these guidelines taken into account.
- The procurement costs and the extra hardware/software expenses necessary for the operation of the data base management system are much smaller not only in an absolute sense, but also relatively (for example, in comparison with the value of the information supplied by the database management system or the entire computer system). This applies especially to the software since a system used on large machines can be sold possibly only to a few hundred customers, whereas 10,000 or upwards of 100,000 copies of a successful PC software may be purchased.
- The introduction of the database technology has no or much less effect on the organization or the organizational structure of the firm.
- In proportion to the price, the 'saturation' of the system occurs much earlier. While well-operating mainframe systems slow down considerably only when they process a very large quantity of data, with PC systems working on a somewhat more complex data structure, this can happen even with a few thousand records.

2.9. Distributed Data Bases

2.9.1. Distributed Systems

While in the Sixties and Seventies the direction of development was marked by the centralization and concentration of computing resources, from the Eighties creating distributed systems and within them distributed data bases came to the forefront more and more.

Distributed systems are categorized mainly by how the attached processors are dispersed locally. In practice, however, this is only one way of characterizing them. The most important tasks which have to be taken into account during the planning and operation of the distributed systems are the following:

- Where does the data processing take place? (Where are the processors?)

- What communication links exist between them?
- Where are the data to be processed stored?
- What rules and standards are valid for processing and passing on the data and information?

From a technical aspect, four main types of distributed systems can be distinguished.

1. Hierarchically Arranged Processors

In the usual three-level model of distributed resources, the central high-performance computer (mainframe) can be found at the top. The local minicomputers are connected to it and the work stations, PC's, and terminals of the end-users are attached to them (see Fig. 2.2.).

Data processing and storage are possible at any level of the hierarchy. Usually a given kind of processing always takes place at the same level. It should be noted that the three-level hierarchy is not mandatory either. The number of levels can be more or less than this.

The processor hierarchy usually matches the organizational hierarchy, too. The mainframes can be found generally at the headquarters of the organization or at the local center. (Naturally, this brings to mind large companies and institutes which need such high-performance equipment.) For local centralized processing (plants, department stores, bank branches, etc.), lower-performance mainframes

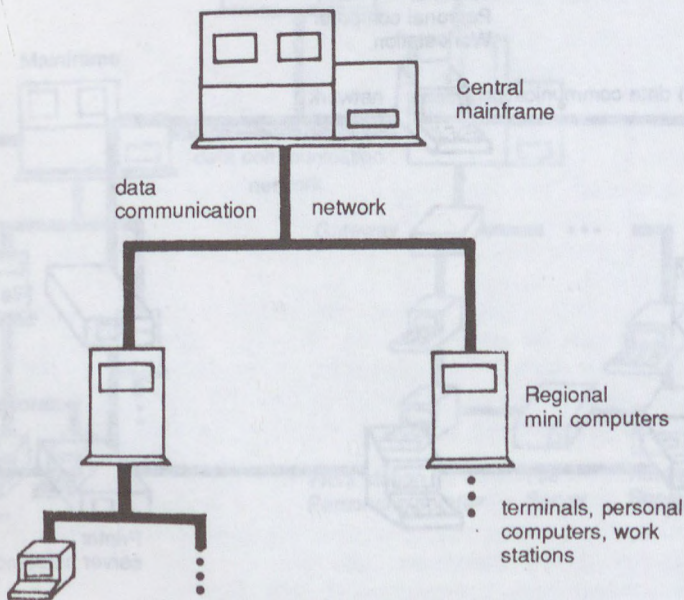


Fig 2.2. Processor hierarchy

might also be used, but minicomputers are preferred instead. Minicomputers can satisfy the individual demands of the large departments of the central organization as well.

At the lowest level of the hierarchy, workstations, PC's, and terminals can be found. These machines are used individually for daily work by independent users or as necessary in common by small groups performing the same tasks.

In this arrangement the placement of computers reflects the company hierarchy.

However, this is not obligatory. At the bottom two levels of the processor hierarchy, machines are often used according to the functions within the organization. With the first system, internal administrative problems are solved. With the second, new products are planned, and with the third, for example, tasks connected with sales are handled.

2. Cooperative Work Stations and PC Networks

This arrangement is becoming more and more prevalent with office applications. Each station or PC is equal; they can communicate freely with any other member of the network. To a certain degree, server machines can play a significant role; they store in one place the shared data used by everyone, or they control the central printer(s).

The network very often includes also a gateway through which it can be connected to other networks (see Fig. 2.3.).

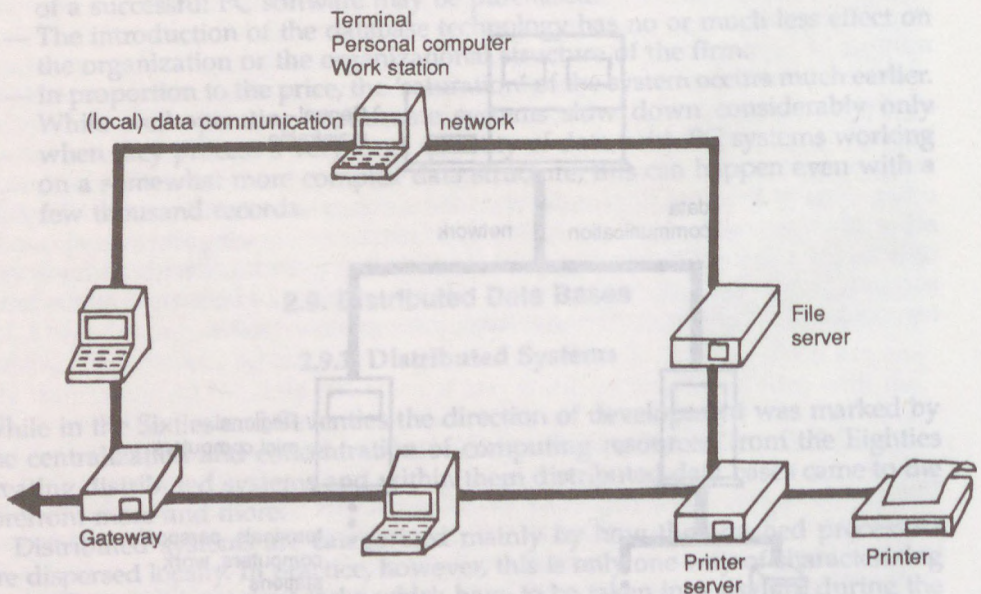


Fig. 2.3. Cooperative workstations network

3. Independent Standalone Machines

In principle, distributed processing can be carried out even with independent standalone machines. The consistency of data can be ensured manually or by changing the volumes containing the data, respectively. Today, this form would no longer be considered distributed processing if there were not systems belonging in the meantime to the first or second groups. These systems can be regarded as distributed among themselves. Although they are parts of a bigger information system, they have no direct relationship with each other. (For instance, a company with many sites has an independent local network or a central machine with work stations at each site, but they are not connected to each other.)

4. Discretionary Combination of the Foregoing

This includes the connecting of the processors standing at the top of the hierarchical processor systems, the linking of different local networks, and the coupling of local networks to different levels of the processor hierarchy, etc. (see Fig. 2.4.).

However, the following conditions have also to be realized for the equipment in the above-mentioned arrangement to truly form a distributed system:

— the possibility of distributed processing,

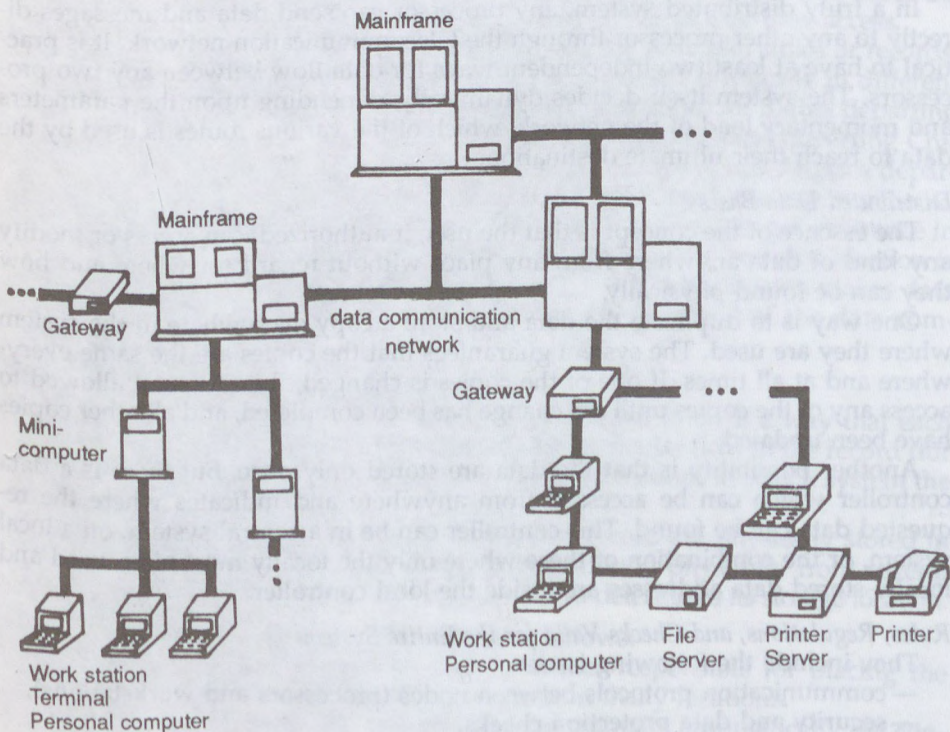


Fig. 2.4. Complex distributed system

- connectivity of processors,
- distributed data bases, and
- rules, regulations, and checks valid for the whole system.

Distributed Processing

Processing which is independent of each other takes place — often even with the same data — simultaneously in several places, but steps are taken to ensure data consistency. The choice of which machine or machines are ultimately used to perform a certain task depends upon which of the following has priority:

- processing is performed where the necessary data or most of them are stored (data traffic decreases);
- processing takes place where the information, i.e., the result, is needed (data and information traffic decrease and are more secure);
- processing is performed at the most suitable machine for solving the problem (processor expenses decrease and local hardware limits do not have to be considered);
- processing takes place at the machine which has free capacity at the requested time (faster).

Connectivity of Processors

In a truly distributed system, any processor can send data and messages directly to any other processor through the telecommunication network. It is practical to have at least two independent ways for data flow between any two processors. The system itself decides dynamically, depending upon the parameters and momentary load of the network, which of the various routes is used by the data to reach their ultimate destination.

Distributed Data Bases

The essence of the concept is that the user, if authorized, can access or modify any kind of data anywhere from any place without regard for where and how they can be found physically.

One way is to duplicate the data and store a copy everywhere in the system where they are used. The system guarantees that the copies are the same everywhere and at all times. If one of the copies is changed, then it is not allowed to access any of the copies until the change has been completed, and all other copies have been updated.

Another possibility is that the data are stored only once, but there is a data controller which can be accessed from anywhere and indicates where the requested data can be found. This controller can be in a central system, on a local system, or the combination of these where only the locally most often used and locally-stored data addresses are inside the local controller.

Rules, Regulations, and Checks Valid for the Entire System

They include the following issues:

- communication protocols between nodes (processors and workstations),
- security and data protection checks,
- rules for transferring programs and data, and
- general working conditions.

Creating distributed data bases is often justified as resulting in better utilization of hardware and software resources. Although this — in case of well-formed systems — is acceptable, yet a much more important argument for distributed systems is that users of data can once again feel in control of their own data and tasks. They no longer have to turn to the distant centralized computer center, the 'Kafkaesque Mansion', for local information, but instead can rely on their own computer.

2.9.2. Features of Distributed Data Bases

Although the distributed data bases described thus far are different from each other in many ways, their common characteristic is that they increasingly approach the 'ideal' objectives described below.

- *Spatial Transparency*: the entire data base appears to the users to be stored in one place. They need not be concerned about where a given record or some parts of it are located. The database management system collects the segments and passes the related data as a whole. Essentially the mapping of the 'conventional data bases' between the external and internal levels is carried out one level higher. There the users need not be mindful of where and how the data are stored in a given data base. With distributed data bases the users are also free to ignore which machine and data base they access.
- *Data Fragmentation*: logically-connected data, for example, tables or records, for security or efficiency reasons may also be stored physically in different places on different machines. For instance, data about all flights departing New York are in the New York computer, data about those departing Chicago are in the Chicago machine, and data concerning Los Angeles departures are in their local computer, etc. On the other hand, the common book-keeping can be done in Dallas, and the recording of passenger insurance in the Hartford system. Most of the data are used locally and that location is responsible for their accuracy and security. Such local division makes data operations at each site faster and reduces the workload of the data communication network.

There are two types of data fragmentation:

- *horizontal fragmentation*: one table (file) is divided in such a way that each record stays in one place. The value of one particular field in the record (for example, the departure location of a flight) determines in which system the whole record is stored.
- *vertical fragmentation*: the various fields of a record are physically stored in different systems. Each record has to be collected physically from several computers. Not the value, but the type of field determines its storage location.

Further characteristics of segmentation are as follows:

- *fragmentation transparency*: the user is neither responsible for placing the fragmented records nor gathering them from many locations.
- *handling of data copies*: the system is able to maintain automatically the consistency of fields and records stored several times in the interest of efficiency or guarantee logical relationships. For example, 'frequent flyer' passenger

data are stored at every location to foster quicker access at all sites. Modification of such a record can be initiated also at any location. The system ensures that during the time of alteration no copy of the record can be accessed at another site (record locking), and once the modification is executed the same new version is available everywhere.

- *copy independence*: if several copies of the same data are found in the system, then each copy which the user could conceivably use must be identical. It is the task of the system to select that copy which can most quickly be accessed.

If the distributed data base meets the above-listed requirements, then it behaves towards the user exactly as a centralized data base. The problems arising from the division are solved by the system itself. Externally, these problems are not visible at the logical level. The conceptual and external levels have to be planned in the same way as in the case of a centralized system; programming languages can be used likewise.

Certainly, the internal administration of the system will be essentially more complicated and, because of this, distributed data bases have not only advantages, but disadvantages as well, which will be discussed in the next point.

2.9.3. Advantages and Problems

Creating distributed data bases results in the following benefits:

- operational costs will be lower because of less data communication,
- data stay near the user,
- the data base and the managing system can be built in modules,
- single nodes can be changed while the system is 'up' without disrupting the system, and
- the failure of a node makes inaccessible only the data stored exclusively there; the other parts of the system are unimpaired.

On the other hand, it is obvious that certain disadvantages and dangers have to be reckoned with; new problems awaiting solution also arise:

- the system can become very complicated,
- in heterogeneous systems translation problems may arise, and
- none of the nodes can close in front of others, even if a node no longer needs the system, some other nodes may need to access it. For example, a node is in a storeroom where the employees' workday has ended, so it could be shut down. Yet data which other users may need are perhaps stored solely in this node.

However, most of the difficulties stem from ensuring the consistency (unanimity) of the system. In a distributed data base, it is hard to avoid redundancy (i.e., certain data can be found in several copies). Let us imagine this: in practice, it is sure that there will be data in the system that are considered by several nodes as their own and which the nodes wish to store in their own machines. This demand can be justified since it is possible that the relevant nodes work with the

data in question in nearly the same ratio. By entering several copies of this data into the system, the network is less burdened and the communication expenses can even be reduced.

In addition, it is prudent to store several copies of data of primary importance in order to guarantee that the failure of any one node of the system will not result in system paralysis.

However, a high price has to be paid for the advantages here as well. It is self-evident that it is inadmissible for two copies of the same data to have different values. (In such circumstances the answer to a question would depend on the data access path.) Serious efforts are required to ensure that modifications do not cause inconsistency even for a short period.

Similarly, it is a difficult task to find data rapidly. It is a vain struggle to keep the data to be used 'on the spot'; every data cannot be stored in every node. (This method would be catastrophic, and not only regarding storage device utilization. Occasional changes would generate so many data transfer requests as to render senseless a 'distributed' organization like this.) That is why it may be necessary, albeit rare, to use the data stored in another node. For reading, only one copy of the data must be found (preferably the one which is the easiest to access). However, if the data have to be modified, then information about the whereabouts of all copies of the data must be obtained. (This is indispensable even if a copy can be found locally.) To check all the nodes from beginning to end would be too cumbersome and expensive a solution. To avoid this, it is practical to make some type of directory about the location of particular data. However, with this all of our problems begin again since the directory contains data which are also read and sometimes modified. If it is stored in a central place, then this center has to be called upon before every transaction, thus increasing the communication volume. In other words, in this case, communication is not at all reduced regarding the frequency of contacting the center, in comparison with the centralized organization. The quantity of transferred information might be less, but the question is whether this outcome makes it worthwhile shouldering the complications inherent in working with a distributed data base.

Another extreme is that a directory is placed in every node. If the data bases are sufficiently static (the data are usually read and modification is very rare), then this method may be appropriate. However, if the data have to be changed often (changing their location has to be considered also, even if the content is constant), then this method is not feasible either. Every node has to be called upon every time to update the directory. The solution is to divide the directories. The global (network) directory — which is needed by all nodes — describes the data base model in principle, and in addition reports the storage location of the individual entity types. (In fact, the required logical unity of the data base can thereby be ensured.) The local directories describe the data of that single node in detail.

Since data secrecy is important also to the users of distributed data bases, it is also expected that the system can distinguish among different authority levels and ensure that individual users cannot go beyond the bounds permitted to them. The question is where the tables containing the necessary data for this are to be placed, and where the appropriate controls are to be carried out. It is a rather common solution to store the access rights in the global directory as well.

2.9.4. Data Distribution

The issue of data distribution was touched upon earlier. However, the importance of the topic warrants more detailed discussion. The efficiency of distributed data bases depends considerably upon how the distribution of data fits into the processes.

Distribution of data can be static when the data base administrator specifies the location and number of copies. (Yet the static attribute is meant in a looser sense since this condition can and must be revised from time to time.) In a dynamic distribution, the system decides — on the basis of evaluating the appropriate statistics — when it is necessary to make a new duplicate or just when it is expedient to abandon one (obviously paying attention to the safeguard that the last occurrence may not be cancelled as a result of any statistical evaluation).

To distribute the data, it is practical to consider first the following:

- a) *Where are those data most often used?* Naturally, the examination has to consider the method of use also, especially if the data in question are read or modified (created) in the node in question or if both forms of use can be found. It also has to be investigated how many reads fall to one alteration, regarding all the use of the data. [Let's imagine, for example, a structure in which locations A and B both engage in selling from a store, but the inventory is prepared in C. At first sight, it seems that since the data ultimately have to go to C for assessment, it is appropriate to store them at C's node, and only there. However, if frequent sales (modification of data) occur prior to an inventory query (reading), then it is probably more useful to place the data in A and/or B.]
- b) *How essential are the given data for using the entire data base?* It has already been stated that it might be desirable to make several copies of the data without which the data base becomes unserviceable. It is possible — and also usual — to classify the data according to the 'degree of indispensability' and to order a minimal number of copies for each class.
- c) *The storage and access cost.* Particularly in heterogenous systems, it may often happen that storage and access of data in one node entail different expenses than in another. However, this can, for instance, stem also from the devices of a node being overloaded.

All of the above-mentioned conditions change during operation. It must always be kept in mind that re-planning and re-structuring of a data base ends only when the data base is no longer used.

2.9.5. Ensuring Consistency

It has already been noted how important and what a difficult task it is to ensure the consistency of the data base. Two levels of consistency can be distinguished:

- a) *Strong consistency.* Every modification of all copies is simultaneous.
- b) *Weak consistency.* During a longer or shorter period copies can be different, that is to say, consistency will be restored only after a certain time.

It is evident that either strong consistency is needed or somehow it has to be ensured that — in case of weak consistency — the as yet unmodified data has to be inaccessible to other processing in the critical period.

Either of these can be achieved with the help of the so-called synchronization protocols (collections of rules and procedures). Synchronization protocols can be centralized or distributed accordingly, based upon whether the method to be followed requires a center to play a favored role or not. (This center is not necessarily always the same node. It can change from time to time, from transaction to transaction.)

It is obvious that synchronization protocols are forced to lock certain data from public use for a longer or shorter period of time. In other words, in case of certain operations, the data become exclusive (and not shared) usable resources.

Even with centralized data bases it has proved impossible thus far to solve perfectly the problems which stem from this. For example, *deadlock*, one of the difficulties referred to above, poses the greatest danger. It occurs in the simplest case when two processes want to use the same two data or group of data in an exclusive way. One process locks the first data while the other locks the second data. To continue, both processes need the data locked by the other, but these cannot be released since neither of the processes are ready to go on.

Unfortunately, as with centralized data bases, there is no effective method with distributed data bases either to prevent or avoid deadlocks. Usually, it suffices to recognize the deadlock, stop the transactions involved in it, except for one, and roll back all activities of the stopped ones, either to their starting point or to a well-defined check point.

The difficulties of a distributed system are aggravated. In integrated systems, the locking lasts for a relatively short time, generally until the changes in the storage and I/O operations on the disk in question have been completed.

In contrast, in a distributed system, the requests for record locking and permission to release have to traverse through the network which increases the locking period in itself. The process is further drawn out if duplicates of modified records are stored also in the system. In this case, the lock is released by the database management system only when an update of all the copies has been successfully completed.

It has already been shown in Section 2.6. how important a role the data base administrator plays. It was stated there that this job is handled often not just by one person, but by a group. In a distributed data base, the role of the administrator is even more complicated. That is why — and because of the geographical division — in practice, it is unavoidable to employ several data base administrators.

On reflection, it is obvious at once that local data base administrators are needed at the individual nodes. They deal with local data, and fine tune the local portion of the data base, etc. In addition to these technical tasks, they develop personal relationships with local users. All these tasks can hardly be handled successfully from a great distance.

On the other hand, the logical unity of the data base requires that the schemas and subschemas be created and modified centrally. Questions such as security or actualizing duplicates have also to be dealt with centrally.

Usually, data base administration has three levels. At the top is the director of data base administration who supervises and coordinates the work of the other administrators. At the second level, but still central, three tasks can be differentiated and administrators are selected for each of them. It is beneficial to assign a coordinator, a communications specialist, and a security administrator. Finally, the local administrators of the nodes are put at the third level.

Although it may also be expedient to act in another way in specific cases, depending on the given task and environment, it is justifiable to accept this pattern as a starting point.

II.

DATA MODELS

General Description

3.1. Classifying Data Models

The database management system is always based upon a data model. *The data model describes the data of different types, the relationships among them, and their related data security processes.* After preparing the model, data entry into the data base and its regular use follows next. While the actual data values do not have to be considered in the data model, when using the data base it is always important to know the particular data values (except when extending or modifying the model). Thus it can be said that record types are dealt with in the data model, while during processing occurrences of the record types are handled.

The data model has three independent levels:

- The *inner or internal level* describing the physical storage and access path of the data. It will be discussed in Chapter 4.
- The *conceptual level* or the *schema* describing the structure of all the data in the data base, all the relationships among them, and the logic of the data access and the access authorities.
- The *external level* or the *subschemas* describing how individual users can view the schema.

In the second part of this book, data model means the complex system that describes the data base at the conceptual and external level.

Three main types can be distinguished:

- hierarchical,
- network, and
- relational models.

In practice, the external and conceptual levels are described by the same model.

The first data base management systems were based on the *hierarchical model*.

They will be discussed in Chapter 5.

From the early Seventies, systems based on *network data models* became more and more widespread. They could handle essentially more complicated data structures than the hierarchical ones. What was once considered their greatest advantage is today viewed rather as a disadvantage. The user himself had to

select the data access path. The advantage accrued to the 'smart' user who was able to select the most appropriate route since he understood both the actual data structure and the relationships connecting data. Thus he could work effectively with the data base in this way. Yet this flexibility proved also to be a disadvantage. As database technology spread, more and more people wanted to use these systems. Yet relatively few of them possessed knowledge deep enough to be able to decide by themselves the efficient access path. That is why a significant portion of such (potential) users were either discouraged by the complexity of use or they were not able to take proper advantage of the possibilities inherent in the data model.

In spite of these disadvantages, from the middle Seventies until the early Eighties, this data model was the most popular one. It is also still used today at a very large number of installations.

The network model is discussed in Chapter 6.

From the mathematical point of view, the most-established and best-elaborated is the *relational model*. It also ensures the greatest data independence. It enjoys the additional advantage that its description method is closest to the everyday view or thinking of the typical user. It stores and supplies the data in two-dimensional tables. Its basis was developed by Codd in the Seventies, but the actual realization was not effective enough. Due to this, mainly theoretical experts were concerned about this model. In practice, it was not very often adopted for use with large data bases until the early Eighties.

With the rapid spread of PC's, the group of users mushroomed and at the same time the volume of data to be handled was no longer so big most of the time. So user-friendly representation of the relational model outweighed the disadvantages of less effective realization. Thus nearly without exception new database management systems marketed from the second half of the Eighties were already based upon the relational model. Many of the network systems have a relational version also (for example, IDMS). The relational model is discussed in Chapter 7.

As earlier mentioned, fact and document format data bases can be differentiated. The data models listed so far all describe data and their relationships in fact data bases. The most important features of the *text retrieval systems* which handle documents will be summarized in Chapter 8.

3.2. Main Guidelines of Model Development

The data model always has to describe the real world, or, rather, that part which is necessary for the entire problem to be solved.

However, the same system and conditions can be described in several ways. Which of the different renderings is chosen depends upon the database designer's attitude, habits, and the efficiency and simplicity of the subsequent uses.

The 'This wine glass is half full.' and 'This glass is half empty.' sentences mean the same in reality. What is said when seeing the glass depends only upon our optimistic or pessimistic outlook. However, if later a question has to be answered such as 'How much more can be drunk from the glass?' 'How much has already been drunk?', 'How much is missing from it?', or 'What is the volume of the

glass?', respectively, and it is not even sure that the dividing line is always exactly down the middle, then it is more practical to say 'There is this much wine in it.' or 'This much wine is missing from it.'

As even the smallest detail of the real world is very complex and complicated, a model can attempt to describe only a portion of the real world. Simplification has to be achieved in such a way that the model remains easy to survey and handle by computer technology. Yet it must reflect precisely at the same time all the data, the task to be done, and the important events. A simple example of this is the cash ledger of a company where it is superfluous to factor into the model how many \$100, \$50, and \$20, etc. banknotes are paid out or received. However, at the same company denominating is most important in the departments at the time of salary payment in cash because an exact amount has to be put into everyone's pay envelope. To what extent the entire model is simplified or how far simplification of its individual parts can proceed varies quite considerably, depending upon the intended use.

In addition to simplicity, clarity, and supervision, changes in demands have to be taken into consideration as well. The data model must be capable of satisfying new requirements as they emerge, and if necessary, it should be easy to expand the model. Yet such changes should not impair the model's ability to solve original problems in the same way as before.

Theoretically, it is deplorable, but, in practice, it must always be borne in mind that the complexity of the data model and data volume, and together with these, the storage of all details, are limited in any case by the available hardware and software. There are cases when this appears as a real physical boundary: take it or leave it, that's the system, and work can be done only within its limits.

In other cases, it presents itself rather as a financial constraint. The budget drawn up for developing the information system is limited, and only a portion of it can be allocated to buying the software and hardware necessary for the system's realization.

Beyond the above, the system's response time can act as a constraint as well. Most information systems work partly or entirely as on-line systems. In other words, they have to produce an answer, provide information, within a given time. If a model contains too many relationships, and the details are considered excessively, then that could result in response times that are so long as to render the processed information obsolete and worthless in practice, albeit accurate.

Consider an automatic city traffic control system that regulates traffic lights. Before switching it even takes into account, among many other things: which cars coming from which direction pollute the air with more lead, and how many people travel in these cars, but it completes the whole evaluation in two minutes; for this reason it is unusable in practice. A much more crude system that changes the lights simply according to the criterion in which direction do more cars want to proceed (subject to the condition that after a certain time, a direction which is always in the minority, for example, the traffic of a side street leading into a main road, also gets priority, in any case, for a short period) is much better.

In addition to these general principles, in the actual realization, the particular demands and habits of the users have also to be taken into account.

All of these factors are discussed in detail in Section 4.1.

3.3. Types of Relationships

Everything is connected to everything else — is said despondently by the developers of a new data base after analyzing and evaluating all of the would-be users' legitimate and illusory expectations — what should be its purpose, what the 'wonderful', new data bank that is supposed to soothe the problems of the company should have to know. Naturally, to realize the above objectives also in the data base makes for a rather hopeless task. Taking into consideration the principles discussed in the previous section, the developers must try to make order out of the jungle of relationships, and lift the indispensable ones for solving the problem out of this morass. After this, the developers do their best to ensure the existence of a database management system and hardware which are able to handle the profusion of data and relationships which remain after the final sifting, and can still satisfy the requisite conditions (budget for procurement and operating expenses, development and completion deadline, operating speed, possibility of expansion to meet further foreseen and unforeseen demands, etc.). If this task cannot be accomplished under the given conditions, then the developers resort to the solution condemned by every professional treatise, yet often applied in practice. They simplify the data model so that it will be usable by their computer system.

Reality is somewhere between the two extremes. Certain concessions have to be made in the data model. (That is why the perfect and comprehensive integrated system, the reincarnation of the Leibnitz World philosophy in computer technology, would be an utopia.) Parallel to this, to create or use the new data base, either existing computer systems must be upgraded in terms of capabilities or replaced by more powerful models.

While the data base administrator can only put forward reasonable proposals for the second option, he can actively contribute to the compromise in the first approach. An analysis of relationships helps, as does knowledge of what kind of relationships can the different database management systems handle and how effectively.

Relationships can be distinguished whenever they exist between two entities (two record types) of the data base or two fields of the same record. How these are handled, i.e., the conversion of one type into another, depends upon the data model. A detailed example of this will be given later in Section 7.9, where normalization in the relational model will be discussed.

However, from the viewpoint of data models, the following classification of relationships between the elements, the objects of the data base, is much more important:

1. *Two elements are independent of each other.* This is the most favorable situation. In this case, no relationship is to be established between them in the data base either. Such 'relationships' can be realized without any restriction in every database system. For example, the data records about the grades of primary school students and the amounts on deposit in local bank accounts can be regarded as independent entities in an integrated data base. (Although it is possible that very skilled mathematicians could find a correlation even here. In case of good marks, students receive a larger benefit, and deposits to some accounts might increase. However, this is already the type of relationship that will surely not be taken into account in the data model.)

Likewise, the fields of the date of birth (month, day, year) can also be considered independently from each other, though, perhaps, a counterexample for this could also be found in New York in the ninth month after the Great Black-out of 1965.

2. There is a *unique and reciprocal relationship* between the two elements. This relation is also called a one to one or 1:1 relationship.

Such can be, for example, the individual Social Security number and the combined values of a last name, first name, mother's maiden name, and a person's date and place of birth. Similarly, a reciprocal and unique relationship exists between the codes and the names of the departments in a consistent, good company code system. The one:one relationship can also be handled in every data model.

3. A *relationship unique in one direction and multiple in the other, the one to many, the 1:N relation*.

The value of the second element type uniquely determines the first one, but the contrary is not true. A typical example of this is the relationship between a mother and her children. A mother can have several (considering strictly the data model, any number of) children, but any child can have only one, moreover exactly one, mother. (Stepmothers are not regarded as mothers — at least in this representation.)

This example has so intrigued experts dealing with data bases that in database terminology, a generally-accepted and used term (although there are not too many like this) for this relationship is 'parent' for the side with one value, and 'child' for the one with several values. (At the same time, this proves that theoretical experts are not limited by the real world in their fantasies, since in most cases, every child has not one, but two parents).

These relationships can also be built upon each other. The database children can have children too, and the parents can have parents. This line of relationships is called *hierarchy* or simple hierarchy. The 1:N (one to many) relationship can also be represented in every data model.

4. *Many to Many, M:N Relationship*

Any element of the first entity type can be connected with any number of elements (0, 1, or more) of the second type, and also inversely, any element of the second type can be connected with any number of the first. This is the most general type of relationship. This characterizes the relation between men and women. One man might have several wives. (Of course, if he is not a Moslem or Mormon, then not at the same time, consecutively, rather than concurrently.) Similarly, one woman can have more than one husband. (If marriage is not the only relationship considered, then even the temporal restriction can be left out of the data model.) A further example of the M:N relationship is the classic database relation: one customer places orders with several suppliers, and one supplier delivers to several customers.

The many to many relationship can be handled only by systems based on the network or the relational models.

The different types of relationships can be seen in Fig. 3.1.

Naturally, just as 1:N relationships can be built upon each other in the hierarchy, the different elements of M:N relationships can likewise be linked to each other. This question will be discussed in Chapter 6 at the network model.

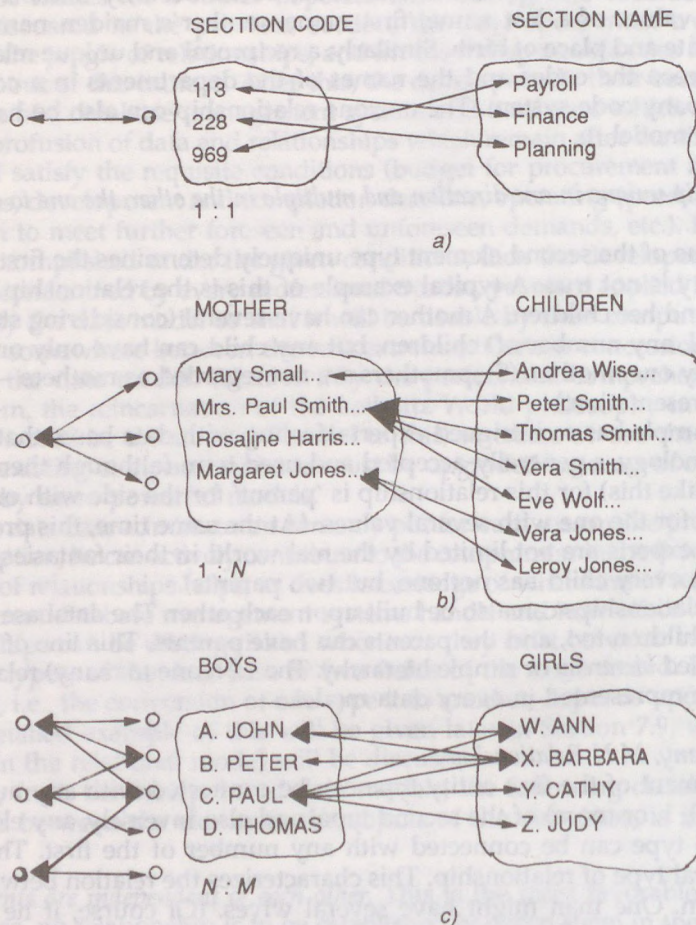


Fig. 3.1. Types of relationship

a) one to one (1:1); b) one to many (1:N); c) many to many (M:N)

Physical Storage Structure and Access Path

4.1. General Considerations

The internal level of the data model describes the real physical storage of data, its physical structure, and the physically realized data access methods.

When creating the internal level, two principal tasks have to be solved. The first is to design a suitable data structure for the data base. Once the data structure has been prepared, the next is to find, modify, enter, or delete particular data. The optimal data base occupies minimal storage, and at the same time, the time needed for its generation, modification, and data retrieval has to be minimized, too. However, these requirements can be satisfied simultaneously only in very simple systems. Generally, it can be stated that storage space and processing time (retrieval + alteration time) are inversely proportional in any database structure.

The decrease of one entails an increase in the other. The same applies to expanding and retrieving, as well as the modification and retrieval time. The faster given data can be found, generally, the longer it takes to modify and expand the data base, modify and delete existing data, or insert new data (see Fig. 4.1.).

The time necessary to create a data base is usually not minimized since this has to be completed only once, while it is frequently necessary to retrieve or alter (add, delete, or modify) data. However, in case of such files where much data have to be stored for security reasons, but the probability of subsequent use at some future time is very low, it is practical to decrease the creation time (or that special modification time when the data base has to be supplemented with a lot of records in one batch) to the minimum at the expense of the alteration time. The expansion time becomes a critical factor if large volumes of new data are regularly added to the data base. For all these reasons, the data base administrator, the organizer of the system, has to be quite sensitive to the following when he/she decides finally in favor of a particular data structure:

- The efficiency of his/her system is influenced decisively by
- what information is most often needed by the users,
 - what hardware is used to store the data,
 - what data volume is to be stored,

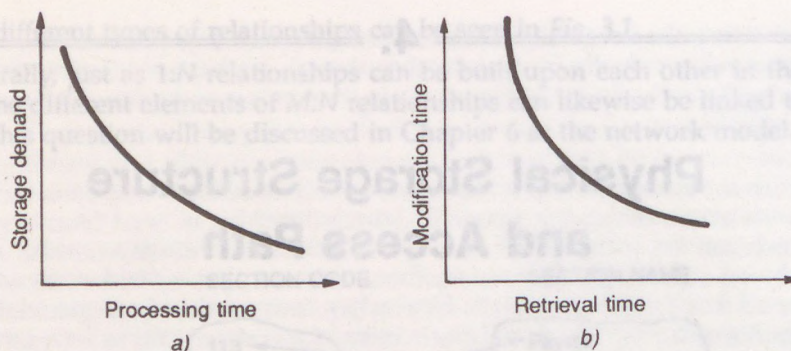


Fig. 4.1. a) Required storage and processing (modification/retrieval) time,
b) Relationship between modification and retrieval times

- how often are the data or a part of them accessed by a predetermined order,
- whether records have always to be retrieved according to the same key fields,
- if the data do not have to be retrieved by a predetermined order, then how many data have to be found, on average, in one processing step,
- whether data have only to be read, or also the content of individual fields has to be changed,
- whether the length of the records may increase, and
- how many new data have to be added to the database or old data are to be deleted, and how often does this occur.

Taking these all into consideration, the data base administrator decides the logical and physical data structures to be created. An expedient compromise develops therefore between the storage requirement and the optimal retrieval and maintenance times. The software system realizing the storage structure, i.e., the database management system, can automatically create the physical level of the data base and look up the requested logical records from it, but only on the basis of guidelines given by the data base administrator.

In this chapter the internal level of data bases is dealt with at relatively great length, though this in principle remains hidden to the average user. As a consequence of data independence, the alterations at the inner level are not observable at higher levels, or rather, not in theory. In practice, the physical data structure and the physically existing access paths influence the efficiency of the system to a very high degree.

A well-chosen storage structure can decrease considerably the area occupied by the data base on the external storage unit. Designing well-defined access paths attuned to the structure of the storage unit and projected user operations can result in response and processing times faster by orders of magnitude. That is why the data base administrator has to be fully aware of what happens at the physical level of the data model in any case. The same applies also to those users who are authorized to create real physical access paths or storage structures to support their own external level, or who can logically define them at the conceptual level.

4.2. Magnetic Disks

4.2.1. Physical Features

Up-to-date data bases are physically stored solely on magnetic disks. Only data saved for safety reasons may possibly be stored on simpler and cheaper magnetic tapes or cassettes. Besides this, at worst, when dealing with very large data bases, it may happen that data files which are big, but used only very rarely, are stored on magnetic tapes or cassettes for economy reasons. They are loaded only temporarily onto the disk for the duration of the processing of the data stored on them.

The magnetic disk consists generally of several surfaces above each other and spinning around a common axis. (At least) one read/write head belongs to each surface. Its position is radially variable. So data storage and access can occur at different concentric circles on so-called *tracks*.

The tracks are located above each other on different surfaces; tracks which are simultaneously accessible without moving the read/write head form a *cylinder* (see Fig. 4.2.).

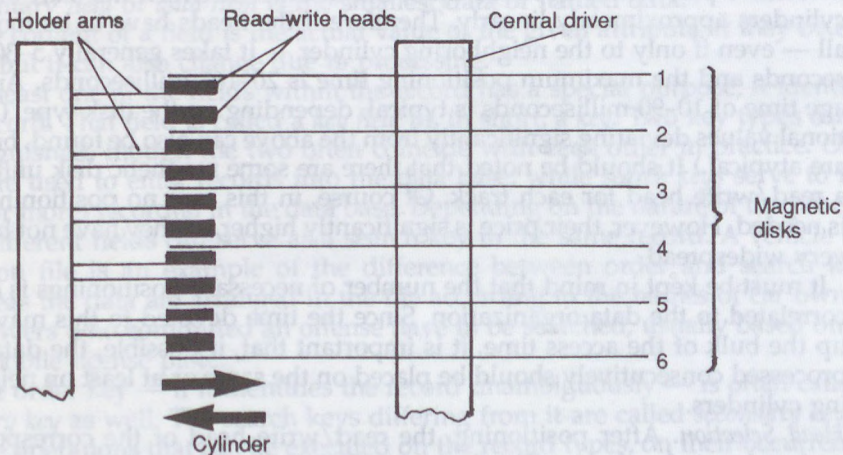


Fig. 4.2. Disk structure

Magnetic disk packages are removable in very many units. The number and size of the disks can vary considerably. In small machines a single surface disk with a capacity of a few hundred kilobytes may be found, but typically 10–20 surface units are used. The capacity of removable disks is usually between a few and two hundred megabytes, whereas non-removable ones boast sizes ranging from twenty megabytes to five gigabytes, or even more.

Earlier in the interest of increasing the actual storage capacity, removable magnetic disks were very widespread. Today, however, fixed, non-removable disks are used for storing a large volume of data. Several of these are connected to the

computer. It is true that this solution is more expensive and the total capacity is limited (in contrast with the storage capacity of removable magnetic disks which can be increased practically ad infinitum). However, the necessary data are accessible at any time and by eliminating disk swapping which results in intensive mechanical stress, disk lifetime increases and its failure rate decreases. A further advantage is that much more precise read/write mechanisms can be installed; thus more data can be packed onto the same surface. These favorable features compensate for the price difference which stems from the larger number of necessary magnetic disk units.

4.2.2. Access Time

The time necessary to access the data on the disk is composed of the following:

- a) *Moving the head, positioning time:* before reading or writing, respectively, the read/write heads have to be positioned to the appropriate cylinder. The time required depends upon which cylinder the unit worked on previously. Naturally, if the heads are already at the correct place, no positioning is needed. The positioning time is the sum of a constant (necessary for accelerating) and a variable part which depends upon the distance between the start and end cylinders approximately linearly. Therefore if the heads have to be moved at all — even if only to the neighboring cylinder — it takes generally 5–30 milliseconds and the maximum positioning time is 20–165 milliseconds. An average time of 10–90 milliseconds is typical, depending on the disk type. (Exceptional values deviating significantly from the above can also be found, but they are atypical.) It should be noted that there are some magnetic disk units with a read/write head for each track. Of course, in this case no positioning time is needed. However, their price is significantly higher, so they have not become very widespread.

It must be kept in mind that the number of necessary positionings is closely correlated to the data organization. Since the time devoted to this may make up the bulk of the access time, it is important that, if possible, the data to be processed consecutively should be placed on the same or at least on neighboring cylinders.

- b) *Head Selection.* After positioning, the read/write head of the corresponding surface has to be chosen. This is executed electronically in a negligible time.
- c) *Rotation Time.* The searched record has to get in front of the read/write head. The time necessary for this is between zero and one entire rotation cycle, if there is one read/write head per cylinder. It can be assumed that rotational positioning accounts, on the average, for one half of the revolution time. For disk types having a headset, $n \geq 2$ per cylinder, the required time is the n th part of the previous.

Since the total revolution time is usually 5–25 milliseconds, the average rotation time in accordance with the above is 1.25–12.5 milliseconds.

- d) *Data Transfer.* The transfer speed of modern magnetic disks is generally between 150–3,000 kilobytes per second.

4.3. Page Structure

Modern computers store data in the main storage on so-called pages whose size is commonly 1–4 kilobytes. The unit of storage on magnetic disks is the block. This is the physically-connected quantity of data that is entered into the main storage with one input/output operation, or saved from there to the disk. Block size can be fixed or variable, depending upon the disk type. Yet in every database management system one page occupies one block (or consists of a whole number of blocks if the block size is fixed by hardware). In the following no distinction will be made between pages and blocks. Furthermore, it is assumed for simplicity that each data base page is equivalent to one physical block, and that it can be identified unambiguously either by the address of the page or by that of the block. At the hardware level, this means that it is specified on which unit, cylinder, track, and block, the page can be found.

At the physical level, data are stored in records. Records can consist of, for example, data of individuals in a personnel file, to which, among others, the following qualities or attributes belong: name, address, occupation, year of birth, etc. with the following values: Glenn M. Schwartz, 534 N. Calvert Street, Chicago, mathematician, 1972, etc.

Each record consists of one or more elementary fields or data items. The elementary *field* or *data item* is the smallest data of named data.

The content of a field is the actual value of the given attribute; it may be constant, but it can also change due to processing.

At least one of the fields within the record has a special purpose; it identifies the record. That field is called a *key field* or, in short, a *key*. Two key types can be distinguished, though the two often coincide with each other in practice. *Order keys* are used to enter records into the data base, while *search keys* serve to find one (or more) record(s) in the data base. Depending on the nature of the processing, different fields can serve as a search key in the same record. A vehicle registration file is an example of the difference between order and search keys. Whereas the data are arranged in the file according to the names of car owners, the drivers who committed an offense have to be searched, usually based on the automobile license plate.

The order key — if it identifies the record unambiguously — is often called a *primary key* as well. The search keys differing from it are called *secondary keys*.

The operations that can be executed on the record types, on their occurrences, or on their individual fields, respectively, can be divided into three main groups.

- a) *File Handling Operations. Creating, deleting, and reorganizing record types* belong here. The essence of the first two operations is self-evident. The third one is needed when the existing data structure can no longer satisfy users' requests with adequate efficiency due to modified demands or changes in the physical data storage structure as a result of maintenance operations. In this case — if there is a way to do this — with another kind of physical arrangement of the same data, a different physical data structure is created that is usually more efficient from the viewpoint of storage space or access time, or both. The data definition language is used for the first two types of operations, and generally, the utility program of the database management system for the third one.

b) *Alteration (Updating)*. This means the *entry* of new records into the database, in other words, *enlarging* the file, or *deleting* existing records from the data file. *Record modifying* operations changing the order or search keys of the record also belong to this category.

Likewise, it can also be considered a maintenance operation when new pages are added to the data base.

c) *File Processing*. *Searching* for records meeting the current requirement or requirements, and *altering* the content of those records belong among these. However, this modification cannot affect the value of any key.

Several kinds of record types can be found in a data base. So, for example, in addition to the record type that is characteristic of persons, a personnel data base can also contain record types which describe the departments and various activities. At the physical level, complete records are always transferred, and input/output operations move a page(s) between the disk and the I/O buffer area of the main memory which includes the record(s).

If the logical record does not coincide with the physical one, then the database management system constructs it for the user from the physical record(s) according to the rules specified by the data definition language.

At the physical level, the physical record is identified unambiguously by its address, which is composed of two parts:

- the address of the page on which the record is located, and
- the address of the record inside the page. This is usually the distance of its first byte from the beginning of the page.

A typical addressing method can be seen in Fig. 4.3.

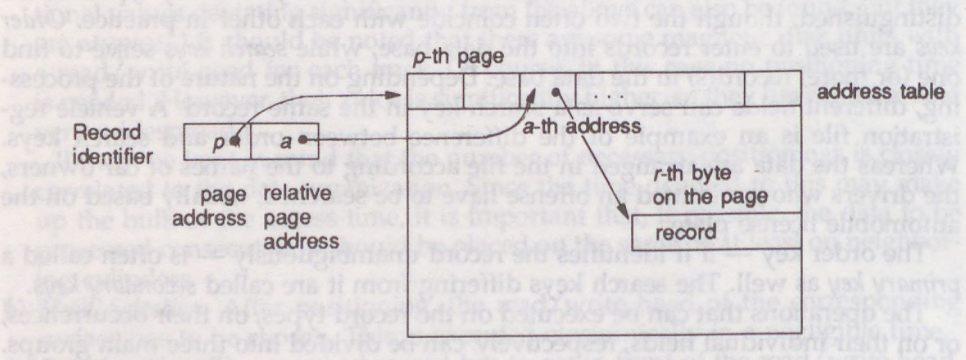


Fig. 4.3. Addressing at the physical level. The address identifying the record is composed of two parts. The first part is the page address (p). The second part usually points to the element of the address table in the given page (a) containing the address of the starting byte of the record within the page (r)

Data present on a page in the buffer area can be accessed faster, generally by 3–5 orders of magnitude, as compared to first having to read the page into the memory. That is why, if possible, the storage structure should be designed so that data used simultaneously or consecutively are on the same page, thereby decreasing the number of slow Input/Output operations. Database management sys-

tems usually store several pages simultaneously in the buffer area of the main memory, and keep a record of them also. So if a record is needed, they first check whether the appropriate page is in memory, and only if not do they load it from the disk, overwriting in the buffer one, temporarily not used page.

However, it is not enough to minimize the number of magnetic disk I/O operations. The execution time of I/O operations depends to a very great degree upon where pages to be read in or written out consecutively are located on the disk, relative to each other. If they are on the same track consecutively, then in one disk rotation all the records of the track can be read in or written out. The situation is not so favorable if they are only on the same cylinder. In this case there is a pause between reading individual pages, usually for $1/2n$ disk revolution, while the record gets under any one of the n read/write heads. The worst case is when records are on different cylinders. In this case the wait time increases along with the positioning time. That is usually between one and a quintuple of one rotation time, depending on the distance between the two cylinders.

Thus, for example, if there are twenty tracks on one cylinder and five pages (blocks) on one track, and if the page reading order agrees with the order in which blocks consecutively pass by the read/write head, then reading one hundred pages requires twenty rotation cycles, with a fast disk, about 100 milliseconds. In this case, it can be said that the data access order corresponds with the physical storage order. If this agreement does not exist, then in the most unfavorable case, it is necessary to reposition when reading every new page. So reading one hundred pages would be one hundred times longer than the average access time. This would be altogether between one or two seconds using a fast access disk whose character is similar to the foregoing, so larger by an order of magnitude.

Summarizing the above, it can be stated that the following guidelines have to be taken into account when deciding the physical storage of data:

1. Records that are expected to be used simultaneously or consecutively should be on the same page, if possible. The number of I/O operations is thereby minimized.
2. If the number and/or size of records needed for the processing is so large that several pages have to be read in and/or written out simultaneously or consecutively, then an effort must be made to store these pages so that their disk storage order matches the reading/writing order. The execution time of individual I/O operations is thereby minimized.
3. The length of individual records should be minimized. A redundancy-free record structure should be created (in practice, only with the necessary minimal redundancy), or the length of the actual occurrences of individual record types should be changed so that only useful data are stored in them. (For example, the length of the address field is 50 bytes. If an actual address occupies only 20 bytes in a given record, then 30 empty bytes at the end of the address field could be left out. Technically, this is usually achieved in the database system by declaring the field (and the record as well) to have a variable length. Therefore when storing the field, a two byte field describing the actual length is placed before it, and the empty bytes at the end of the field are omitted. In

The synonym for direct access is random access.

this example, the saving is 28 bytes, concerning this field more than 50%.) In this way, more records find room on the same number of pages. Thus the number of I/O operations is reduced.

In view of the above, in connection with speeding up physical I/O operations, it is necessary to introduce the concept of *clustering*. Its principle is that, if possible, those records expected to be processed together or consecutively are placed physically on one page, or, if this cannot be realized, then on pages which follow each other according to the processing order. If, for example, a record type is processed regularly using one key order, then it is expedient to enter these records into the data base in order according to this key. Consequently, records which are close to each other according to the logical key will also be close to each other physically. (Directly after loading, records logically in succession will be located physically consecutively.) In this instance, it can be said that the given record type is clustered according to this key. So during the most frequent processing — executing sequential processing according to the clustering key — paging is kept low.

Clustering can speed up processing even when it is present among different record types. When processing hierarchically, for example, when every individual order has a record of order-head type and several of order-detail type are connected to it, and during processing the head is processed first, and afterwards in turn its detail records, it is also advantageous to store these different record types on the same page.

The database management system can promote clustering if the data base administrator specifies the guidelines according to which it has to be realized. At initial load he loads records so that they will cluster. When entering new records it is attempted, if possible, to insert them on that page(s) which retains clustering.

For a given record type or record types related to each other, clustering can be realized only according to one key or one relationship, respectively. (Of course, by chance it can happen that this also means clustering according to another aspect.) One significant advantage of hierarchical models is that they can handle effectively and simply clustering based upon the relationship between different record types.

Naturally, as more and more new records are inserted clustering may cease. Or users' habits or demands may change, and now they most frequently access data by a different key. Of course, this does not alter the operation of the data base, but it can considerably increase processing time. In this case, the task of the data base administrator is to restore clustering by reorganizing the data base. This means saving the relevant data and then reloading it in the order of the key, according to which data are to be clustered.

This task, as well as others, can be executed relatively simply by using existing utility programs included in most database management systems.

4.4. Access Methods

4.4.1. Addressing

In modern database management systems, multi-level addressing and, on the basis of addressing, a multi-step access path are applied also at the physical level. Data are stored on data base pages (disk blocks), and the address of individual records is composed of two parts, the page address (block), and within this page, the record address. Records at the conceptual level are mapped on physical pages in a way determined by the data definition language. Therefore it is always known on which pages records of a given type can be found. Moreover, it is also often known, depending upon the mapping method, exactly on which page or possibly exactly in which location of that page a given record should be searched for. The page address always refers to a block of a track on a cylinder of a disk unit, while the address inside the page is the distance from the beginning of the page of the record's first byte. However, at the user level, the data base is searched not according to the physical addresses, but according to the data content. It must be known where to look — on which page(s) and where within it — for a record of a given type, or going into further detail, for a given key. The mapping system between the conceptual and inner levels translates this content-based search into a physical address.

In most cases, this address transformation is unambiguous. The record is definitely found at the definite physical address, if in the data base; or it has to be stored there. However, it can also happen that it defines only a starting point, whence, following a given search algorithm, the record is found, or it is determined that it is absent.

4.4.2. Basic Access Paths

Some of the methods discussed hereafter are suitable for locating pages, as well as record addresses inside the page. Therefore they are discussed on the basis of the same principle. In this case, the wording 'data access' or 'addressing' is used. When the procedure can be used only for locating either pages or records, then this will be mentioned separately.

The data organization and access path can be:

- serial, i.e., based on physical order,
- sequential, based on a given logical order,
- direct¹ (indexed),
- direct (hashing), and
- hierarchical.

Generally, the relational models use only the first three methods; they will therefore be discussed in more detail. As for hierarchic arrangement and hashing, only their essence will be discussed briefly.

¹ The synonym for direct access is random access.

It should be noted that even for the same key, data can be organized and accessed differently in the same data base.

Any data base or any part of it can be processed serially, based upon the physical order. At worst, the efficiency will be very low in most cases.

In addition, most database management systems enable sequential processing of the same data, following a given logical order of the key fields. They also permit direct, indexed access using the same key when the record address can be gotten directly from the key value.

4.5. Serial Access

When storage is based on physical order, there is absolutely no relationship between the keys and the addresses of the stored data, nor between the consecutive data keys, nor between any other features of the data. The data follow each other continuously (perhaps they split into two or more groups and within each group remain continuous as a result of insertion by different types of data) on the storage device. Only serial processing, following the physical location, is possible.

It is simple to create and maintain serially-organized data bases. New data must be written at the end of the data base. As there is no relationship among the different data, any of them can be deleted or modified without influencing the data base structure. When data modification does not enlarge the length of the data, then the data can be changed in place. Yet to search for data having a defined key value is very circuitous. In an unfavorable case, the whole data base must be examined.

It is obvious that in a serially-organized data base, in order to find a record with an individual key value (it occurs only once in the entire data base), on the average, half of the records must be examined. When the given key value does not exist or more than one can exist in the data base — and all of them must be found — then the entire data base must be examined from beginning to end.

For this reason, if only a small part of the data is needed, then this kind of processing is fairly slow due to the large number of comparisons. It will be seen that using the data organization methods reviewed below, the number of comparisons can be decreased significantly, thereby markedly reducing the search time. Yet on the other hand, because the rotation and positioning times are generally zero during serial page processing, access of consecutively-stored data is usually quicker to a large degree. For processing records stored on the same page, the time saved is not so large or is often negligible. However, the simplicity of the processing is still a benefit in this case. A very simple algorithm is used to locate the next record; it involves only increasing a counter by a fixed value. So the processing time, which benefits from a smaller number of comparisons, decreases in practice only when there are far less data to examine in a non-consecutive manner.

For this reason, serial processing is used in those cases where a significant part of the data must always be accessed. Working all the time in this way, it is not worthwhile to create any other data structure except for the always available

serial one. Saving data for security reasons and reconstructing the data base from the thereby saved data is handled mostly with a serial, structure-free file.

An additional benefit of this data organization and access path is that instead of magnetic disks, much cheaper magnetic tapes and cartridges can be used for storage.

4.6. Sequential Organization and Access

In sequential organization, there is an unambiguous logical relationship between the search keys of the consecutive data. In ascending order, all elements' keys are larger (or equal, if duplicates may occur) than the keys standing before them (in descending order, they are less or equal). In sequential access, after any data only the directly consecutive one can be accessed. (In some systems, the directly previous one can be accessed, too.)

In order to fill up an empty sequential file or data base, the data to be loaded must first be ordered by search key and then entered in this order. Building up a sequential structure, inserting new data into, and deleting them from an existing one, will be discussed in Section 4.6.1.

Several sequential access paths based upon different keys can be created for the same data. This is important to access already-ordered records according to other aspects (for example, based on family name, date of birth, or place of work).

The unambiguous relationship among the search key values of the same field does not mean that the actual physical storage order is identical to the logical one. If data are retrieved often based upon one key, then it is also expedient to realize this to a large degree, to exploit the benefits of serial access, too. Naturally, if sequential structures are built on several keys, then the physical order can be realized only for one of them.

With sequential page access, the physical and logical order match is important because processing consecutive pages minimizes the rotation and positioning times.

With records it is not so important to achieve total or near total match. The concept of clustering introduced in Section 4.3. (a lighter form of matching of the two types of ordering) accelerates the process to a large degree. In this case, the logically (sequentially) consecutive records are written preferably to the same page. So in sequential processing paging is rarely necessary. This decreases the number of I/O operations and the number of pages to be kept in the buffer.

Naturally, clustering can be realized for only one key for a given record type. This should be the most frequently used key in the sequential processing. In order to realize this, the records are first arranged in sequential logical order, and then entered in this way in the data base. The sequential processing will thereby be essentially faster. The database management system will try to store new records — as long as there is room for them on the page — according to the clustering criteria. Processing slows down significantly only when so many new records have been inserted or search keys modified that clustering disappears or weakens. In this case, by unloading and reloading the given record type in the proper sequential order, clustering and quick processing can be restored.

To accomplish this, most database management systems include utility programs.

4.6.1. List Structure

Logically consecutive pages, logically consecutive records of a given record type, fields of the same record stored in different locations, and records of different record types which are hierarchically arranged in a data base can be connected by list structure.

In the list structure the logically consecutive elements are elements of a sequential structure, but there is no relationship between the actual physical location and the key. The logical relationship among the data is defined by pointers, and these enable sequential processing, too. Every data element, otherwise called a cell, is composed of two parts:

- data part, which includes the actual data, and
- pointers containing the address of the cell(s) connected logically directly to it.

The usual terminology does not distinguish between the data part and the cell. In a loose sense, the whole cell is called data.

In the pointer field, the address of the connected record can be given in three forms:

- as an absolute hardware address,
- as a relative address; in data bases, for pages it is usually the relative address of the next page counting from the beginning of the data base, and for records, the relative address of the page containing the next record and its relative location on that page as discussed in Section 4.3, and
- as a symbolic address, containing the order key of the record that logically follows. From this the system defines the actual physical address.

The majority of current database management systems utilize the second method, relative addressing, if list structure is required at all. The relative address is transformed into an absolute address by the operating system.

Furthermore, in the following figures, relative or symbolic pointers are usually given for clarity, but it does not mean that only such addresses are expedient to use for actual pointers.

A simple list structure is shown in Fig. 4.4. The leftmost field of every element (cell) contains the key value of the field defining the sequential structure. In the shaded area, there are irrelevant data regarding the list structure. The rightmost field contains the pointer, the address of the logically next data element. Using this pointer, it is possible to proceed sequentially through the elements of the list in a logical sequence. That is why this data structure is called sometimes *chained list*. The last element of the chain is marked with a special sign (on the figures *) in the pointer field.

Naturally, to process a list its beginning or anchor point must be found. Therefore the so-called starting index table which maintains a pointer whose purpose is to show always the first element of the list is usually in a given place. However, in order to insert new elements rapidly among the already existing ones it must be known — without starting to scan the entire storage device — where can be found a free location to store physically this element. Thus another list, the list

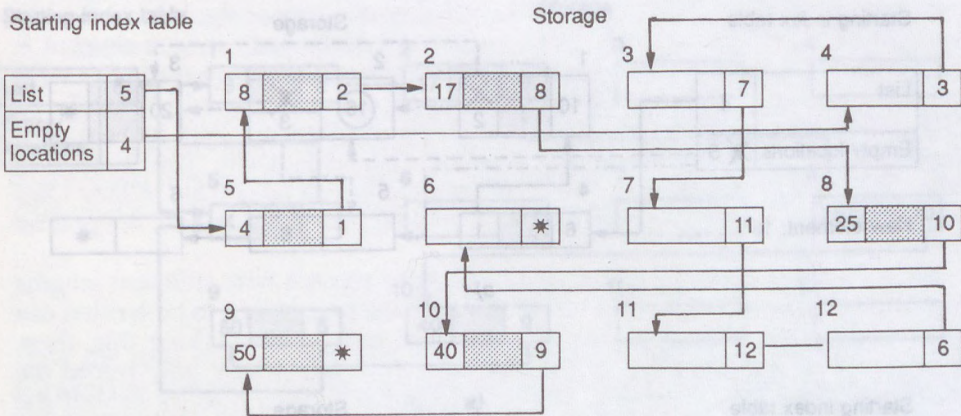


Fig. 4.4. Realization of list structure

of free places, is prepared. This includes in chained form the addresses of all momentarily empty storage locations.

A new element can be inserted in the list by storing it in the first element of the empty list whose address can be gotten from the starting index table pointing to the beginning of the list of free locations.

Naturally, the pointer indicating the first empty location in the list must be updated next. The original value, which was formerly in the pointer field of the just-entered cell and pointed to the second element of the empty list, is inserted in the starting index table. The physically thus defined element's new logical place is defined based on its key, finding it by traversing the chain sequentially. After the insertion, the value of the new pointer will be the content of the pointer field of the logically previous element. The pointer of this previous record will be changed to the address of the inserted new record. Similarly, the deletion of an element which is in the list will trigger the modification of its own pointer, the previous element's pointer, and the free list starting place pointer because the deleted member will be placed at the top of the empty list. The insertion and deletion processes can be seen in Fig. 4.5. If the records are inserted into the data base (or deleted from there) with the help of the data manipulation language, then the database management system automatically completes the updating of these pointers.

In practice, it often causes a problem when the cells include only the logically next element's address. If, for example, the process is started (as well as continued) not from the beginning of the list in order to process data which is present there, then from the end of the chain the way back to the beginning of the list cannot be found automatically. A more serious problem arises when the element to be deleted is not located sequentially by the chain. Then the location of the logically previous element whose pointer must be changed as a result of the deletion is unknown. This latter problem can be surmounted if only one chain goes through the data. The content of the cell following the deleted cell will be trans-

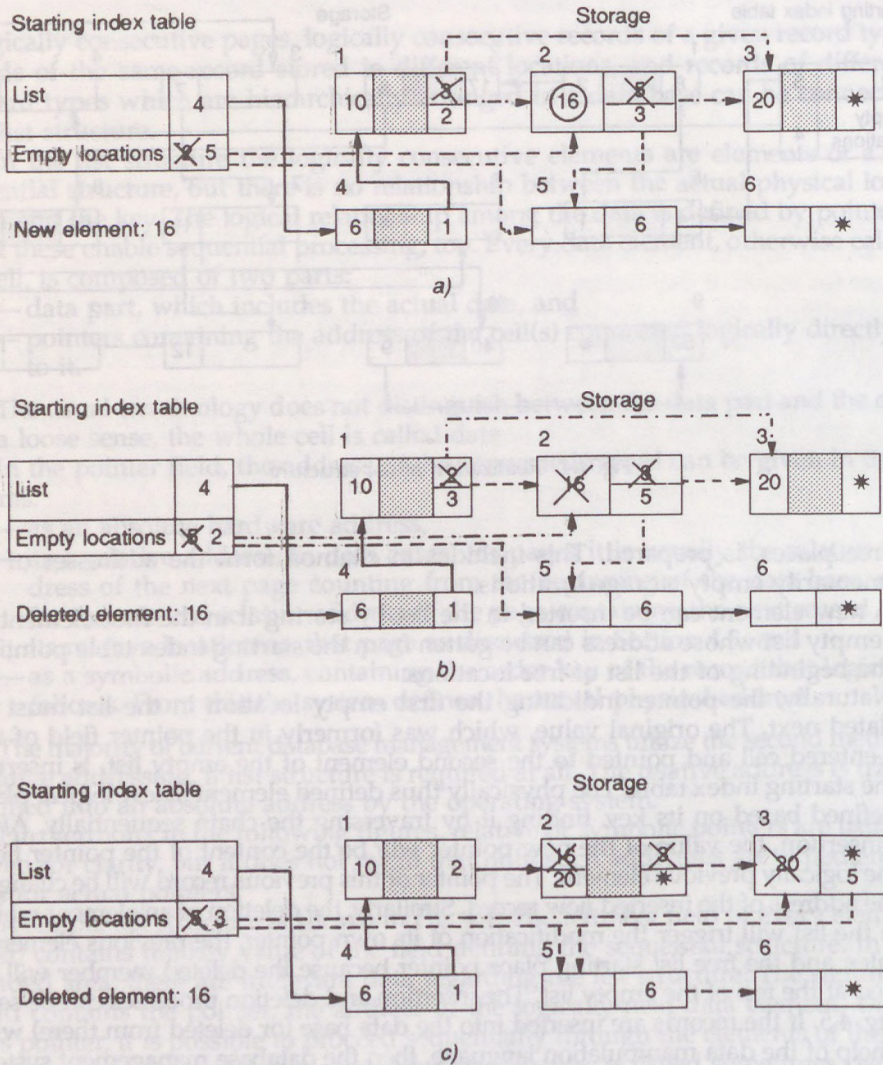


Fig. 4.5. List modification

(The broken line --- indicates the status before the modification.

The dotted broken line -.- the status after it.)

a) Insertion of a new element; b), c) Deletion of an existing element

ferred to it, and once emptied, this cell will be placed at the beginning of the empty list (Fig. 4.5.c). However, this method cannot be adopted for the last element of the list, or in those instances where the deleted cell is simultaneously an element of other lists.

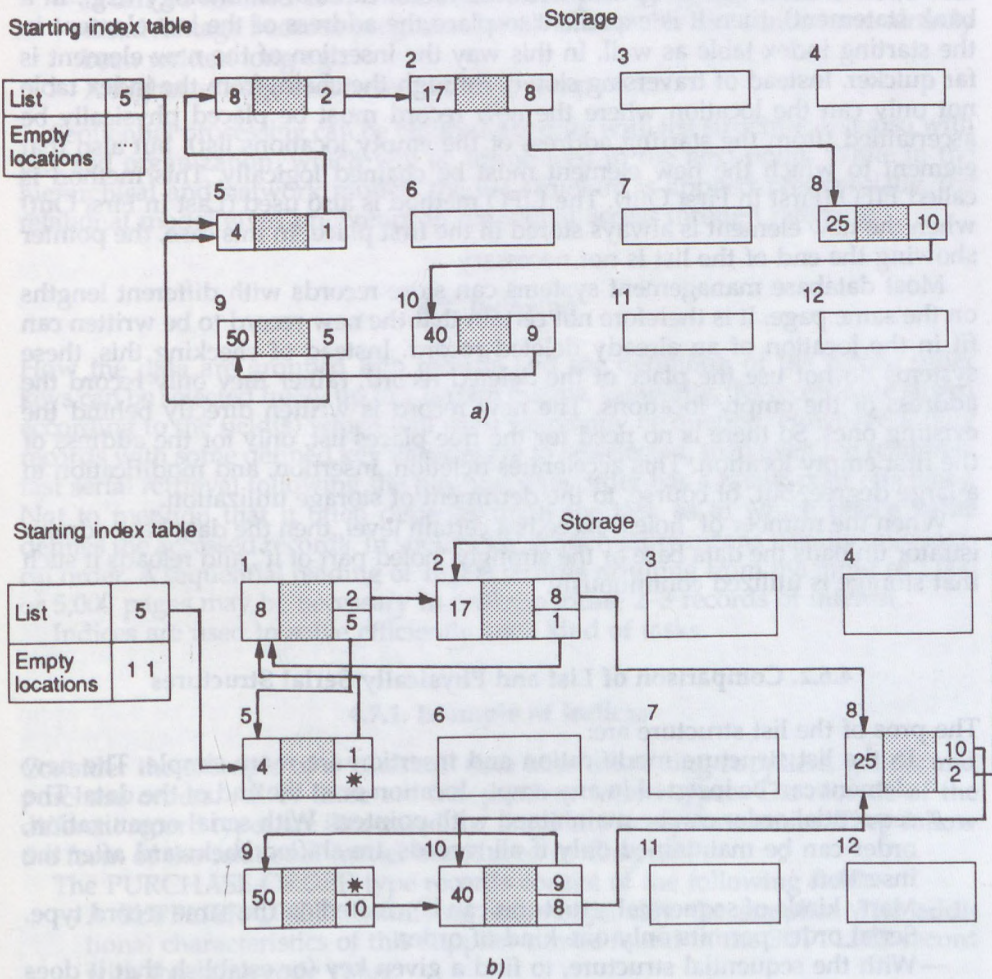


Fig. 4.6. a) Ring b) Double-chained list

(The pointers connecting the empty cells were not shown in the figure for brevity.)

These problems can be overcome by increasing the number of pointers. The two most frequently used expanded list structures are the ring and the double-chained list (Fig. 4.6.).

With the *ring*, the pointer of the last element points back to the beginning of the list. So the list can be traversed from any starting point. In the *double-linked list*, all cells include two pointers. The address of the logically previous element is given by the first, the other notes the next element. The list can therefore be processed sequentially in both directions, and deletion is faster, too.

Insofar as the order inside the list is not interesting (e.g., the list fosters searching based on a secondary key and connects the elements possessing the same key

value), or the order is congruent with the modifications' chronology (e.g., in a bank statement), then it is expedient to place the address of the last element in the starting index table as well. In this way the insertion of the new element is far quicker. Instead of traversing slowly through the chain, from the index table not only can the location where the new record must be placed physically be ascertained (from the starting address of the empty locations list), but also that element to which the new element must be chained logically. This method is called FIFO (First In First Out). The LIFO method is also used (Last In First Out) when the new element is always stored in the first place. In this case, the pointer showing the end of the list is not necessary.

Most database management systems can store records with different lengths on the same page. It is therefore not certain that the new record to be written can fit in the location of an already deleted record. Instead of checking this, these systems do not use the place of the deleted record, rather they only record the address of the empty locations. The new record is written directly behind the existing ones. So there is no need for the free places list, only for the address of the first empty location. This accelerates deletion, insertion, and modification to a large degree, but, of course, to the detriment of storage utilization.

When the number of 'holes' exceeds a certain level, then the data base administrator unloads the data base or the strongly 'holed part of it', and reloads it such that storage is utilized continuously.

4.6.2. Comparison of List and Physically Serial Structures

The pros of the list structure are:

- In the list structure modification and insertion are more simple. The new element can be inserted in any empty location or at the end of the data. The sequential order can be maintained with pointers. With serial organization, order can be maintained only if all records are shifted backward after the insertion.
- Many kinds of sequential structures can be created on the same record type. Serial order permits only one kind of order.
- With the sequential structure, to find a given key (or establish that it does not even exist) a given key, on average, half of the records must be examined. With the serial, this is true only when the searched key value occurs precisely in one record. In all other cases, every record must be examined.
- With the help of lists, other data structures can also be realized on the same data. If the pointer chain connects records that occur in the same hierarchy (e.g., the customer's master record and his orders), then it allows a hierarchic access path. If a pointer chain to all possible values of a field is built up whose first element can easily be found, then this is essentially an implementation of the indexed organization which will be discussed in Section 4.7.

The cons of the list structure are:

- Processing, especially deletion and insertion are slower, but for not clustered keys the processing time can increase dramatically, even for retrieval.

- It can be realized only with a more complex program. (This is not a real disadvantage because programs for all imaginable situations have already been written long ago.)
- A higher storage requirement due to the pointers.

Sequential processing can be realized not only with list structure, but also with indexed organization, which will be introduced in the next point. While in the hierarchical and network models the list structure is applied preferentially, the relational model gives preference to sequential access through indexing.

4.7. Indexing

How the data are grouped into records and the definition of the record order keys can be selected freely up to a certain degree. Generally, the records are stored according to the field(s) which will most frequently be queried. However, if only records with some defined key value(s) are to be selected, then even the relatively fast serial retrieval following the physical data order has a rather poor efficiency. Not to mention that it often depends upon the task as to which field's value defines the accessed records. In such cases, sequentiality means usually only logical order. A sequential reading of 10,000 records resulting from the input of 1,000 or 5,000 pages may be necessary in order to locate 2–3 records of interest.

Indices are used to solve efficiently such kind of tasks.

4.7.1. Example of Indices

Consider the example of a 'classical' data base containing suppliers, goods, and purchase orders. All of these are independent record types. The records of the different record types are illustrated for simplicity in separate tables. They follow the form of the relational model introduced in Chapter 7.

The PURCHASE-ORDER type records consist of the following fields:

- A SUPPLIER-CODE identifying unambiguously the supplier. The additional characteristics of this supplier can be found in the SUPPLIER record with the same code value.
- A GOODS-CODE identifying unambiguously the ordered goods. The additional characteristics of the goods can be found in the GOODS record with the same code value.
- DATE-1, the purchase order date,
- DATE-2, the delivery date, and
- QUANTITY, the ordered quantity in the measure characteristic of the goods (stored in the related GOODS record).

From the data base, quick responses must be given regularly to the following type of questions:

1. What are the standing orders of a given supplier (code)?
2. What are the standing orders of a given goods (code)?
3. What orders are placed in a given day?
4. What orders arrive in a given interval?

After the order records are loaded into the data base, the first question can be answered easily once an auxiliary table consisting of two fields has been prepared. Its first field and in the meantime its key is the SUPPLIER-CODE. The second field contains the address of the PURCHASE-ORDER record, having SUPPLIER-CODE equal to the first field. If there are several purchase orders for one SUPPLIER-CODE, then this code occurs in the auxiliary table as many times as in the orders. Every address field points to another purchase order (Fig. 4.7.). (If a SUPPLIER-CODE figures in very many PURCHASE-ORDERS, then this table can be organized in a different way as will be touched upon in Point 4.7.3.)

The table which includes the *value* of the SUPPLIER-CODE and the *address* of the PURCHASE-ORDER record is called *index table*. It is said that purchase order records, the PURCHASE-ORDER record type (table), are *indexed by the SUPPLIER-CODE*.

Orders from a given supplier can be retrieved by searching for the suitable code in the index table. Based on the address(es) found there, the PURCHASE-ORDER record(s) of the given SUPPLIER-CODE will be retrieved.

This is essentially faster than retrieval of all the PURCHASE-ORDER records and individual examination of the SUPPLIER-CODE's values. It stems from the index table not only being sequential, but also having a physical order consistent with the search order. In addition, the index table is far smaller than the PURCHASE-ORDER table. Besides this, as will be shown in Point 4.7.4., it is not necessary to read through sequentially the index table up to the selected supplier code. With the help of further indices built upon the index table, any supplier code and its address(es) can be found in a few steps.

To answer the questions of type 2, 3, and 4, similar indices can be built upon the GOODS-CODE, DATE-1, and DATE-2 fields of the PURCHASE-ORDER records.

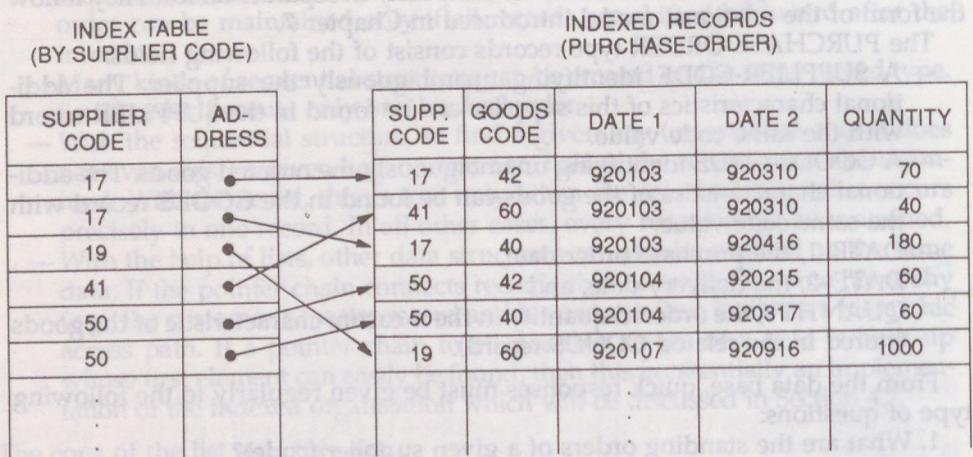


Fig. 4.7. Indexing PURCHASE-ORDERS records by DELIVERY-CODE

4.7.2. Indices

To generalize from the above, indexed data (record types and tables) can be divided into two parts:

- the actual data part, and
- the index table(s) including the suitable record address(es) for each value of the indexed fields.

Indices can be built on every field or every field combination (Point 4.7.6.) of any record type (table). Several index tables can belong to one record type.

The index built on the order or the primary key is called *primary index*. Indices built on other fields are called *secondary indices*.

If there is only one entry in an index table for any value, that is to say, a record with a given index value occurs a maximum of once in the indexed record type, then the index is *unique*; if more are permitted, then it is *duplicate*. The index built on the primary key is always unique.

It follows from the logic of our example, that indices built upon the fields SUPPLIER-CODE, GOODS-CODE, DATE-1, and DATE-2 of PURCHASE-ORDER's record type, are all duplicates. However, if for the purpose of quickly finding the suppliers, an index is built upon the SUPPLIER's record type, based on the value of the SUPPLIER-CODE, then it will be unique.

It is superfluous to build an index for the QUANTITY field. Questions like these 'From which goods were 50 or 100 gallons ordered?', or 'Enumerate the orders in increasing quantity.' — are unlikely to be asked.²

From the above-mentioned, it can be seen that in different tables fields with the same logical meaning can occur in different ways as indices.

If in the index table, the key values are ordered in increasing order, then the index is *ascending*; if in decreasing order, then the index is *descending*.

If in the index table, the values are defined by the value of only one field (key), then the index is *elementary*; if the values are defined by the combined values of several fields, then the index is called *composite*.

The previous examples were all elementary indices. A typical example of the use of composite indices will be shown in Point 4.7.6.

The index table enables the sequential retrieval of indexed data even if the data are not in sequence. Retrieval consists always of two steps. First, the index table is read sequentially; then, based on the address found there, the indexed record which belongs to it is read.

In the sequential read command of the data manipulation language (Point 5.3.3. and Section 6.7.), naturally, it does not matter how and in how many steps does this operation take place at the physical level. The database management system completes all of these. Of course, the database management system can use only indices defined previously by the data base administrator. If he or she

² It is conceivable that orders for a given good must be displayed in order by the quantity ordered. But in this case, the records are selected at first based on the GOODS-CODE, and the quantity order is interesting only within any selected group. The index constructed only for the QUANTITY field refers to an order of magnitude more records, so neither for this reason would it be useful to rely on this index alone for this task.

neglects to do this, even though that would be more efficient, then the system will not define and construct indices.³

By indices not only can records with given key values be retrieved, but also those falling into a given key domain. As an example, the 4th question of Point 4.7.1. can be answered in this way: the index table of the delivery deadline is read sequentially from the starting point to the last time point; all purchase orders belonging to the addresses found in the desired range are retrieved.

Some questions can be answered directly from the index table without examining the data. Above all, these are existence-like questions or questions concerning only the number of records which meet a given condition. For example, 'How many orders are there for a given day?', or 'Are there any orders for a given goods code?', etc. The efficiency of the indexed organization is more evident for such kinds of questions.

The construction and use of indices accelerates search. But, of course, there is a price to be paid. This is examined next.

The first is additional storage. The more fields or field combinations are indexed, the more space is needed for the index tables. Roughly it can be said that in an efficient data base, the index area is about half the size of the data area.

The second disadvantage manifests itself when modifying, deleting, or inserting. (Modification means here the change of an indexed field.) Namely, in this case, not only must the operation be completed on the data record, but all relevant indices must be updated.

If, for example, the delivery deadline of an order is modified from 101693 to 110193 in one of the PURCHASE-ORDER records indexed by this field, then the data base management system

- notes the address of the modified record,
- searches in the DATA-2 index table for entries equal to the 101693 value,
- deletes from these the entry containing the address of the modified record, and
- inserts a new entry into the relevant location of the DATA-2 index table in which the value of the key is 110193 and the address is that of the modified record.⁴

4.7.3. Duplicate Indices

Most secondary indices are duplicates. The same key value may occur in several records. The solution of Point 4.7.1. whereby in all lines of the index table the key value is repeated, that is to say, each indexed data record is an individual entry in the index table, is efficient, provided there are only a few duplicates. However, if too many duplicates exist for many key values, say, several hundreds

³ Some of the most recent relational database systems can perform even this task. In certain cases, they make temporary indices if it seems that this will accelerate the solution of the task.

⁴ There are systems which insert only logically the new entry into the relevant place of the index table. While there are others – if feasible – which insert it physically in accordance with the logical order in the correct place. It means that the index entries which are behind it must be shifted further. In the former case, the process slows down after many changes; while the latter's update takes longer.

(for example, for an order or delivery date where many delivery services and goods are involved), then the constant repetition of the keys enlarges the index table redundantly. This problem can be solved in two ways.

The first is that the index table for every key includes only one address, that of the first record having this value. In the indexed record itself, the address of the next record with the same field value is stored as well and this serves as a pointer. The records belonging to the same index value form a chained list whose first address can be found in the index table. So the index table consists of as many entries as *different* values exist in the indexed field. The length of the chained list is equal to the number of data records possessing the same key value.

The price paid for decreasing the size of the index table is that the record length increases.

The second method results in indexed table entries of varying length. (Or their length is fixed, but can be chained if too many records have the same key value.) Behind the key will be as many addresses as data records having the identical key value. Here the trade off for the size reduction is that the handling of varying length index entries is more difficult.

The above-mentioned methods are illustrated by an example shown in Figs 4.8. and 4.9.

The example in Fig. 4.8. shows data indexed by several fields. From the register of university students, the students' records must be located and lists prepared based upon students' names, their years of birth, and the course attendees (for example, a name list of those students who study Physics or Christopher Robin's personal data). Records having the same key value are chained and the index table points to the first element of this chained list. All physical records are 100 bytes long. Since some students' data can be longer than this, a student's logical record may consist of several physical records.

Every student record starts with three mandatory key fields. The first includes the name of the student. The second is the abbreviation of the courses he/she attends, and the third is the last two digits of his/her year of birth. The first and the third fields are elementary while the second is a group field whose length is included in some kind of form — which is irrelevant in this respect — in the record.⁵ A pointer is attached to every key field pointing to the next record having the same key value. After the key, there are different data (address, former schools, marks, etc.) not used for search.

Based upon the following example, it is useful to show how different kinds of information can be retrieved from this data base.

To gather all information on Christopher Robin, first the student name index is examined, giving 1300 as the first record of Robin. (For simplicity, storage locations are numbered. In practice, instead of this, the actual cylinder, track, and record numbers are given, or by an algorithm, they are digitized into the physical address.) Starting from location 1300, all data about Christopher Robin can be found following the pointers of the name field reading the records at locations 2000 and 2400 (see Fig. 4.8.).

⁵ It should be noted that the relational model does not allow this type of record structure. To work with such data, records have to be restructured in a different way.

To obtain the names of students attending the Physics course, the relevant index table entry is retrieved from the course index table. It is learned thereby that the first student's data are at location 1100. From this, the student's name (Allen), and his

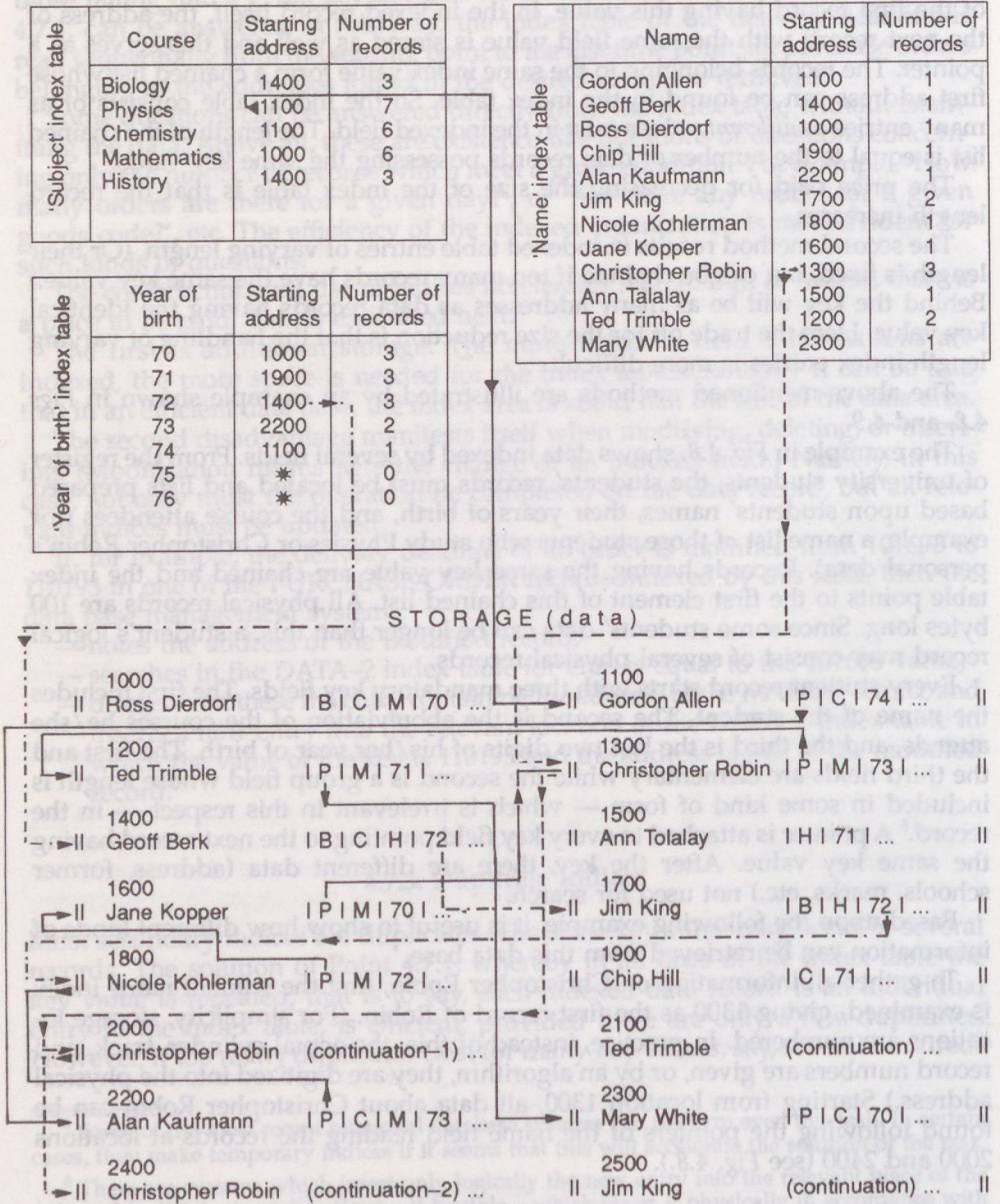


Fig. 4.8.
Handling of duplicate indices with a list structure

'SUBJECT' INDEX TABLE

Course	Record Address						Chain
Biology	1000	1400	1700	*			0
Physics	1100	1200	1300	1600	1800	2200	6
Chemistry	1000	1100	1400	1900	2200	2300	0
Mathematics	1000	1200	1300	1600	1800	2200	0
History	1400	1500	1700	*			0
Physics	2300	*					0

'NAME' INDEX TABLE

Name	Record Address					Chain
Gordon Allen	1100	*				0
Geoff Berk	1400	*				0
Ross Dierdorf	1000	*				0
Chip Hill	1900	*				0
Alan Kaufmann	2200	*				0
Jim King	1700	2500	*			0
Nicole Kohlerman	1800	*				0
Jane Kopper	1600	*				0
Christopher Robin	1300	2000	2400	*		0
Ann Talalay	1500	*				0
Ted Trimble	1200	2100	*			0
Mary White	2300	*				0

'YEAR OF BIRTH' INDEX TABLE

Birth Year	Record Address					Chain
70	1000	1600	2300	*		0
71	1200	1500	1900	*		0
72	1400	1700	1800	*		0
73	1300	2200	*			0
74	1100	*				0
75	*					0
76	*					0

Fig 4.9. Handling of duplicate indices with an index table consisting of rows with varying lengths. The storage structure can be seen in Fig. 4.8.

The special mark * indicates the end of the record addresses. The value differing from 0 in the chain column gives the relative address of the subsequent record with an identical key value in the index table

other data can be gotten. By navigating through the list about Physics, all the other students of this course will be reached. (Starting from addresses 2200, 1800, 1600, 1300, 1200, and 2300, Kaufmann, Kohlerman, Kopper, Robin, Trimble, and White.)

If there is a query about students born in 1972, then, based upon the 'year of birth' index, the first record reached is at location 1400 (Berk). King and Kohlerman at addresses 1700 and 1800 are attached to the list.

Figure 4.9. shows the index tables for the same data where the addresses of all records having identical key values are listed in the index tables. In this case, pointers are superfluous in the student records. Insofar as there is no more room in an index table entry for adding the address of a new record, then a new entry is chained to the full one, and the following addresses will be entered there (see the entry of the Physics course attendees in the figure). The records having the same key are alphabetically ordered according to the student's name in Fig. 4.8. They are unordered in the index table of Fig. 4.9. In the index table entries, the logical and the physical order are the same for both figures. These requirements can be ignored without any adverse consequences. Naturally, this organization accelerates search in the index table, but at the same time complicates the insertion of new records. Especially for this reason, the index table can be a chained list by itself, and for a few elements it can be unordered, too.

The above-mentioned duplicate handling methods are effective if there are many duplicates. Despite this, they are rarely applied in the relational model because it is difficult to fit them into the conditions arising from the conception of the model.

4.7.4. Index Table Search

Indices accelerate record retrieval. By getting in one or two steps via indices the page containing the searched record, the number of I/O operations is considerably reduced. By locating rapidly records on the page which was brought earlier into memory, the CPU time is reduced. As there is a 3-4 orders of magnitude difference between the two times, the most important factor is to minimize the paging I/O time. (It is true that a multi-programming computer during this time can execute other jobs. Yet that accelerates only the solution of other jobs, with no appreciable impact on the examined first job's run time.)

In large data bases, there are a lot of entries in the index table, too. Due to this, even locating just a given index and the attached addresses can result in the retrieval of several pages, and within those a longer search may be necessary.

For this reason, several methods have been elaborated. One of them is indexed-sequential organization which will be described at this point because it provides rapid, sequential, and direct access to the data. However, the precondition for this is that the order key order be identical to, or, at least, roughly similar to the search key order. The balanced tree (B^+ tree) in which this nearly identical ordering need not be realized will be introduced in Point 4.7.5. Accordingly, that can be applied to any disk storage structure.

When creating the file,⁶ the indexed-sequential organized records are loaded in ascending order. The file includes, besides the data, index areas which foster direct access. A simple, but clear example of the indexed-sequential organization

⁶ The methods introduced here were elaborated originally on files in database technology. Depending on the data model, similar record types are collected into tables (relations), sets, or segments as files. For unification and shorter notation, only the word 'file' will be used here, but in the more general sense, it covers all otherwise named record types.

method is the telephone book. Its first index is the capital letter at the side which shows that names starting with this letter can be found from this page. Because this letter is isolated in a truly visible way from the real information, names and telephone numbers, when searching for the telephone number of say, Stewart Hecht, the book can be opened immediately at the letter 'H'. Thus in the worst case, the names must be read consecutively starting with 'H'. As the number of these can be fairly large, another index serves to accelerate search. This is the page index.

The page index is also isolated from the real information. It can be found at the top of the page. It gives the first and/or last name on that page. The search thereby continues expediently, and instead of the complete name order, only the page index is read. By means of the page index the Hecht name is reached or it is passed by. Then name by name reading starts. So direct search is also possible in the indexed-sequential file.

Since new records cannot always be inserted at the place desired for them, if necessary, another area must be prepared. This is called the overflow area. Based on this, any indexed-sequential file consists of three parts whose sizes are established at the time of their creation. Later their sizes can be changed, but usually only during reorganization. The three parts are:

- the primary data area,
- the overflow area, and
- the index area.

The *primary data area* includes data loaded in ascending order during file creation. If insertion of new data is unlikely, then for the sake of efficient space utilization, this area is filled up fully with data, even at file creation. However, if it is probable that many new records will later be added to the file, then at first only a certain part of this area will be filled up (for example, just 60% of every page). New records added subsequently will be placed in the remaining space (at least in the beginning). This accelerates processing.

Overflow areas include records which do not fit, due to the presence of already-existing records, in the primary data area designated by their key. These records are organized in a chained list. The reference to the first element of this list is on the page to which the overflow record is assigned.

The *index areas* facilitate rapid writing of records into the suitable location of the data base, as well as retrieval.

The indices can have several levels in the indexed-sequential organization.

The fundamental index is the page index.⁷ Its key value is equal to the highest index value of that page; the pointer section gives the address of the page (Fig. 4.10.). The parts of the data areas which are kept free are available for the reception of the subsequent expansion.⁸ If the values of the index table in ascending order are $ik_1, ik_2, \dots, ik_n$ and the page addresses belonging to them are $a_1, a_2, \dots, a_n$, then by sequential processing:

⁷ In older systems, it was essentially equivalent to the track index. The index above it was similar to the cylinder indexes.

⁸ Certain database systems allow also that different types of data records can exist on the same page.

- the first element of the index table is read,
- all records of the page given by the index table will be processed in order, and
- the next element of the index table is retrieved and all the records on the page belonging to that element are processed sequentially until the end of the index table is reached.

Direct search means that the index table is traversed in order while $ik_{j-1} < k \leq ik_j$, where k is the key of the searched record. That record should be stored on the page a_j .⁹ So this page will be read, and it will be searched sequentially for the record. It is shown in Fig. 4.10. that, for example, by searching for the record with key 50, from the index table it can be established that this key falls between indices 25 and 52, so it will be on page 4, which contains maximal index 52.

In order to decrease positioning time, the index page and indexed records are on the same cylinder.

When there are so many page indices that they can fit only on several pages, then a new index level, the page index table, will be created as well. Its entries are the highest values of the particular index pages and the addresses of these index pages. This table is often called the master index table.

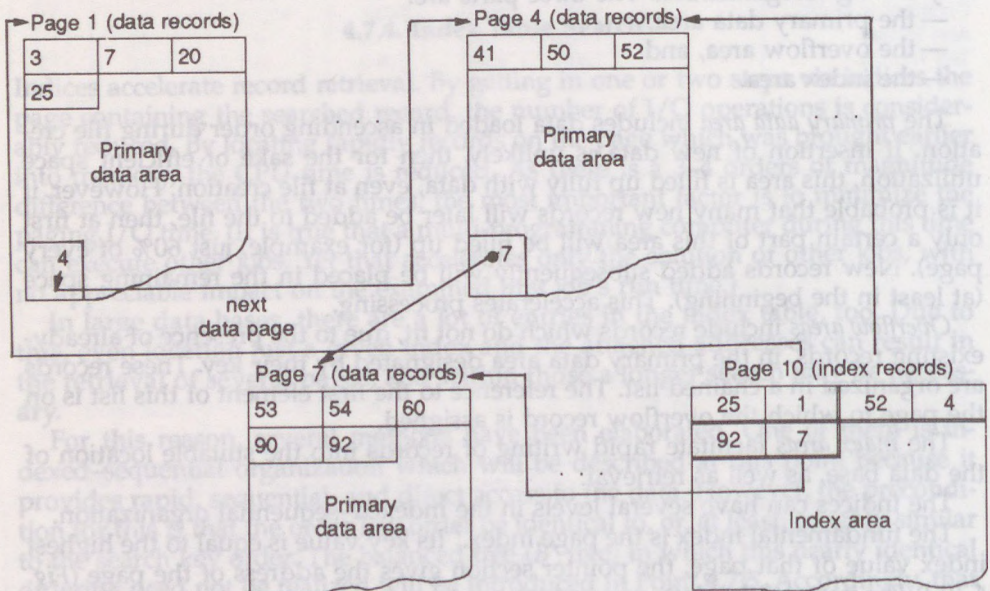


Fig. 4.10. Indexed-sequential organization. The indices are on page 10, the data records on pages 1, 4, and 7, arranged sequentially by a pointer chain. Every index value equals the highest index stored on the page under the pointer field address

⁹ Or as a consequence of the expansion, it gets to the overflow area. Nevertheless, it is still connected to this page with the pointers. That is why it is said that logically it is on the assigned page.

If, for example, the addresses of the index pages are $ia_1, ia_2, \dots ia_m$, and the highest indices on them are in order

$$ik_{i1}, ik_{i2}, \dots ik_{im},$$

then the entries of the master index are

$$(ik_{i1}, ia_1); (ik_{i2}, ia_2); \dots (ik_{im}, ia_{im}).$$

The search principle is the same as for single level index tables. However, in this case, from the master index table is chosen first which index table page to retrieve in order to find in it the page address containing the record.

By creating an indexed-sequential file, the data are loaded according to the key. At this time the system concurrently stores the data on the page and builds up the index table. After loading, the following operations can be executed on the records:

- sequential processing based on the key,
- direct retrieval and processing based on the key,
- adding new records and deleting old ones, and
- modification of existing records.

The first two operations were discussed earlier. The modification can be regarded always as the retrieval and rewriting of the suitable record, insofar as this modification does not affect the search key. If the key is to be modified, then the operation can be decomposed into the deletion of this record and the insertion of a new record whose key is identical to the modified key. The insertion of new records will be shown next. The deletion occurs essentially in a similar way with the exception that the place of the deleted record remains empty in many systems and fills up only if reorganized (of course, with a different record).

The insertion of a new record is shown in Fig. 4.11. At file creation time, the primary data area was not totally filled up, so some additional records can also be stored there without causing overflow. The record with key 5, for example, can be inserted trouble-free among the existing records after its place is found, based on the indices using the method defined in the above paragraphs. All records whose key is greater than 5 must be shifted to the right by one location. (In many systems, the records remain in their original place. The new record is written at the first free place on the page. An inner page index, which can be found in a defined location on the page, points to the individual records inside the page. The key of the new record, as well as its internal page address, will be inserted by the system into the corresponding inner page index fields.)

The next record to be inserted has the key value 16. However, there is no more room for it in the primary data area. Therefore, when it is inserted into its location, the last record (key = 28) on the track will be moved to the overflow area. Due to the overflow, the value of the overflow pointer of this page will be changed. However, the search index indicates the highest key value belonging logically to the page, and it will remain 28.

In order to insert the record with key 2, the record with key 16 must be shifted to the overflow area, occupying the next empty location there. It is therefore inserted after the record with key 28. Yet with the modification of the pointers, its logical order can be kept henceforward. The record with key 20 must be inserted directly into the overflow area. In this case, there is no need to shift any records.

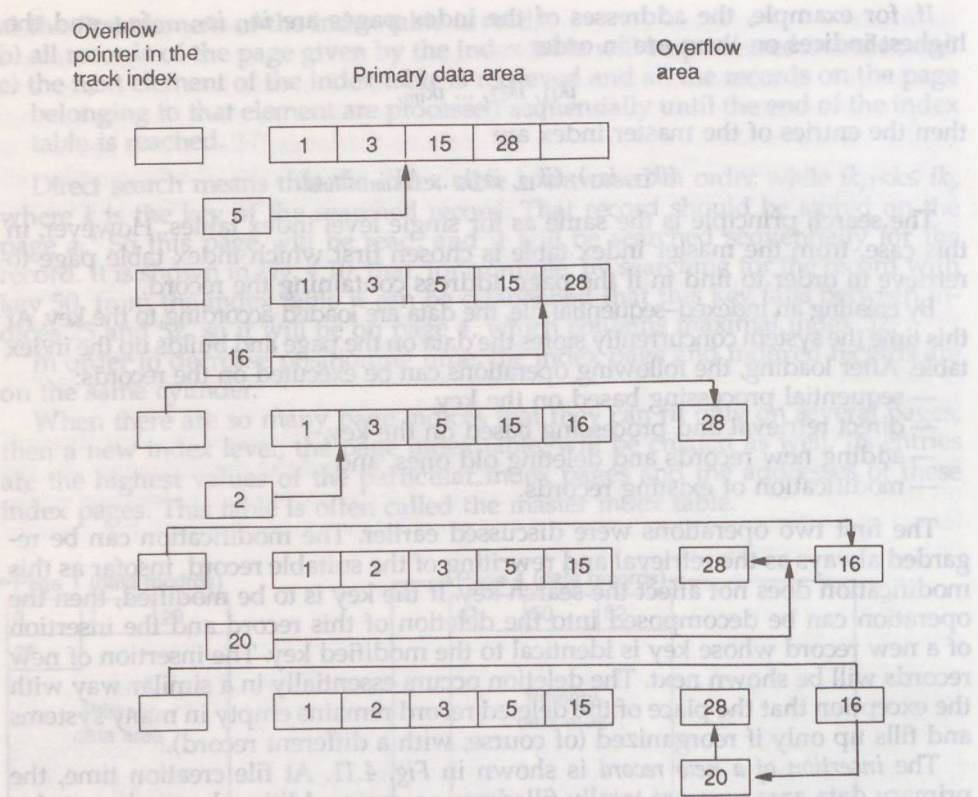


Fig. 4.11. Insertion of a new record into an indexed-sequential structure

Only the pointers of the chained list must be updated. If a record is deleted from the file, then its place will be freed up only if it was in the primary data area. (In some systems, even this is uncertain.) Hence in that case, instead of the deleted record, another record can later be written which logically belongs to the page.

The deleted records' locations in the overflow area can be used only when the data are reorganized. The deleted data are marked with a special sign symbolizing deletion. In spite of that, they exist physically, but do not take part in the logical process. It should be noted that the search keys entered into the index table do not change until the file has been reorganized.

Hence, for example, when the record with key 29 must be entered, then it will be written in any case on the second page even if there is place physically on the first page, but none on the second.

It is worth noting that with both sequential and direct processing, if there is an overflow, then the overflow records are found automatically by the system. Naturally, it increases the processing and access time, especially when overflow records get to another page. For this reason, the database management system monitors the number of different types of overflow records.

With the reorganization of the file, the original access times can be restored. The records are unloaded in key order. This file will serve as input to recreate the indexed-sequential file. After reloading, it will contain no overflow records.

4.7.5. B⁺ Tree

In relational systems, indexed-sequential access is realized by the B⁺ tree structure. By enabling efficient, sequential, and direct access, the B⁺ tree features the benefits of the indexed-sequential organization, but at the same time, overflow records do not hinder access efficiency and it can be applied on data stored in any physical order.

The index area of the B⁺ tree structure is divided into two parts:

1. *Sequential indices.* They are similar to the index table described in Point 4.7.2. They include the keys and addresses of the records with that key value. They are ordered by the key. During sequential access, these indices, better to say, the pages which are gotten from these, are examined. The pages containing the sequential indices are chained in a list for sequential access.
2. *B⁺ tree indices.* To provide rapid, direct access. In practice, it is a multi-level tree structure. Its operating principle is introduced in the example shown in Fig. 4.12.

For simplicity, the indices are examined only on one field of one record type. (The explanation can be applied in the same way for the further record types and fields.)

At the highest level of the tree, there is room for n_1 index (in this example, for two, with values 40 and 75). Their pointers point to the indices at the next level of

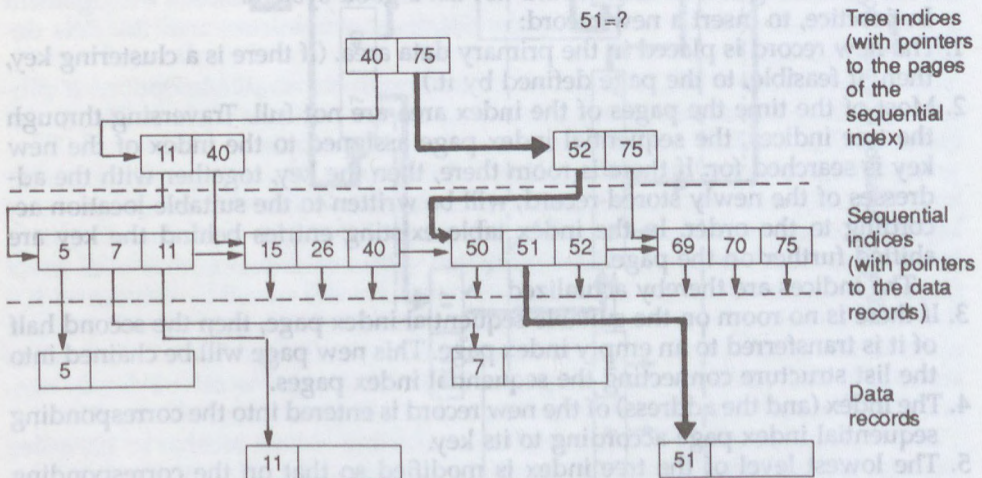


Fig. 4.12. B⁺ tree structure

the tree at which further indices and addresses point to the deeper levels (generally to index pages) until the pages of the sequential indices are reached. At the specific tree index levels, there is room for $n_2, n_3 \dots$ indices (generally $n_1=n_2=n_3=\dots$), from the sequential indices, n can be placed in one group, generally on a page. (In the example, for simplicity $n_1=n_2=2, n=3$. In practice, far more indices can fit on one page.) The address points always to the page one step lower on the tree, and the index gives the highest index value existing at the lower level page.

In this example there are two pages at the second level. On the first page, two indices can be found with values 11 and 40; because on the first sequential page, the largest index is 11, and on the second 40. The addresses of the indices of the second level point to the first and second page of these sequential indices. Similarly, indices 52 and 75 occur on the second page of the second level. The indices present at the first level refer to the highest indices of the second level's pages (40 to the first page and 75 to the second page of level 2).

The search for the record with key 51 is as follows:

1. Examining the indices of the highest level $40 < 51 \leq 75$, so in the following step that index page at the address belonging to 75 will be loaded.
2. Here $51 \leq 52$ (it is already smaller than the first index), so the page with the address belonging to index 52 will be read.
3. The sequential indices are reached thereby. Reading these sequentially, the address belonging to key 51 is found whence the record is retrieved. If key 51 is not on this page, then for the above-mentioned record type, there is no record with this key in the data base.

The storage structure can be seen in Fig. 4.12. It is very well-arranged, but strongly idealized. In practice, a tree can almost never be symmetrically built and so uniformly filled with data. In this example, trouble occurs immediately if, say, the record with key 17 is inserted in such a way as to preserve the index structure. Naturally, solving such a task would not tax a good system.

In practice, to insert a new record:

1. The new record is placed in the primary data area. (If there is a clustering key, then, if feasible, to the page defined by it.)
2. Most of the time the pages of the index area are not full. Traversing through the tree indices, the sequential index page assigned to the index of the new key is searched for. If there is room there, then the key, together with the addresses of the newly stored record, will be written to the suitable location according to the order. In the index table existing entries behind the key are shifted further on the page.

The indices are thereby actualized.

3. If there is no room on the suitable sequential index page, then the second half of it is transferred to an empty index page. This new page will be chained into the list structure connecting the sequential index pages.
4. The index (and the address) of the new record is entered into the corresponding sequential index page according to its key.
5. The lowest level of the tree index is modified so that on the corresponding page of the index in the halved pages' index the address is changed to the address of the new index page, and a new index is placed which includes the

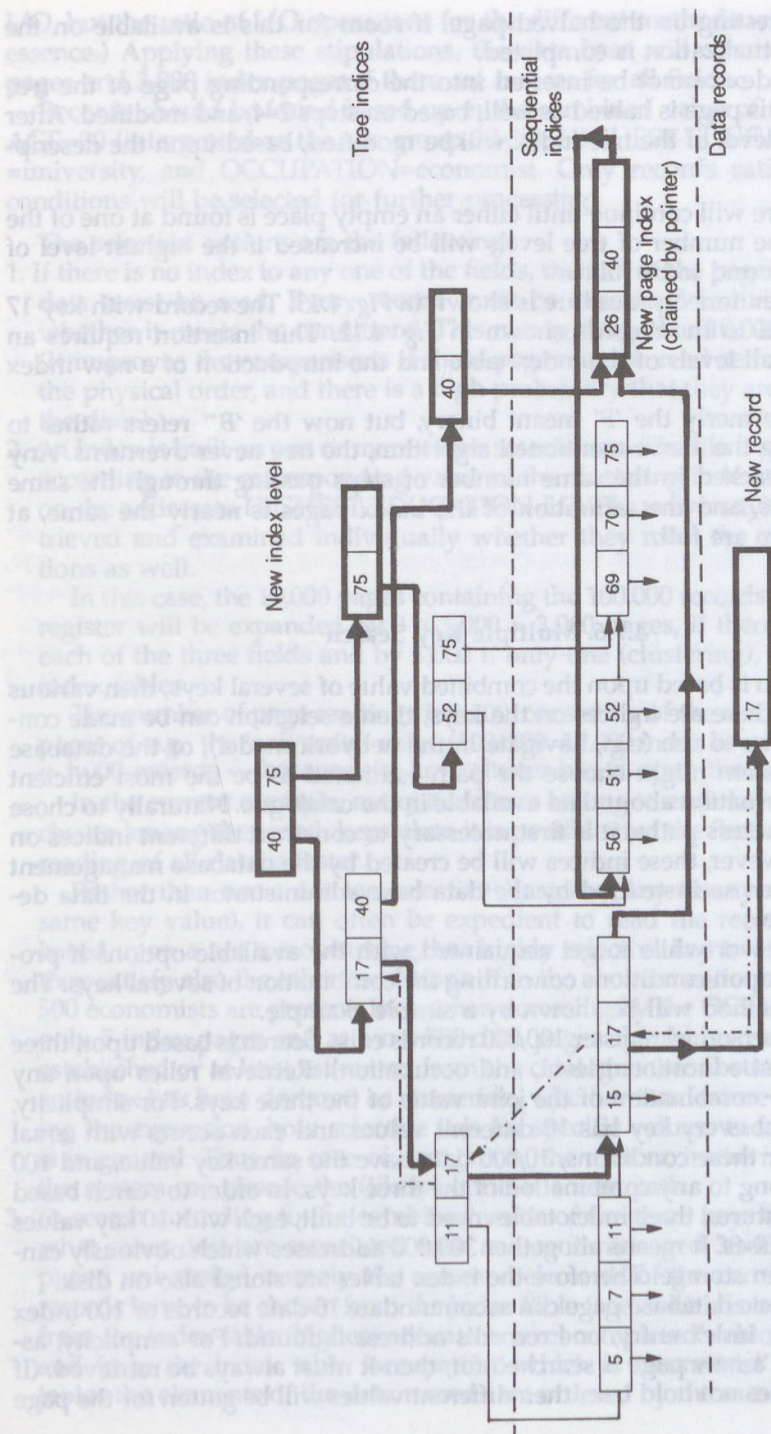


Fig. 4.13. Insertion of a new record into the B^+ tree structure and the updating of the indices.
New pages and relationships are in bold

highest key resting on the halved page. If room for this is available on the page, then actualization is completed.

6. If the new index cannot be inserted into the corresponding page of the tree index, then this page is halved as well, based on steps 2–4, and modified. After this the next level of the tree index will be modified, based upon the description in step 5.

This procedure will continue until either an empty place is found at one of the tree levels, or the number of tree levels will be increased if the highest level of the tree is also found to be full.

The above-mentioned procedure is shown in Fig. 4.13. The record with key 17 must be inserted to the records shown in Fig. 4.12. This insertion requires an arrangement at all levels of the index table and the introduction of a new index level.

In the tree, formerly the 'B' meant binary, but now the 'B+' refers rather to 'balanced'. Using the above-mentioned algorithm, the tree never overturns. Any record can be reached in the same number of steps passing through the same number of levels, and the saturation of the index pages is nearly the same, at least half of them are full.

4.7.6. Multiple Key Search

If record selection is based upon the combined value of several keys, then various options exist. If there are indices on the keys, then a selection can be made concerning which one to use (e.g., navigate in the network model), or the database management system might choose the path estimated to be the most efficient based upon information about data available in the catalogue. Naturally to choose among various access paths, it is first necessary to construct different indices on the records. However, these indices will be created by the database management system only when so instructed by the data base administrator in the data definition language.

It is therefore worthwhile to get acquainted with the available options if processing is based upon conditions concerning the combination of several keys. The essence of the method will be shown by a simple example.

Suppose in a personnel register 100,000 records exist. Search is based upon three keys (age, highest educational level, and occupation). Retrieval relies upon any key value or any combination of the joint value of the three keys. For simplicity, it is assumed that every key has 10 different values and each occurs with equal frequency. Under these conditions, 10,000 data have the same key value, and 100 data records belong to any combination of the three keys. In order to search based upon all three features, three index tables need to be built, each with 10 key values and 10,000 addresses. It means altogether 300,000 addresses which obviously cannot fit in the main storage. Therefore the index tables are stored also on disk.

Assume that one database page can accommodate 10 data records or 100 index records. In every index entry, one record's address is found. For simplicity, assume also that if a new page is searched for, then it must always be retrieved. (If this condition does not hold true, then different values will be gotten for the page

I/O, but the ratio of I/O operations for the different methods will not change in essence.) Applying these stipulations, the data base will consist of 10,000 data pages and 3,000 index pages if there are indices for all fields.

Records should be found based upon the combined value of three fields: e.g., AGE=30 (interpreted as the age group 30-35), HIGHEST-EDUCATION-LEVEL=university, and OCCUPATION=economist. Only records satisfying all three conditions will be selected for further processing.

The selection options are the following:

1. If there is no index to any one of the fields, then all of the pages with personnel data must be read. Every record must be examined individually to check whether it meets the conditions. This means altogether 10,000 page readings. (It improves the comparisons if these pages can be read serially according to the physical order, and there is a high probability that they are consecutive on the disk.)
2. An index is built on one or more of the search keys. The file is read sequentially according to the corresponding value of the clustering index table (e.g., based on the addresses HIGHEST-EDUCATION-LEVEL=university). Records are retrieved and examined individually whether they meet the other two conditions as well.

In this case, the 10,000 pages containing the 100,000 records of the personnel register will be expanded by $3 \times 1,000 = 3,000$ pages, if there is an index for each of the three fields and by 1,000 if only one (clustering), according to the index tables.

The number of page readings is 1,100, consisting of the reading of 100 index pages of, e.g., the 'university' value ($100 \times 100 = 10,000$ index keys), and $10,000/10 = 1,000$ records — because the keys cluster based upon the retrieval index. *11 pages*

In the current example, many addresses belong to each key. If none of them cluster among the search keys, then it is possible that the first-mentioned serial reading of all data is faster.

Rather than use a not too selective clustering key (many records with the same key value), it can often be expedient to read the records individually based upon a not too clustering, but highly selective key to establish whether they satisfy also the other conditions. If in the above-mentioned example, only 500 economists are present, then when accessing by the OCCUPATION index, only 5 index pages and at most 500 data pages should be retrieved. It can be established or at least estimated from the catalogue (see Section 7.10.) whether an index has been declared as clustering, whether this feature remains following the expansion, how selective it is for specific values, and how selective it is in general. Thus, in case of navigation, the user or, in the relational model, the system can choose the (likely) optimal access path.

3. To search according to the combined value of the three keys (e.g., to establish what other data are associated with an economist aged 30-35 who has completed university), namely, the common elements of the corresponding index records have to be chosen from the index table 'age' which is 'age group 30-35', from the index table 'highest education level completed' which is 'university', and from the index table 'occupation' which is 'economist'. If in all index tables the elements of the data records are ordered by the same principle, then

it is sufficient to read through only once the records of all index tables with the corresponding values. So to find the 100 records which satisfy all three conditions, $3 \times 100 = 300$ index pages and (maximum) 100 data pages must be read, that is to say, the disk is accessed 400 times.

The retrieval of records fulfilling several conditions simultaneously can be greatly accelerated without expanding the storage requirement of the file if a separate index table is created for every combination of the search fields, in the example for all three. The number of these index combinations is $10^3 = 1,000$. Every entry of it includes 100 data record addresses; hence the addresses of one combination can fit on one index page.

Considering that there are 10,000 data pages and 3,000 index pages, these 1,000 additional index pages do not increase storage demand considerably. In the meantime, performing the multi-key search, 101 disk references, rather than the previous 400, are sufficient to retrieve the 100 data records.

It suffices to make one reference to the index page which includes the addresses of those records that satisfy simultaneously all three conditions and a hundred references to retrieve the corresponding records. (It is assumed that the index table is directly organized, and so the record or, rather, records belonging to the selected key value can be found in one step. It is generally true, but if not, then in a well-organized index table, the addresses of records belonging to the selected key value can be found within a few steps.)

If one of the search keys is clustering, then it is expedient to declare this as the first component of the composite key. In this case, the addresses are read from the index table composed of the three keys according to that clustering key. It means that retrieving 100 selected records will not require 100 page changes. The probability that after the retrieval of any one record the next record will be on the same page increases significantly.

4.8. Hashing

Indexed access causes a relatively big administrative overhead for the system. Index tables must be created, modified, and within them separate searches are required which entails increased storage space and run time.

To search for data directly based upon one key, the so-called *hashing method* is more appropriate. Here the correspondence between the key and data address is 'almost' one-to-one. Accordingly, a mapping is applied among the keys and the addresses (or rather address domains) which in most cases assigns various addresses to different keys. However, in certain cases, it will assign the same address (address domain) to them. (If the mapping is symbolized with a function f , then preferably $f(K_1) \neq f(K_2)$ if $K_1 \neq K_2$, but in some – possibly only in a few – cases $f(K_1) = f(K_2)$ even though $K_1 \neq K_2$.) Elements assigned to the same address by mapping, but having different keys, are called synonyms. Obviously, their storage has to be taken care of separately. This will not be discussed in detail. However, it should be clear from the above, that for processing speed, every effort is made to minimize as much as possible the number of synonyms because

they can only be accessed using a more complicated and slower way involving more steps.

The other goal should be to economize on storage space, i.e., not to reserve a larger space than is deemed necessary. So the applied mapping must be such that among the actually occurring records should not be too large a 'gap', that is to say, unutilized address.

These two endeavors cannot be satisfied entirely; in practice, they are indeed contradictory. That mapping which makes the addresses 'denser' usually causes many synonyms. In contrast, the one that produces few synonyms results in poor storage utilization.

Therefore a rational compromise must be reached. In general, mapping can be considered as adequate if 80–85% saturation does not yield more than 20% synonyms.

The simplest hashing algorithm is when, in a numeric or hexadecimal representation, the numerically considered key is divided by a prime number and the remainder is considered as the address of the record.

This will be its relative location in the data base from which it can be reckoned unambiguously on which page, and perhaps, where on the page it can be found. The divisor is chosen that approximately will equal or be a bit less than the number of available storage places. Today, several other essentially better hashing algorithms have been elaborated. The algorithm is a part of a database management system (perhaps the data base administrator can chose from among several), and by storing or searching based upon a key, the system defines the physical address, and if necessary, examines the synonyms.

In general, for direct processing, if access is only through one key, then a good hashing is more efficient than indices. That is why in hierarchical and, especially, in network data bases, this is the preferred direct access method. In contrast, in relational data bases, hashing is not used or only in very special cases because

- it does not allow sequential processing, not even for the hashing key, and
- it permits direct access to a record type only according to one key (although this can be a combined key).

The main attraction of the relational model is its flexible use and with hashing, in contrast to the indices, precisely this cannot be ensured.

4.9. Data Condensing and Encoding

4.9.1. Data Condensing Methods

Despite increases in the storage capacity of magnetic disks and their price decline, during database planning the proper utilization of the external storage capacity remains an important objective. If the useful data occupy a smaller area on the storage, then, first of all, not the number of disk units are decreased, rather more information will be stored in the data base. Moreover, the processing will also be faster for two reasons:

- More data can be stored on one page. Paging and the number of I/O operations will be reduced.

- The same information can be transmitted in less bytes through the communication network. Data reaches the user (and the distributed data bases) faster.

Condensing is achieved in data bases by working with fields and records having variable length. If the valuable data content can vary in length, (e.g., names, addresses), then the declared field length must be long enough to contain the longest data. However, at the same time, in most cases, in character type fields only the leading characters contain valuable information, while the spaces following them are redundant. (With numbers, the situation is the contrary. In that case, the worthless zeros occurring at the beginning of a field can be trimmed.) In the name field the name T. Bryce Wadsworth Cadwallider, Jr. should fit, but perhaps only Big Tom will be stored there. The length difference is even more dramatic in the text retrieval systems introduced in Chapter 8.

To abandon the worthless spaces at the end of a field, variable field lengths can be declared at the physical level. The database management system inserts a field length indicator in front of these fields (in general, two bytes). So the maximum N byte length character type field occupies $N-n+2$ bytes if there are n spaces at the end of the field.

If a record includes variable length field(s), then the record length is variable, too. That is why the system automatically places a record length indicator (usually 4 bytes long) before these record types.

Since the length indicator occupies space, too, and it requires time to handle it and establish the compressed record length, and during retrieval to fill it up with spaces, in practice, such indicators are favorable only if the average space saved is generally at least 12–15 bytes per field.

It is also a typical (though in data bases rarely used, while in data transmission a usually automatically applied procedure) that for several times consecutively repeated characters only the first character and the number of repetitions are transmitted. In front of the repeated character, a special, otherwise not usable, bit combination indicates that a repeated character string now follows, but in a condensed form. Should this special bit combination be transmitted, the system duplicates it automatically prior to transmission. This indicates when receiving that not recurrent characters arrive, but rather this character string.

When storing text, the greatest space economy can be achieved by the so-called Huffman code. Characters are represented by a variable length bit series; the most frequent characters consist of the fewest bits. Although in certain circumstances, the space saving achieved reaches 30–40%, this method has not become widespread in practice. The reason is, above all, its clumsiness and the very time-consuming nature of the transformation.

4.9.2. Encoding Algorithms

Although the database management system offers protection against unauthorized access, competent system programmers may possess such means to bypass these safeguards and access directly the volume. It causes even more trouble that the data base management system does not offer any protection for the copies made for safety reasons. Anyone who manages to gain access to backup tapes or disks can read what exists on them. Tapping of data transmission lines is also a serious security risk for sensitive data.

Therefore very confidential or secret data should not be stored in their original form in the data base, but encoded.

The encoding process is based upon an E encoding algorithm whose actual realization depends upon a K_E parameter(s) and a cipher key. Consequently from the T text the $T_E = E_{K_E}(T)$ text can emerge from which the original text can be restored only with the R restoring algorithm, knowing the K_R parameter(s), that is to say $T = R_{K_R}(T_E)$. In practice, $K_E = K_R$ often, but it is not necessary.

The number of encoding algorithms is in principle unlimited, but for computer implementation the American National Bureau of Standards published in 1977 a standard encoding and decoding algorithm for whose application ready-made softwares are now available for most types of computers. Public knowledge of the algorithm is harmless since the decoding of the enciphered text is not possible without knowing the decoding key. Using the suggested procedure ²⁵⁵ different variations exist; trying them and evaluating the results cannot be realized within a realistic time period even with the highest speed supercomputers. In contrast, if the key is known, then text encoding and decoding can be completed in a matter of seconds or minutes.

4.10. Sorting

Computers can store and process a vast amount of data. However, for this process to be efficient and well-arranged, it is generally necessary that the data be organized, sorted by suitably chosen criteria. Efficient sorting algorithms have played a key role in data processing from the very beginning. General, user-friendly sort programs which arrange data in any requested order are available. Every commercial operating system includes a Sort/Merge Generator. Its input is the unsorted, its output is the sorted data file. Besides this, control commands are required. They define the input and output files, their record forms, the sorting keys, and their location in the record.

In database technology, these sort/merge programs are not really used directly. At most, input data are sorted earlier, before the initial loading of the data base to increase the efficiency of further processing.

When working with the data base it is often necessary to sort in an ad hoc manner all data or a selected portion of them. A short introduction to sorting methods is important since it is one of the most time-consuming operations of data processing. This is especially true for large volumes of records. Sorting time does not increase linearly with the number of records to be sorted. Depending

upon the method, its order of magnitude is proportional between $n \times \log(n)$ and n^2 (n is the number of records). The typical database user does not come into direct contact with the sort program, even less with the sorting algorithm generated by this program. The database management system completes automatically the operations needed for sorting, insofar as they are necessary. Despite that, it is useful to introduce briefly the most important sorting procedures. The user can thereby sense what kind of burden is imposed on the system, i.e., in general, to what extent does it slow down the operation when he would like to solve a task which requires an occasional sort of the data.

In general, after completing the sort, the sorting program of the database management system does not put into physical order the data which had to be sorted. Rather it creates only an index file containing the selected sorting key or keys in order, and the record addresses in the data base, corresponding to these keys.

The processing order is defined by this index table.

4.10.1. Automatic Sorting

In practice, subsequent sorting is necessary in the data base in the following cases:

- creating a new index for existing data,
- sorting occasionally processed data according to a key for which there is no index, and
- filtering records with the same values.

All three occurrences will be discussed below.

Creating a New Index for Existing Data

As applications multiply, it is often necessary to regularly process data directly or in ascending/descending order, based upon the value of a given field (or fields) on which no index was earlier built. In this case, efficiency can be significantly increased if a new index for this field is created.

To construct an index, first the records must be sorted by the new index's value. Then an index table for the so-ordered keys must be constructed. It is usually a relatively long process due to the large quantity of data, but need be completed only once.

Changes in the data are followed automatically in the index table. Neither a reorganization nor a new sort is required.

Index maintenance is performed either with the modification of the list structure introduced in Point 4.6.1., or with the B^+ tree discussed in Point 4.7.5.

Naturally, as has already been stated, to create a new index it is sufficient for the user to specify to the database management system using the data definition language

- which record type,
- from that which fields,
- in which order (ascending or descending) are to be indexed, and
- whether to check the uniqueness of the key values, or allow duplicates (see the CREATE INDEX command in Section 10.4.).

Occasional Ordering of the Results of Individual Operations

Several records are selected from the data base (possibly from various record types) to be displayed or passed for further processing ordered by a field(s) on which there is no index. This often stems from ad hoc-like retrievals where the users need the data in the most diverse order or grouping.

Since the selected data and ordering fields vary in an unforeseen manner, it is neither possible nor would it even be expedient to construct an index for all such possible cases. Therefore if such a task needs to be solved, then the database management system generates commands necessary for sorting at the time of execution, creates the sorted temporary file, and after the completion of the task deletes it (see Point 11.2.3., ORDER BY clause). To achieve the sorting, time and storage space are necessary. For storage the system uses a work area, reserved especially for such purposes. This enlarges the size of the data base. The time needed for sorting slows down the processing, often even by an order of magnitude.

Filtering Duplicates

It may often happen that not all fields of a record are necessary. After eliminating the unnecessary fields from the selected records, some which were originally different became identical. But for further processing, only one is needed from each of them (see Point 11.2.2., SELECT DISTINCT ...). The duplicates which thereby emerge can be filtered efficiently if the result records are ordered. Identical records follow each other. So they can be detected easily and the duplicates deleted.

An example of this situation can be a company register from which is sought the names (or the codes) of the sections where there are employees earning less than \$ 1,500 monthly. The task is solved by selecting from those record types (tables) containing personnel data, all the records where the monthly wage is less than \$ 1,500, and from these only the field section name (or section code) is kept. The first result of this operation is a draft list where every section occurs as many times as it has employees earning less than \$ 1,500. The report will be made clearer by dropping the duplicates from the result.

4.10.2. Sorting Methods

When sorting, as when solving every computer task, the efficient use of different resources, that of time, operative memory, external storage capacity, investment, and operating expenses conflict. Fast sorting requires a large operative memory and a large temporarily-used disk area in case of a big data volume, both of which cost money. However, if smaller memory and storage are used, then the sorting time will increase. In any case, machine time, especially in case of interactive processing, and user time are more scarce and valuable.

To solve the dilemma — as always in computer technology — an expedient compromise has been struck. As far as the operative memory, the operating speed is much greater as compared to writing to or reading from the disk. That is why as much data as possible is read into the memory and sorted there. (By data here

is meant, in general, only the sort key of the record and its disk address.) The thus organized subset is written out to a temporary file on the disk and a new group of unordered data will be loaded into the memory where it is sorted and subsequently written out to disk. This process continues until all data to be ordered have been put out to the disk as ordered subsets. Thereafter they are merged to produce the ordered data.

In case of a small amount of data it may be possible to sort the whole volume in the operative memory at one time. However, with a larger data volume it is necessary to use external data storage, temporary disk area. As a consequence of this, the sorting efficiency depends upon the efficiency of the following processes:

- the inner sorting in the operative memory, and
- the merging of the ordered subsets written temporarily to disk.

For the inner sorting, different algorithms are used.

Most of them are based on the concept that from the set of records in the memory, the smallest (the biggest) is selected and placed alongside the first (last) place of the set. Or from two elements selected by a defined algorithm, the smaller (bigger) advances continuously forward. The procedure omits the element which has been suitably placed in the final order and repeats itself until all the elements are in the right place.

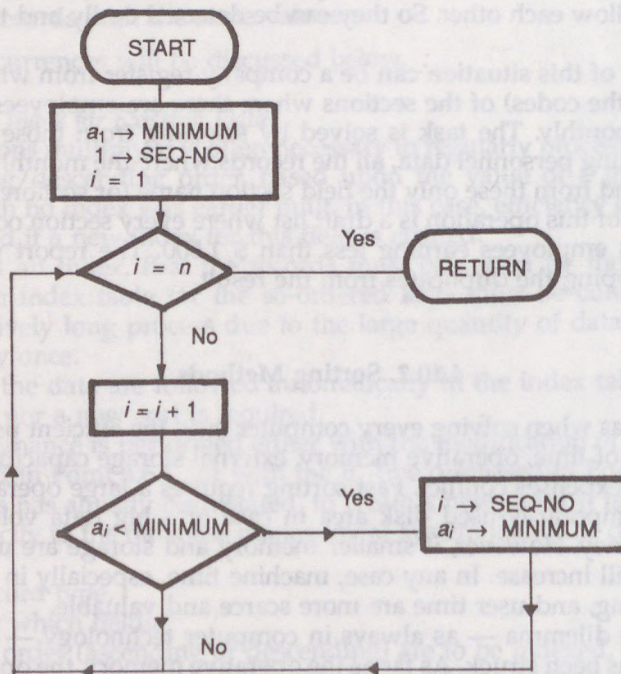


Fig. 4.14. The block diagram of the routine searching for the smallest element

Element order number	Data								
	Step 1		Step 2		Step 3		...	Next to last Step	
	Dis- ordered	Or- dered	Dis- ordered	Or- dered	Dis- ordered	Or- dered	...	Dis- ordered	Or- dered
1	8	2	8	2	8 ← max	2		max	2
2	16		16	3	16	3		max	3
3	2 ← max		max		max	8		max	8
4	3		3 ← max		max			max	12
5	29		29		29			29 ← max	16
6	17		17		17			max	17
7	12		12		12			max	21
8	21		21		21			max	29

Fig. 4.15. Selection sorting

Such an algorithm is shown below. Its principle can be followed easily, but it should be noted that more efficient, albeit more complicated, procedures also exist which are preferable in practice. Their complexity causes no problem for the user since the algorithm is hidden from him anyhow.

The first step of this very simple algorithm is to select the smallest element. This can be achieved in many ways. One of the simplest variations of this is shown in Fig. 4.14.

Since this procedure must be repeated many times, it is essential that this routine use a good algorithm and be coded efficiently. The above-introduced method is easy to program. Yet those methods which always select the smallest element of the set are far from optimal regarding efficiency. Therefore when sorting a large data volume, it is often more expedient to compare elements which are neighbors or exist in a computable distance calculated by a given algorithm. The smallest element is thereby determined and additional steps for sorting can also be prepared.

After the introduction of the simple method designed to choose the smallest element, a well-arranged procedure will be shown by which elements of a set can be ordered with its repeated use.

The method is called *selection sorting* and its principle is shown in Fig. 4.15. The smallest element of the unsorted set is selected (2) and placed in the first position of the sorted set. To exclude this element from further examinations, its sorting key will be changed to the highest possible value. (In every computer there is a character string which is bigger than all letters, numbers, and symbols.) In this example it will be indicated as *max* (maximum).

Next the so-modified unsorted set is again examined and its smallest element is searched for. This is not the smallest element of the original set because that key is now equal to *max*.

The thus-found smallest element is put into the next place of the sorted set and the *max* value will be assigned to its key. This procedure is repeated until all the elements of the unordered set have been carried into the sorted set, i.e., after the modification *max* is present in the key field of all elements.

This sorting method requires a relatively large amount of storage and time. The sorted set occupies at a different location the same space as the unordered set. All the elements are examined for selection; nonetheless, it is sure that those which have already been moved to the ordered set will not be considered.

The modification shown in Fig. 4.16. helps to solve this problem. All elements of the unsorted set are first examined. The smallest one (2) is exchanged with the first element (8). During the second step, the unordered set is examined only from the second element, and the thus-found smallest element (3) is exchanged with the second (16). Next the examination resumes from the third element and that will be exchanged with the smallest. (In this example they are identical by chance.) The procedure is repeated until the final element is reached.

Element order number	Data					
	Step 1	Step 2	Step 3	Step 4	...	Next to last Step
1	8	2	2	2		2
2	16	16	3	3		3
3	2	8	8	8		8
4	3	3	16	16		12
5	29	29	29	29		16
6	17	17	17	17		17
7	12	12	12	12		21
8	21	21	21	21		29

Fig. 4.16. Selection with exchange

The smallest element is selected from among those below the bold line

Although the exchange of the elements requires additional operations, disregarding the exchanged elements saves essentially more time than is needed to examine and compare those elements marked with *max*.

The following algorithm can be used to merge the sorted subsets into the complete, ordered file:

- The smallest element of the subsets' first elements is chosen. (If there are several identical ones, then one of these, e.g., the element from the first subset, will be selected using a defined algorithm.)
- This element will be written as the next element of the sorted set.
- The next element of the chosen subset becomes the first element of this subset. If there are no more, then the value *max* will be assigned to the first element.

- d) The procedure will resume from a) until all the values of the comparable elements equal *max*.

Naturally, by sorting in a descending order the pre-sorted subsets' order is also descending. From the first elements always the highest will be selected. If some subset is exhausted, then its first element will be considered the fictional smallest value in the record.

2.1. Hierarchical Systems

The first database management systems were based on the hierarchical model. The reason for this was that numerous aspects of real life can be described well enough with simple hierarchies as the first approach. In such cases, when the dependencies are unambiguous, a hierarchical model represents reality well and in a manner that can be realized effectively on the machine. In practice, the most widespread hierarchical system is the Information Management System (IMS) developed by IBM and first marketed in 1968. Since then it has been considerably expanded and modified. Due to this — as with FORTRAN and COBOL, despite the frequent prediction of its extinction by theoretical computer science pundits — it is applied even nowadays to several areas. Admittedly, the reason for this is mainly that these systems already exist, have been used to store a huge quantity of data, and it is more convenient or cheaper to retain them as independent systems or to integrate or connect them somehow into the newer, more modern systems, than to rebuild the whole data structure and volume and rewrite all programs according to the requirements of the new model.

Besides this, if the processing order matches the hierarchy, then hierarchical systems, though considered obsolete nowadays, allow an essentially more efficient realization on the machine than up-to-date relational databases. The limits of the model do not truly disturb the users or programmers if the actually realized data model is relatively stable, and most of the information requests can be formulated, put in a procedure, or programmed in advance.

2.2. Tree Structure

A simple example of a hierarchical structure is the personnel structure of a company. This consists of the department heads subordinate to the general manager, then section heads, group leaders, those subordinate to the latter, and finally, individual employees. This structure describes an unambiguous official channel and is also an excellent example of the tree structure in one. Its characteristic feature is that any element of the hierarchy can be accessed logically only through one path. The logical

The Hierarchical Model

5.1. Hierarchical Systems

The first database management systems were based on the hierarchical model. The reason for this was that numerous aspects of real life can be described well enough with simple hierarchies as the first approach. In such cases, when the dependencies are unambiguous, a hierarchical model represents reality well and in a manner that can be realized effectively on the machines. In practice, the most widespread hierarchical system is the Information Management System (IMS) developed by IBM and first marketed in 1968. Since then it has been considerably expanded and modified. Due to this — as with FORTRAN and COBOL, despite the frequent prediction of its extinction by theoretical computer science pundits — it is applied even nowadays in several areas. Admittedly, the reason for this is mainly that these systems already exist, have been used to store a huge quantity of data, and it is more convenient or cheaper to retain them as independent systems or to integrate or connect them somehow into the newer, more modern systems, than to rebuild the whole data structure and volume and rewrite all programs according to the requirements of the new model.

Besides this, if the processing order matches the hierarchy, then hierarchical systems, though considered obsolete nowadays, allow an essentially more efficient realization on the machine than up-to-date relational databases. The limits of the model do not truly disturb the users or programmers if the actually realized data model is relatively stable, and most of the information requests can be formulated, put in a procedure, or programmed in advance.

5.2. Tree Structure

A simple example of a hierarchical structure is the personnel structure of a company. This consists of the department heads subordinate to the general manager, then section heads, group leaders, those subordinate to the latter, and finally, individual employees. This structure describes an unambiguous official channel and is also an excellent example of the *tree structure* in one. Its characteristic feature is that any element of the hierarchy can be accessed logically only through one path. The logical

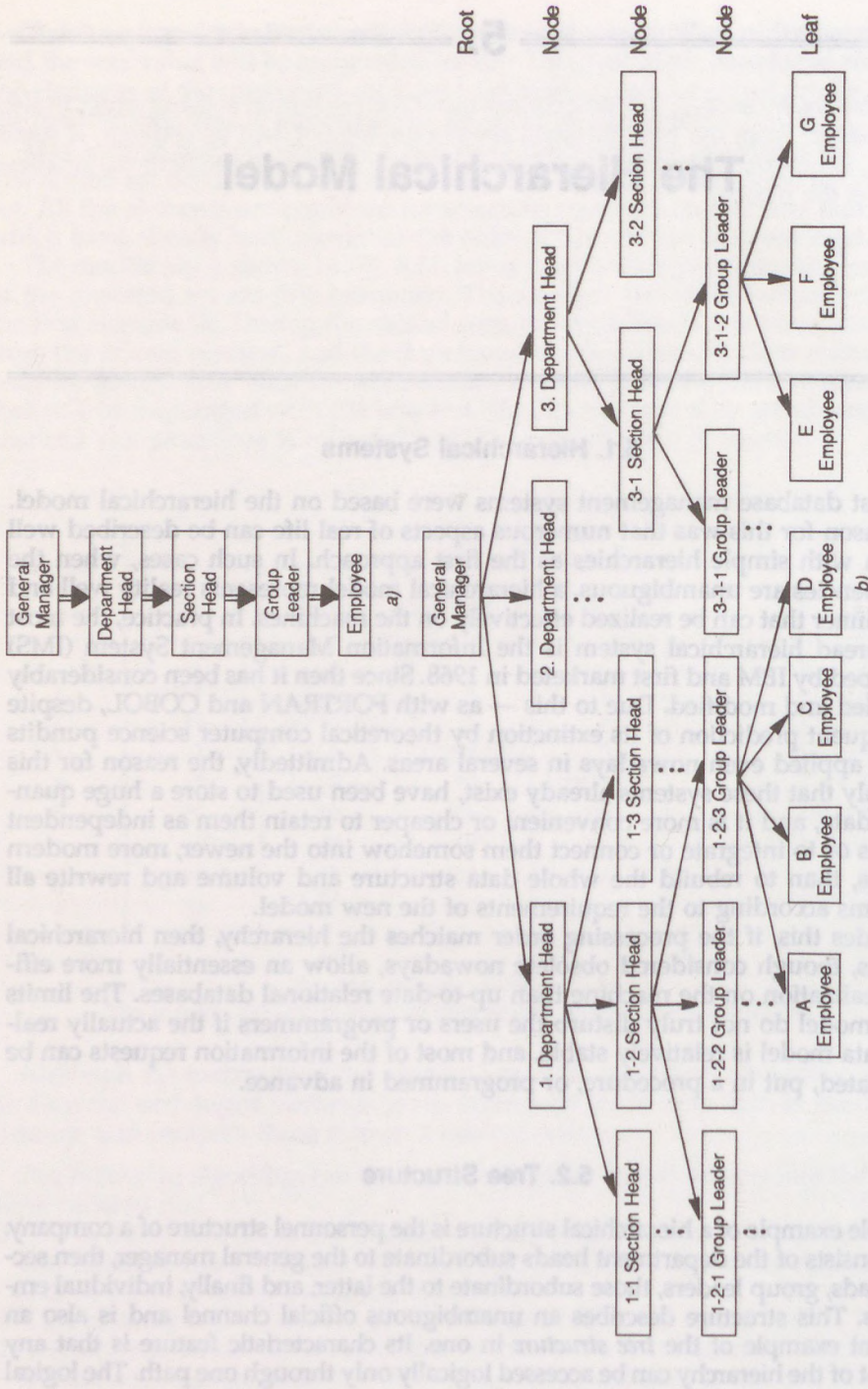


Fig. 5.1. An example of tree structure
a) represented by general data structure, b) occurrences of the individual elements

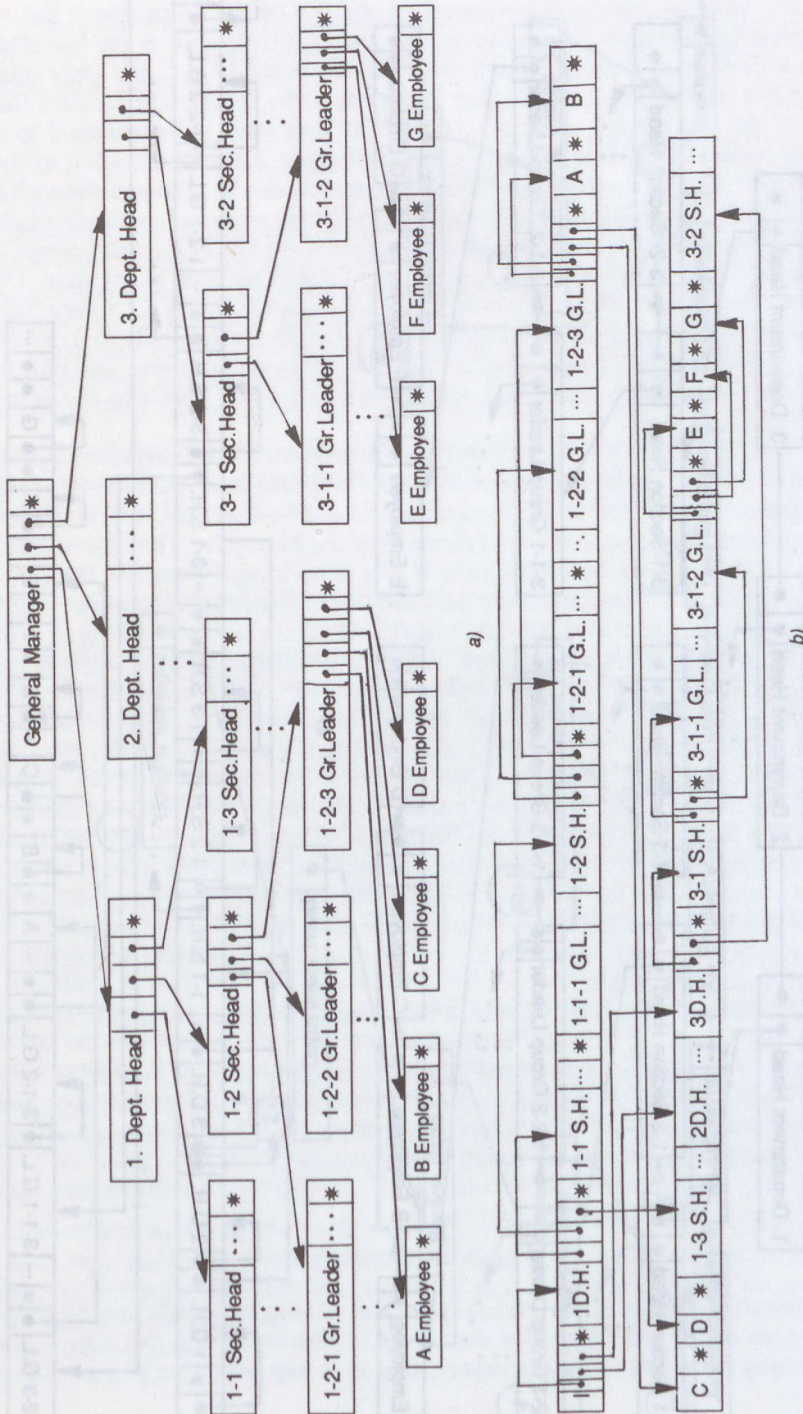


Fig. 5.2. a) The storage and b) the physical structure of the logical data structure seen in Fig. 5.1. (Arrows indicate the pointers and the * marks the chain's end.)

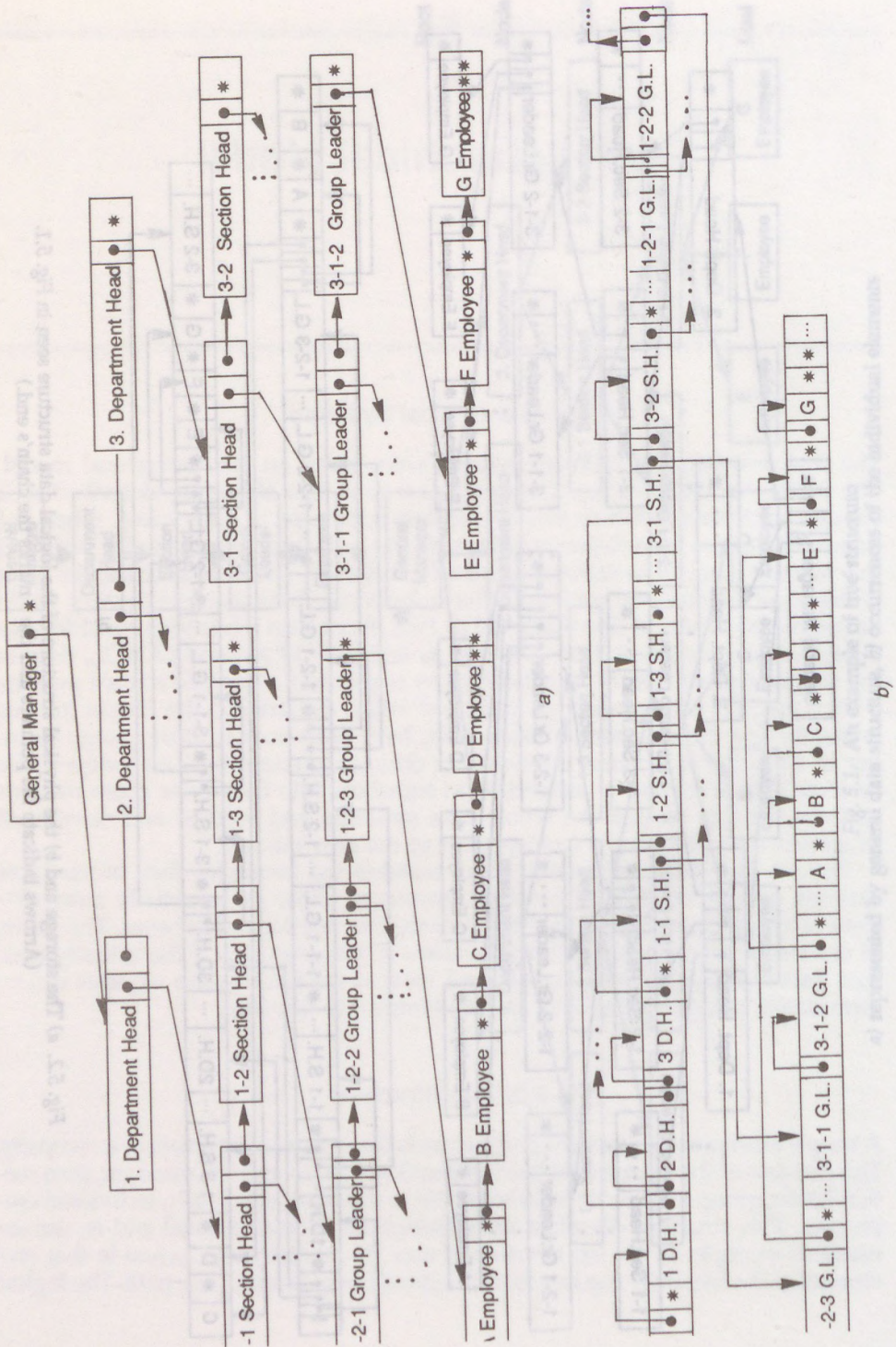


Fig. 5.3. Another possible a) storage and b) physical structure of the logical data structure seen in Fig. 5.1. (Arrows indicate the pointers and the * marks the chain's end.)

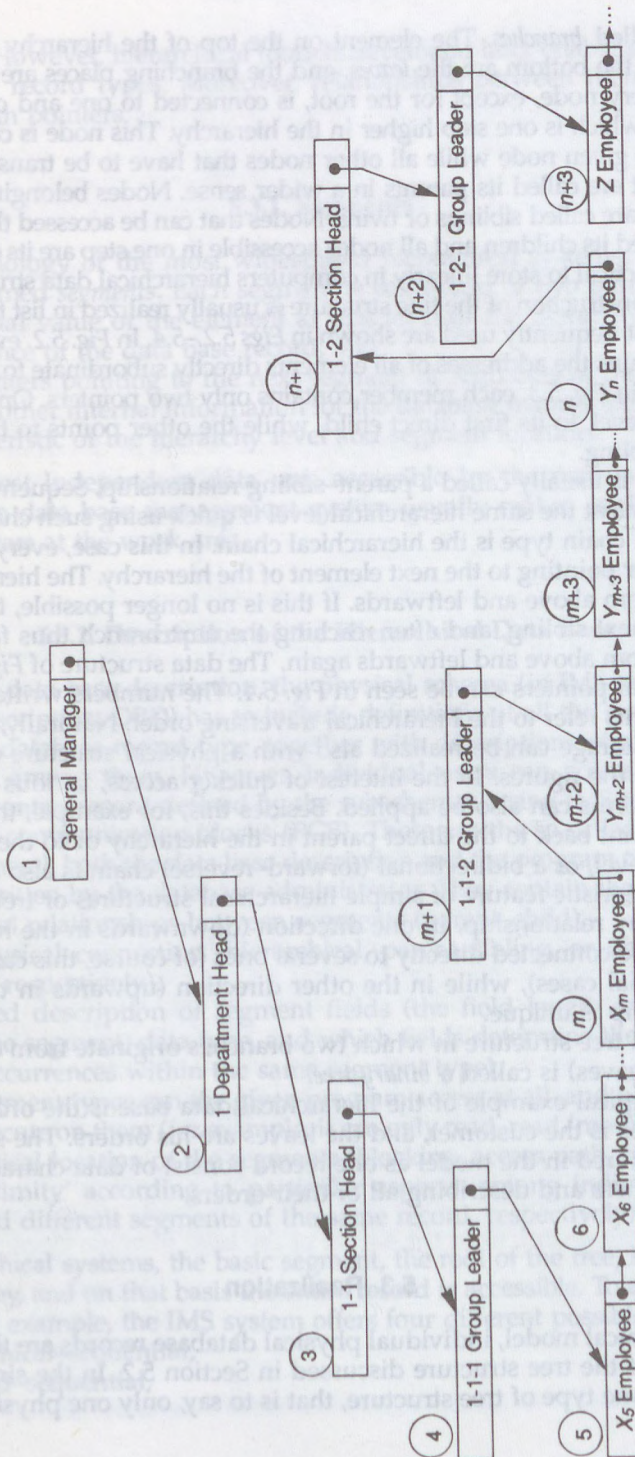


Fig 5.4. Tree structure connected by hierarchical pointers

paths are called *branches*. The element on the top of the hierarchy is the *root*, the members on the bottom are the *leaves*, and the branching places are the nodes (see Fig. 5.1.). Every node, except for the root, is connected to one and only one of the other nodes which is one step higher in the hierarchy. This node is called the direct parent of the given node while all other nodes that have to be transited to reach it from the root are called its parents in a wider sense. Nodes belonging to the same direct parent are called siblings or twins. Nodes that can be accessed through a given node are called its children and all nodes accessible in one step are its direct children. Since it is practical to store linearly in computers hierarchical data structures as well, the storage construction of the tree structure is usually realized in list form. The three methods most frequently used are shown in Figs 5.2.-5.4. In Fig. 5.2. every individual element contains the addresses of all elements directly subordinate to it, while in the version seen in Fig. 5.3. each member contains only two pointers. One points to the subordinate level, to its first direct child, while the other points to the same level, to its next sibling.

This chain is usually called a parent-sibling relationship. Sequential processing of the elements at the same hierarchical level is quick using such chains. The third often-applied chain type is the hierarchical chain. In this case, every element contains a pointer pointing to the next element of the hierarchy. The hierarchical order is descend from above and leftwards. If this is no longer possible, then upwards, right to the next sibling, and after reaching the next branch thus far untouched, to descend from above and leftwards again. The data structure of Fig. 5.1. realized by hierarchical pointers can be seen in Fig. 5.4. The numbers written to the individual elements refer to the hierarchical traversing order. Naturally, data arrangement in the storage can be realized also with a physical structure different from those seen in the figures. In the interest of quicker access, various combinations of the basic forms can also be applied. Besides this, for example, the pointer can be used to point back to the direct parent in the hierarchy or to the first member at the same level, or a bidirectional (forward-reverse) chain is also available.

The characteristic feature of simple hierarchical structures or tree structures is the one : many relationship. In one direction (downwards in the hierarchy) any element can be connected directly to several ones (of course, this can also be zero in some special cases), while in the other direction (upwards in the hierarchy) the relationship is unique.

The special tree structure in which two branches originate from any node (except for the leaves) is called a *binary tree*.

Another typical example of the hierarchical data base is the order book. The root of the tree is the customer, and the leaves are his orders. The segments (see Point 5.3.1.) stored in the model as one record consist of data characterizing particular customers and describing all of their orders.

5.3. Realization

In the hierarchical model, individual physical database records are the individual occurrences of the tree structure discussed in Section 5.2. In the simplest model there is only one type of tree structure, that is to say, only one physical data base

record type. However, hierarchical models developed later enable storage of several kinds of record types. Moreover, relationships between them can also be described with pointers.

5.3.1. Segments

In the terminology of the most widespread hierarchical systems, tree structure nodes are named *segments*. Each segment is composed of two parts:

- the actual value of the element at that level of the hierarchy in the given occurrence of the data base record, and
- the pointers pointing to the next segment or segments of the data base, as well as other internal information for the database management system and characteristic of the hierarchy level and segment location.

The smallest independent data unit accessible by the user is the segment. However, the data base management system usually makes available only the actual data part at the work area.

5.3.2. Description of the Hierarchical Data Base

The physical data base description, the physical schema (in IMS terminology the data base description, DBD) has to include definitions of all the segment types of the physical database record type, together with descriptions of all parent-child relationships among them. However, individual users can access only the segments of the entire record defined by the subschemas, using IMS terminology, by the program communication blocks (PCB). These are the so-called sensitive segments. In general, both the data base description and the program communication blocks are written by the database administrator. They contain the following:

- a) The logical relationships between segments (parent-child),
- b) Their physical connection (hierarchical, parent-sibling, or one or two-way pointers, respectively),
- c) A detailed description of segment fields (the field length and its location within the segment, data type, and which fields determine the order of individual occurrences within the same segment type),
- d) What segment types can the given program access at all, and what operations can it execute on them (for example, it can only read, read/modify, delete, etc.).
- e) The physical location of the segments (blocking, access path, ensuring physical 'proximity' according to particular aspects, among individual segment types and different segments of the same record, respectively).

In hierarchical systems, the basic segment, the root of the tree, is always identified by a key, and on that basis the entire record is accessible. To access the basic segment, for example, the IMS system offers four different possible access paths:

- hierarchical-sequential,
- indexed-sequential,

- direct, and
- indexed-direct.¹

Simpler systems generally offer only one of these access paths to the user.

In IMS HSAM organization, basic segments, as well as all subordinated segments connected to them in various orders, are in one sequential file. The physical sequence of individual segments agrees completely with the order stemming from the hierarchy.

In HISAM the basic segments are stored in an indexed sequential file based upon keys, in HDAM in a directly organized file where the relationship between the key and the storage address can be obtained by hashing, while in HIDAM in an inverted file to the key of the basic segment.

In all three of the above organization methods, the children attached to the same occurrences of basic segments are located partly or entirely at physically discrete places. Relationships among segments stored at different locations are made by pointers created and maintained by the data base management system.

5.3.3. Hierarchical Database Operations

The data manipulation language of the hierarchical model contains instructions for data retrieval, insertion, deletion, and modification. To enable modification or deletion, first the record has to be identified. This is usually done by the key value identifying the segment. The operation of the GET UNIQUE command reads the first record of the data base satisfying the condition. Additional records meeting the condition can be found with either the GET NEXT or the GET NEXT WITHIN PARENT command. The former locates the next record at the same level of the hierarchy while the latter repeatedly supplies all children of a given parent.

If there is no record satisfying the given condition or if there are no more records, then the system informs the user.

In addition, in most systems, it is also possible to create secondary indices. They enable non-key values to be searched for, e.g., according to order date.

The most frequent hierarchical operations are summarized in Table 5.1. The English names and abbreviations are used in the chart and text since most systems work with English commands. These commands agree in their form and function with DL/I commands. DL/I is a data manipulation language used in earlier IBM database management systems.

The simple hierarchical model's greatest advantage is that it very often gives an easily understandable natural mapping of the hierarchically-constructed real world. When the simple hierarchy described by the data model agrees with the natural hierarchy arising from the nature of the task, then the system's efficiency is very high.

¹ These access methods are usually called by their abbreviated forms: HSAM, HISAM, HDAM, HIDAM.

Table 5.1.
The most often used data manipulation commands
of simple hierarchical systems

Command	Abbreviation	Function
GET UNIQUE	GU	Reads a definite segment
GET NEXT	GN	Reads the next segment at the same level in the hierarchy ^{1, 2}
GET NEXT WITHIN PARENT	GNP	Reads the next segment of the same direct parent ^{1, 2}
GET HOLD UNIQUE	GHU	The same as GU, but it temporarily locks the segment from other programs ³
GET HOLD NEXT	GHN	The same as GN, but it temporarily locks the segment from other programs ³
REPLACE	REPL	Replaces the modified segment following a previous command of the GET HOLD ... type
DELETE	DLET	Deletes the segment following a previous command of the GET HOLD ... type
INSERT	ISRT	Inserts a new segment into the database

Remarks

1. Next is interpreted as compared to the segment occurring in the prior operation executed in the data base.
2. The return code indicates if there are no more records.
3. Such types of GET commands are necessary for modifying or deleting. This ensures that other programs can access unambiguously and in a traceable way these segments in their form prior to the modification or after it.

However, various dependencies are possible among the same data. As soon as the relationships arising from the task diverge from the model's hierarchy, then the database management system or the user himself has to solve problems due not to the complexity of the task, but rather the unsuitability of the model. Resolving these problems decreases mainly the efficiency of the system, and its application becomes more difficult.

5.4. An Example of the Hierarchical Data Base

To illustrate the problems resulting from the simplifications of the hierarchical model, the following simple data model will be discussed. In a vehicle registry the following data are needed about vehicles and owners:

- license plate,
- make of the vehicle,
- model of the vehicle,
- color of the vehicle,
- engine size, and
- other vehicle features.

About every former or current owner are needed:

- name,
- permanent address,
- present occupation,
- period of ownership, and
- other information.

(For simplicity, both for vehicles and owners, data not germane to the example, e.g., chassis and engine number, traffic offenses, etc. were included in the collective term 'other'.)

Two possible hierarchical representations of this task can be seen in Fig. 5.5. In Fig. 5.5.a the basic segment is the VEHICLE segment and its direct children are the CURRENT-OWNER and FORMER-OWNER segments while in Fig. 5.5.b the basic segment is the CAR-OWNER segment and its direct child is the VEHICLE segment which is the direct parent of the FORMER-OWNER.² Which of the two potential hierarchical organizations and representations are more favourable depends upon the nature of the processing. The method shown in Fig. 5.5.a can answer relatively quickly queries whose selection is based on the license plate (for example, answers to queries about the name of the current vehicle owner or the address of all former owners for a given vehicle), while in Fig. 5.5.b queries are based upon the owner's name. (For example, 'What is the make of X's car(s)?' or 'From whom did he buy it?') The following query 'What cars has X owned previously?' cannot be answered efficiently by any of the hierarchical models shown in Fig. 5.5. (Naturally, by arranging the available data in a different hierarchy at the cost of answering other kinds of questions more slowly (trade-off), this one could be answered efficiently as well.)

Finally, all logical relationships are built upon physical connections in every system. So retrieval efficiency depends not only upon the logical, but also upon the actual data storage structure. However, it is usually true that if the logical data structure does not satisfy the information needs, then the efficiency of the system will suffer. (The converse is not true. An appropriate logical structure can be built even upon an uneconomical physical structure.) The storage arrangement of a hierarchy corresponding to the one in Fig. 5.5. is illustrated in an indexed (HIDAM) organization in Fig. 5.6. For simplicity, it is assumed that each segment occupies one storage location, and they are linked with a one-way hierarchical pointer chain.

The above-mentioned problems arising from the simple hierarchical model are illustrated by posing the following question: 'What are the data of all the vehicles owned by X?' (it is assumed that the name unambiguously identifies the person).

² It is assumed that in both representations a car can have only one owner at a given time, and in Fig. 5.5.b also that the name identifies the owner unambiguously. In the latter case, further problems can arise from the possibility that in a record the FORMER-OWNER can be the same as some CURRENT-OWNER. This can cause a redundancy or a deviation from the tree structure. However, if FORMER-OWNERS are considered from the start as objects of a different type than the CURRENT-OWNERS (since if the name is disregarded, then the different values of the same attribute at various times are valid), then this problem will not arise.

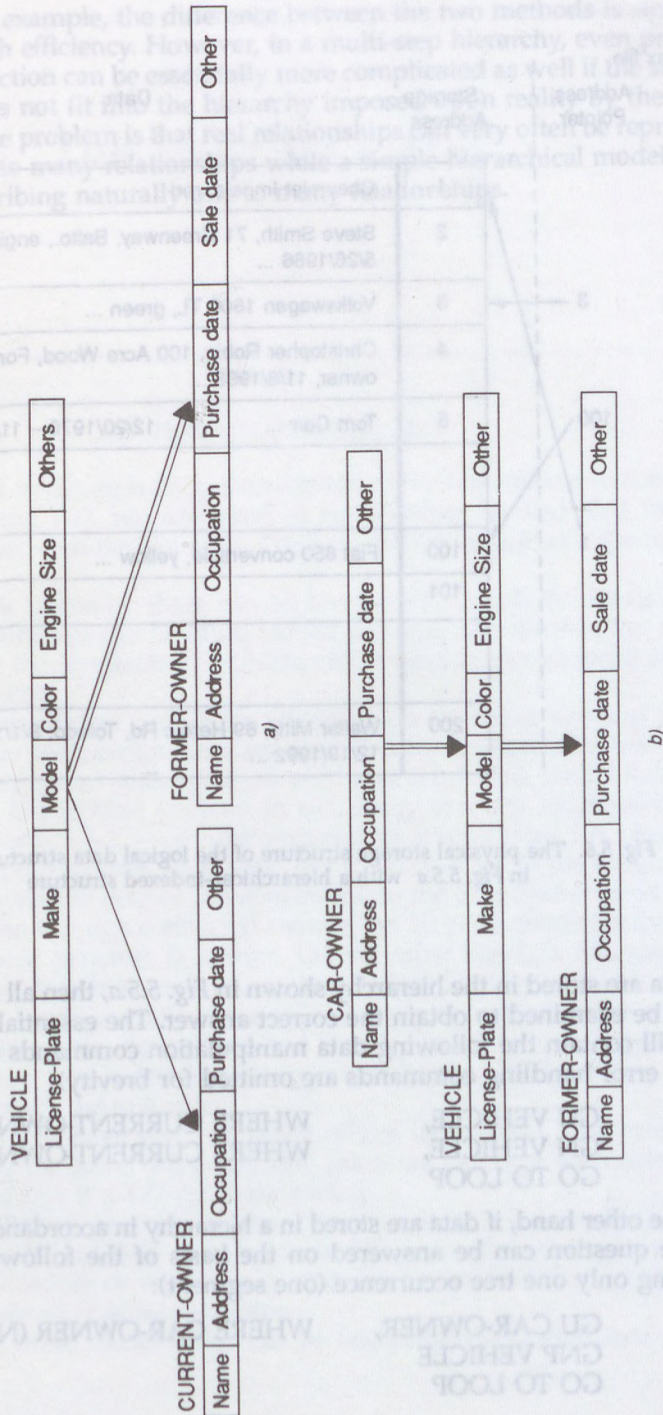


Fig 5.5. Two possible methods of describing the logical relationships among vehicles and their owners in a simple hierarchical model

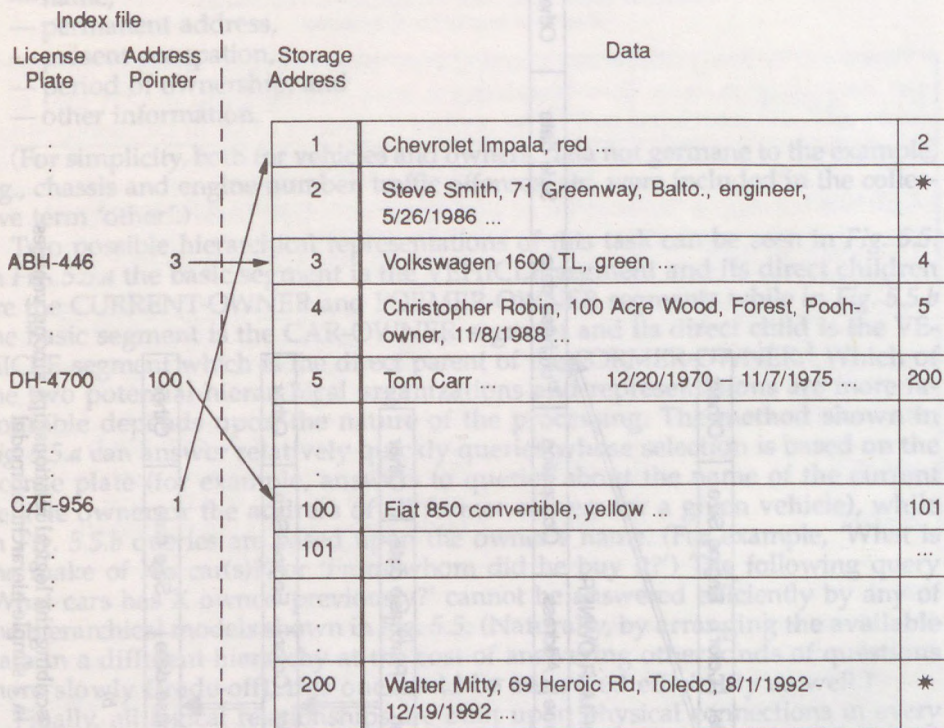


Fig. 5.6. The physical storage structure of the logical data structure shown in Fig. 5.5.a with a hierarchical-indexed structure

If data are stored in the hierarchy shown in Fig. 5.5.a, then all database records have to be examined to obtain the correct answer. The essential part of the program will contain the following data manipulation commands (further processing and error handling commands are omitted for brevity):

```

LOOP.      GU VEHICLE,      WHERE CURRENT-OWNER (NAME = X)
           GN VEHICLE,      WHERE CURRENT-OWNER (NAME = X)
           GO TO LOOP
  
```

On the other hand, if data are stored in a hierarchy in accordance with Fig. 5.5.b, then the question can be answered on the basis of the following program by examining only one tree occurrence (one segment):

```

LOOP.      GU CAR-OWNER,    WHERE CAR-OWNER (NAME = X)
           GNP VEHICLE
           GO TO LOOP
  
```


In this simple example, the difference between the two methods is significant only in the search efficiency. However, in a multi-step hierarchy, even programs dealing with selection can be essentially more complicated as well if the structure of the query does not fit into the hierarchy imposed upon reality by the model. The essence of the problem is that real relationships can very often be represented only with many to many relationships while a simple hierarchical model is only suitable for describing naturally one to many relationships.

6.1. Network Data Structure

The network data structure is the generalization of the hierarchical structure. Not only 1:N (including 1:1), but any kind of relationship, so also N:M (many to many) can exist between the elements. The network needs neither a specific route nor any leaves.

In the network hierarchy there can be levels (perhaps all the levels are like this) where relationships can be multi-valued not only downwards, but also upwards, i.e., using the terminology of hierarchical models, one element can have several direct parents.

In practice, a significant part of the relationships are many to many relationships. In computer technology these are modelled by network structures.

Database management systems based on the network data model have much in common with hierarchical systems. In fact, many to many relationships — as will be seen in Section 6.2. — are decomposed to several clear 1:N hierarchical relationships. In both models, additional records connected to the anchor record can be obtained with the help of pointers built into the data base records. In both models, navigation through connected records can be performed effectively only when the data base structure is known. On the other hand, in this case, if the access path to the data is known in advance and defined by pointers, then the data base can be used generally more efficiently, or at least with the same efficiency as if the data were represented and handled in a relational model.

An example of a simple network structure can be seen in Fig. 6.1.

The conditions are the following:

Any customer can have several branch offices (addreses). One customer is always served by the same salesman. One salesman can deal with several customers. Therefore this is a 1:N type relationship.

The same applies to consumer orders. There is a unique one to one relationship between goods and their specifications. However, the same goods may be found in several orders or warehouses, and can be ordered from several suppliers. So these are all N:M relationships.

Consequently, these relationships can be faithfully represented in a network model.

The Network Model

6.1. Network Data Structure

The network data structure is the generalization of the hierarchical structure. Not only 1:N (including 1:1), but any kind of relationship, so also N:M (many to many) can exist between the elements. The network needs neither a specific route nor any leaves.

In the network hierarchy there can be levels (perhaps all the levels are like this) where relationships can be multi-valued not only downwards, but also upwards, i.e., using the terminology of hierarchical models, one element can have several direct parents.

In practice, a significant part of the relationships are many to many relationships. In computer technology these are modelled by network structures.

Database management systems based on the network data model have much in common with hierarchical systems. In fact, many to many relationships — as will be seen in Section 6.2. — are decomposed to several clear 1:N hierarchical relationships. In both models, additional records connected to the anchor record can be obtained with the help of pointers built into the data base records. In both models, navigation through connected records can be performed effectively only when the data base structure is known. On the other hand, in this case, if the access path to the data is known in advance and defined by pointers, then the data base can be used generally more efficiently, or at least with the same efficiency as if the data were represented and handled in a relational model.

An example of a simple network structure can be seen in Fig. 6.1.

The conditions are the following:

Any customer can have several branch offices (addresses). One customer is always served by the same salesman. One salesman can deal with several customers. Therefore this is a 1:N type relationship.

The same applies to customer orders. There is a unique (one to one) relationship between goods and their specifications. However, the same goods may be found in several orders or warehouses, and can be ordered from several suppliers. So these are all N:M relationships.

Consequently, these relationships can be faithfully represented in a network model.

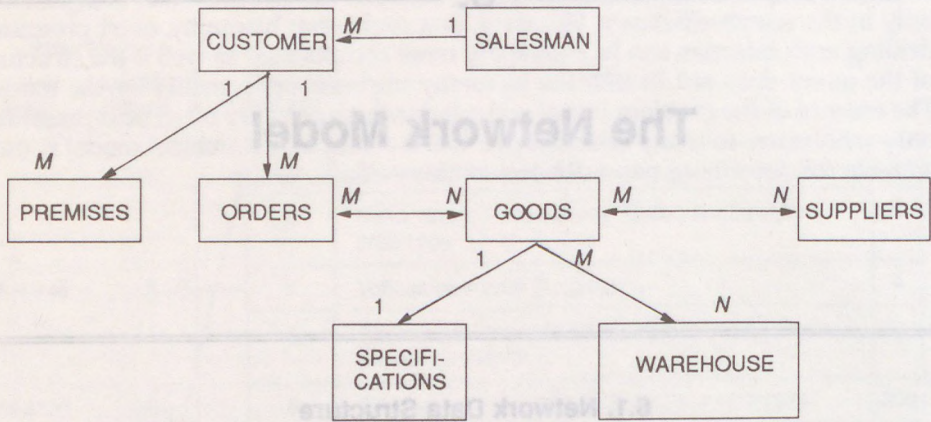


Fig. 6.1. A complex network structure to model the relationships between salesmen, customers, orders, goods, suppliers, and warehouses

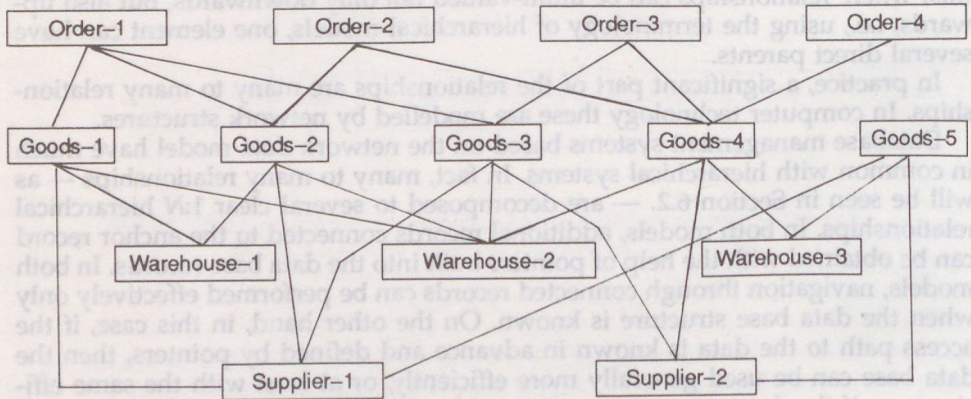


Fig. 6.2. Actual occurrences and relationships among the record types described by the many to many relationship in the data model seen in Fig. 6.1.

Some actual occurrences of the network portion of this model are shown in Fig. 6.2. Goods-2, goods-3, and goods-4 are stored in warehouse-2, but goods-2 can also be found in warehouse-1, and goods-4 in warehouse-3 as well. Thus there is a many to many relationship between the **GOODS** and **WAREHOUSE** record types.

6.2. Connecting Records

Let us look again at the simple data model described in Section 5.4. As one owner can possess more than one vehicle and one vehicle can be owned by several owners, the many to many relationship between owners and vehicles cannot be represented by the simple hierarchical model. However, in the network model this relationship can be handled with the help of a new node type, the so-called *connecting record*. Any single occurrence of it generates one and only one relationship between one VEHICLE and one CAR-OWNER record occurrence. The connecting record generally contains the data depending upon both records. (In our example, the start and end of ownership.) However, its main function is not the storage of these data, but linking the appropriate entities of different record types. From a logical aspect, connecting records can be considered as members of various chains. One of the chain types (WHO-OWNS) contains connecting records belonging to various cars, the former and the current ones of the same owner, while the other chain type (WHICH-VEHICLE) contains the connecting records of all the owners of the same car.¹

A part of a network structure with connecting records is represented in Fig. 6.3. In this model a 1:N relationship exists, both between the VEHICLE and the CONNECTING record types and the CAR-OWNER and the CONNECTING ones. That is why both the WHICH-VEHICLE and WHO-OWNS relationships can be described by a simple hierarchy. In this model — in contrast to the hierarchical models represented in Fig. 5.5. — queries that are symmetrical to owners and vehicles can be answered similarly and with the same simplicity.

The above example shows clearly the main features of the network model. With the help of connecting records, a simple hierarchy can be created in the model whose elements can participate simultaneously in several hierarchies. Naturally, in complicated data relationships, several types of connecting records may have to be created. Yet beyond that no distinction is drawn in the data model between connecting record types and those that originally existed. The name connecting record refers at most to the origin of the record.

In the network model, any record can be connected to several records of different types, generally without restriction. (Thus in particular a connecting record can also be attached to several records of different types on the basis of the WHO-OWNS and WHICH-VEHICLE relationships in the example.) However, the relationship can be only 1:1 or 1:N (or N:1, respectively) between any two record types linked directly with each other. The M:N relationships that may exist in reality are decomposed into 1:N relationships in the model by inserting connecting records. It follows from this that a theoretically indeterminate number of records of R_2 type can be directly connected to any member of a different R_1 type (R_1 and R_2 can also be a connecting record), but if there can be an r_1 element of R_1 that is directly linked with more than one element of R_2 , then any element of R_2 can be directly connected at most to one element of R_1 .

¹ The physical organization can differ from the chain. However, for the user it always appears to be realized by a chain.

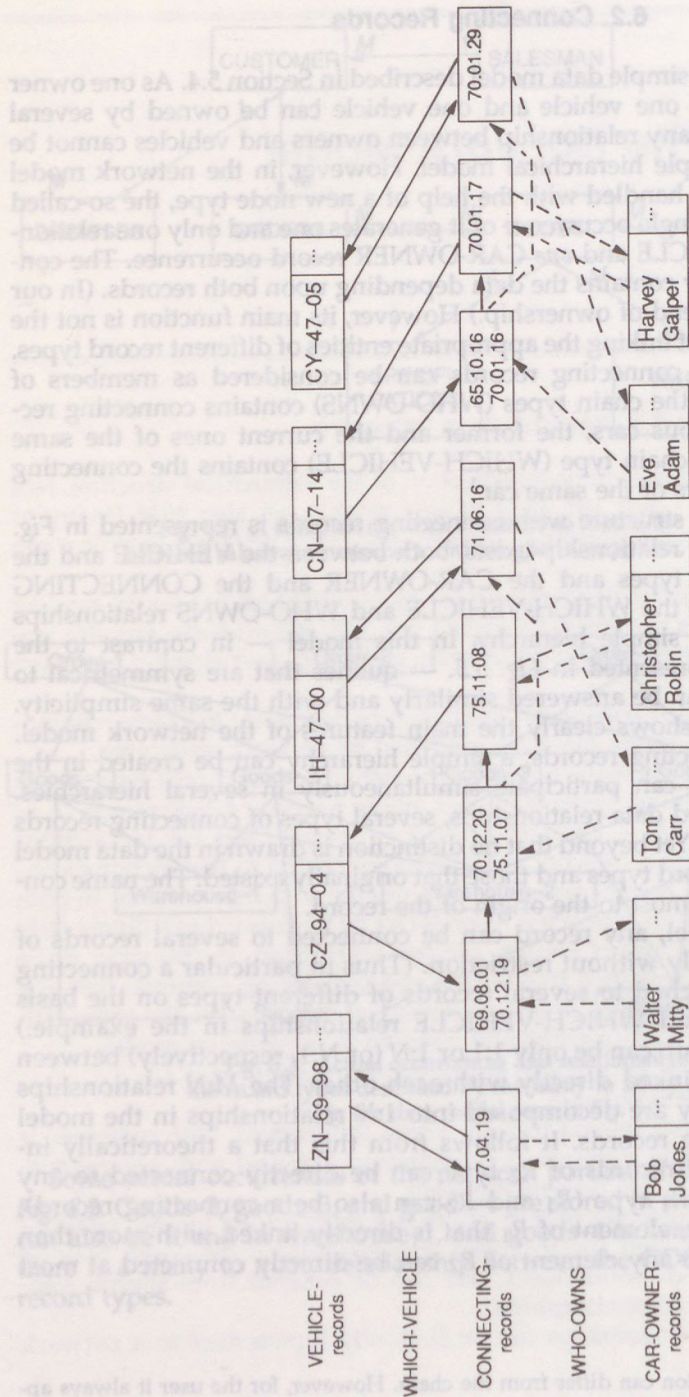


Fig. 6.3. Network representation of (a part of) the vehicle registry
(For clarity, only a part of the chains is shown.)

6.3. The DBTG Proposal

The CODASYL Committee established a special group (Data Base Task Group, DBTG) with the objective of unifying database management systems. This group studied and compared the database management systems then in use, and next prepared a report proposing what features a database management system should have. Its first draft was published in 1969, followed by the final report in 1971. It is usually called the DBTG report or proposal in honor of its authors. After preparing the report, DBTG disbanded, and two new committees were established, the Data Base Language Task Group and the Data Definition Language Committee. Their task was to elaborate the proposal in detail and embed it into COBOL and other host programming languages.

The object of the DBTG was to define a system with the following features:

- It selects the most appropriate data structure for every application independent of the fact that the same data or a part of them are used also by other applications. This flexibility must be achieved without redundancy.
- Several users or programs can simultaneously access the data base.
- Various search strategies can be used in the entire data base or in some parts of it, respectively.
- It protects the data base from unauthorized users and programs.
- It ensures central control of data storage.
- It makes the programs independent of the storage devices.
- It allows the declaration of various data structures, from the simplest data independent of each other to complicated network structures.
- The user can work with the data without regard for the maintenance of the declared structural relationships.
- It has to make the program independent of the data to the greatest possible degree.
- The data description in the data base and program have to be made independent of each other.
- The description of the data base must not be limited to one programming language.
- The data base and its description have to be compatible with various programming languages.

To achieve these goals, they advanced a proposal for three languages:

- schema definition language,
 - subschema definition language, and
 - data manipulation language.
- a) The *schema definition language* describes the data model and physical storage organization. (It leaves open certain physical assignments, e.g., the peripherals; they can actually be defined during run time.) This can also be a programming language with an independent syntax.
 - b) The *subschema definition* connects the schema and the user's program; therefore its syntax has to harmonize with the host programming languages from which the database management system can be called. As the authors of the proposal were mindful mainly of the biggest users' interests, the language specified in the DBTG proposal conforms with COBOL.

- c) Likewise, the *data manipulation language* elaborated in detail by them is also associated with COBOL. These commands can be used in a host language expanded with database manipulation features, in addition to its own original commands.²

The schema of the network model equals in essence the conceptual level of general database terminology and the subschema equals the external one (see Section 3.1.). However, at the same time, the data definition language (DDL) proposed by DBTG — and implemented in network data base management systems used in practice — does not remain only at these levels. As will be seen in Section 6.4., specific instructions must be given about storage organization and access path in this language, thus the inner physical level of the data base is defined rather profoundly.

The examples below use the syntax proposed by DBTG or the commands of one of the most widespread network database management systems, IDMS (Integrated Database Management System) marketed by Cullinet Software Inc.

6.4. Schema Definition

The schema definition is composed of four main parts:

- the name of the schema,
- the definition of the areas,
- the definition of the records, and
- the definition and description of the sets.³

Their sequence is settled to a certain degree. Schema naming always comes first. As for the others, definitions of individual areas, records, and sets can alternate, but the area definition has to precede the definition of those records that are located in that area, and any set can be given only after the definition of all its records.

The schema definition of the vehicle registry is shown in Fig. 6.4.; the registry has already figured in prior examples. The syntax proposed originally by DBTG is used in the example. Commands of particular database management languages can differ formally from this to a certain degree.

Statement 1 gives the name of the schema (VEHICLE-REGISTRY) and 1a and 1b define the conditions under which the schema can be modified or copied into subschemas.

If the program would like to change the schema (of course, this is not the same as changing data in the data base), then it must fulfill the conditions of the procedure CHECKING. To copy the schema or parts of it into the subschema, the password CAN-BE-COPIED must be given. Both were supplied by the data base administrator to ensure the secrecy and protection of the data base. A similar protection can be ensured against displaying or printing the schema or changing the record locks that provide control.

² Naturally, most of the commercial network systems presently available can be embedded into other programming languages, for example, into PL/1 or Assembler.

³ To avoid confusion, the concept 'set' which is used in the DBTG proposal is not equivalent to 'data set' used in IBM terminology.


```

1  SCHEMA NAME IS VEHICLE-REGISTRY;
1a  PRIVACY LOCK FOR ALTER IS PROCEDURE CHECKING
1b  PRIVACY LOCK FOR COPY IS 'CAN BE COPIED'.
2  AREA NAME IS CAR-OWNER-AREA.
2a  AREA NAME IS VEHICLE-AREA;
2b  ON OPEN FOR UPDATE CALL DEALER.
3  RECORD NAME IS VEHICLE;
3a  LOCATION MODE IS CALC LICENCE-PLATE-LOCATION USING LICENCE-PLATE
3b  DUPLICATES ARE NOT ALLOWED;
3c  WITHIN VEHICLE-AREA.
3d  02 LICENCE-PLATE.
    03 LETTER-SYMBOL; PICTURE IS A (2).
    03 DASH1; PICTURE IS X.
    03 NUMBERS.
    04 NUMBER1; PICTURE IS 9 (2).
    04 DASH2; PICTURE IS X.
    04 NUMBER2; PICTURE IS 9(2).
3e  02 MAKE; PICTURE IS X(12).
3f  02 MODEL; PICTURE IS X(12).
3g  02 COLOR; PICTURE IS X(10).
3h  02 ENGINE-SIZE; PICTURE IS 9(4).
3i  02 OTHERS.
    03 ...
    :
3j  RECORD NAME IS CAR-OWNER;
3k  LOCATION MODE IS CALC NAME-PLACE USING OWNER NAME
3l  DUPLICATES ARE NOT ALLOWED;
3m  WITHIN OWNER-AREA.
    :
3n  RECORD NAME IS CONNECTING;
3o  LOCATION MOD IS VIA WHO-OWNS SET;
3p  WITHIN OWNER-AREA.
3q  02 FROM-WHAT-TIME.
3r  03 FROM-YEAR; PICTURE 9(4).
3s  03 FROM-MONTH; PICTURE 9(2).
3t  03 FROM-DAY; PICTURE 9(2).
3u  02 FOR-HOW-LONG.
    :
4  SET NAME IS WHICH-VEHICLE;
4a  MODE IS CHAIN;
4b  ORDER IS SORTED;
4c  OWNER IS VEHICLE
4d  MEMBER IS CONNECTING
4e  MANDATORY AUTOMATIC;
4f  ASCENDING KEY IS FROM-WHAT-TIME DUPLICATES ARE NOT ALLOWED;
4g  SET OCCURRENCE SELECTION IS THRU LOCATION MODE OF OWNER.
4h  SET NAME IS REGISTERED-VEHICLES;
4i  MODE IS CHAIN,
4j  ORDER IS SORTED;
4k  OWNER IS SYSTEM
4l  MEMBER IS VEHICLE
4m  MANDATORY AUTOMATIC;
4n  ASCENDING KEY IS LICENCE-PLATE DUPLICATES ARE NOT ALLOWED.
4o  SET NAME IS OWNER;
    :
    :

```

Fig. 6.4. Schema description in the schema definition language proposed by the DBTG

In the data definition language, the complete syntax of the schema-protecting clause is the following:

$$\text{PRIVACYLOCK FOR action IS } \left\{ \begin{array}{l} \text{literal} \\ \text{lockname} \\ \text{procedure} \end{array} \right\}$$

where action can mean the altering (ALTER), copying (COPY), displaying (DISPLAY) of the schema, or changing of the record locks (LOCKS).

The data base administrator can divide the data base into one or more areas (AREA).

One area can be, for example, a file in the conventional sense, a magnetic disk pack, or an assigned cylinder, etc. With the appropriate assignment of areas and data stored in them, the data base administrator can speed up the system (he assigns a rapidly accessible area for frequently-used data or stores physically close to each other data that are used together, etc.) and tighten its security to a high degree. (For example, he gathers important data to one area protected by special security measures, passwords or procedures, or which is often saved, etc.) In some systems, the data base administrator can also specify the area where a given record type should be stored physically.

Statement 2 defines the CAR-OWNER-AREA area, and statement 2a defines the VEHICLE-AREA area. (In the example the two different areas were defined only for the purpose of introducing these options.) The latter can be modified (UPDATE) as a consequence of statement 2b only by programs that comply with the requirements of the procedure called DEALER that is defined by the data base administrator. This is checked at run time before writing into the area when the program executes the OPEN command.

Record descriptions (3-3u) define the names of individual record types, their access paths, location mode, as well as their storage location, and all fields of the record.

Statement 3 describes the VEHICLE record. In 3a-3c it is defined that individual records are accessed by hashing (see Section 4.8.) (LOCATION MODE IS CALC). The algorithm defined by the procedure called LICENSE-PLATE-LOCATION is used with the actual value of LICENSE-PLATE (USING). The address will be assigned within the VEHICLE-AREA area (WITHIN) and no two cars can have the same license plate (DUPLICATES ARE NOT ALLOWED). Consequently, if an attempt is made to enter into the data base a VEHICLE record having the same license plate as another record already existing in the data base, then the system will report an error and refuse the insertion. Further statements (3d-3i) describe the individual fields and data elements of the VEHICLE record. The clause PICTURE IS gives the length of the field and stored character types (A=alphabetical, 9=numeric, X=alphanumeric. The field length is indicated in the clause either by a number in parentheses or by its number of repetitions.) Data elements can be grouped also. To illustrate this, for example, the LICENSE-PLATE group field is composed of several elementary fields, though alternatively, it could be stored more compactly in a smaller place — for example, by dropping hyphens which are obviously unnecessary. Statements 3j-3m define the same for the CAR-

OWNER record type. The simplifying assumption that the NAME identifies unambiguously the owner is maintained here also.

Statements 3n-3u define the CONNECTING records. They maintain the relationship between CAR-OWNER and VEHICLE type records. The data base administrator wishes to place the CONNECTING type records near to the occurrence of the CAR-OWNER records to which they are connected (VIA WHO-OWNS set, 3p-3o) in the CAR-OWNER-AREA area on the basis of some other criteria (WITHIN).

6.5. The Set Conception

One of the most fundamental points of the DBTG proposal was the introduction of *set* conception. Sets establish the various network and hierarchical relationships among the data in the data base.

In its simplest form, a set is composed of two record types. Both of them can contain several fields. One record type is the *owner*, the other is the *member*. Each set occurrence contains exactly one occurrence of the owner type. Any number of occurrences of the member type can be linked to it. (If the number of members is zero, then the set is called empty.) So there is a simple hierarchical relationship between the occurrences of owner and member records in which the owner record is at the higher level of the hierarchy. It follows from the hierarchy that inside the *same* set one member can belong to *only one* owner. Naturally, this does not rule out that in a set of *another type*, the same occurrence can belong to another owner record. So, for example, in Fig. 6.3. the WHICH-VEHICLE set establishes a relationship between records of VEHICLE type as an owner and CONNECTING type records as members. In that figure, five occurrences of the set can be seen whose owner records are cars with license plate numbers ZN-69-98, CZ-94-04, ... and CY-74-05. There is only one member in the first, third, and fifth set occurrences, while there are three of them in the second and two in the fourth.

As the hierarchy can have several levels, similarly, it is also possible that in one set *several types* of members belong to the same owner type, and those records that occur as members in one set are the owners of one or more records of another type in another set. In this sense, the set matches the tree structure of the simple hierarchy where the owner is the root of the tree and the members are its direct children. Additional owners are all the non-leaf nodes and additional members are all the nodes except for the root. The relationship between the parent and its direct children is the set. Thus, the nodes, except for the root and the leaves, are owners and members simultaneously, but naturally in two different sets.⁴

Another set, the WHO-OWNS set, can also be seen in Fig. 6.3. Its members are the same CONNECTING records that are also members in the WHICH-VEHICLE set, but the owner is the CAR-OWNER record type. With the help of the common member records, the two sets describe a network data structure. In fact, they realize a many to many relationship between the VEHICLE and CAR-OWNER records types.

⁴ It is essential that the owner and member records are not only different from each other, but also ones of different types. Likewise, it is impossible that if in a S_1 set the owner is the A type record and the member is B , then in a S_2 set B is the owner and A is the member. Structures of this nature can be realized in DBTG data models only by inserting connecting records.

A special type of set is the *singular set* owned by the system. Every singular set can have only one occurrence, but any number of them can be declared in a data model. So, for example, the sequential file in the usual sense can also be considered as a singular set.

The above example contains two different singular sets containing the VEHICLE and CAR-OWNER records, respectively, as members.

The data base administrator establishes the relationships among various data by defining sets. That is precisely why the schema description must contain also a detailed description of each individual set since the database management system builds the physical relationships according to the logical relationships defined there. The set defining part of the data base declaration seen in Fig. 6.3. is also described in the schema definition language proposed by the DBTG as seen in Fig. 6.4. It contains in addition to the two sets presented there, the singular REGISTERED-VEHICLES set for sequential vehicle processing.

Statement 4 (SET NAME IS) declares the WHICH-VEHICLE set; 4a, 4b, and 4f specify that the member records, ordered by date of sale (ORDER IS SORTED, ASCENDING KEY IS FROM-WHEN), are linked by a chain (MODE IS CHAIN). 4c and 4d define the types of owner (OWNER IS) and member (MEMBER IS) records. Statement 4e defines how members can be inserted into or deleted from the set. This can be any one of the four combinations of the options MANDATORY or OPTIONAL and AUTOMATIC or MANUAL.

The MANDATORY specification means that once the record has been connected into the set, then it can be removed from there only by physical deletion from the data base (DELETE). (Naturally, within the *same* set it can be moved to *another occurrence* with a modifying (MODIFY) command.) Records that are specified by OPTIONAL can be removed from the set with the REMOVE command without their mandatory deletion from the data base. (So these records can continue to be members or owners of *other* sets, even after they have been deleted from this set.) While deleting, in addition to this, it has to be kept in mind that the deletion of an owner record automatically deletes all its member records as well.

In a set defined by the AUTOMATIC specification, if a new member is entered into the data base, then the system automatically inserts it into the set by producing and/or modifying the appropriate pointers. If MANUAL is selected, it does this only after an additional INSERT command.

Since any newly-inserted member record has to be placed into the proper set occurrence by the database management system, it also has to be defined how this selection process must be conducted. The clause SET OCCURRENCE SELECTION IS THRU serves this purpose. Only the two most important methods are mentioned here. The first is when the owner is assigned by the method defined in the schema description (LOCATION MODE OF OWNER). The new record becomes a member of the set occurrence of the owner obtained in this way. This solution was used in statement 4g in Fig. 6.4. Thus in the WHICH-VEHICLE set, the system inserts a new CONNECTING record into that occurrence of the VEHICLE set whose owner has the value for LICENSE-PLATE equal to the value provided by the user. In the other oft-used selection method, the system connects the new record into the current set occurrence (CURRENT OF SET); i.e., the one

which was last referenced within the given set type either by its owner or by one of its member records. In statement 4f, just as in the record definition, the clause (DUPLICATES ARE NOT ALLOWED) means there cannot be two members with the same ordering field value inside the same set occurrence. (This does not mean that the value of the FROM-WHEN field cannot be the same in two CONNECTING records; it means only that it cannot be the same *within one set occurrence*. This arises from the reasonable assumption that the owner of a car will not change twice in one day.)

The 4h-4n statements describe similarly the REGISTERED-VEHICLES singular set (OWNER IS SYSTEM). Its members are the VEHICLE records ordered according to the unambiguous LICENSE-PLATE. This set enables serial processing of VEHICLE records.

The WHO-OWNS set has to be defined similarly.

The DBTG-proposed schema definition allows different data representations in the physical storage and in the program. FOR ENCODING CALL procedure-1 and FOR DECODING CALL procedure-2 statements serve this purpose. So, for example, if data are condensed for better storage utilization (see Point 4.9.1.), or stored encoded for safety reasons (see Point 4.9.2.), and de facto physically stored data are derived from the original ones by a procedure called ENCODE and data are restored to their original form from the stored ones by a procedure called DECODE, then as a result of the following field description,

```
02 VEHICLE-MAKE; PICTURE IS X(12);
   FOR ENCODING ALWAYS CALL ENCODE;
   FOR DECODING ALWAYS CALL DECODE;
```

the user will always see the VEHICLE-MAKE field as an alphanumeric field with twelve characters. However, its stored value in the data base will always be the mapping of this character string produced by a procedure called ENCODE, for example, a one byte long binary number between 0 and 255.

Redundancy can also be decreased in the record description by the VIRTUAL clause. Data defined in this way are not stored physically by the system. They take their value directly from the data element given by SOURCE, or produce it as a result of the procedure defined in the RESULT OF clause. Thus, for example, if the procedure AVERAGING produces the average of the A1, A2, ..., AN variables, then the variable declared below occupies no real space in the data base, but authorized programs can refer to it and use its value, which will be the average of the actual values of A1, A2, ..., AN

```
03 AVERAGE; IS VIRTUAL RESULT OF AVERAGING...
```

The entire syntax of the schema definition language (DDL=Data Definition Language) and the detailed explanation of individual commands are contained in the DBTG report and the systems handbooks designed on the basis of their proposal, respectively. Most of the systems on the market offer generally only a portion of the options permitted by DBTG.

6.6. Subschema Description

Individual users deal usually with only certain parts of the data base described in the schema. It is the duty of the data base administrator to define in the subschema(s) what parts of the data base can be accessed by various users and programs. In addition, he also has the option to change the field names, data representation, selection methods, and security locks within the subschema. Any number of subschemas can be declared on one schema, any number of programs can use the same subschema, and different subschemas can overlap one another.

In Fig. 6.5. a subschema description is shown allowing only relationships among vehicle license plates and ownership periods on the schema defined in Fig. 6.4.

```

1  SUB-SCHEMA IDENTIFICATION DIVISION.
2    SUB-SCHEMA NAME IS CAR OF SCHEMA NAME
      VEHICLE-REGISTRY
2a   PRIVACY KEY FOR COPY IS 'CAN-BE-COPIED'.
3  SUB-SCHEMA DATA DIVISION.
4    RENAMING SECTION.
4a   RECORD NAME CONNECTING IN SCHEMA IS CHANGED TO OWNERSHIP.
5    AREA SECTION.
5a   COPY ALL AREAS.
6    RECORD SECTION.
6a   01 VEHICLE.
6b     02 LICENSE-PLATE; PICTURE IS X(8).
6c   01 OWNERSHIP.
6d     02 FROM-WHAT-TIME,
6e       03 FROM-YEAR; PICTURE IS 9(4).
6f       03 FROM-MONTH; PICTURE IS 99.
6g       03 FROM-DAY; PICTURE IS 99.
6h     02 FOR-HOW-LONG.
      :
7  SET SECTION
7a   COPY WHICH-VEHICLE, REGISTERED-VEHICLES.

```

Fig. 6.5. A subschema description
in the subschema definition language proposed by the DBTG

Statement 1 says that a subschema description follows. Statement 2 gives the name of the subschema (SUBSCHEMA NAME IS) which is CAR in our example, and this is a part of the VEHICLE-REGISTRY schema (OF SCHEMA NAME). Statement 2a gives the password necessary for copying the VEHICLE-REGISTRY schema (PRIVACY KEY FOR COPY IS). This is necessary because the ability to be copied was related to the 'CAN-BE-COPIED' password in the schema definition by the data base administrator (see statement 1b in Fig. 6.4.). Statements 4 and 4a enable the record, defined under the CONNECTING name in the schema, to be referred to by the OWNERSHIP name in the subschema.

With statements 5 and 5a the descriptions of the schema area are copied by the system into the subschemas (COPY). Statements 6-6h describe the records

Command	Function
CLOSE	Releases previously occupied set type(s) and area(s)
DELETE	Deletes the given record(s) from the data base
FIND	Based upon the record selection condition, selects the appropriate record(s)
FREE	Inactivates the previous KEEP command
GET	Transfers the actual data to the user's work area
IF	Enables the evaluation of conditions connected to the database
INSERT	The record will be a member of the given set(s)
KEEP	Informs the system that the program would like to use further the actual record
MODIFY	Modifies the given record(s)
MOVE	Transfers the actual attributes
OPEN	Opens the given area(s) for the program, checks the access authority
ORDER	Rearranges the members of the set given in the command
REMOVE	Terminates the connection between the record(s) and the set(s) given in the command
STORE	Stores physically the given record(s) and inserts them logically into the suitable sets
USE	Defines an error handling procedure

Fig. 6.6. Commands of the data manipulation language proposed by the DBTG

defined in the subschema while statements 7 and 7a define the WHICH-VEHICLE and the REGISTERED-VEHICLES sets in the same way as defined in the schema (COPY). (In the set description, the data base administrator can change the selection method and set protection in the subschema, but he cannot modify its other features.)

6.7. Data Manipulation Language

The commands and functions of the data manipulation language proposed by DBTG are summarized in Fig. 6.6. Their realization in the IDMS system is summarized in Fig. 6.7. A detailed commentary about individual commands would be beyond the scope of a general guide; that is why only the most essential features of some commands are shown in the example that has already been mentioned.

The task is akin to that in the hierarchical model: 'Look up for the purpose of further processing the names and addresses of all owners of green Buicks.' Only those parts of the program — written in COBOL as a host language — which are affected by interacting with the database management system are shown in Fig. 6.8. The program works with the COMPLETE-DATABASE subschema of the VEHICLE-REGISTRY schema defined in Fig. 6.4.; it was defined by the data base administrator to be identical to the schema. Since only the UPDATE function was secured by a checking procedure (2b in Fig. 6.4.) in the schema description, a program that seeks only to read can access the entire data base without further ado.⁵

⁵ If it was not so, then proper values for the PRIVACY KEY or appropriate procedures, respectively, would have to be given in the IDENTIFICATION DIVISION.

Command	Function
BIND	Binds the appropriate record types to the program operating on the database
CONNECT	Connects the given record to the appropriate set
DISCONNECT	Terminates the connection between the record and the set given in the command
ERASE	Erases the given record from the database
FIND	Based upon the record selection condition, selects the appropriate record(s)
FINISH	Finishes the database handling operations, releases the previous occupied record and set types and areas
GET	Transfers the actual record to the user's work area
IF	Enables the evaluation of the conditions connected to the database (for example, whether a given record is the member of a set type or a concrete set occurrence)
MODIFY	Modifies the suitable record in the database
OBTAIN	On the basis of the record selection condition, selects the appropriate record and transfers it to the user's work area
READY	Opens the appropriate area(s) for the user for the requested processing method (for example, for exclusive use, for read-only, also for modification)
STORE	Stores physically the given records and inserts them logically into the suitable sets

Fig. 6.7. Data manipulation commands of the IDMS system

In the data definition section of the COBOL program (DATA-DIVISION), it has to be specified with which subschema of which schema will the program work (1a and 1b). Besides this, the ERROR-STATUS variable is defined in the WORKING-STORAGE-SECTION (2-2a). After error-free execution of any data definition or manipulation command, its value will be set to zero. Any error or unforeseen event during execution results in a non-zero value. For example, after executing the FIND command the value of ERROR STATUS will be 0307 when the end of set is reached.

In the procedure part (PROCEDURE DIVISION), DECLARATIVES indicate what steps must be taken when a data manipulation operation is not executed error-free (4-4h). The USE command serves this purpose. The end-of-set condition is handled separately; it is not considered an error, but only the end of the processing steps.

After opening the sets (5), the first VEHICLE record is searched in the REGISTERED-VEHICLES set with the FIND command (6).⁶ Note that locating a record does not also entail its reading; so the program will get the record after the GET command. Beginning from this point the program executes a loop (7-7b) from which it exits if there are no more vehicle records (7a).

⁶ The most important forms of the command FIND are the following: the command FIND FIRST record-name RECORD OF set-name SET searches the first member record, the command FIND LAST the last one, the command FIND NEXT the next one, and the command FIND OWNER searches the owner of a given actual set. The command FIND record-name USING-identifier enables direct locating on the basis of the data base key being in the identifier, while the command FIND NEXT DUPLICATE WITHIN record-name RECORD selects the next record having the same identifying value on the basis of the record key value that serves direct access. (Naturally, only when the existence of such records was permitted by the schema description.) The command FIND CURRENT OF record-type gives the current (put into use most recently) record of the given record type, while the command FIND CALC record-type gives the directly-accessible record that has the given key value.


```

DATA DIVISION.
1  SCHEMA SECTION.
1a  INVOKE SUB-SCHEMA COMPLETE-DATABASE OF SCHEMA
    VEHICLE-REGISTRY.
.
2  WORKING-STORAGE SECTION
.
2a  77 ERROR-STATUS PICTURE 9999.
.
3  PROCEDURE DIVISION.
4  DECLARATIVES.
4a  FORESEEN-ERROR SECTION.
4b    USE IF ERROR-STATUS = '0307'
4c  SET-END-HANDLING.
4d    EXIT.
4e  ACTUAL-ERROR SECTION.
4f    USE IF ERROR-STATUS = ...
4g  ERROR-HANDLING.
    error-handling commands
4h  END DECLARATIVES.
.
5  OPEN ALL.
.
6  FIND FIRST VEHICLE RECORD OF VEHICLE-REGISTRY SET.
7  END-OF-SET-CONTROL-1.
7a  IF ERROR-STATUS = '0307' PERFORM END-OF-DB-PROCESSING
    ELSE PERFORM VEHICLE-SELECTION.
7b  GO TO END-OF-SET-CONTROL-1.
.
8  VEHICLE-SELECTION.
8a  GET VEHICLE.
8b  IF MAKE = 'VOLKSWAGEN' AND COLOR = 'GREEN'
    PERFORM OWNER - SELECTION THRU THERE-ARE-NO-MORE.
8c  FIND NEXT VEHICLE RECORD OF VEHICLE-REGISTRY SET.
.
9  OWNER-SELECTION.
9a  FIND FIRST CONNECTING RECORD OF VEHICLE.
10  END-OF-SET-CONTROL-2.
10a  IF ERROR-STATUS = '0307' GO TO THERE-ARE-NO-MORE.
10b  FIND OWNER RECORD OF WHO-OWNS SET.
10c  GET OWNER,
    processing and copying of suitable data
10d  FIND NEXT CONNECTING RECORD OF VEHICLE.
10e  GO TO END-OF-SET-CONTROL-2.
10f  THERE-ARE-NO-MORE. EXIT.
.
11  END-OF-DB-PROCESSING.
11a  CLOSE ALL.
.

```

Fig. 6.8. Data retrieval with the help of a network type database management system

If a VEHICLE record is found with the FIND command, then the VEHICLE-SELECTION procedure must be executed (8-8c). The actual VEHICLE record is read, namely, the one defined previously with the FIND command. If its data satisfy the condition, then the procedure defined in 8b is executed (from OWNER-SELECTION until THERE-ARE-NO-MORE); if not, then the next record of the REGISTERED-VEHICLES set is searched (8c) and the loop 7-7b resumes, namely the sequential reading of VEHICLE records. It should be noted that the COBOL program can refer only by name to individual fields of the database records defined in the subschema without further declarations.

If the read VEHICLE record fulfills the selection conditions, then the first CONNECTING record of the actual WHICH-VEHICLE set is obtained (9a), as well as its owner CAR-OWNER record in the WHO-OWNS set (10b). The latter is read and processed. Then the same is executed with the owner of the next CONNECTING record (10d,e, and 10b,c, respectively) until all the owners of this car have been processed (10a). Once this occurs, the next VEHICLE record is searched and processing continues with it. It should be noted that with command 10d the CONNECTING records are read via the WHICH-VEHICLE set.

Naturally, the wide range of the entire DBTG data manipulation language cannot be shown with a relatively simple example like this. However, if the user ever tried himself to program in any programming language the task discussed above, then he would soon realize the merits of such data manipulation languages.

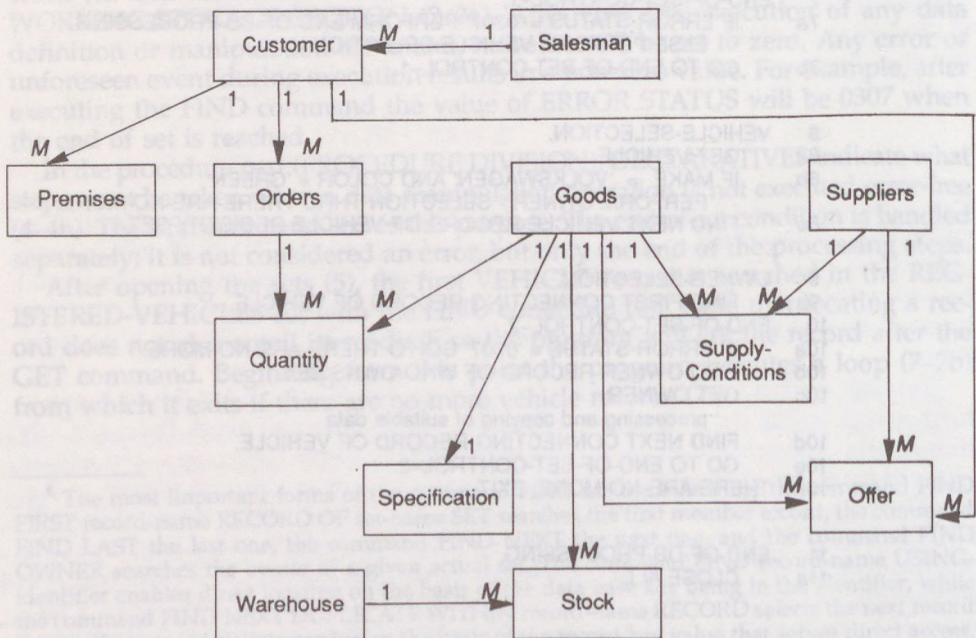


Fig. 6.9. Realization of the logical model shown in Fig. 6.1. with the insertion of connection records

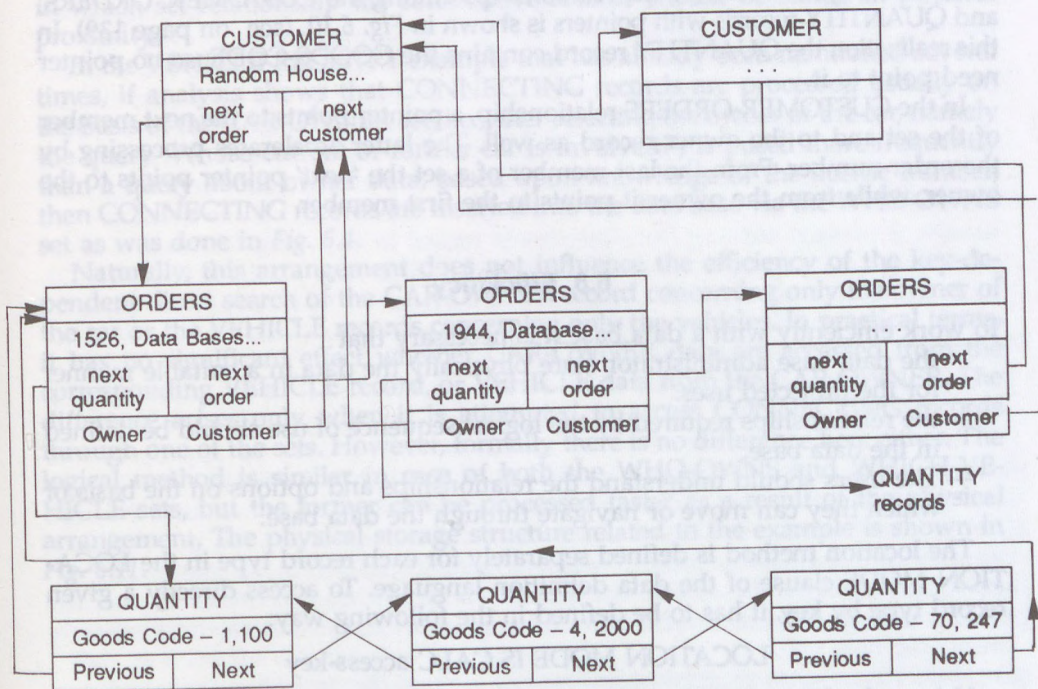


Fig. 6.10. Part of Fig. 6.8. Relationships are realized with pointers between the records

To illustrate further the features of the network data models, it is shown in Fig. 6.9. (see on page 138) how the data structure shown in Fig. 6.1. can be realized.

The M:N relationship among the records of ORDERS and GOODS is removed by inserting the connecting record QUANTITY. This indicates the ordered quantity of a specific goods in a given order. (Besides this, it also contains the unit price if it can vary within orders.) Which goods code and order number this quantity refers to can be stored directly in the QUANTITY record as well, but it is also possible that they can be obtained only as the result of two additional 'find owner' operations.

Two connecting record types can be created among GOODS and SUPPLIER records. One, DELIVERY-CONDITIONS, specifies under what conditions the given goods can be shipped by the supplier and the typical delivery terms (delivery deadline, shipping cost, quantity discounts, etc.) while OFFER concerns the actual delivery and depends also upon which CUSTOMER is involved.

GOODS and WAREHOUSE records are connected by the STOCK record type. The model also includes, in addition to the 1:M and 1:1 set relationships shown in Fig. 6.8., five singular sets whose members are the CUSTOMER, GOODS, SUPPLIER, WAREHOUSE, and SALESMAN records.

How the system realizes the relationships among the CUSTOMER, ORDERS, and QUANTITY records with pointers is shown in Fig. 6.10. (see on page 139). In this realization the QUANTITY record contains the GOODS-CODE, so no pointer need point to it.

In the CUSTOMER-ORDERS relationship, a pointer points to the next member of the set and to the owner record as well. The latter accelerates processing by the order number. From the last member of a set the 'next' pointer points to the owner, while from the owner it points to the first member.

6.8. Efficiency

To work efficiently with a data base it is necessary that

- the data base administrator store physically the data in a suitable manner for the projected uses.
- the relationships required by the logical sequence of use should be defined in the data base.
- The users should understand the relationships and options on the basis of which they can move or navigate through the data base.

The location method is defined separately for each record type in the LOCATION MODE clause of the data definition language. To access directly a given record type by key, it has to be defined in the following way:

LOCATION MODE IS CALC access-key

Using a hashing algorithm (see Section 4.8.), these records are placed within the given area by the system, so any of them can be obtained very quickly by key value, usually in one step.

However, on the other hand, these records can be processed in a predefined sequence only if a set relationship has been defined among them in accordance with this condition. Yet the efficiency of sequential processing would not be optimal in this case since the physical data sequence and their logical processing order would not agree. Logically consecutive records would be read by the system from different physical disk pages using the set relationship pointer chain. This would entail a lot of paging, therefore relatively slow processing.

If it is known that processing occurs generally on the basis of the set relationship, — namely, after the owner is found, the member records of the set will also be needed later — then it is expedient to store the entities of the member record type physically according to the set relationship. This is ensured in the following manner:

LOCATION MODE IS VIA set-relationship

In this case, the system tries its best — provided there is a way to accomplish it — to store members also physically close to their owner records, if possible, on the same page. In this case, processing via the set relationship can be very fast since the physical storage and logical processing orders agree, or, at least, the records are clustered; paging is rarely needed.

Naturally, if the same record type occurs in several sets, then this optimal processing arrangement can usually be realized only for one set. The data base administrator's task is to decide, on the basis of projected processing requirements,

in which set relationship the consecutive records should be stored in physical proximity.

In the VEHICLE-REGISTRY example that has already been mentioned several times, if analysis shows that CONNECTING records are processed usually on the basis of the CAR-OWNER set, i.e., after selecting the owner of the car, namely the query 'Whose current or former car is involved?', is posed more frequently than a query about owner data, based upon knowledge of the license number, then CONNECTING records are inserted into the data base via the WHO-OWNS set as was done in Fig. 6.4.

Naturally, this arrangement does not influence the efficiency of the key-dependent direct search of the CAR-OWNER record concerning only the owner of the car or the VEHICLE records concerning only the vehicles. In practical terms, it has no significant effect whether CAR-OWNER data are accessed from the corresponding VEHICLE record, or VEHICLE data from the CAR-OWNER. The difference arises only when it is attempted to access CONNECTING records through one of the sets. However, formally there is no difference here either. The logical method is similar in case of both the WHO-OWNS and WHICH-VEHICLE sets, but the former can be processed faster as a result of the physical arrangement. The physical storage structure related to the example is shown in Fig. 6.11.

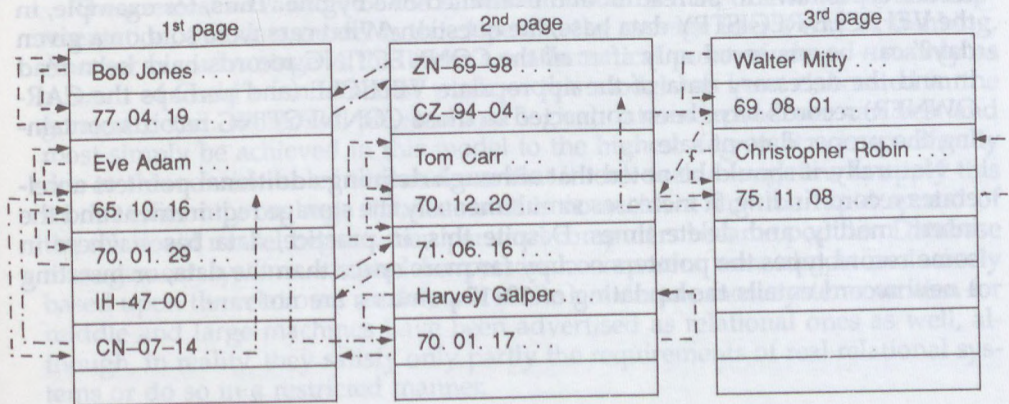


Fig. 6.11. Physical location of records based on set relationship

The pointer of the WHO-OWNS relationship is indicated by a broken line (- -) and that of the WHICH-VEHICLE relationship by a dotted broken line (-.-.-). The physical insertion was based on the WHO-OWNS set, therefore the records belonging to one set occurrence are on the same page. It can be seen that on the same page, independent of the storage defined in the set relationships, there can be occurrences of various record types depending upon the CALC location

The pointers defined in the sets also influence considerably the processing efficiency. The system constructs a forward pointer automatically in every set.

However, to deduce the owner from individual members of the set, it is expedient to define one more pointer in the set pointing to the owner from each member. This is particularly efficient if there are many members in the individual set occurrences. Having a pointer to the owner, it is not necessary to traverse the remainder of the forward pointer chain to find the owner. (On the average, the remainder is half of the members of the given occurrence.) If the members are usually processed starting from the owner record in a given order, then it is expedient to create the set pointer chain sorted on this field (ORDER IS SORTED). In that case, records of the set occurrence do not have to be read first, temporarily stored, and then sorted; rather the FIND NEXT command finds the records automatically in their appropriate order.

Individual set members are often stored within the set in the chronological order of their entry into the data base, and they are also processed in this way. If set occurrences consist of many members, then it may be appropriate to define also a backward pointer in the set. If this is not done, then the system can determine the location of a new member only by first proceeding through the entire chain (or the remainder of it, if the owner is not the entry point). This is rather time-consuming when several hundreds of elements are involved. If backward pointers exist, then starting from the owner record the system can find the last record in one step and after this it can immediately insert the new one.

Unfortunately, no set relationship can be created dynamically in network systems. So if there is no set defined among the records based on a given relationship, then processing according to this relationship, all records of the appropriate record types have to be read in and examined one by one. Thus, for example, in the VEHICLE-REGISTRY data base, the question 'What cars were sold on a given day?' can be answered only after *all* the CONNECTING records have been read in and the necessary data of the appropriate VEHICLE (and perhaps the CAR-OWNER) records have been connected to those CONNECTING records containing the proper date of sale.

Naturally, it should be noted that although defining additional pointers accelerates record finding, it increases simultaneously the storage requirement and the insert, modify, and delete times. Despite this, in practice, data bases where in some record types the pointers occupy far more space than the data, or inserting a new record entails the updating of 10-15 pointers, are not rare.

The Relational Model

7.1. Introduction

In the relational model, data are logically represented in relations, visualized as tables. The various relationships among the data are established with relations, and among the relations with relational operations. Data structures and the data are defined through the tables. Data manipulation is realized by relational operations. In addition to these two basic tasks, the relational model facilitates data integrity, and knowing the various tasks to be solved, structures can be created that increase the efficiency of applications.

Besides this, like all database management systems, it contains the data base definition itself. Based upon this, the data base administrator and users can work with the system and obtain current information from it.

The greatest advantage of the relational model is that, due to its tabular representation, this data model is the closest to the average user's way of thinking. Another big advantage is that from a mathematical point of view — mainly as a result of Codd's fundamental work — this model has been elaborated in the greatest detail. Moreover, physical and logical data independence can best and most simply be achieved in this model to the highest degree. It was specifically due to this considerable independence that it took such a long time to apply this model efficiently on large data bases. It is no accident that the rapid spread of the relational model began when personal computers became popular. Database management systems adopted for personal computers were almost exclusively based upon the relational model, and in the last decade most systems written for middle and large machines have been advertised as relational ones as well, although, in reality, they satisfy only partly the requirements of real relational systems or do so in a restricted manner.

7.2. Features of Relations

Mathematically speaking, the relational model consists of one or more two-dimensional special tables, in other words, relations. These relations define various entities of the real world and their attributes. The relations have columns and rows.

The names of the columns form the heading of the relation (table). The entries of the rows are the data of the relation. The rows characterize the entities of the given relation and the columns the individual attributes. In classical file organization terminology, the relation is considered one file; the entries, the rows, are the individual records of the file, the columns are the elementary fields of the records, and the individual data elements of the table are the values of the appropriate record fields.

To form truly relational data bases, the individual tables need to satisfy the following conditions:

- Every relation has an unambiguous name identifying it in the data model.
- There are the same number of columns in each row of a given relation. The number of columns (n) is usually called the *degree* of the relation. The rows are often called tuples or n tuples. The degree of the relation is constant in the model. If it is changed, then it means actually the end of the original relation, and the creation of a new one.¹
- Every column can be identified by an unambiguous name within the given relation.
- Every column can contain only well-defined values. (They can also come from a continuous domain.) In their entirety, they are called the domain of the attribute values of the given column. The domain contains only the same type of elementary data identical from the model's point of view. The domain is always application-dependent.
- Each column of any row can have only one value from among the allowed ones. This means that the column values are elementary. They cannot contain repeating fields.

A simple vehicle owner (V&O) registry relation can be seen in Fig. 7.1. It contains only the minimum data necessary for determining ownership. It is assumed throughout this chapter that persons are identified unambiguously by their names. (If, for example, there were two Bob Jones, then they would be named Bob Jones 1 and Bob Jones 2 to ensure thereby the uniqueness of the name.) This relation defines an appropriate part of the real world only if vehicle ownership cannot be shared. Each car has one and exactly one owner at any given time. (If it has not yet been sold, then, for example, the owner will be considered the car dealer hoping to sell it later.)

To complete the relational model of the vehicle registry, the definition of two additional entity types, the vehicles and the owners, as relations is also needed. This can be achieved most expediently by two independent relations (tables).

The columns of the VEHICLE table (see Fig. 7.2.) are the LICENSE-PLATE, MAKE, MODEL, COLOR, ENGINE-SIZE attributes and the OTHER vehicle attributes, depending only upon the vehicle itself, but the latter is not detailed now.²

¹ In practice, several relational systems allow an increase in the degree of a table to insert new columns (see command ALTER, Section 10.6.), but as was earlier noted concerning the model, it means a new relation.

² Both in the VEHICLE and CAR-OWNER tables—the latter will be detailed—data summarized in the column OTHER must be considered as either a single continuous explanatory field which provides textual information or an abridging mark written out in lieu of several independent columns that, however, have well-defined values (for example, chassis number, engine number, horsepower...).

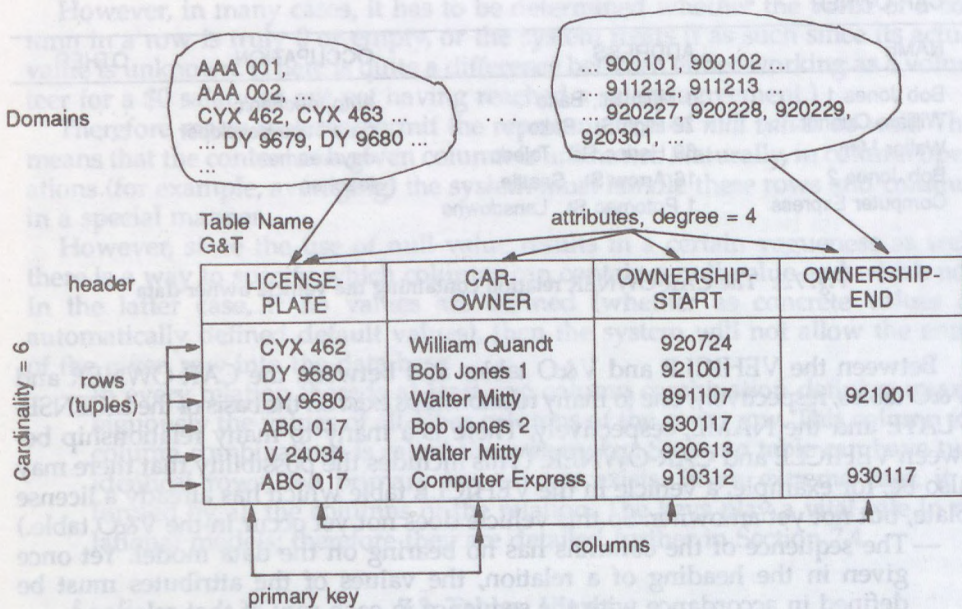


Fig. 7.1. Representation of a relation

The degree of the relation is 4; the number of domains is 3. (The OWNERSHIP-START and OWNERSHIP-END attributes are based upon the same domain, the set of the valid dates.) The cardinality is 6 at the current moment. Since the same person can own simultaneously several vehicles, and a vehicle (although not simultaneously) can have several owners, the rows are identified jointly by these two attributes only. Therefore the primary key is composite; the LICENSE-PLATE and CAR-OWNER columns form jointly the primary key

The second, the CAR-OWNER table (see Fig. 7.3.), contains data about the individual owners. These data are independent of the cars. Its columns are the (unique) owner name, his/her address and occupation, and OTHER necessary data which are not detailed.

VEHICLE					
LICENSE-PLATE	MAKE	MODEL	COLOR	ENGINE-SIZE	OTHER
CYX 462	Mitsubishi	Colt GL	white	1298	...
DY 9680	Ford	Mustang L	metallic silver	1996	...
ABC 017	Daihatsu	Charade	blue	1298	...
V 24034	Ford	Escort D	white	1796	...
CZ 9404	Volkswagen	Fastback TL	green	1594	...
KF 9547	Volkswagen	Fastback TL	green	1594	...

Fig. 7.2. The VEHICLE relation containing the vehicle data

CAR-OWNER			
NAME	ADDRESS	OCCUPATION	OTHER
Bob Jones 1	1 Main St., Balto	auto mechanic	...
William Quandt	72 High St., Balto	data base developer	...
Walter Mitty	69 Herioc Rd., Toledo	daydreamer	...
Bob Jones 2	16 Arrow St., Seattle	teacher	...
Computer Express	1 Potomac St., Lansdowne		...

Fig. 7.3. The CAR-OWNER relation containing the vehicle owner data

Between the VEHICLE and V&O tables and between the CAR-OWNER and V&O tables, respectively, one to many relationships exist on the basis of the LICENSE-PLATE and the NAME, respectively. There is a many to many relationship between VEHICLE and CAR-OWNER. (This includes the possibility that there may also be, for example, a vehicle in the VEHICLE table which has already a license plate, but not yet an owner. So this vehicle does not yet occur in the V&O table.)

- The sequence of the columns has no bearing on the data model. Yet once given in the heading of a relation, the values of the attributes must be defined in accordance with the sequence in each row of that relation.³
- Since the entities of the relation of the table are unique rows identified by their content, the table cannot have two totally identical rows — two entries where each value in each column is the same. (This condition reflects the logical real world restriction that if no distinction can be drawn between two elements, then they should not be considered different. Their characterization stems from the definition of their features, and at most — if necessary — it can be defined how many items exist.)
- The number of rows in a relation used to be called its *cardinality*. This changes dynamically when new data are entered or existing ones deleted. The sequence of the rows has no significance in the relation. The rows have no favored order, at least in theory.⁴
- In principle, one and exactly one value has to be assigned to each column in every row of the relation. However, in practice, it happens very frequently that certain columns' values are not known when entering the data record into the data base; nevertheless, the record is to be stored. One possibility to solve this contradiction is that every column should have a default value. (In general, for numeric columns it is zero, and for character-types spaces.) Thus every column of every record is assured of a value, either supplied directly by the user or given by the default value.

³ Due to reading from left to right, however, it is typical that the 'more important' columns are placed left and ahead, while the 'less important' ones are behind. However, this is only a convenient arrangement of no special importance to the model.

⁴ However, in practice, in order to increase efficiency data are very often stored in a given physical order.

However, in many cases, it has to be determined whether the value of a column in a row is truly 0 or empty, or the system treats it as such since its actual value is unknown. (There is quite a difference between either working as a volunteer for a \$0 salary or not yet having reached a salary agreement.)

Therefore many systems permit the representation of *null values* as well. This means that the content of a given column is unknown. Naturally, in column operations (for example, averaging) the system must handle these rows and columns in a special manner.

However, since the use of null value results in a certain vagueness as well, there is a way to specify which columns can contain a null value and which not. In the latter case, if no values are defined (whether as concrete values or automatically defined default values), then the system will not allow the entry of the given row into the data base.

- In every relation there is at least one column combination defining unambiguously the values of all other columns of the given row. This column (or column combination) is called the *primary key*. Since no table can have two identical rows, the primary key always exists. In an extreme case, it is formed by all the columns of the relation. The keys play a vital role in relational models; therefore they are detailed further in Section 7.4.

7.3. Tables, Views

Up to now it has been implicitly assumed that in the relational model the data of the relations (tables) are actually stored in the data base in that form as well. If this were always true, then the absence of redundancy, one of the most important benefits of database management systems, would be lost since in that case all elementary data (column values) would be stored as many times as there were relations containing them. Precisely to avoid this, the relational model allows the creation of additional virtual tables from the physically-stored ones. In most cases, the former can be used just as the latter.⁵

Note that while in theory the relational model permits the following relation types, not every one of the currently-marketed systems supports at present each of the below-listed features.

Base Relation or Table

It is a physically-existing relation, containing actual data in the defined format. In general, this is called a table in the more narrow or exact sense.

Virtual Relation or View

It is a user-defined relation. It does not contain data itself. When defining the view, it is specified from which tables, columns, and rows, as well as relational operations, are produced the individual rows and columns of the view.

⁵ In general, insertion, modification, and deletion can be limited in virtual tables because in certain cases their results are not defined unambiguously, or a contradiction can arise among the original and defined data. (For example, the virtual $C=A+B$ column is defined from columns A and B . An attempt to increase the C value would fail as the system would not know how to change A and B so as to maintain the equality.) These topics will be detailed in Point 10.3.3. when discussing read-only views.

Snapshot

It contains well-defined data of a given table(s) and view(s) at a definite moment. It differs from the view in that its data are also stored physically. It is usually read-only and serves data security and restoration. A snapshot can be, for example, the routine daily saving of certain tables.

Query Results

As the name says, it supplies useful information for the user. Strictly speaking, it is not a data base relation. For example, it usually has no independent name, may contain identical rows, and exists only until the obtained information is evaluated or forwarded for further processing.

Intermediate Tables

To solve more complicated tasks, the database management system must store temporarily the intermediate results. (For example, selected rows must be sorted by another field, or identical rows must be sifted in the final result.) They are usually real, physically-stored tables (though it is possible that they refer to some portion of the data only by pointers). Yet they exist only temporarily in the data base. Generally they are deleted automatically from the system once the complex operation requested by the user has been completed, or sometimes when the data base session ends.

In the relational model, the user or the data base administrator creates relations to define the data structure only at the conceptual and external levels of the data model. The relationships among the data are established by relations. Relations can be connected with each other by the fulfillment of given relationships between attributes built upon the same domain. Generally, this means the attribute values should be equal. In contrast to the hierarchical and network models, here there are no chains and pointers defined by the user or the data base administrator to produce the relationships among the various entities with whose full knowledge the user can navigate through the data base and check data integrity. As for data model entities of the same and different types, the system produces and ensures data integrity with the help of the primary and foreign keys, to be detailed in the next section, while various operations executable on the relations enable entities to be combined. They will be detailed in Sections 7.5. and 7.6.

7.4. Keys

7.4.1. Data Base Integrity

Data base data integrity and the contradiction-free status of various relations must be ensured by the database management system. Some of these rules are so critical that their fulfillment is checked automatically by the system. If the data fail to satisfy these conditions, then the system does not allow their entry, modification, or deletion. Other integrity conditions are specific and application-dependent. For example, the latter includes the generally valid condition that each attribute can possess only the value of the appropriate domain.

There may be further application-dependent cases when well-defined relationships must exist among the various attributes or attribute combinations. (For example, in any row of the CAR-OWNER table, the values of both the OWNERSHIP-START and OWNERSHIP-END attributes can be selected from the domain of real dates, but the START value cannot be larger than the END.)

Some of these special conditions can be built into a given data base definition (for example, date checking, the value of a column can be only numeric, or can take only some well-defined values, etc.). Whereas for other conditions the user must run a special program each time to check their validity.

For example, the degree to which a particular database management system supports the attribute-domain relationship varies markedly. (In most cases, only to a limited degree. In prior versions of IBM SQL/DS only the data type was checked; with AS/400 value limits and a limited number of actual values can also be defined for the domain, while in DB2 and SQL/DS release 3 the user can prepare his own checking program which is activated automatically during every data entry and modification.) Database management systems also differ as far as which connections and terms can be examined automatically.

That is precisely why only the general integrity rules valid for every relational data base are discussed in this point. Among them the first is the *primary key*, often called simply — and incorrectly — only the key as well. The primary key serves to unambiguously identify the entities of the individual relations.⁶

The other group of integrity-ensuring keys is the *foreign key*. With its help the system can check automatically that no contradictions have arisen among the values of those attributes of various relations based upon the same domain. (For example, it is not allowed to fill out a consignment note for a non-existing customer or good.)⁷

The above two basic integrity rules come from the modelling of real life. That is why the database management system uses them only to check real objects, basic relations, and the integrity of tables.

The integrity of views and other secondary relations is already application-specific. It depends upon the definition of the formed relation. The user or data base administrator must therefore arrange it, preferably by building it into the definition of the formed relation (so checking is automatic) or with a special application program.

⁶ This carelessness could be allowed only if the rows were identified and searched for always by the same field. Despite this, it can be found often in every day language, mainly in sales brochures, and occasionally even in technical literature. In this book to avoid ambiguity, if a primary key is involved, then the adjective is always written before it, except for that situation where it was already repeatedly put out and it is obvious from the context that key means primary key in its subsequent occurrences.

⁷ Unfortunately, there are a lot of systems on the market which do not support foreign keys. Thus, for example, even the SQL/400, developed for IBM's quite widespread AS/400 computer family, does not permit — or rather at the time these remarks were written — the definition of foreign keys, and thus their automatic checking.

7.4.2. Primary Key

As listed in Section 7.2. where the features of relations were defined, any row of a relation is identified based on its content, and, at a minimum, in every relation exists one column combination which is always unique in the given relation; it defines therefore always unambiguously the values of the other columns in the row.⁸

In practice, it can happen that not only one column combination identifies unambiguously the rows of a relation. Thus, for example, in a personnel register, in addition to his/her social security number, a person can be identified by these column combinations: his/her name, mother's maiden name, and year and place of birth, or if the register is such — as assumed in Section 7.2. — that *always* unique names are used, then by the name alone as well.⁹ In this case, these columns (or column combinations) are called *candidate keys*, and any of them can be selected as the primary key. After assigning the primary key of the relation all other candidate keys are called *alternative keys*.

In principle, it is insignificant which of the candidate keys is finally selected to be the primary key; however, it matters from a practical standpoint.

A great number of systems store physically the individual relation rows according to the primary key. Thus records cluster on that. So it is expedient to select the most often used candidate key as the primary one. Thus, for example, if a relation contained the Periodic Table of Elements, then from among the candidate keys, the chemical symbol, element name, and atomic number (for example: H, hydrogen, 1), the atomic number would likely be scrapped at once, and if mostly physicists or chemists worked with the relation, then the chemical symbol, otherwise the element name, would be defined as the primary key.

The unique primary key must satisfy within the relation the following conditions as well:

- *neither the keys nor any of their components — if composite keys — can contain the null value.* This rephrases into relational model terminology the plausible logical condition that only those things whose values are known can identify anything else.
- *in case of composite keys, the number of components is minimal.* If any column were left out from the key, then uniqueness would cease. In the example mentioned at the beginning of this point, if there are names that always identify unambiguously, then the combination of name, mother's maiden name, year, and place of birth is not a primary key, since it is not minimal. Naturally, here these conditions should apply also to future data. Even if the name is not definitely unambiguous, it may still be very unlikely that

⁸ In the relational model, identification is based upon content in an associative way. However, this is true only at the logical level of the model. To implement the model and for physical addressing, no associated memory hardware is needed. In contrast, most of the successful systems up to now have used traditional, location-based addressing and are based upon hardware and operating systems which apply traditional I/O techniques.

⁹ It is essential here that this uniqueness must be fulfilled not only for the current data, but also for the entire period of the data base relation's existence.

there are two persons in the relation with the same name. However, for this reason, the name still cannot be considered as the primary key since a person bearing the same name as someone already in the register could conceivably appear for entry into it at anytime.

The selection of primary keys is one of the most important tasks of data base development. However, the actual data model usually makes obvious what should be the relation's primary key. If additional alternative keys also exist, then their unambiguousness must be identified separately when defining the data base so that the system can ensure it automatically. This can be realized by a unique index defined on the alternative key (see Section 4.7., and CREATE UNIQUE INDEX, Section 10.4.).

The primary key of the V&O table shown in the example is a composite key. It is formed on the LICENSE-PLATE and CAR-OWNER columns.¹⁰

7.4.3. Foreign Keys

To ensure that the data base is and remains contradiction-free, well-defined relationships must always be valid among certain relations. The most important is when in one relation, in the *referring* relation, a column or column combination can have only values equal to a primary key value of another relation, the *referenced* relation, or at most it can contain the null value. This column (combination) is called a *foreign key* in the referring relation. Additional common names for relations permanently connected to each other are *dependent* and *parent* relations or tables.

Such foreign keys in the V&O table are LICENSE-PLATE and CAR-OWNER which refer to the primary keys of the VEHICLE and CAR-OWNER tables. It is evident that the names of the individual columns can differ in the dependent and parent tables, but naturally they apply to data whose content are of the same type. (Generally, they are based upon the same domain, possibly upon a part of the appropriate domain of the parent table.) It does not cause any trouble either if the column name in a table matches another table's name. In the above example, the foreign keys are concurrently part of the dependent table's primary key. However, this is not mandatory, either. The foreign key relationship would also be valid even if the foreign key was not identical to the primary key, any alternate key, or any of its components in the dependent table. An example of the latter is a personnel register where the EMPLOYEES table contains personal data about the employees, and among them also the code of the section employing them (see Fig. 7.21. in Section 7.9.). The primary key of the relation is the social security number of the employee, and the section code is a foreign key. (The first name – last name combination is an alternate key, in case of unique name formation.)

The parent and dependent relation can be the same. An example is an extension of the EMPLOYEE table seen in Fig. 7.21. containing the social security number of the employee's direct supervisor. (It is assumed that there is only one.)

¹⁰ Thus in this model the same person cannot buy the same vehicle twice. For simplicity, this restriction is accepted now.

Naturally, due to data base consistency, this could only be a value already existing among the employee numbers. At the same time, this is a good example of why it is necessary to allow the null value for foreign keys. The most senior person of the company (president or chief executive officer) does not have a boss; so the value of this attribute will be null in this row.

A typical representation of the relationships defined on the basis of foreign keys between relations is shown in Fig. 7.4. The arrow points from the dependent table to the parent one and generally contains the methods of key modification and deletion as well (see later in this point).¹¹ The name of the foreign key is indicated either in the relation or next to the relationship arrow.

In principle, the dependent table can also be referred by another table. It can also happen that a reference loop ends by referring to several tables (Fig. 7.4.c). In this case, only those employees can be in the EMPLOYEE-1 table whose DIRECT-SUPERVISOR number occurs in another row of the same table as EMP-NUMBER, or it is the null value (self-reference), and their S-CODE is the null value or already exists in the SECTIONS table (simple reference), but there can be only such rows in the SECTIONS table whose S-LEADER value is the null value or a valid EMP-NUMBER in the EMPLOYEE-1 table (simple reference and simultaneous creation of a reference loop).

The existence of foreign keys defines various conditions for various relations, namely *reference conditions* or *dependency relationships*. Their essence can be briefly summarized in this way: *there can be no foreign key in the data base whose value does not equal the value of a primary key connected to it, or which does not contain the null value.* (The contrary is not true; in the parent relations there can be primary key values for which there are no matching foreign keys. This is understandable since, for example, it would cause no problem if there were such goods for which no order has yet been placed. The good code is the referred primary key in the relation containing goods and the foreign key in the relation containing the orders.)

So the database management system must ensure automatically that the reference conditions remain valid even after any changes in the data.

The change in data can be:

- insertion of new data,
- modification of existing data, or
- deletion of existing data.

(The restriction is examined now only regarding the foreign key value.)

Insertion

Any new row can be inserted into the parent table if its primary key is a permitted value (unique and not null).

A row can be entered into the dependent table only if the foreign key value is null value (if this value is allowed at all for this column (combination) by the data model), or it is equal to an existing primary key value in the parent table. This condition is completely rigid. It does not allow any deviation.

¹¹ It is usual also to direct the arrow from the parent toward the dependent table. This contradicts the logic of the reference, but is consistent with the representation of the 1:N relationships, realized by the foreign keys.

Deletion

Deletions are not restricted in any way in the dependent table. Unrestricted deletion in the parent table might inadvertently destroy dependency relationships. To avoid this, the data model permits only the following deletion methods:

- restricted,
- setting to null value, or
- cascade.

When deletion is *restricted*, the row of the parent table can be deleted only if there is no foreign key in the dependent table equal to its primary key value. If it was still necessary to delete such a record, then this could be executed only whereby first all the rows referring to it were deleted or their foreign key values were changed to other allowed ones.

From the standpoint of data consistency and accuracy, this is the safest method, but sometimes it poses difficulties since all of the dependent tables must be checked. Therefore *setting to null value* is also a common method whereby the foreign keys whose values are equal to the deleted value will be changed to the null value automatically in the dependent table. Naturally, this is possible only if the foreign key column(s) allows the null value.

Its execution is simpler. Deletions can be performed without further operations. Yet because of the automatic modification potentially useful information may also be lost. (For example, the code of a good which is no longer delivered is deleted, but as a result the good becomes unidentifiable in previous orders as well. Naturally, such a model can be correct formally, but is logically false in any case!)

Cascade change is even more convenient, but poses even bigger risks. When executing it, together with the row containing the primary key, all rows in which the foreign key equalled the deleted value will also be deleted automatically from the dependent table. This is particularly dangerous if other tables refer to the dependent table because further deletions ensue in these dependent tables as well. To avoid this, in practice, a lot of systems permit only one-degree references for cascade deletions. Other tables cannot refer to the so-referring table at all or only with restricted deletion.

The following example (see Fig. 7.5.) illustrates what problems can arise from unrestricted multi-degree permission to set a value to null or cascade.

In table *A* *PK* is a primary key (its values are *a1* and *a2*) to which the *FK* columns of tables *B* and *C* refer as foreign keys with cascade deletion. (At this point in the example the other columns of *A* are irrelevant.)

The composite primary keys of *B* and *C* are *PK1* and *PK2* to which refer *FK1* and *FK2* foreign key values, as well as *FK2* and *FK3* ones, respectively, from table *D* with setting to null value at deletion.

From *A a1* is deleted. The cascade deletes *x11* and *x12* from *B* first, and as a consequence *FK1* and *FK2* will be set to the null value in the *d1* row of *D*, then because of *FK2* being null, the foreign key of *d1* has the null value for *C*. Thus in the second cascade the deletions of *x11* and *x13* from *C* do not affect *D*, so the value of *x13* will remain.

Similarly, it is evident that if the cascade begins through *C*, then in the *d1* row of *D* the value of *x11* remains unchanged, and *x12* and *x13* are set to null value.

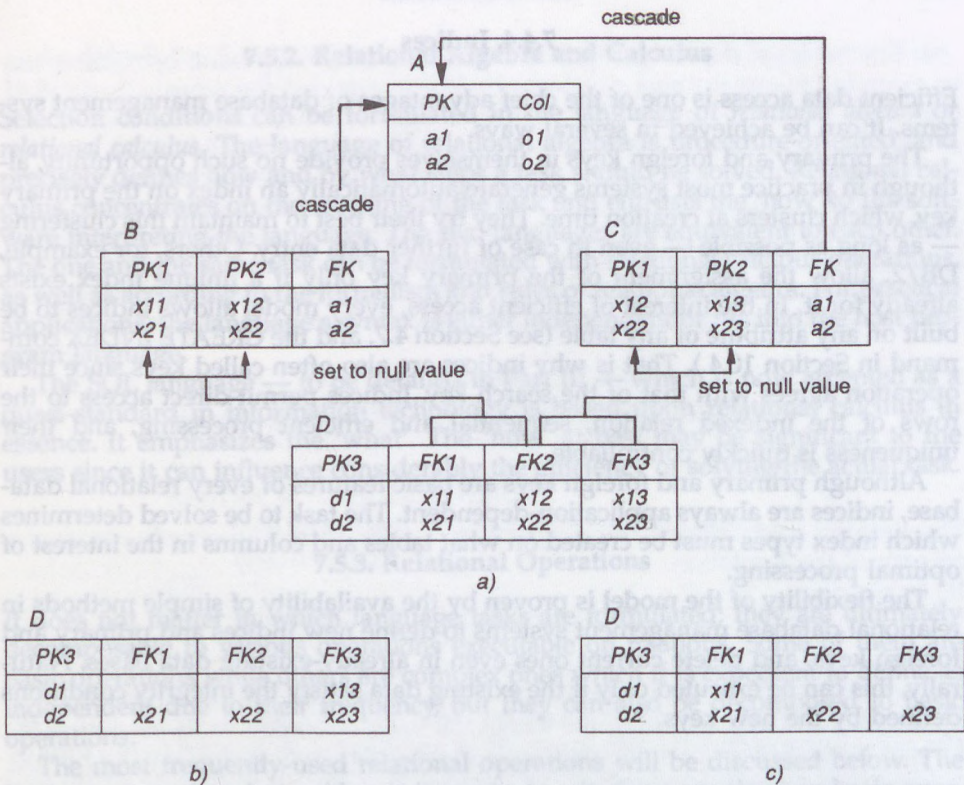


Fig. 7.5. An example of problems with foreign keys, multi-stage references, and not restricted deletion: a) the data definition and dependency relationships, b) the effect of the deletion of the *a1* record if cascade deletion occurs first through Table B and then through C, c) the same as b, but the cascade sequence is C, then B (The empty column entry means the null value.)

Thus the result of the operation would depend upon its execution sequence at a lower level; this is inadmissible since it cannot be controlled by the user. That is precisely why database management systems do not allow the creation of such dependency relationships.

Modification

The options are the same as with deletion, except that here cascade results in the automatic change of the foreign key value to the key's new value in all relevant rows of the dependent table.¹²

Reference conditions must be defined along with the primary and foreign keys when defining the data model, creating the data base, or modifying its structure.

¹² The actual options for deletion and modification may differ even in the same system. Thus, for example, DB/2 and SQL/DS allow all three options for deletion, but only restricted modification is allowed.

7.4.4. Indices

Efficient data access is one of the chief advantages of database management systems. It can be achieved in several ways.

The primary and foreign keys in themselves provide no such opportunity, although in practice most systems generate automatically an index on the primary key, which clusters at creation time. They try their best to maintain this clustering — as long as possible — even in case of further data entry. Others, for example, DB/2, allow the assignment of the primary key only if a unique index exists already for it. In the interest of efficient access, every model allows indices to be built on any attribute of any table (see Section 4.7. and the CREATE INDEX command in Section 10.4.). That is why indices are also often called keys since their operation agrees with that of the search key. Indices permit direct access to the rows of the indexed relation, sequential and efficient processing, and their uniqueness is quickly controllable.

Although primary and foreign keys are basic features of every relational database, indices are always application-dependent. The task to be solved determines which index types must be created on what tables and columns in the interest of optimal processing.

The flexibility of the model is proven by the availability of simple methods in relational database management systems to define new indices and primary and foreign keys, and delete current ones even in already-existing data bases. Naturally, this can be executed only if the existing data satisfy the integrity conditions defined by the new keys.

7.5. Relational Operations

7.5.1. The Purpose of Operations

The most important benefit of the relational model is that the entities — existing also physically and satisfying the varying conditions, or producible from them in a logical way, or temporary — can easily be determined, identified, selected, and defined. Relational operations are used to perform the following tasks:

- determining the data to be selected when searching,
- selecting data to be modified or deleted,
- determining the data to be entered, and
- defining virtual data. They can be constant relations of the data model identified by name (views, snapshots) or just temporary ones needed to solve the given task, and dropped automatically afterwards.

Besides the above, relational operations are also used to define the data base access authorities and integrity conditions, namely selecting which group of objects to interconnect or which relations needing protection must be locked.

As the most frequent operations are data selection, generally, the phrase 'selection' or 'select operation' will be used in the following, but it means not only selection in the narrow sense, but any of the above-listed operations.

7.5.2. Relational Algebra and Calculus

Selection conditions can be formulated in the language of *relational algebra* or *relational calculus*. The language of relational algebra is procedure-oriented, and precisely defines how and by what steps a task should be solved. Relational calculus concentrates on the wording of the task and entrusts the 'how' to the software interpreting the language. The two languages are equivalent to each other. The one and the same task can be formulated with relational calculus operations, as well as according to the formalism and rules of relational algebra. For practical applications, the language's syntax is either independent or fits into the host program language.

The SQL language — to be detailed in Part III — which is now accepted as a quasi-standard in information technology is based upon relational calculus in essence. It emphasizes the 'what'. The 'how' at best may be significant to the users since it can influence considerably the efficiency of solving the actual task.

7.5.3. Relational Operations

It does not matter in which language tasks are formulated; they are ultimately decomposable to various operations executable on relations. Some of them are basic operations while others are complex ones which it is expedient to define as independent due to their frequency, but they can also be decomposed to basic operations.

The most frequently-used relational operations will be discussed below. The list is not complete, but neither is it scanty, to wit, there are also non-basic operations among the listed ones, as earlier mentioned. However, it should be stressed that the system is *closed* for each of these operations. This means that if any of these operations is applied to any relation or relations, then the result will be another relation to which these operations are applicable again.

The relational operations considered by the author fundamental and therefore described separately, are the following:

- renaming,
- restriction,
- projection,
- cross product,
- union,
- intersection,
- difference,
- join,
- join with default value (external or outer join),
- extension,
- group forming operation, and
- assignment.

Relational operations can be classified also as to whether they effect one or two operands.

It follows from the foregoing that the validity of two-operand operations can be extended to any number of operands. One operand of the repeatedly-applied operation will be the result of the previous operation and the other will be one of the additional operands which can be considered the third, ... etc.

Operands can be transposed in most multi-operand operations. Generally, the sequence of the various operations is also discretionary. These features are very significant when translating relational operations into real physical I/O commands. If they are executed in their optimal as opposed to an erroneous order, then the resultant response time will differ markedly, even by orders of magnitude. Operations for which these conditions are not valid will be separately discussed.

7.6. Definitions of the Most Important Operations

7.6.1. Renaming

The simplest basic single relation operation is RENAME. This simply renames a column of the R relation.

Its form is:

$R \text{ RENAME } (r_1, r_2)$

where r_1 is the original and r_2 is the new column name.

The renaming of several columns of the same relation can be executed either by the repeated application of the RENAME basic operation or its extended definition to $R \text{ RENAME } (r_{11}, r_{12}; r_{21}, r_{22}; \dots, r_{m1}, r_{m2})$ where r_{11} and r_{12} are the related old and new column name pairs.

This operation is needed because of closure. Without it the result of two-relation operations could be a 'relation', for example, containing columns with identical names.

7.6.2. Restriction

Another single-relation operation is RESTRICT.¹³ The result contains only those rows of the R relation where the F_1 restrictive condition is fulfilled for the value of the r_1 column. This means generally the $<$, $>$, $=$, ... etc. comparisons, where the item to be compared is a constant, the value of another column of the row, or a scalar value obtained by a relational operation.

Its general form is:

$\text{RESTRICT } R \text{ WHERE } r_1:F_1$

The names of the columns of the result relation and its primary key are the same as in the original relation. The different comparison options will be detailed in Section 11.2. in the WHERE clause of the SQL SELECT command.

¹³ It is often called SELECT. However, this can cause confusion since the selection command of the SQL language is called SELECT as well. This is, however, far more complex than the former and by its application all relational operations can be executed.

As default the F_1 restrictive condition is generally a simple logical condition.

E.g., RESTRICT VEHICLE WHERE MAKE = 'Ford'

gives as a result all cars in the VEHICLE table manufactured by Ford.

However, in practice, complex restrictive conditions are very often formed to select rows. These conditions may apply to the same column or different columns, they can be connected by a logical OR or AND, or they can contain negations (NOT) and parentheses as well.

For example, Ford Mustang and Ford Escort cars with an engine size larger than 100 cubic inches can be selected by any of the following restrictions (or other similar ones):

```
RESTRICT VEHICLE WHERE MAKE='Ford'
AND ENGINE-SIZE > 100
AND (MODEL='Mustang' OR MODEL='Escort')
```

or

```
RESTRICT VEHICLE WHERE
(MAKE='Ford' AND MODEL='Escort' AND
ENGINE-SIZE > 100)
```

OR

```
(MAKE='Ford' AND MODEL='Mustang' AND
ENGINE-SIZE > 100)
```

(If it were known when formulating the query that only Ford produces 'Mustang' and 'Escort' model cars, then the condition concerning the MAKE could be left out of the restriction, simplifying it. However, this kind of simplification option arises only from the peculiar characteristics of the actual data model. So it is neither discussed here nor when examining additional relational operations.)

Most database management systems handle a restriction containing a logical expression as a basic relational operation as well. However, the relational realization of a complex condition can also be defined so that the restriction operation includes only one elementary condition. The result of complex conditions can be produced by additional two-operand relational operations obtained from the relations as a result of the elementary restriction (see, for example, the operations UNION and INTERSECT).

7.6.3. Projection

The third single-operand basic operation is PROJECT. It keeps only those columns from the R relation which are defined in the projection. The columns of the projection are identified by the same name.

R PROJECT [$r_1, r_2, \dots, r_n$]

where $r_1, r_2, \dots, r_n$ are the columns of the original R relation remaining in the result as well. Their order can be different from what it was in the original relation. Since the set of relations is closed for the operations, it is also necessary for the result of the projection to fulfill the conditions given in Section 7.2. concerning the relations. So the above definition must still be supplemented thus: if after deleting the columns, identical rows are produced (for example, as in the case of

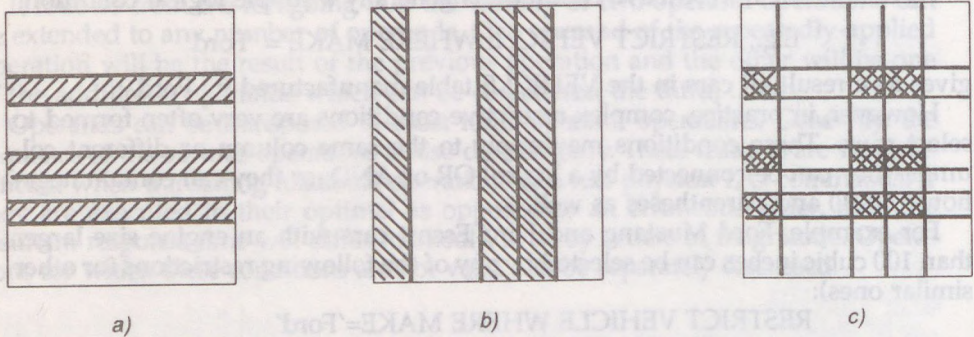


Fig. 7.6. Single relational operations a) restriction, b) projection, c) restriction and projection (The execution sequence of the two operations is discretionary.)

the projection of the MAKE and ENGINE-SIZE columns in the VEHICLE relation's data seen in Fig. 7.2.), then only one of them is retained. If the projection contains the primary key of the original relation, then it will be the primary key of the result as well, otherwise it must be defined separately.

The PROJECT operation can also be used to reorder the columns of a relation. The effect of restriction and projection on a given relation is illustrated in Fig. 7.6.

7.6.4. Cross Product

Among the two-relation operations, the most fundamental is the cross product. The cross product of the $R(r_1, r_2, \dots, r_m)$ and $S(s_1, s_2, \dots, s_n)$ relations with a degree of m and n , respectively, is $R \times S$ or R TIMES S . It is a relation with a degree of $m+n$ whose columns are obtained whereby all the columns of S are written after the columns of the R relation, and its rows are obtained by writing each row of S after each row of R . If the number of rows in the original relations, their cardinality, are i and j , respectively, then the cardinality of the cross product will be $i \times j$ (see Fig. 7.7.).

The primary key of the product will be the concatenation of the primary keys of the two factor relations.

The names of the product columns agree with the column names of the individual factors. It follows from the closed nature that the operation can be executed only if the factors are *product-compatible*, namely there are no columns in them with the same names. If this were not true, then the corresponding columns of the R and/or S relations would have to be renamed first using the renaming relational operation (RENAME).

The usual procedure is (and the SQL language to be discussed in Part III automatically applies this) that every column name is qualified automatically by the name of its relation. So if the column name is not unambiguous, then it can and must be made unambiguous thus:

relation-name.column-name
E.g., VEHICLE.LICENSE-PLATE or V&O.LICENSE-PLATE

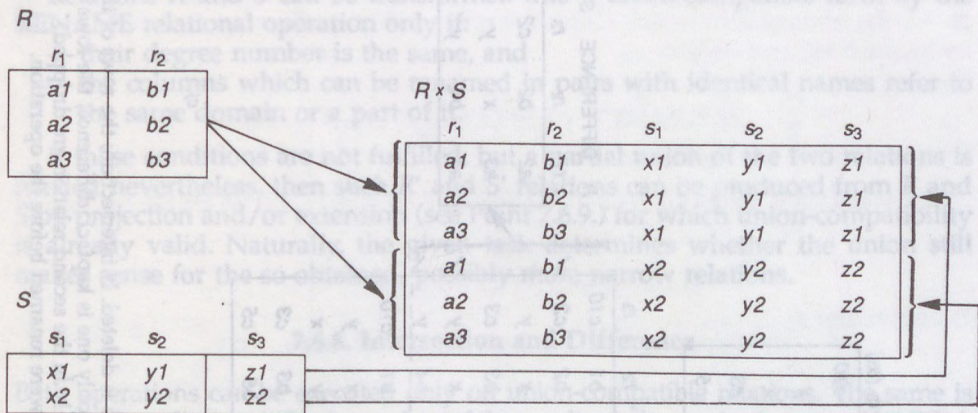


Fig. 7.7. The product of two relations (R and S)

The column names are different in the two relations; therefore there is no need to execute the RENAME operation before the multiplication.

This procedure can be applied automatically except if the components of the cross product are the same. (This operation is allowed without further ado; moreover, it is often needed in practice.) In this case, column names must be made unambiguous by the user himself.

Note that although the cross product is a very important basic operation in theory, in practice, its restriction-free application is avoided as much as possible. A simple reason for this is that while handling relations with, say, 10,000 rows, would not in itself entail efficiency problems, their cross product would contain $10,000 \times 10,000 = 10^8$ rows. Its storage and serial search or the construction of temporary indices for direct access could entail major storage and response time problems.

7.6.5. Union

The union of two relations, $R \text{ UNION } S$, can be formed only if the two relations are union-compatible, namely, both relations contain the same columns. In this case, the union includes all rows existing in either the R or the S relation, but includes only once those columns occurring in both of them (see Fig. 7.8.a).

Both the columns and the primary key of the resultant relation are equivalent to the (common) columns of the operand relations and their key, respectively.

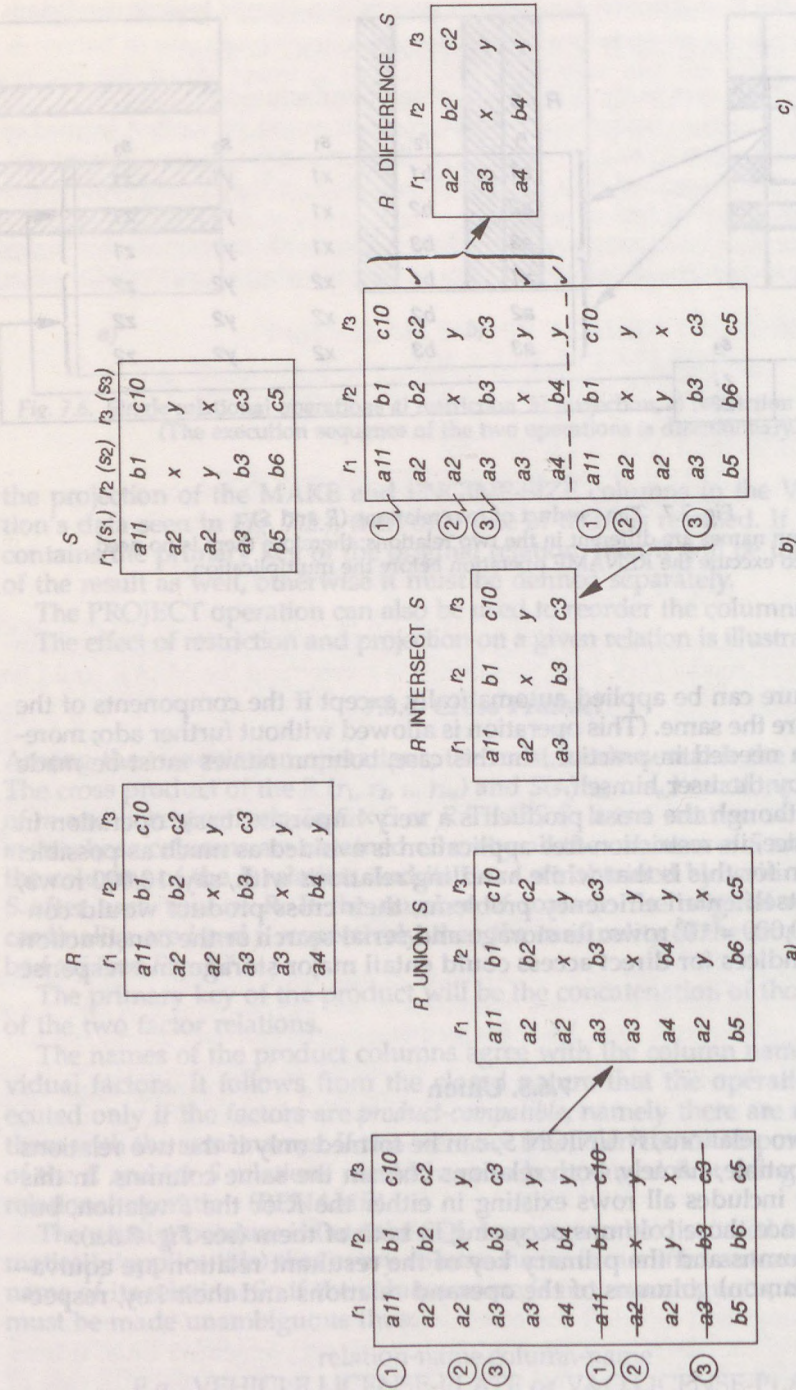


Fig. 7.8. Two-operand relational operations
 a) union; the rows of the two relations are written one after another. One of the identical rows is deleted. b) intersection; the rows of the two relations are written one after another. Only the identical rows are retained, and of them only one is kept. c) difference; the rows of the two relations are written one after another. All duplicate rows and those occurring only in the second relation are dropped.
 The R and S relations were transformed into union-compatible form; the columns of S were renamed before the operation.

Relations R and S can be transformed into a union-compatible form by the RENAME relational operation only if:

- their degree number is the same, and
- the columns which can be renamed in pairs with identical names refer to the same domain or a part of it.

If these conditions are not fulfilled, but a partial union of the two relations is needed nevertheless, then such R' and S' relations can be produced from R and S by projection and/or extension (see Point 7.6.9.) for which union-compatibility is already valid. Naturally, the given task determines whether the union still makes sense for the so-obtained, possibly more narrow relations.

7.6.6. Intersection and Difference

Both operations can be executed only on union-compatible relations. The same is true for the columns and primary key of the result relation, as in the case of UNION.

The $R \text{ INTERSECT } S$ relation contains only those rows of R and S occurring in both relations (Fig. 7.8.b).

The rows of the $R-S$ or $R \text{ DIFFERENCE } S$ relation are composed of only those rows of R which cannot be found in S (Fig. 7.8.c). Unlike the two-operand operations described thus far, the operands of the difference are not transposable, $R-S \neq S-R$.

The union, intersect, and difference operations could be better illustrated by disregarding for a moment the rule that the sequence of rows plays no role in the relation and putting the identical rows of the R and S relations at the end of R and beginning of S , respectively (see Fig. 7.9.).

It is obvious that if only elementary conditions are permitted in the restriction operation (see Point 7.6.2.), then the 'non-existing'

$R \text{ RESTRICT WHERE } (r_1 : F_1 \text{ OR } r_2 : F_2)$

can be produced as the result of the following relational operations

$(R \text{ RESTRICT WHERE } r_1 : F_1) \text{ UNION } (R \text{ RESTRICT WHERE } r_2 : F_2),$

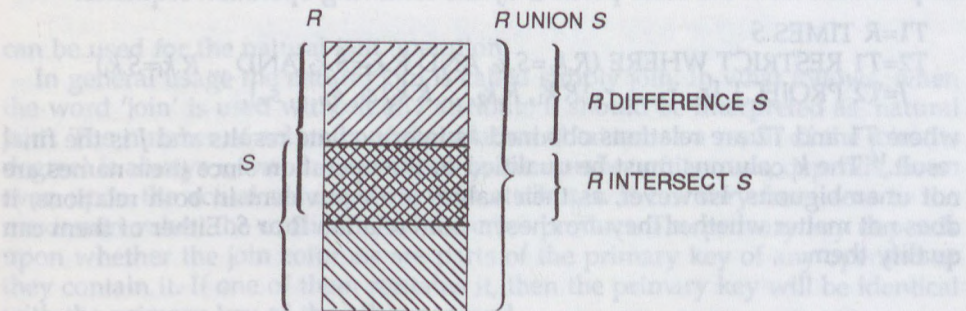


Fig. 7.9. The representation of union, intersection, and difference with 'sorted relations'.

The identical rows of R and S overlap each other due to the sorting

since the columns and primary keys of both 'results' are equivalent to those of R . All the rows of R in which either the relationship $r_1 : F_1$ or $r_2 : F_2$ is fulfilled are included once and only once.

Similarly

$R \text{ RESTRICT WHERE } (r_1 : F_1 \text{ AND } r_2 : F_2)$

is equivalent to the operations

$(R \text{ RESTRICT WHERE } r_1 : F_1) \text{ INTERSECT } (R \text{ RESTRICT WHERE } r_2 : F_2),$

and

$R \text{ RESTRICT WHERE } (\text{NOT } r_1 : F_1)$

is equivalent to

$R \text{ DIFFERENCE } (R \text{ RESTRICT WHERE } r_1 : F_1)$

7.6.7. Join

Working with databases, natural JOIN is the most important and most often used two-operand relational operation. As will be seen, although it is not actually a basic operation, due to its importance it is expedient to define JOIN as an independent operation interpretable at once by the database management system. This command makes it much easier for the user to formulate requests than if the JOIN result had to be assembled from basic operands in stages.

Let the columns of the R relation be $r_1, r_2, \dots, r_n, k_1, k_2, \dots, k_i$, and those of the S relation be $k_1, k_2, \dots, k_i, s_1, s_2, \dots, s_m$, where the k_j columns apply to the same attribute in both relations and their names are identical by pairs while there are none with identical names among the r_j and s_j columns. (If this was not valid for the column names, then before the join the RENAME and PROJECT operation would have to be applied on the operand(s).) The natural join of R and S is the relation whose columns are $r_1, r_2, \dots, r_n, k_1, k_2, \dots, k_i$, and $s_1, s_2, \dots, s_m$, and its rows are obtained by writing every row of S to each row of R where the column values of $k_1, k_2, \dots, k_i$ are equivalent in both relations (see Fig. 7.10.a). If the two relations lack common columns, then the natural join is equivalent to the cross product, otherwise it can be produced from the cross product by the following operation sequence:

$T1 = R \text{ TIMES } S$

$T2 = T1 \text{ RESTRICT WHERE } (R.k_1 = S.k_1 \text{ AND } R.k_2 = S.k_2 \text{ AND } \dots R.k_i = S.k_i)$

$J = T2 \text{ PROJECT } [r_1, r_2, \dots, r_n, R.k_1, R.k_2, \dots, R.k_i, s_1, s_2, \dots, s_m],$

where $T1$ and $T2$ are relations obtained as intermediate results and J is the final result.¹⁴ The k_j columns must be qualified in the operation since their names are not unambiguous. However, as their values are equivalent in both relations, it does not matter whether they are chosen from relation R or S . Either of them can qualify them.

¹⁴ The join could also have been written without intermediate relations as one composite operation. However, with this decomposition the individual steps are more understandable.

R			S		
r_1	r_2	k_1	k_1	s_1	s_2
a1	a2	x	x	f1	f2
a1	b1	x	z	f1	f2
c1	c2	y	a	g1	g2
d1	d2	z	x	g1	g2
x1	x2	z			
x1	x2	y			

R JOIN S					
r_1	r_2	k_1	s_1	s_2	
a1	a2	x	f1	f2	
a1	a2	x	g1	g2	
a1	b1	x	f1	f2	
a1	b1	x	g1	g2	
d1	d2	z	f1	f2	
x1	x2	z	f1	f2	

R OUTER JOIN S					
r_1	r_2	k_1	s_1	s_2	
a1	a2	x	f1	f2	
a1	a2	x	g1	g2	
a1	b1	x	f1	f2	
a1	b1	x	g1	g2	
c1	c2	y	0	0	
d1	d2	z	f1	f2	
x1	x2	z	f1	f2	
x1	x2	y	0	0	

Fig. 7.10. Join operation. Join is based upon the common k_1 column in both relations.
 a) natural join, b) default (outer join). The default values of columns s_1 and s_2 are 0

Instead of this time-consuming description, the notation

$$R \text{ JOIN } S (k_1, k_2, \dots, k_i)$$

can be used for the natural join operation.

In general usage the natural join is called simply join. In what follows, when the word 'join' is used without any attribute it should be interpreted as 'natural join'. The number of columns of the relation obtained as a result of the join (its degree) is always $n + m + i$. Its cardinality, the number of its rows, depends, however, upon the actual values of the join attributes. It can vary from zero to the maximum value, the cardinality of the cross product. The primary key depends upon whether the join columns are parts of the primary key of any operand or they contain it. If one of them contains it, then the primary key will be identical with the primary key of the other operand.

In practice, the join is formed on the basis of one common column ($R.k_1 = S.k_1$) ($i=1$) in most cases. This kind of join is shown in Fig. 7.9. as well.

Note that if the values of the columns in both relations are null values, then they are not identical. Rows containing null value in the join columns will not appear in the result.

7.6.8. Join with Default Value (outer join)

With the natural join, if the join column(s) is a foreign key(s) in the first operand referring to the join column(s) in the second operand, then there is a match to each row of the primary relation; therefore each of them appears in the result. However, if this is not valid, then there can be rows in the first operand to which no row matches in the second one; therefore their values will be absent from the rows of the result.

However, in practice, it is often necessary for each row of the first operand to occur at least once in the result relation as well. Yet if there is no match in the join columns, then it is not known which row of the second operand to select in order to complete these rows of the first operand. The join with default value (outer join) solves this problem. In these cases — for lack of something better — the default value of each column of the second operand is written into the result relation (see Fig. 7.10.b).

Thus the cardinality c_{oj} of the result of the outer join is

$$c_r \leq c_{oj} \leq c_r \cdot c_s,$$

where c_r is the cardinality of the first operand, and c_s is the cardinality of the second one.

The outer join is not a basic operation; it can be produced as the union of natural joins. The operands are not transposable; the first operand plays a favored role.

7.6.9. Extend

A new relation can be formed from an existing one by adding a new column. Its general form is

EXTEND R TO R' ADD *method of column forming* AS r_{new}

where r_{new} is the new column of the new R' relation. (The operation could also be defined so that the name of the extended relation remains the same and the original relation ceases simultaneously. For physical tables, the ALTER TABLE command (see Section 10.6.) is interpreted also in this way.

The new columns are often formed from the available columns of the relation by arithmetic operations, for example, PERCENT = VALUE / BASIS * 100. In this form, R' can only be a virtual relation (a view or query result(s)) because most database management systems have no automatic procedure to ensure the maintenance of such relationships among the columns of the same row for actually-stored data.

Practical tasks would be more convenient if it were possible to extend a physical table with a totally new column independent of the previous data, without

having to transfer the existing data into a new extended table, delete the old one, and then rename the new table with the old one's name.

This can be handled by a special form of the previous extension where $R = R'$ and the method of column forming gives only the definition of the column domain.

Naturally, a default value must also be given to the new column (or it must be allowed to contain the null value). This will be the value of the new column in the already-existing rows of the relation as well.

7.6.10. Group Operation Forming (Summarize)

It very often happens in practice that it is necessary to assign a value typical of the entire group to a group of selected data.

For example, this can be the total, average, minimum, or maximum value of employee salaries in a personnel register, or the same values applying to individual groups of employees, for example, those who work in the same section or have a common task.

This complex operation is a generalization of the extension where the rows of the new table contain only the values characterizing the group or the formed values (total, minimum, etc.), respectively, from among the rows of the original table. Its form is:

SUMMARIZE R TO R' GROUP BY *method of grouping* ADD $f_1(r_1)$ AS r'_1

Here $f_1(r_1)$ is a value characterizing the group; for example, the sum of the r_1 column values $SUM(r_1)$ in the rows belonging to the same group, or their minimum $MIN(r_1)$.

If the grouping method is omitted, then the aggregates apply to the entire relation, and R' will have only one row. Otherwise, it has as many rows as different groups can be formed from them.

Just as with the other relational operations, several groupings can also be defined in one SUMMARIZE operation. For example, a new relation contains by sections and within them by tasks (grouping aspects) the average, maximum, and minimum salary of the individual groups, as well as the number of persons in a given group (COUNT).

Note that the grouping operation can be considered a true relational operation only when certain restrictions apply. In most implementations, if it is applied to a relation, then it is not sure that its result will be a relation. (Namely, it is not sure that it will fulfill the conditions defined in Section 7.2.) Thus its use in practice is generally restricted; it cannot be used, for example, in some operations defining views (see CREATE VIEW in Section 10.3.). Further grouping options will be detailed in Part III when discussing column functions (Point 11.2.5.) and the GROUP BY clause of the SELECT command (Point 11.2.2.).

7.6.11. Assignment

It is expedient to supplement the operations thus far defined with an additional one which names the relation produced by the above-discussed operations. In this way, a new, generally virtual object is produced in the data base. It can be used easily in the future, independent of in how many steps and by how complicated a method it was first produced from the real data.

The usefulness of this command will be truly appreciated when working with views (see Section 10.3.).

7.7. Examples of Relational Operations

To aid understanding, some practical examples will be given for the above operations. Their data are taken from the VEHICLE, CAR-OWNER, and V&O tables of the VEHICLE-REGISTRY data base shown in Figs 7.1.–7.3.

The formulation is not connected to any specific data manipulation language. Moreover, in the nowadays, already quasi-standard SQL language — to be detailed in Part III — it is not even necessary, in general, to describe in as much detail as in the examples how to obtain the data from the data base. It is sufficient to define what is required. The language available to the user corresponds to the formalism of relational calculus far more than to that of relational algebra, but, naturally, that is only the most external layer.

7.7.1. Example of Restriction and Projection

The task is to supply the names of those people who sold or bought cars after December 31, 1992. All data are available in the V&O relation. Project and restrict with proper conditions must be executed on them. The sequence of the two operations is irrelevant.¹⁵

```
[RESTRICT V&O  
WHERE (OWNERSHIP-START > 921231 OR OWNERSHIP-END > 921231)]  
PROJECT [CAR-OWNER]
```

The result can be seen in Fig. 7.11.

7.7.2. Example of Join, Restriction, and Projection

Here the make and model of cars owned by individuals is of interest. Data must be selected from the V&O and VEHICLE tables since the former contains only the name of the owner from among the required data while the latter only the make and the model. The connection between the two different tables is established by the common license plate (join columns).

¹⁵ To compare dates as numbers, they must be represented in the standard international (ISO) format: yymmdd.

(RESTRICT V&O WHERE (OWNERSHIP-START > 921231 OR
OWNERSHIP-END > 921231)) PROJECT [CAR-OWNER]

V&O

LICENSE-PLATE	CAR-OWNER	OWNERSHIP-START	OWNERSHIP-END
CYX 462	William Quandt	92 07 24	
DY 9680	Bob Jones 1	92 10 01	
DY 9680	Walter Mitty	89 11 07	92 10 01
ABC 017	Bob Jones 2	93 01 17	
V 24034	Walter Mitty	92 06 16	
ABC 017	Computer Express	91 03 12	93 01 17

Fig. 7.11. The names of the owners of cars sold or bought after 12/31/1992 (from the V&O table, see Fig. 7.1.). 921231 must be entered in the comparison since the date is stored as a number in the international (yymmdd) form

In the V&O table the OWNERSHIP-END value is unknown for the current owner; so it can be obtained with the null value restrictive condition.

(RESTRICT V&O WHERE OWNERSHIP-END IS NULL)
JOIN VEHICLE (LICENSE-PLATE)
PROJECT [CAR-OWNER, MAKE, MODEL]

The result of the operation is shown in Fig. 7.12.

(V&O JOIN (RESTRICT VEHICLE WHERE OWNERSHIP-END IS NULL)
(LICENSE-PLATE)) PROJECT [CAR-OWNER, MAKE, MODEL]

V&O		VEHICLE			
CAR-OWNER	LICENSE-PLATE		LICENSE-PLATE	MAKE	MODEL
William Quandt	CYX 462	←→	CYX 462	Mitsubishi	Colt
Bob Jones 1	DY 9680	←→	DY 9680	Ford	Mustang
Bob Jones 2	ABC 017	←→	ABC 017	Daihatsu	Charade
Walter Mitty	V 24034	←→	V 24034	Ford	Escort

Fig. 7.12. The make and model of cars owned by individuals (from the VEHICLE and V&O tables, see Figs 7.1. and 7.2.)

As in the previous example the sequence of the individual operations is irrelevant. The same result could be obtained, for example, by the following operation sequences:

```
RESTRICT (V&O JOIN VEHICLE (LICENSE-PLATE))
WHERE OWNERSHIP-END IS NULL
PROJECT (CAR-OWNER, MAKE, MODEL)
```

although its execution would likely be more time-consuming since all the rows of the V&O table would have to be taken into account when joining.

That is precisely why the languages of relational data base management systems usually do not require step-by-step definition of the access procedure. The sequence of the operations (if several) is optimized by the system on the basis of available information about the actual data.

This information can be found in the system catalogue (see Section 7.10.). Access optimization will be detailed in Section 7.8. and when discussing the EXPLAIN command (see Point 13.6.1.).

7.7.3. Grouping and Group Function Example

The task is to learn the maximum and minimum engine sizes of each make. To do this, the rows of the VEHICLE table have to be grouped according to MAKE. Next the largest and smallest values of the ENGINE-SIZE column must be selected from among the individual groups with the MAXIMUM and MINIMUM group functions.

```
SUMMARIZE VEHICLE TO MAX-MIN-ENGINE-SIZE
GROUP BY MAKE
ADD MAKE (MAKE) AS MAKE,
MAXIMUM (ENGINE-SIZE) AS MAX-ENGINE-SIZE,
MINIMUM (ENGINE-SIZE) AS MIN-ENGINE-SIZE
```

Here the symbol MAKE (MAKE) indicates only that the new table contains under the MAKE column name the value of MAKE for each group for which the maximum and minimum values were obtained.

The result of the above operation is shown in Fig. 7.13.

If these aggregates are routinely needed, then it is expedient to define the MAX-MIN-ENGINE-SIZE relation as a permanent virtual relation (view) in the data base, so it can later be used directly.

For example, then it is not necessary to apply a grouping function to the VEHICLE table in order to learn within what engine size bounds move the Ford-manufactured cars; rather, the operation:

```
RESTRICT MAX-MIN-ENGINE-SIZE
WHERE MAKE = 'Ford'
```

is sufficient. (This operation applied to a view is naturally decomposed further by the system into group forming operations affecting the physical table.)

SUMMARIZE VEHICLE ADD
 MAKE (MAKE) AS MAKE.
 MAX (ENGINE-SIZE) AS MAX-ENGINE-SIZE
 MIN (ENGINE-SIZE) AS MIN-ENGINE-SIZE

GROUP BY MAKE

LICENSE-PLATE	MAKE	MODEL	COLOR	ENGINE-SIZE	OTHER	MAX-ENG-SIZE	MIN-ENG-SIZE
CYX 462	Mitsubishi	Colt	white	78	...	78	78
DY 9680	Ford	Mustang	metallic silver	120	...	120	96
V 24034	Ford	Escort	white	96	...		
ABC 017	Daihatsu	Charade	blue	60	...	60	60
CZ 9404	Volkswagen	Fastback	green	96	...	96	96
KF 9547	Volkswagen	Fastback	green	96	...		

Fig. 7.13. Maximum and minimum engine sizes of various make cars (from the VEHICLE table, see Fig. 7.2.)

Similarly, as in the examples of Points 7.7.1. and 7.7.2., if the so-obtained relation is not only needed ad hoc, but regularly required, then it is expedient to define the result as a constant relation or view.

7.8. Optimizing the Access Path

The efficiency of database management systems based on the relational model has already been examined by many authors. At first mathematical calculations were made about the efficiency of join and that of various operators in abstract machines. However, the cost of the external I/O operations, precisely the most significant factor, was not taken into account in them. Later it was considered as well, and the newest systems determine the access algorithm from the data access independent query of the relational model on the basis of the physical storage structure, rather than optimizing mathematical operations.

Look at the following simple relations:

```
SPARE-PART (SP-NUMBER,SUPPL-NUMBER,STOCK,PRICE)
SUPPLIER (SUPPL-NUMBER,NAME,ADDRESS,CREDIT-RATING)
```

where there is a many to many relationship between spare parts and suppliers, to wit, any spare part can be obtained from several suppliers, and any supplier can deliver several kinds of spare parts.

The following query: 'Give those SP-NUMBER's — and the SUPPL-NUMBER's and ADDRESSes of their suppliers — whose PRICES are more than 100 and where the CREDIT-RATING of the supplier is less than 5.' can be algebraically formulated in the relational model, for example, in the following way:

```
[SPARE-PART RESTRICT WHERE PRICE > 100]
JOIN
[SUPPLIER RESTRICT WHERE CREDIT-RATING < 5]
(SUPPL-NUMBER))
PROJECT
[SP-NUMBER,SUPPL-NUMBER,ADDRESS]
```

(To aid reading, parentheses, braces, and brackets indicate the operations belonging logically together.)

As seen above, the result is a relation produced by the natural join of the SPARE-PART and SUPPLIER relations based on the SUPPL-NUMBER, a restriction for the PRICE and CREDIT-RATING domains, and a projection to the SP-NUMBER, SUPPL-NUMBER, and ADDRESS columns.

Since the relational model and algebra contain no restriction for physical data access, the required result relation can be produced in several ways. The following four are worth examining from a practical point of view. However, in the interest of simpler coding and generalization, the above specific query is rewritten in the following general form:

```
R=([R1 RESTRICT WHERE r1:F1] JOIN [R2 RESTRICT WHERE r2:F2]) (e)
PROJECT [p]
```


where r_1 , r_2 , e , and p may refer to several columns of the relations. The e joining column(s) is interpreted on the E domain(s). Based on this, the result of the operations will be a projection of the join of the R_i ($i=1,2$) relations which are restricted by the $r_i:F_i$ conditions and projected to the P domain(s).

Suppose that R_1 and R_2 are physically-existing tables whose rows are obtained from one physical record; namely, the entire row is always transferred by an I/O operation. In this case, the projection and when it is executed does not influence directly the number of I/O operations. Naturally, in the temporary relations arising from the intermediate steps, the sooner the projection, the sifting out of intermediate columns, is executed, and those unnecessary for the final result or for any further operations are dropped, the less unnecessary data will be stored from among the rows. That is why it is accepted as a general rule that those columns neither necessary in the final result nor in the intermediate steps are excluded at once while those used only in any of the intermediate steps (for example, for join), are kept only as long as they are needed.

Since relational operations contain no instructions how data are actually to be accessed, the above task can be solved by the system in several ways.

a) *Index Examination for the Join Domains*

If both relations are indexed on the join domains, then both indices are examined. From among the indices, those value pairs are selected one by one, for which $e_{1,n}=e_{2,m}$, where $e_{1,n}$ and $e_{2,m}$ are the values of the E_1 and E_2 domains in any entry of R_1 and R_2 , respectively. Next, for example, based on the addresses belonging to the e_1 index, it is examined whether R_1 contains any entry fulfilling at all the $r_1:F_1$ condition. If there is none like this, then the examination shifts to the following $e_{1,n}=e_{2,m}$ value pair. If there is such an entry, then after finding the first entry, processing follows with the entries under the addresses belonging to the e_2 index in the R_2 relation, and temporarily store the projections of all those to P in a work area fulfilling the $r_2:F_2$ condition. Next R_1 is returned to. The projections to P from among the relations under the addresses of the e_1 index and fulfilling $r_1:F_1$ are concatenated one by one with all the partial entries stored temporarily in the work area. The so-obtained new entries are parts of the results; they are therefore transferred to the output work area of the data base management system or the user. The above operations are executed on all the equivalent $e_{1,n}=e_{2,m}$ pairs.

b) *Join with Presort*

Using the access path of the storage structure, both R_1 and R_2 are examined. This means either a serial access or an indexed one if indices exist on r_i . The W_i ($i=1,2$) auxiliary files are produced containing the projections of those R_i entries to P_i fulfilling the $r_i:F_i$ restriction. The auxiliary files are sorted according to the join domain. Next the two sorted files are merged in one round on the basis of the $E_1=E_2$ values. The rows are concatenated in accordance with the relational join operation.

c) *Selection in Several Rounds*

Instead of sorted auxiliary files, the projected entries of R_1 satisfying the conditions are loaded into the storage in several rounds, sorted there according to E_1 , and then joined with the appropriate R_2 elements one by one.

d) Address Matching

If there are indices for both the join and restricting domains, then on the basis of the latter the $W'_i (i=1,2)$ sorted auxiliary files are produced containing the addresses of those entries satisfying the restrictive conditions. As with procedure *a)* all of the addresses belonging to an $e_{1,n}=e_{2,m}$ pair are examined and if the same addresses can be found both in W'_1 and W'_2 , then the projections of the entries defined by the addresses are produced and joined.

Procedure *a)* is indisputably the most efficient if the relations are not indexed for the restricting domains, and if the entries of the individual relations cluster according to the index on the join domains.

However, it is very disadvantageous if they do not cluster around the indices of E_1 and E_2 , but instead around some other S_1 and S_2 indexed domains [e.g., the physical storage is sequential or indexed-sequential on the basis of $S_i (i=1,2)$, and all the entries of the R_i relations are accessible according to S_i . In this case, procedure *b)* or *c)* is much more advantageous. Procedure *c)* is the most efficient if the work place is big enough to receive in one round the projections of all those entries of R_1 satisfying the conditions, to wit, the relations are not too large or the restrictions are very selective. However, if the relations are indexed for both the join and the restricting domains, then procedure *d)* is expedient to apply mainly if the relations cluster according to the indices of the join domain.

Actual numeric results demonstrating the efficiency of the algorithms detailed above and how the number of external storage requests depends upon the size of the relations, the selection, and the physical data structure are shown in

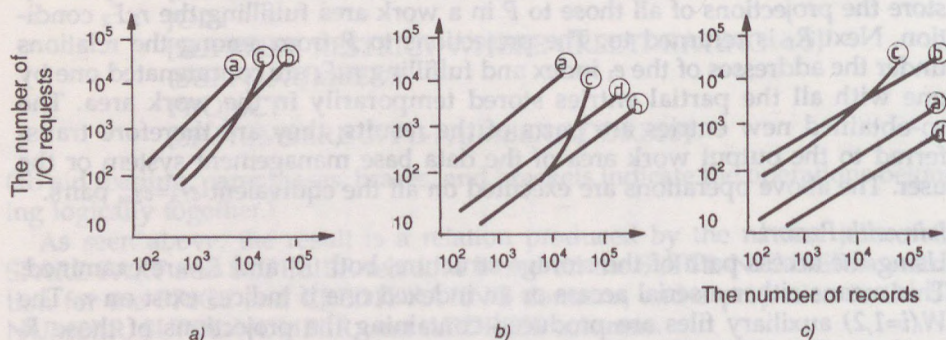


Fig. 7.14. Efficiency of various access paths

Rows are selected from the joined relations by restriction. The restrictions are fulfilled by 10% of the rows in both relations. *a)* Both relations are indexed for the domain serving the join, but they do not cluster. *b)* The relations are indexed for both the join and restricting domains, but cluster according to neither of them. However, both relations cluster according to another domain. *c)* The relations are indexed for both the join and restricting domains, and they cluster according to the join domain in both relations. *d)* Address matching. The relations are indexed for both the join and restricting domains. The indicators written next to the graphs refer to the access paths cited in the text

Fig. 7.14. It has been assumed when calculating that the selection condition retains 10% of the rows.

In the relational model, precisely as a result of significant data independence, the user does not have the option of defining the actual data access path. However, in the more-developed systems, he can observe how it is finally achieved and can obtain an estimate as well about the resource requirement of the query, and, in practice, the predicted average access time (see the EXPLAIN command in Section 13.6.). On this basis, in large data bases when solving complex tasks, the user can perhaps make access of the actual data more efficient by reformulating the query.

7.9. Normalization

7.9.1. Concept of Normalization

The first step of data base development is to collect those data needed by various applications (or possibly needed in the future). The next step is to try to organize the thereby assembled 'chaos' into easy to handle units. These two steps are — independent of the data model — the alpha and omega of data base organization. A great advantage of applying the relational model is that it offers a procedure to create relationships and produce reasonable data classifications that can be used well in practice. The procedure has been theoretically established and can be written in a formal algorithm. (So future 5th generation softwares promise to automatically and mechanically generate it.) This procedure is called *normalization*.

Normalization is always done at the logical data level. Generally, it consists of several steps. A significant part of them could be carried out also by a skilled data base developer without any theory, just by using common sense. However, knowledge of the normalization rules helps precisely to avoid trying ad hoc solutions which may overlook some important data classification possibilities or poorly embed them into the relationships during development, thereby resulting in a failure or some difficulties with the actual work.

Naturally, normalization is not a panacea either. Neither does its systematic application solve every problem every time. The data base administrator encounters frequently the problem that the system works rather slowly with theoretically perfectly-normalized relationships. In that case, he is often forced to make a compromise. In the interest of efficiency he may be satisfied with a lower-level normalization which is perhaps logically less easy to survey and less clear, but ensures greater efficiency. This will be seen later in the last example of Point 7.9.5.

Despite all these factors, it can be said generally as a consequence of a good normalization that:

- the data storage demand shrinks,
- elementary data can be modified faster and with less risk of error, and
- the data base will be logically more understandable.

To facilitate comprehension, the process and the problems of normalization will be illustrated by an example.

However, before moving to this, a new concept — *dependency* — must be introduced. Essentially, this is the basis of the entire normalization theory and process.

7.9.2. Dependencies

In any relation the values of some attributes depend upon the values of other attributes.

If in the R relation the value of one attribute (X), the independent variable, unambiguously determines the value of another attribute (Y), the dependent variable, then it is said that Y depends functionally upon X in the R relation. Naturally, this unambiguous relationship applies not only to the actual content of R, but it is a restriction that is independent of time and exists throughout the entire life of the data base.

Both attributes X and Y can also be composite; they can be composed of several columns.

The usual symbol of functional dependency is:

$$R.X \longrightarrow R.Y$$

This can be represented in a dependency diagram (see Fig. 7.15.).

Naturally, the general model can tell nothing about what dependencies exist among the individual attributes. They have always to be clarified by the data base developer who is familiar with the circumstances of the given task. It follows from the definition of the relation that all attributes depend upon the primary key and obviously — if they exist — upon all alternative keys as well.

The definition of functional dependency identical to the above is more useable in practice.

In a given R relation column Y depends upon X if and only if in any two rows of it in which the values of X are the same, the values of Y are always the same as well.

Naturally, the reverse of the statement is not true. To the same Y value different

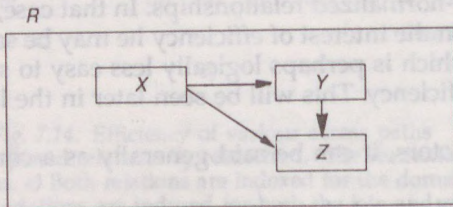


Fig. 7.15. Representation of the functional dependency. An arrow points to the dependent attribute from the independent one. In the figure also Y and Z depend functionally upon X, and Z depends upon Y as well.

X 's may belong. For example, in the VEHICLE table of Fig. 7.2. (MAKE, MODEL) $\rightarrow$ VOLUME. On the other hand, many (in theory, any number of) cars with the MAKE, MODEL and (MAKE, MODEL) combinations can have the same VOLUME value.

Functional dependency means a 1:N relationship within the relation. A special case of this is the 1:1 relationship when in the R relation not only the value of column X defines unambiguously the value of Y , but also, inversely, the value of Y defines unambiguously X . The usual symbol of this is

$$R.X \longleftrightarrow R.Y$$

An example of reciprocal functional dependency is the reciprocally unique relationship between the goods code or section code and their names in an unambiguous goods or section code system or the relationships between the primary and the alternative keys in any relation.

In the previous example, the independent attribute was composite and the dependent one was dependent only upon the combination of the two. This kind of dependency is called *total functional dependency*.

Generally speaking, in the R relation attribute Y possesses total functional dependence upon (the composite) attribute X if and only if it depends functionally upon X , but does not depend upon any real component of X .

If X is not composite, then functional and total functional dependency mean the same.

An additional example of functional dependency, but not of total dependency, is a multi-item consignment note containing among others, the consignment note number, the goods code (these two together form the primary key), the goods name, its quantity and price, and the delivery date.

Here the quantity depends functionally and totally upon the primary key while the date and the name or the price of the goods do not. The date is unambiguously defined by the consignment note number while the other two by the goods code.¹⁶

In addition to these fundamental dependency relationships, later multi-valued functional dependency will be needed also. In this case, the value of X defines not one Y value, but a group of them. (For instance, a given course can be taught only by some particular persons from among the teachers.)

In a general wording, in the R relation that has attributes A , B , and C , B is in a multi-valued functional dependency on A if and only if the potential B values which belong to any given A and C values depend only upon the value of A and are independent of C .

(If B can have always only one value, then this is equivalent to the usual functional dependency.)

¹⁶ It is assumed that the good's price is constant for everyone; at most, some can obtain a discount which can be another attribute. If periodically changing prices are used, then the goods depends functionally not upon the good code, but, for example, upon a combination (goods code, delivery date). Naturally, these conditions have to be recognized during the logical planning of the database.

The symbol of the multi-valued functional dependency is

$$R.X \multimap R.Y$$

The database management system can guarantee automatically the dependencies among various relations by defining the foreign keys. However, there is no similar automation for maintaining the dependencies among the columns of the same relation. This should be checked always by the program, otherwise contradictions can arise within the relation.

By normalization — as will be seen — these dependencies will be eliminated. Thus contradictory data are blocked from entering into a correctly normalized data base.

7.9.3. First Normal Form (1NF)

To understand the process of normalization, the following task will be handled:

A register should be prepared about the employees of a company. The following data are needed about each employee (the name of the column is in parentheses):

- social security number (EMP-SSN),¹⁷
- last name (LAST-NAME),
- first name (FIRST-NAME),
- the code of the section where he works (S-CODE),
- the name of the section (S-NAME),
- the code of the task he works on (TASK-CODE),
- the name of the task (TASK),
- what percentage of his job is devoted to the task (PERCENT),
- year of birth (BIRTH), and
- the social security number of his supervisor (SUP-SSN).

Based on additional analysis of company conditions, it is found that:

- the social security number is unique,
- the section codes and names are unique, and a one-to-one relationship exists between them,
- the same applies to the task codes and names,
- an employee can be a member of one section, but work on several tasks,
- it is not necessary to establish a relationship among tasks and sections, and
- it is not necessary to check automatically whether the time devoted to all tasks is the employee's entire work time (100%).

Data are collected by sections. So the starting table contains one record, one 'row' per section. The first field is the section code, the second is the section name; then come the above-listed data about individual employees as many

¹⁷ For clarity, social security numbers are represented only as 5 digit numbers.

S-Code S-Name Emp-SSN Last-Name First-Name Task-Code Task Percent Birth Sup-SSN

recurring
as many times as there are employees in the section

a)

Starting data. One row for each section.

S-Code	S-Name	Emp-SSN	Last-Name	First-Name	Task-Code	Task	Percent	Task-Code	Task	Percent	Birth
F01	Finance	06502	Townes	Alex	P1	Payrl	30	A1	Acct	70	52

Emp-SSN	Last-Name	First-Name	Task-Code	Task	Percent	Birth
05110	Taylor	Mary	A1	Acct	50	55

Emp-SSN	Last-Name	First-Name	Task-Code	Task	Percent	Birth	Sup-SSN
01340	Harris	Frank	A1	Acct	20	71	01340

S-Code	S-Name	Emp-SSN	Last-Name	First-Name	Task-Code	Task	Percent	Birth	Sup-SSN
M02	Marketing	03500	Hemmings	Bill	B1	Mkt.Rs.	35	71	00358

b)

Fig. 7.16. Data collected by departments: a) Logical representation, b) Actual data.

EMPLOYEES – 1NF

S-CODE	S-NAME	EMP-SSN	LAST-NAME	FIRST-NAME	TASK-CODE	TASK	PER-CENT	BIRTH-YEAR	SUP-SSN
F01	Finance	06502	Townes	Alex	P1	Payroll	30	1952	01340
F01	Finance	06502	Townes	Alex	A1	Accounting	70	1952	01340
F01	Finance	05110	Taylor	Mary	A1	Accounting	50	1955	01340
F01	Finance	01340	Harris	Frank	A1	Accounting	20	1971	01340
F01	Finance	03250	Jones	Clara	B1	Market res.	50	1952	01340
M02	Marketing	03500	Hemmings	Bill	B1	Market res.	35	1971	03500
A11	Admin.	00130	Dederich	Devon	A1	Accounting	80	1962	00177
A11	Admin.	00200	Matheson	Ellen	A1	Accounting	50	1970	00177
A11	Admin.	00200	Matheson	Ellen	B1	Market res.	50	1970	00177
A11	Admin.	00298	White	Joseph	B1	Market res.	100	1947	00177
A11	Admin.	00177	Noble	Mark	B1	Market res.	40	1971	00177

primary key
(EMP-SSN, TASK-CODE)

Fig. 7.17. By eliminating the repeating fields, the data are brought into the EMPLOYEES-1NF 1NF relation. The primary key is the combination of the EMP-SSN and TASK-CODE columns.

times as are employees in the section. Finally, the last field is the social security number of the section supervisor (see Fig. 7.16.).

This 'table' is not a relation since there are repeating fields in it. The column combination that contains the individual employees occurs in the various rows in a varying number. To be able to work with it in the relational model, it first has to be transformed into a relation; the repeating fields have to be eliminated. This can be achieved by creating a new table having the same columns as the original one, but new rows are formed from each of the repeating fields differing from each other in the content of the originally-repeating field (group). The table obtained thereby already satisfies the relational conditions; it can be said that the relation is in the *first normal form* (1NF) (see Fig. 7.17.)

In general, it is said that the data of a table are in the 1NF, i.e., they form a relation, if the conditions detailed in Section 7.2. are fulfilled, namely:

- every column refers to one and only one attribute,
- every row is different,
- the order of attributes is the same in every row,
- there are no repeating fields, and
- there exists (at least) one unique key in every row upon which all the other attributes functionally depend.

Every relational database management system requires data to be at least in the first normal form (1NF). Although at this normalization level many problems can still arise, a higher degree is not needed. Yet as will be seen later, it is generally recommended.

To truly fulfill the conditions of 1NF in the example, it has to be checked whether the rows possess their own unique keys. The social security number itself is not one since a person can work on several tasks. However, for example, if combined with the task code, then it identifies unambiguously. So the primary key of the table will be the social security number and task code composite key. (The social security number-task name combination is an alternative key.)

7.9.4. Second Normal Form (2NF)

It does not require a great deal of wisdom to see that the 1NF table in Fig. 7.16. contains an enormous number of redundancies. It is obvious at once that it is unnecessary to separately present the names of every section and task in every row. However, besides these obvious redundancies, there are also other more obscure potential failures in this table. Their common name is *modification anomalies*. This includes insert, update, and delete anomalies.

Hereafter it will be discussed what happens if, for example, Alex Townes or Bill Hemmings leaves the company, that is to say, their data are deleted from the table. In that case, the P1 task code automatically disappears from it, and the M02 marketing section code as well. (At the same time, this means also that the P1 task and the M02 section names will not be known in this case.) In effect, information has been deleted inadvertently from the data base which will be needed later on. This is the *delete anomaly*.

It can cause additional problems if Ellen Matheson gets married, or, for example, the A11 section gets a new supervisor. In both cases, several rows have to be changed simultaneously in order for the data to remain consistent. The update anomaly is when updates have to be carried out in several locations after data are changed. The *update anomaly* results also in contradictions in the data base if modifications are not executed consistently. (For example, at Devon Dederich in the A11 section, the code of the supervisor could remain 00177, while at the others it would change to 00298.)

Finally, what problems may arise when entering new rows? What is the *insert anomaly*?

If a new section is formed in the company, or a new employee is hired, but no task is assigned to him, then neither of these data can be entered into the data base. They would produce a row in which the primary key (or a part of it, respectively) had a null value. To see clearly the source of all these problems, the dependency diagram of the relation in Fig. 7.17. should be studied (Fig. 7.18.).

It shows that only the percentage distribution (PERCENT) depends functionally totally upon the composite primary keys, the social security number (EMP-SSN) and the task code (TASK-CODE) combination. All of the other attributes depend only upon a component of the primary key.

A relation is in the second normal form (2NF) if and only if it is in the 1NF and all the non-key attributes depend totally functionally upon the primary key.

1NF relations with elementary primary keys are automatically in the 2NF also. However, relations with composite keys have to be transformed into the 2NF in the interest of ending modification anomalies. (It is not certain that this will eliminate all anomalies, but it can considerably reduce their number.) This is called the *decomposition* of the relation.

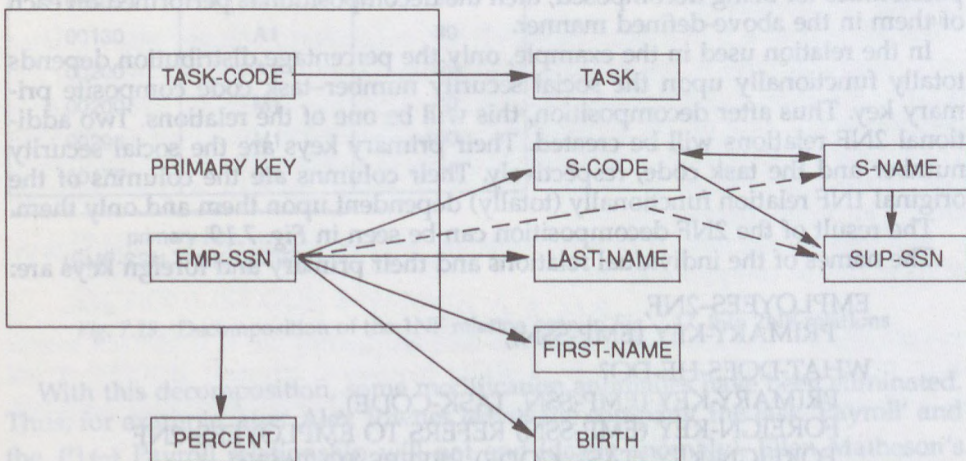


Fig. 7.18. Dependency diagram of the personnel register described with the relation seen in Fig. 7.17. The $\text{EMP-SSN} \rightarrow \text{DEPT-NAME}$ and $\text{EMP-SSN} \rightarrow \text{SUP-SSN}$ dependencies are not shown in the interest of clarity since they are automatically realized through the S-CODE anyway

To decompose a 1NF relation into 2NF, new relations are created. The first 2NF relation is its projection whose primary key is the primary key of the original relation, and whose columns are those and only those that depend functionally totally upon this key. The primary keys of the other 2NF relations are the components of the original primary key on which other columns are dependent. They contain the columns dependent on this primary key.

In a general wording, if in a relation with columns A , B , C , and D , the primary key is the combination of A and B , and D depends functionally upon A , then (with short, clear symbols)

$R(A, B, C, D)$
 PRIMARY-KEY (A, B)
 $R.A \rightarrow R.D$

then the decomposition into the second normal form means the substitution of the original R relation with the following two new relations:

$R_1(A, D)$
 PRIMARY-KEY (A)

and

$R_2(A, B, C)$
 PRIMARY-KEY (A, B)
 FOREIGN-KEY (A), REFERS TO R_1

The foreign key is necessary to ensure data base integrity. By this the join of the new R_1 and R_2 relations over A , the

$R_1 \text{ JOIN } R_2(A)$

operation, always produces all the rows of the R relation and only them.

If the complex key has not two, but several totally functionally dependent possibilities for being decomposed, then the decomposition is performed on each of them in the above-defined manner.

In the relation used in the example, only the percentage distribution depends totally functionally upon the social security number-task code composite primary key. Thus after decomposition, this will be one of the relations. Two additional 2NF relations will be created. Their primary keys are the social security number and the task code, respectively. Their columns are the columns of the original 1NF relation functionally (totally) dependent upon them and only them.

The result of the 2NF decomposition can be seen in Fig. 7.19.

The names of the individual relations and their primary and foreign keys are:

EMPLOYEES-2NF,
 PRIMARY-KEY (EMP-SSN),
 WHAT-DOES-HE-DO?
 PRIMARY-KEY (EMP-SSN, TASK-CODE),
 FOREIGN-KEY (EMP-SSN) REFERS TO EMPLOYEES-2NF
 FOREIGN-KEY (TASK-CODE) REFERS TO TASKS
 TASKS
 PRIMARY-KEY (TASK-CODE),

The columns of the so-created 2NF tables can be seen in the Fig. 7.19.

EMPLOYEES – 2NF

S-CODE	S-NAME	EMP-SSN	LAST-NAME	FIRST-NAME	BIRTH	SUP-SSN
F01	Finance	06502	Townes	Alex	52	01340
F01	Finance	05110	Taylor	Mary	55	01340
F01	Finance	01340	Harris	Frank	71	01340
F01	Finance	03250	Jong	Hady	52	01340
M02	Marketing	03500	Hemmings	Bill	71	03500
A11	Admin.	00130	Dederich	Devon	62	00177
A11	Admin.	00200	Matheson	Ellen	70	00177
A11	Admin.	00298	White	Joseph	47	00177
A11	Admin.	00177	Noble	Mark	71	00177

primary
key
(EMP-SSN)

WHAT-DOES-HE-DO?

EMP-SSN	TASK-CODE	PERCENT
06502	P1	30
06502	A1	70
05110	A1	50
01340	A1	20
03250	M1	50
03500	M1	35
00130	A1	80
00200	A1	50
00200	M1	50
00298	M1	100
00177	M1	40

primary key
(EMP-SSN, TASK-CODE)

TASKS

TASK-CODE	TASK
A1	Accounting
M1	Market Res.
P1	Payroll

primary key
(TASK-CODE)

Fig. 7.19. Decomposition of the 1NF relation seen in Fig. 7.17. into 2NF relations

With this decomposition, some modification anomalies have been eliminated. Thus, for example, after Alex Townes leaves the company, the task 'Payroll' and the $P1 \leftrightarrow$ Payroll relationship will not end (delete anomaly). Ellen Matheson's name has to be changed at one location only (update anomaly) and, in addition, new employees can be employed without giving them tasks (insert anomaly). However, the other problems will persist in the future. As will be seen in the next point, those can be eliminated only by decomposition into the third normal form.

This decomposition is *complete*, i.e., it contains all the information that could be found in the original relation. However, in addition to this, it contains more as well, namely that the name of the P1 task code is 'Payroll' and the M02 section name is 'Marketing', while the original relation contains them in the data base only as long as there is at least one employee working on that task or in that section, respectively.

7.9.5. Third Normal Form (3NF)

In relations transformed into the 2NF, the following modification anomalies still exist:

- after Bill Hemmings leaves the company (deletion), information concerning the marketing section disappears,
- if the A11 section gets a new supervisor (update), it has also to be modified at several locations in the future,
- it is not possible to enter a new section (insert) until it has at least one employee.

If the dependency diagram of the 2NF relations is drawn (see Fig. 7.20.), then it can be seen that the only functional dependency not based upon a primary key originates from the section code (section name), and somehow all the anomalies are also connected to this attribute.

If these dependencies are also eliminated by forming new relations, then each of the new relations will be in the third normal form (3NF). In this case, as will be seen, the remaining modification anomalies will also diminish.

A relation is in the 3NF if and only if it is in the 2NF and all attributes that are not part of the primary key depend functionally only upon the primary key.

In a different wording, the 3NF means that functional dependencies can originate only from the primary and the alternative keys.

The relation EMPLOYEES-2NF is not in the 3NF because, for example, the section code (S-CODE) is not a primary or an alternative key, and other columns depend functionally on it.

To decompose a 2NF relation into 3NF, first a projection is made. This includes only those attributes which depend solely upon the primary key. Its primary key remains the same. An additional new relation is formed containing the column(s) depending not only on the primary key and the column on which it depends. The primary key of the new relation (or relations, if there are several dependencies) is the independent attribute of the decomposed relation, and its columns are the attributes depending on it.

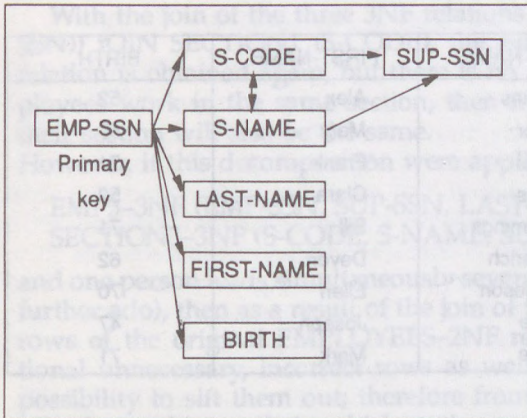
In a general wording, if in a 2NF relation with columns *A*, *B*, *C*, and *D* (any of them can be composite as well),

R (*A*, *B*, *C*, *D*)
PRIMARY-KEY (*A*)
R.B → *R.C*

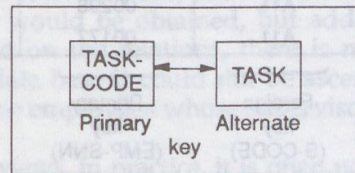
then decomposition into the 3NF means the creation of the following relations:

*R*₁ (*B*, *C*)
PRIMARY-KEY (*B*)
and *R*₂ (*A*, *B*, *D*)
PRIMARY-KEY (*A*)
FOREIGN-KEY (*B*), REFERS TO *R*₁

EMPLOYEES-2NF



TASKS



WHAT-DOES-HE-DO?

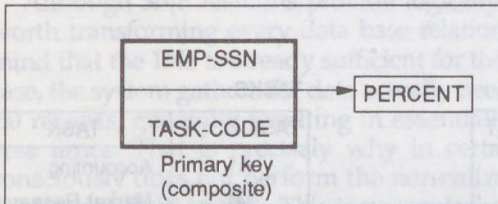


Fig. 7.20. Dependency diagram of the 2NF relations seen in Fig. 7.19.

The R relation can be restored unambiguously by the join of R_1 and R_2 (over B).

The decomposition of the EMPLOYEES-2NF relation following this rule is shown in Fig. 7.21. However, it is essential that the decomposition should proceed according to the above-defined principle since, for example, EMPLOYEE-2NF could also be in principle decomposed into two 3NF relations in the following way:

EMP1-3NF (EMP-SSN, S-CODE, LAST-NAME, FIRST-NAME, BIRTH)

EMP2-3NF (EMP-SSN, SUP-SSN)

SECTION1 (S-CODE, S-NAME)

where EMP-SSN is the primary key of both EMP relations and the SECTION1 key is S-CODE. Integrity is ensured by the foreign keys S-CODE in EMP1-3NF and EMP-SSN in EMP2-3NF.¹⁸

¹⁸ If the primary key in SECTION1 was the S-NAME alternative key instead of S-CODE, then S-CODE could not be defined as a foreign key. So integrity would not remain automatically. This is also an example that it is not immaterial which of the candidate keys is finally selected to be the primary key.

EMPLOYEES

S-CODE	EMP-SSN	LAST-NAME	FIRST-NAME	BIRTH
F01	06502	Townes	Alex	52
F01	05110	Taylor	Mary	55
F01	01340	Harris	Frank	71
F01	03250	Jones	Clara	52
F02	03500	Hemmings	Bill	71
A11	00130	Dederich	Devon	62
A11	00200	Matheson	Ellen	70
A11	00298	White	Joseph	47
A11	00177	Noble	Mark	71

Foreign
key
(S-CODE)

Primary
key
(EMP-SSN)

WHAT-DOES-HE-DO?

EMP-SSN	TASK-CODE	PERCENT
06502	P1	30
06502	A1	70
05110	A1	50
01340	A1	20
03250	M1	50
03500	M1	35
00130	A1	80
00200	A1	50
00200	M1	50
00298	M1	100
00177	M1	40

Primary key
(EMP-SSN, TASK-CODE)

Foreign key
(EMP-SSN) (TASK-CODE)

EMPLOYEES

TASKS

TASKS

TASK-CODE	TASK
A1	Accounting
M1	Market Research
P1	Payroll

Primary key
(TASK-CODE)

SECTIONS

S-CODE	S-NAME	SUPERVISOR
F01	Finance	01340
M02	Marketing	03500
A11	Administration	00177

Primary
key
(S-CODE)

Foreign key
Supervisor
refers to
EMPLOYEES

Fig. 7.21. Decomposition of the 1NF relation seen in Fig. 7.17. into 3NF relations
(Using the 2NF decomposition seen in Fig. 7.19.)

With the join of the three 3NF relations (EMP1-3NF JOIN EMP2-3NF (EMP-SSN)) JOIN SECTION1 (S-CODE), the exact content of the EMPLOYEES-2NF relation is obtained again, but there is no automation to ensure that if two employees work in the same section, then their supervisor and the supervisor of their section will also be the same. However, if this decomposition were applied

EMP3-3NF (EMP-SSN, SUP-SSN, LAST-NAME, FIRST-NAME, BIRTH)
SECTION3-3NF (S-CODE, S-NAME, SUP-SSN)

and one person leads simultaneously several sections (which can happen without further ado), then as a result of the join of the two 3NF relations, not only all the rows of the original EMPLOYEES-2NF relation would be obtained, but additional unnecessary, incorrect rows as well. Based on the relations, there is no possibility to sift them out; therefore from this data base it could not be ascertained unambiguously in which section work those employees whose supervisor leads several sections.

Although 3NF relations provide logically clear views, in practice, it is often not worth transforming every data base relation into this form. (It should be borne in mind that the 1NF is already sufficient for the database management system.) In that case, the system gathers the data actually needed by the user from many more physical records, generally resulting in essentially more I/O operations and longer access times. That is precisely why in certain cases the data base administrator consciously does not perform the normalization up to that level; he keeps certain relations in 2NF or 1NF. This is particularly expedient if the data depending upon each other are regularly used together and change comparatively rarely, and the space occupied by the individual dependent data is relatively compact. Naturally in this case, the data base administrator has to be aware that he himself or the user has to arrange for the elimination of potential update anomalies.

A typical example of this is a register containing the zip code and exact address in addition to other data. As the exact address (house number, street, city, state, country) unambiguously defines the zip code, this relation is not in the third normal form. At the same time, every reasonable application uses the address, together with the zip code. Therefore it would be unnecessary to decompose the relation into two 3NF relations and slow down processing by always joining with the ZIPCODE (ADDRESS, ZIPCODE) relation.¹⁹

¹⁹ Not to mention the practical problem that should not at all be neglected: if the data base administrator does not want to assign a different entry for every individual house number in the ZIPCODE table, then how can he expediently solve the problem that while, for example, in the town of Honeyoe, NY all streets have the same zipcode, in New York one street may cut across several zip code zones (for example, the zip code for Park Avenue varies, based on whether the house number is odd or even, or is above or below a given value).

7.9.6. Post-Normalization Problems, Higher Normalization Forms

In practice, it is sufficient in most cases to organize the data into 3NF relations using the algorithms detailed in the previous points, or because of the compromise made in the interest of efficiency, into 2NF or 1NF relations, respectively. However, life is very diverse, and such conditions may exist which cannot be ensured automatically by well-decomposed 3NF relations either. Next two will be shown, of which the second example leads to even better-normalized 4NF relations. Bear in mind that the specialized literature already acknowledges 5NF relations as well, and it is possible that higher forms will soon be introduced also. However, their importance is rather theoretical. The systematically-created third normal form is almost always sufficient in practice, and moreover, as could be seen in the previous point's example, even it is sometimes excessive.

As the first example, the SCL relation is shown. Its columns are STUDENTS, COURSE, and LECTURER. The rows of the SCL relation define what courses the students take and from whom.

Additional conditions arising from real world restrictions are:

- every student takes a given course from only one lecturer (but can take several courses), and
- every lecturer gives only one course (but a course can be given by several lecturers).

The primary key of the SCL relation is (STUDENT, LECTURER), but it is not in the 3NF because the functional dependency $LECTURER \rightarrow COURSE$ exists.

Thus when inserting a new row into SCL, it needs to be checked separately, independent from the database management system, whether the given lecturer actually teaches the given subject.

If the solution chosen for this was to decompose the SCL relationship into the 3NF on the basis of what has been discussed thus far, then the following relations would be obtained:

SL (STUDENT, LECTURER)
 PRIMARY-KEY (STUDENT, LECTURER)
 FOREIGN-KEY (LECTURER) REFERS TO LC

and

LC (LECTURER, COURSE)
 PRIMARY-KEY (LECTURER)

Joining the two relations, exactly SCL will be obtained. In spite of this, the solution is not perfect.

Suppose John Scherlis studies computer science from Machiel van der Loos, but this course is also given by Mark Pevsner. There is no data base automation to ensure that the row John Scherlis, Mark Pevsner will not be entered into the SL relation, although — since according to the conditions Mark Pevsner teaches only computer science — this contradicts the condition arising from the real world model that a student takes a course from only one lecturer.

A second example where the 3NF does not produce the desired result is the following task with these entities:

- subjects (a specific lecture on a particular subject is a course),
- lecturers, and
- books.

Additional conditions:

- one subject can be taught by several lecturers, but a given course can be given by only one,
- several books are assigned for one course and all of them must be used, no matter who gives the course, and
- every lecturer gives only particular courses, i.e., any subject can be taught only by given lecturers.

SUBJECT-LECTURER-BOOK

SUBJECT	LECTURER	BOOK
Computer Science	{ Michael van der Loos } { Mark Pevsner }	{ Introduction to Computer Science } { Computer Science for System Analysts }
Physics-III.	{ Michael van der Loos }	{ The Way to Nuclear Physics } { Applications of Radioactive Isotopes } { Physics III. }
Mathematics-I.	{ Esther Schwartz } { Richard Zeney } { Kenneth Sokolow }	{ Calculus }

Fig. 7.22. The relationship among courses, lecturers, and books in a non-normalized form

The non-normalized data can be seen in Fig. 7.22.

By eliminating the repeating fields, data can be transformed into the 1NF. The so-created SLB (SUBJECT, LECTURER, BOOK) relation consists only of the primary key. Thus it is automatically in the 3NF (see Fig. 7.23.). Despite this, this model contains numerous redundancies. (For example, it is unnecessary to define three times that Michael van der Loos teaches Physics III or two times which books are compulsory for Computer Science, etc.)

SLB

SUBJECT	LECTURER	BOOK
Computer Science	Michael van der Loos	Introduction to Computer Science
Computer Science	Michael van der Loos	Computer Science for System Analysts
Computer Science	Mark Pevsner	Introduction to Computer Science
Computer Science	Mark Pevsner	Computer Science for System Analysts
Physics-III.	Michael van der Loos	The Way to Nuclear Physics
Physics-III.	Michael van der Loos	Applications of Radioactive Isotopes
Physics-III.	Michael van der Loos	Physics III.
Mathematics-I.	Esther Schwartz	Calculus
Mathematics-I.	Richard Zeney	Calculus
Mathematics-I.	Kenneth Sokolow	Calculus

Fig. 7.23. Data arrangement seen in Fig. 7.22. in a normalized relation
The composite primary key is (SUBJECT, LECTURER, BOOK)

In addition, the usual modification anomalies crop up. For example, the addition of a new Physics III lecturer results in the entry of three new rows, and moreover, in such a way that BOOK can have only well-defined values which, however, cannot be ensured automatically by integrity conditions. Similarly, the deletion or update of a compulsory book usually refers also to several rows.

All these difficulties stem from the existence of a multi-valued functional dependency between the individual columns of the primary key:

SUBJECT —————→ LECTURER
SUBJECT —————→ BOOK

To eliminate the problems stemming from the multi-valued functional dependency, the fourth normalized form (4NF) will be introduced.

A R relation is in the 4NF if and only if it is in the 3NF, and if it contains any multi-valued functional dependency, for example, $R.A \twoheadrightarrow R.B$, then all the other $C, D, \dots$ columns of the relation depend functionally upon A , that is to say, $R.A \rightarrow R.C$; $R.A \rightarrow R.D \dots$

The relation cited above as an example is obviously not in the 4NF since

SUBJECT —————→ LECTURER is true, but
SUBJECT —————→ BOOK is not valid
(only SUBJECT —————→ BOOK)

The decomposition into the 4NF decomposes the columns in multi-valued dependencies into separate relations, namely in the current example:

SL (SUBJECT, LECTURER)
PRIMARY-KEY (SUBJECT, LECTURER)

and

SB (SUBJECT, BOOK)
PRIMARY-KEY (SUBJECT, BOOK)

With this decomposition algorithm the redundancy is eliminated, and it is possible to fully restore the original data by the operation SL JOIN SB (SUBJECT) without any loss.

However, one problem still remains, although this may be advantageous in practice. A new subject could have been entered into the original SLB relation only if at least one lecturer *and* at least one book were associated with it (if not differently, for example, with the 'NONE' default value, instead of the null value). In contrast, in the 4NF relation a new subject can be entered if either a lecturer who gives it *or* a book assigned to it exists.

If analysis of the actual circumstances reveals the existence of multi-valued functional dependencies, then it is generally expedient to decompose the relations to the 4NF.

7.9.7. Defining Views from the Normalized Relations

Thus far — not counting the exception made in the interests of efficiency — only the advantages of normalization have been shown. However, all these advantages would shrivel to nothing for the average user if, for example, he could formulate the most frequent query of the normalized relation seen in Fig. 7.21.:

'What are the names of the section and task, and the percentage with which a given employee is involved?' only always by the explicit definition of the join of four tables.

In this case, the use of the relational model would not be at all simpler than navigating in a network data base; the former is handicapped since more efficient access paths based on full knowledge of the set connections cannot be defined. The skilled user is able to exploit such paths in a network data base. Precisely to ease the users' work, only the physically-stored data, the real tables, are normalized. From them all the virtual relations and views which may be regularly needed by the users can be created by relational operations. These views are often only 1NF relations. However, users refer to these views in which the data appear redundant, but are joined in the desired way.

Thus, for example, for the query formulated at the beginning of this point, a view named ANSWER1 can be defined in the data base. It is produced as the projection of the join of the four tables shown in Fig. 7.21. and contains the following data about all the employees:

- social security number,
- last name,
- first name,
- section name,
- task name, and
- percentage of job devoted to the task.

The new relation is created from the existing ones by the following operations:

```
ANSWER1 ≡ [(EMPLOYEES JOIN SECTIONS (S-CODE))
JOIN WHAT-DOES-HE-DO? (EMP-SSN)]
JOIN TASKS (TASK-CODE))
PROJECT [EMP-SSN, FIRST-NAME, LAST-NAME, TASK, PERCENT]
```

In the view there will be one row per employee per task.

Knowing the column definitions of the ANSWER1 view, the question can be simplified thus:

'Give the rows of ANSWER1 identified by the given employee.'

In order to answer the query, the user need not know how the ANSWER1 view was produced from the original tables and by which paths the system accesses them.

Besides this, views are often created for data-protection and security reasons. If a user is not permitted to work with every row/column of a relation, then a view is created by restriction/projection containing only that portion of it which can be processed by him. The user will receive access authority not for the relation, but only for that view.

In summary, the steps of developing and creating a relational data base are:

1. Define which data will be needed.
2. Define the entities, arrange the data belonging together.
3. Define the relationships between the entities.
4. Decompose all many to many relationships by defining new relations.
5. Create the relations, arrange data (at least) into the first normalized form.
6. Define foreign keys to ensure data integrity among the dependent relations.

7. Perform the necessary higher level normalizations.
8. Create the tables of the normalized relations.
9. Define the relations which combine the data of the expected tasks.
10. Create constant views from the so-obtained relations.
11. Enter data, work with them.
12. If necessary, modify the data base structure or create new tables and views, according to points 1–11.

Besides the above, naturally, it is necessary to ensure data protection, grant access authorities, and enforce them. This will be detailed in Section 12.2.

7.10. Catalogue / Data Dictionary

7.10.1. The Purpose of the Catalogue

The data base must contain, together with the data, their definitions and the definition of their use and organization. This independent part of the data base is called the *catalogue* or the *data dictionary*. In the relational model the catalogue itself also consists of several relations which in principle can be used just as the tables containing the data and the views created from them.

The catalogue is composed of several tables and views containing the definition of every object of the system. If anything changes in any object, the system automatically modifies the catalogue relations keeping track of the data of the changed object. Thus, for example, if a new table is created, then its characteristics will be stored automatically in the catalogue table describing the tables. Likewise the characteristics of the columns of the new table will be stored automatically in the catalogue table keeping track of all the columns of the data base.

What information is in the directory, in what relations, and in what forms, is highly system-dependent. So much so that within the IBM Standard Applications Architecture (SAA) the DB2, SQL/DS and SQL/400 systems each contain different catalogue tables, store different information about the same objects, and what is more, for example, name differently the catalogue relations containing the data of the most important objects, those of the relations (SYSIBM.SYSTABLES, SYSTEM.SYSCATALOG, SYSTEM.SYSTABLES). That is exactly why instead of a definite system, only the general features of the most important catalogue relations will be discussed below. Note that the catalogue of large systems may contain as many as 20–30 different tables.

Most of the catalogue tables can generally be read without restriction by anyone. However, there are limits upon other operations. So, for example, it is not possible to:

- drop a catalogue relation,
- explicitly lock it from other users, or
- insert, modify, or delete rows in it with a direct command.

The rows of the catalogue are automatically updated by the system. If the user or data base administrator wants to explicitly accomplish a modification in it for which he possesses authority, then there are special catalogue handling commands for this.

7.10.2. Catalogue Tables

Catalogue tables can be classified into four main groups, depending upon the data they contain and their subject:

- catalogue tables about the system objects and their constituent parts, respectively,
- catalogue tables about the access authorities of the individual objects,
- catalogue tables about the dependencies among the various system objects and elements, and
- catalogue tables containing general information about the system.

7.10.3. Object-Oriented Catalogues

From the information related to objects, the following are generally stored in independent tables:

Table Catalogue

This contains one row about each table and view of the system (including also the catalogue tables). Its most important columns are:

- the table name,
- the table creator or its owner, respectively,²⁰
- whether it also contains physically the data, or is only a virtual relation (view),
- the number of columns,
- the number of rows,
- the number of pages it occupies,
- where it can be found in the data base,
- how many tables it depends upon, and
- how many tables depend upon it.

Besides the above, for example, it may also contain information about the keys, the number of records which overflowed the original pages, the average record length, clustering, and a remark about the function of the table.

Column Catalogue

It contains one row about each column of every table and view. Its most important columns are the following:

- the column name,
- the table/view name that contains the column,
- the creator of the table/view,
- the column serial number in the table/view,
- what data type it contains (character, integer, decimal, date, etc.).

²⁰ This is generally the user identifier used to login at the computer or to connect to the database management system. In such systems where the user does not possess the authority to physically create a table, the data base administrator performs this task instead of him. In that case, the creator of the table is actually the data base administrator who grants the ownership, and together with this, all creator rights that are equivalent to the former, to that user at whose request the (empty) table was formed.

- the length of the column, and
- can it contain null value or not.

In addition to the above, it may also contain a remark about the function of the column, as well as such data needed for the evaluation of the system's efficiency, for example, the number of different values in the given columns, whether it clusters, is an index built on it, etc.

Index and Key Catalogues

They contain one row about every index and key. For example, the most important columns of the index catalogue table are the following:

- the index name,
- the creator of the index,
- the table name upon which the index is based,
- the columns' names upon which the index is based,
- the index type (unique, ascending, primary, etc.)
- whether it clusters, and, if yes, to what degree, and
- the number of different index values.

With composite indices, the key catalogue gives similar information about the individual columns of the index as well.

Other Catalogues

Additional catalogues may contain:

- view definitions,
- the logical division of the data base into parts,
- the compiled and stored access paths of the user's programs, and
- the synonym dictionary enabling every user to refer to various system objects using names defined by him.

Bear in mind that not every system contains all the catalogue tables listed above with the content detailed here. However, on the other hand, there are systems that supply more detailed information about objects than the above.

The same remark applies to all of the following catalogue types as well.

7.10.4. Catalogue about Access Authorities

When a data base is accessible by several users it is indispensable to define exactly who is authorized to do what. For example, 'Who may create or delete tables?', 'Who may read, write, delete, or update data in a given relation?', 'Who may refer to a certain table, and how?', etc.

The general principle is that he who creates a relation (also usually called the relation owner) may do anything with it, including granting these privileges to or revoking them from anyone, respectively (see the GRANT and REVOKE commands in Sections 12.2. and 12.3.). That is precisely why object describing catalogues always contain the creator of the given object also.

However, it is necessary for the system to store this information in a readily-accessible manner in order to check at run time the privileges and authorities for each data base query.

The following catalogues serve this purpose:

General Authority

The list of those authorized to use the data base. This generally contains special data base passwords as well; therefore it can be accessed only by the data base administrator. The users usually may look at only a view of it which excludes the password.

If not everyone is allowed to create or delete independent objects, then this table's columns show what the given user is authorized to do.

Privileges on Tables/Views/Columns

It lists all the privileges of the individual users on all tables, views, or — within this if the privilege is detailed down to the column level — then on the column, respectively.

The most important columns of this table are:

- the userid of the privilege grantor,
- the userid of the grantee,
- the type of the object,
- the name of the object,
- the date of granting the privilege, and
- what operation is allowed.

The latter can separately be a privilege to read, insert, update, or delete data, to create indices, or to refer with a foreign key.

In most systems, there is a special user identifier (PUBLIC). If this is entered in the catalogue, then the operations it authorizes can be executed on the objects by anyone at all authorized to use the data base.

The Authority to Use Application Programs

If the data base management commands are given not interactively, but built into application programs, then the compiled programs contain the data base access commands as independent modules. If any user was able to link them with his own program and run it, then data base access would not be controllable. That is precisely why an independent catalogue table contains the most important features of these data base access modules, as well as which users are authorized to run them.²¹

7.10.5. Dependency Relations

This catalogue table contains the dependencies of the data base views, indices, and programs on the tables. It is important since, for example, by dropping a table all the views based upon it will automatically be dropped too. As a corollary, users who could previously work only with the view, and were not even

²¹ The running authority in itself is not sufficient. It is also necessary to receive permission for the data base operation to be executed in the module (for example, to delete rows in a table). If it is not so, then an error will arise during the run of the program since the system does not permit unauthorized operation.

allowed to access the table, will not be able to work further as well. Similarly, also those programs seeking to use a deleted table or a view based upon it will be crippled.

If a view is based upon or a program works with several tables, then the dependency is marked by one separate row in the catalogue for each of them.

The most important columns of the table are the following:

- the name of the base object,
- the creator of the base object,
- the type of the base object (view, table, index, etc.),
- the name of the dependent object,
- the type of the dependent object (view, program), and
- the creation date of the dependency.

Here is some practical advice — before dropping an object, first refer to this catalogue to learn the consequences.²²

7.10.6. General System Information

It defines, among other things, which character set may be used, date and time representation (for example, 12-31-1993 or 31.12.1993; 1:30 PM, etc.), in what language the system communicates with the user, etc.

These options are defined by the data base administrator, usually when installing or starting the system, and they apply to all users of the system (as a default value).

7.10.7. Catalogue in Distributed Data Bases

In addition to the services provided by the catalogue of a centralized system, the catalogue of a distributed data base has also to supply information about:

- where the individual records are located physically,
- if a logical record has to be collected from several locations, then where its components can be found, and
- what data are stored in duplicates in the system, where they can be found, and how their consistency must be ensured.

These can be obtained from properly-defined new catalogue tables taking into account the requirements of distributed systems as well. In practice, there arises a more serious problem, namely, where to store the catalogue itself. Basically, one of the following options can be chosen:

1. A central, centralized catalogue.
2. Several copies of the entire catalogue are stored in the system. Every independent processing site has its own complete duplicate.

²² There are database management systems which, for example, do not even allow the drop of a table as long as there exists any view in the data base referring to it. (For example, the database management system of the IBM AS/400 is like this.)

3. Partitioned. Every independent site keeps track of only those objects stored there. The entire catalogue is the union of the catalogues of the individual sites.
4. Every site monitors its own objects, and, in addition, a complete central register can also be found at a certain site (the combination of 1 and 3).
5. A unified name convention and catalogue content is to be used within the system.

Each solution has its advantages and disadvantages.

If finally on the basis of various analyses, a distributed data base is chosen, then most of its advantages will be lost if the first solution is selected. Due to the central catalogue, individual sites will depend considerably upon the center in the future, too. However, this dependence will be mitigated since the regular updating of the catalogue does not depend upon the center's pleasure; it is automatic, but the physical dependency will remain completely.

The second approach is nearly unrealizable with rapidly changing systems, unless data are allowed to be inconsistent for a period of time. The simultaneous updating of all catalogue copies would necessitate such complicated procedures that practical work would be paralysed.

The third method is advantageous only when operations concern almost solely the local data base. If data stored at another site are needed, then, in the worst case, to find them, all the system catalogues will have to be searched (and based on Murphy's Law, almost every case will be like this).

The fourth method eliminates this disadvantage, but restores once more the dependency upon the central site.

In practice, the fifth method promises the best results, if the following rules are observed:

- a) every object name is complemented, qualified by
 - the identifier of the creator,
 - the location of its creation, and
 - its first storage location.
- b) in every local directory are stored
 - the synonyms of the objects used there,
 - the definition of the objects stored there, and
 - the momentary storage location of the objects created there.

When moving objects from one location to another, these data are updated automatically by the system.

When referring to any object, it can be ascertained from the qualification (or if it is not qualified, then the system searches for it locally and either finds it there or complements it on the basis of the synonym dictionary) where the object was created.

First, the system also searches for it there, and if it can no longer be found, then the system will learn the momentary storage location from the local catalogue.

It is evident that any object can be found by this method, at worst, in two steps. The distribution of the data base remains. Only the synonym catalogue containing the complete names of the objects has to be stored redundantly. Its update has to be carried out by the system automatically.

7.11. Summary of the Relational Model

Although it was already detailed in the foregoing, the basic requirements of the relational model are summarized again. The reason for this is that all up-to-date data base management systems are based on the relational model, and it is expected in the following decade that they will follow increasingly the requirements set out by the model.

The relational model is based upon three pillars. They are as follows:

- realization of the relational data structure,
- ensuring data integrity, and
- data manipulation options.

The relational data structure means that every entity type is defined in a relation (table) of the N th degree whose individual occurrences (rows) are composed of attribute values (columns) based upon elementary fields (domains). The primary key serves their unambiguous identification within the relation.

Within the data base every element can be identified unambiguously by the relation name and by the value of its primary key.

To create or modify the data structure, it is necessary to have a data definition language. This is used to create the base relations (tables) and the virtual relations derived from them by relational operations, respectively.

Data integrity is realized by primary and foreign keys. The primary key value is always unique within a relation, and none of its components can be the null value.

The values of foreign keys must agree with a primary key value of the parent relation or be equal to null.

Using the operations of relational algebra or calculus, data manipulation enables data selection for creating, inserting, reading, updating, or deleting data from the relations.

Systems satisfying the above basic requirements in every aspect, in a way that will be detailed below, are called *complete relational systems*. Such a system — to the best knowledge of the author — has not yet been marketed at the time of writing this book.

Systems which represent data using tables (and only so), and allow, at a minimum, restriction, projection, and natural join without the user being required to define a data access path, are called *minimal relational systems*. Most of the inexpensive relational systems written for PC's are minimal relational systems. However, for example, spreadsheet softwares which are very widespread and popular in practice, cannot be considered as relational systems — even when they are advertised as including database management facilities — since they do not support join, the most important relational operation.

Systems complete from the relational aspect contain independent commands to execute the relational operations detailed in Section 7.6., or most of them, and they generally ensure data integrity to some degree. Among the better known systems on the market are, for example, DB2, SQL/DS, SQL/400, INGRES, ORACLE, etc.

Based on current criteria, *complete relational systems* have to satisfy all of the following requirements:

1. They store every information in relations and only in relations in the data base.
2. All information is accessible unambiguously by specifying the name of the appropriate relation and within this, the name of the given column and value of the row's primary key.
3. Null values are handled by the system unambiguously and properly.
4. They include an always up-to-date, accessible catalogue.
5. They include an interactive language, as well as a version which can be embedded into high-level languages as well; this language enables
 - defining data and relations,
 - executing relational operations,
 - defining data protection and integrity conditions, and
 - handling complex transactions uniformly (see the COMMIT and ROLL-BACK commands in Section 12.5.).
6. Tables can be modified through views if that can be handled unambiguously and without contradiction.
7. They include operations which facilitate inserting, deleting, and updating groups of data.
8. They ensure physical data independence.
9. They ensure logical data independence.
10. Integrity conditions are contained in the data base. Their enforcement is application-independent. The conditions can be modified, but revised conditions must also be valid for all data already present in the data base.
11. The above rules remain in force whether the data base is centralized or distributed.
12. Integrity and security constraints must also remain in force when it is possible to access directly individual records at a lower level.

The above-mentioned systems which are complete from the relational aspect satisfy already a significant portion of these conditions, but not yet each one of them or not entirely. Nevertheless, for an everyday, typical user they can be considered in practical terms as already complete (or nearly so) relational systems.

via the operations executed on the relations. In text retrieval systems the records obtained in response to a query agree with those records (or a certain portion of them, respectively) whose fields partly or entirely contain (or do not contain) the character strings and the logical relations among them defined at the time of search. A typical example of this is when articles and books of interest are to be selected *based upon their summary* which has no additional keywords for search. Let's look, for example, at books and papers in whose synopses the words 'popular' or 'educational' and, additionally, the word 'computer sciences' (exact definition of the character string) appear, or such words beginning with either of these character strings 'popular ...' or 'educational ...' and the character string 'computer sciences ...'. In the former case works where '...' which makes popularly known the newest achievements of computer science occurs will not appear among the resulting records, but in the latter case they will.

In text retrieval systems the search and checking can be performed with the serial processing of the individual documents. (Most word processors operate in

Text Retrieval Systems

Text retrieval systems form a special category of data bases. Information is stored in them as continuous or partly-distributed text in a non- or, in certain aspects, partly-formatted way. In a narrower sense, these systems are not considered as data bases in data base terminology, or are supplemented with the adjective 'document format'. The software for their use is also named differently — text retrieval system.

However, in practice, classical fact database management systems, the systems which contain formatted data, and systems which handle document formats, overlap frequently. Almost every up-to-date relational data base management system includes a feature that enables retrieval based upon the agreement of a character string in the text fields. Using this feature — although with less efficiency, fewer facilities, and less convenience — fact databases can be operated as text retrieval systems.

Therefore it is worthwhile to summarize the most important facilities and features of text retrieval systems.

In (fact) databases all of the records obtained in response to a given query can be composed of data of various relations or data of several various record types collected by the data base management system through the set relationships or via the operations executed on the relations. In text retrieval systems the records obtained in response to a query agree with those records (or a certain portion of them, respectively) whose fields partly or entirely contain (or do not contain) the character strings and the logical relations among them defined at the time of search. A typical example of this is when articles and books of interest are to be selected *based upon their summary* which has no additional keywords for search. Let's look, for example, at books and papers in whose synopses the words 'popular' or 'educational' and, additionally, the word 'computer sciences' (exact definition of the character string) appear, or such words beginning with either of these character strings 'popular ...' or 'educational ...' and the character string 'computer sciences ...'. In the former case works where '...' which makes popularly known the newest achievements of computer science' occurs will not appear among the resulting records, but in the latter case they will.

In text retrieval systems the search and checking can be performed with the serial processing of the individual documents. (Most word processors operate in

such a way.) This requires a truly long time for large data volumes. However, it can be carried out also via indices built on each word of the text or on those words of the material that were marked in advance at the time of data entry. The storage requirement will then be enlarged by the size of the index tables, in practice — depending upon the number of indexed words — generally about two or three times. The entry and modification of the documents takes an essentially longer time, but the retrieval will be much faster than in systems which retrieve with serial search.

Figure 8.1. shows a simplified version of a part of a retrieval system containing television news. The documents consist of the following fields:

1. Videotape identifier (Kxx/xxx)
2. Event time (MM/DD/YY)
3. Document identifier in the computer system, in the data base (ISIS: xxxx)
4. Brief description of content
5. Place of event
6. Subject catalogue number
7. Remark (a more detailed description, etc., the figure does not show it.)

The materials seen in the figure were supplied by the system as a result of searching for the word 'armament' in the 1985 news.

A part of the index table pointing to the listed documents is shown in Fig. 8.2. The system constructs an inverted file on each word of the fields marked for this, except for some special words of the short content description. (the, that, against, and, This itemized list is stored in the so-called STOPWORD FILE.)

The keys of the index table contain those words of the document that can be used for search. Further entries contain those document identifiers (addresses) in the data base and within them, the serial number of the fields in the record that contain that key value. In certain systems the relative location of the field key also appears; this enables, mainly in case of complex search conditions, the differentiation and selection of those texts where the relative position of the searched character strings is significant. (For example, when searching for a document containing a reference to the person named John Paul, the search keys are 'John' and 'Paul', but the order of their occurrences is critical; materials referring to Paul John are not needed.)¹ In the interest of faster search and establishing logical relationships among the individual search conditions, both the keywords and, within individual keywords, multiple document identifiers, are sorted. The document selection consists of searching for their addresses from the index table on the basis of the keywords and then their retrieval from those addresses. The latter may involve the complete document or a part of it. The format may be defined by the system.

Thus, for example, in the example shown in the previous figures, the system selects the 1985 video materials dealing with ARMAMENT in any form (only these were shown in the figure). It selects from the index table the identifiers of the records having a date field ending with 1985 and also including the word ARMAMENT in their content description. The common part of the two sets pro-

¹ Naturally, if the index was built not on the individual words, but on the whole name, then the above problem would not arise.

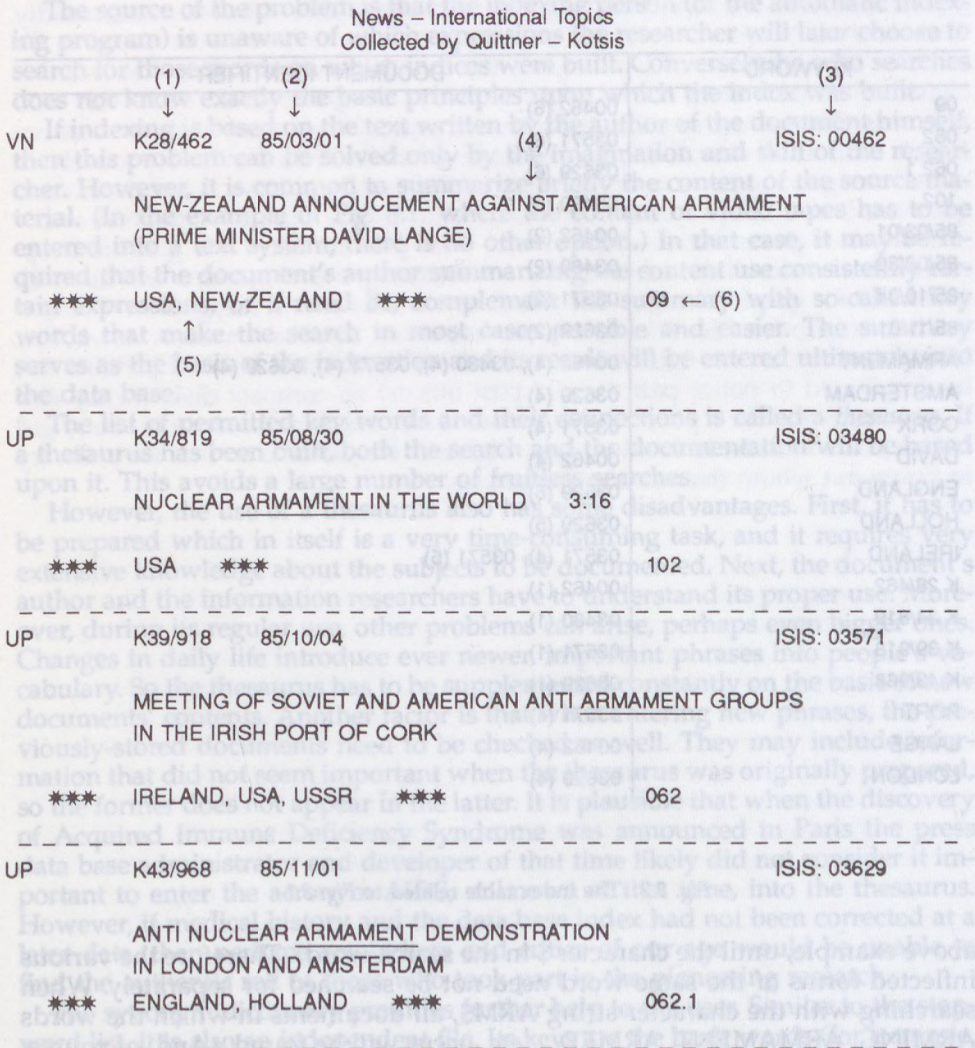


Fig. 8.1. A simplified version of a news retrieval system for television news.
The items seen in the figure were provided by the system from the 1985 material
in response to the 'ARMAMENT' word

duces the individual identifiers of those documents in the system — from which their addresses can be deduced — that satisfy both conditions. Naturally, for meaningful work to be possible, it is necessary that the searched text (mainly the brief content description) should be suitable, not only from a technical aspect, but also usable for computer processing. This is aided by truncated search. The keyword is compared only until the appearance of a special character (in the

KEYWORD	DOCUMENT IDENTIFIER
09	00462 (6)
062	03571 (6)
062.1	03629 (6)
102	03480 (6)
85/03/01	00462 (2)
85/08/30	03480 (2)
85/10/04	03571 (2)
85/11/01	03629 (2)
ARMAMENT	00462 (4), 03480 (4), 03571 (4), 03629 (4)
AMSTERDAM	03629 (4)
CORK	03571 (4)
DAVID	00462 (4)
ENGLAND	03629 (5)
HOLLAND	03629 (5)
IRELAND	03571 (4), 03571 (5)
K 28/462	00462 (1)
K 34/819	03480 (1)
K 39/918	03571 (1)
K 43/968	03629 (1)
PORT	03571 (4)
LANGE	00462 (4)
LONDON	03629 (4)

Fig. 8.2. The index table related to Fig. 8.1

above example, until the character \$ in the search word). Therefore the various inflected forms of the same word need not be searched for separately. When searching with the character string ARM\$, all documents in which the words ARMING, ARMAMENT, ARMED ... etc. occur will be found. (And, of course, IBM Headquarters ARMONK, and in a German text ARM meaning poor, also.)

Key words must be selected carefully in order to avoid problems during search. It can happen that a chemist is interested in the specialized literature dealing with certain attributes of sodium, but this word may not occur in the given article. The author writes instead, for example, about the alkali metals. If this phrase is not given as an alternative search word, then this article will not be among those selected.

Likewise, a physicist needs to know that instead of the phrase 'nuclear bomb', the words 'atomic bomb', 'hydrogen bomb', or simply, 'A-bomb', etc. are frequently used. A carefully-formulated query should include all of the above words to initiate search for all of them.

The source of the problem is that the indexing person (or the automatic indexing program) is unaware of which expressions the researcher will later choose to search for those words on which indices were built. Conversely, he who searches does not know exactly the basic principles upon which the index was built.

If indexing is based on the text written by the author of the document himself, then this problem can be solved only by the imagination and skill of the researcher. However, it is common to summarize briefly the content of the source material. (In the example of Fig. 8.1. where the content of video tapes has to be entered into a text system, there is no other option.) In that case, it may be required that the document's author summarizing the content use consistently certain expressions, or if need be, complement the summary with so-called key words that make the search in most cases possible and easier. The summary serves as the basis of the indexation and its result will be entered ultimately into the data base.

The list of permitted key words and their connections is called a *thesaurus*. If a thesaurus has been built, both the search and the documentation will be based upon it. This avoids a large number of fruitless searches.

However, the use of a thesaurus also has some disadvantages. First, it has to be prepared which in itself is a very time-consuming task, and it requires very extensive knowledge about the subjects to be documented. Next, the document's author and the information researchers have to understand its proper use. Moreover, during its regular use, other problems can arise, perhaps even bigger ones. Changes in daily life introduce ever newer, important phrases into people's vocabulary. So the thesaurus has to be supplemented constantly on the basis of new documents' contents. Another factor is that when entering new phrases, the previously-stored documents need to be checked as well. They may include information that did not seem important when the thesaurus was originally prepared, so the former does not appear in the latter. It is plausible that when the discovery of Acquired Immune Deficiency Syndrome was announced in Paris the press data base administrator and developer of that time likely did not consider it important to enter the acronym AIDS, unknown at that time, into the thesaurus. However, if medical history and the data base index had not been corrected at a later date, then perhaps researchers and critics of our age would be unable to find the authentic roll of those who took part in the pioneering research.

The synonym dictionary provides further help to the user. Similar to the stop-word list, it is also an independent file. Its keys are the basic words (for example, BENELUX); the other field(s) is that word that substitutes for or is equivalent to it (for example, BELGIUM, NETHERLANDS, HOLLAND, or LUXEMBOURG). If the search is to be extended to all possible substitutes or synonyms of a given word, then the system is so instructed by giving a special expression (in this example, ANY). In this case, the indices that belong to each synonym of the word acting as a key in the index file belonging to all the documents are automatically searched. The materials matching the selection conditions will be the total of the items listed there (logical OR).

To maintain balance between documentation and searching, the most important factor is to choose the proper *hit ratio*. This means that no documents containing valuable information should be overlooked and not too many irrelevant

documents from the aspect of the given query should be selected unnecessarily. This measure or criterion consists of two parts. Both have to be compared to the number of records actually satisfying the search conditions. The first member characterizes how many records were found, as compared to the effectively suitable ones. The second member indicates the ratio of how many of them were not found. In an ideal case, the values of these quotients are 1 and 0, respectively. In a well-designed system, a skilled user can generally achieve 0 for the second quotient, even if, in practice, he scores a value greater than 1 for the first one.

Modern high resolution displays and laser printers allow the graphical representation of stored data. In the near future, text retrieval systems will likely be expanded for graphical (or video) search or even for sound recognition. Computerized fingerprint identification is an important application even nowadays. Maybe it is not too far-fetched that in the future pictures of wanted persons will be distributed to police patrols (and post offices) on compact disk. It may also become common for a proud father, rather than parading photos of his baby girl before jaded friends, to hand out her digitized bit patterns so that they can try to figure out whom the newborn resembles.

III.

THE SQL LANGUAGE

to the SQL Language

9.1. SQL as a Standard

SQL is an abbreviation for Structured Query Language. However, in fact it means much more; it is an entire relational data definition and manipulation language which can be viewed as a standard or nearly so in practice.

The development and wide acceptance of SQL is one of those rare cases in computer technology where IBM's market and application-oriented developers and the computer science experts started in the same direction and came essentially to the same conclusion. Just as with the chicken and the egg, it would be very difficult to judge whether SQL's theoretical foundations helped IBM's SQL systems to gain acceptance as a quasi-standard, or IBM's weight forced the theoretical and standard-setting experts to consider its system as fundamental to the development of and interface to other systems. However, not this is important, but rather the heartening fact that simultaneous with the commercialization of relational database management systems, a language became available which was accepted by most developers and users. All others were therefore compelled in their own self-interest to accommodate to it. Naturally, there are differences among different firms' SQL products. Even, for example, SQL systems developed by IBM for various operating systems (DB2 for MVS; SQL/DS for VM and VSE; SQL/400 for OS/400, as well as OS/2 EDM for the OS/2 environment), 'speak in different dialects'. Their command sets differ and even identical commands have different features. Especially for this reason, in this section, the exact syntax of any particular SQL implementation is not provided, but rather the language's basic elements already realized in almost all versions on the market. These will be complemented by some very important additional features used and applied by the most advanced systems.

In a formal sense, SQL is close to relational calculus and nonprocedural languages. In essence, the user says only what he or she wants. How this is solved, that is, the concrete steps needed to realize his/her request, is defined by the language's command interpreter, taking into account the data base structure. To define the problem-solving algorithm, good SQL systems consider not only the logical and the physical data structures, but also the actual data content and data

General Introduction to the SQL Language

9.1. SQL as a Standard

SQL is an abbreviation for Structured Query Language. However, in fact it means much more; it is an entire relational data definition and manipulation language which can be viewed as a standard or nearly so in practice.

The development and wide acceptance of SQL is one of those rare cases in computer technology where IBM's market and application-oriented developers and the computer science experts started in the same direction and came essentially to the same conclusion. Just as with the chicken and the egg, it would be very difficult to judge whether SQL's theoretical foundations helped IBM's SQL systems to gain acceptance as a quasi-standard, or IBM's weight forced the theoretical and standard-setting experts to consider its system as fundamental to the development of and interface to other systems. However, not this is important, but rather the heartening fact that simultaneous with the commercialization of relational database management systems, a language became available which was accepted by most developers and users. All others were therefore compelled in their own self-interest to accommodate to it. Naturally, there are differences among different firms' SQL products. Even, for example, SQL systems developed by IBM for various operating systems (DB2 for MVS; SQL/DS for VM and VSE, SQL/400 for OS/400, as well as OS/2 EEDM for the OS/2 environment), 'speak in different dialects'. Their command sets differ and even identical commands have different features. Especially for this reason, in this section, the exact syntax of any particular SQL implementation is not provided, but rather the language's basic elements already realized in almost all versions on the market. These will be complemented by some very important additional features used and applied by the most advanced systems.

In a formal sense, SQL is close to relational calculus and nonprocedural languages. In essence, the user says only what *he or she wants*. How this is solved, to wit, the concrete steps needed to realize his/her request, is defined by the language's command interpreter, taking into account the data base structure. To define the problem-solving algorithm, good SQL systems consider not only the logical and the physical data structures, but also the actual data content and data

volume in order to optimize access time. The system catalogue provides the necessary information.

9.2. Syntax Notation

To become acquainted with a language, the syntax of its commands must be known. These can be defined in a meta language in Bachus–Naur Form. In this book the general features of the SQL language are introduced, rather than citing a particular user manual for a given system. It has therefore been considered expedient to use lucid graphical syntax rather than the exact, but complicated BNF notation. In practice, the precision of the former is sufficient. In addition, if the reader would like to work with a commercial data base management system, then it is very likely that he/she will encounter these syntax notations in the reference books and user's guides.

9.2.1. Syntax Notation

The syntax notation differentiates among:

- SQL keywords, written in the syntax diagram with capital letters¹
E.g., DELETE
SQL keywords must be written as given in the syntax; if they are not or incorrectly given, then the SQL interpreter indicates a syntax error. SQL keywords can be used only for the given purpose.
- Names of objects defined by the user or data base administrator, for example, table or column names. There are various language-dependent rules about how to form these names, (e.g., maximal length, allowable characters). They are indicated with lower case letters in the syntax notation, E.g., table_1
- Arithmetic and logical operators
(+, -, *, /, **, >, <, >=, <=, <>).
should be considered as reserved words. They can be used only in this meaning and specific form.
- The additional specific characters of the syntax diagram (., ; etc.) must be written exactly in that way and at the precise location in the command as shown in the diagram.
- The mandatory and optional clauses must be given in the same order as in the syntax.

¹ It depends upon the SQL interpreter whether in actual commands both lower case and upper case letters can be used. If so, they can be considered as identical or different. Or except for character strings serving as operands (literals), everything must be written with upper case in the commands. Differentiation in the use of lower and upper case letters shown in this section is valid only in the command syntax definition in the formal description.

9.2.2. Interpretation of the Diagram

The commands are read from top to bottom; along the lines (route) from left to right; from the end of a line downward, starting from the beginning of the next line.

Some special symbols:

- ▶▶—— beginning of the command
- ▶ the command continues in the next line
- ▶—— continuation of the previous line
- ▶▶ end of the command.

In those cases when only the independent clause of a command is written, the symbols ▶—— and ——▶ are used, but inside the complete commands they are not noted in that way.

Commands are written as follows:

- Mandatory elements are written *on* the main route.

▶▶—— COMMAND —— mandatory element ——▶▶

- Optional elements are written *under* the main route.

▶▶—— COMMAND optional element ——▶▶

- If there is a choice from among several elements, then they are all written below each other.

- If exactly one must be chosen

▶▶—— COMMAND option-1
option-2 ——▶▶

- If among one or more choices, it is possible that none need be chosen, then

▶▶—— COMMAND option-1
option-2 ——▶▶

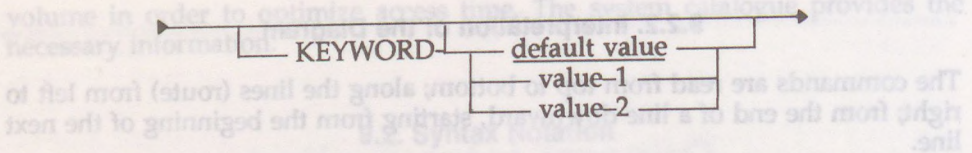
- The notation of repetitive elements is

▶▶—— COMMAND repeatable element ——▶▶

▶▶—— COMMAND repeatable option-1
repeatable option-2 rep.op.-3
rep.op.-4 ——▶▶

In the first example, the repeatable element must be given at least once; in the second example, option-1 and option-2 can be ignored, while between option-3 and option-4, one of them must occur at least once.

If a clause is fully ignored in an option which allows only one value among several and its value must be defined, then this default value is underlined.



So 'KEYWORD default value' need not be given. If neither 'KEYWORD value-1' nor 'KEYWORD value-2' is written in the command, then it is tantamount to giving the default value.

The general command form is shown in Fig. 9.1.

Some comments about interpreting the syntax diagram:

1. The command name. It is a mandatory keyword and must be written exactly so.
2. Optional keywords. None or at most one of them can be given. When given, it must be written exactly so.

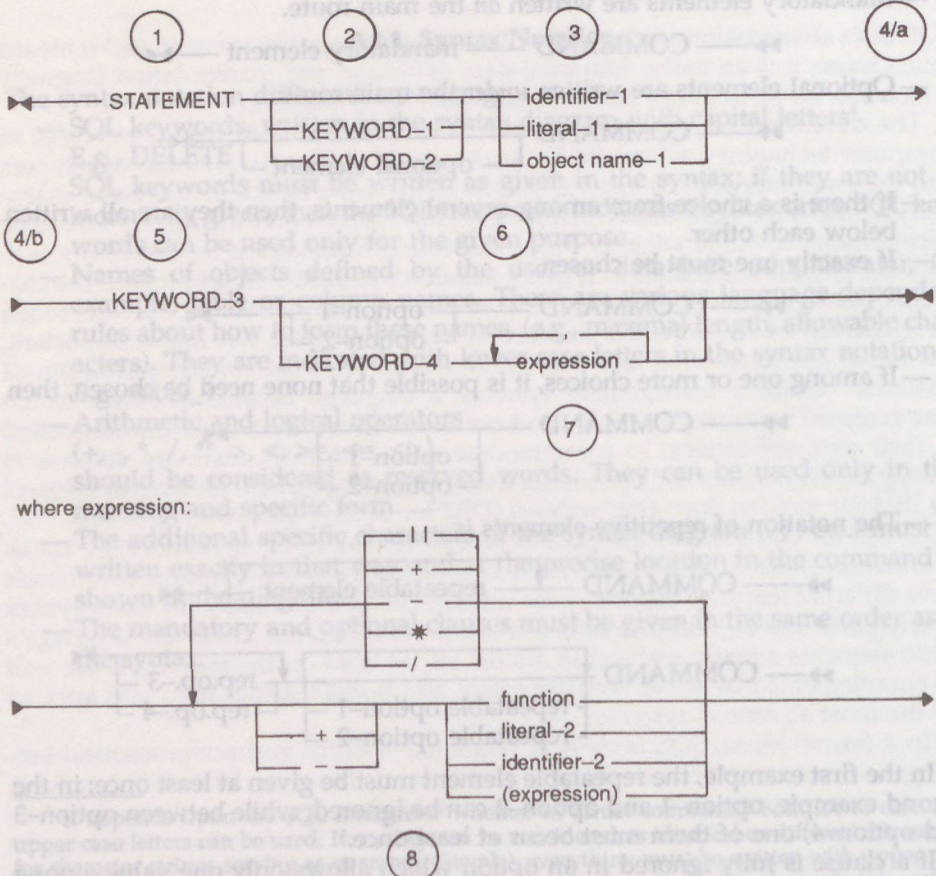


Fig. 9.1. General definition of an SQL command

3. Mandatory operand. Exactly one must be given. It can be a user-selected identifier, a literal, or an object name.
4. a/4. b The command continues on the following line.
5. Mandatory keyword. It must be written exactly so.
6. The entire clause can be omitted. If it is given, then its format is determined by the auxiliary path under the main route. In this case, KEYWORD-4 is mandatory.
7. If clause 6 is given, then after KEYWORD-4 the 'expression' must be given at least once and can be repeated any number of times. Repeated expressions are separated by a comma.
8. The expression's detailed explanation. For brevity, it is shown in a separate (sub)diagram.²

9.3. Identification of SQL Commands

SQL commands can be divided into four main groups:

- data definition commands,
- data manipulation commands,
- control commands, and
- other commands.

The *data definition commands* define and modify the data base structure. They serve to assign the data base areas and create, modify, and drop tables, views, and indices.

The most important among them are the

CREATE,
DROP, and
ALTER

type commands.

SQL's data definition commands (analogous to the Data Definition Language) will be introduced in Chapter 10.

The *data manipulation commands* select, modify, or delete existing rows from existing tables or views, or write new rows there, respectively.

The most important among them are the

SELECT,
DELETE,
UPDATE, and
INSERT

type commands.

² The above example actually springs from the guidebook of a well-known system and illustrates the imprecise definition of this notation. Rigidly interpreting the syntax would allow in one expression, e.g., even two + signs after one another, which obviously does not make too much sense. In these cases (mistakenly, but hopefully not without grounds), it is entrusted to the user's common sense that the meaningless and solely formalistic variations will be filtered out (in the worst case, after understanding the error message issued by the system).

These are supplemented by further commands necessary for embedding them into the host program language.

SQL's data manipulation commands (analogous to the Data Manipulation Language) will be introduced in Chapter 11.

Control commands enable the use of different SQL objects, ensuring the consistent execution of multi-step command groups or the restoration of the original condition in case of any error.

The most important control commands are

GRANT (give authority),
REVOKE (withdraw authority),
COMMIT (confirm command results),
ROLLBACK (restore original condition),
LOCK (lock data base objects),
WHENEVER (error-handling in program),
SET (system parameters setup)

and type commands.

SQL control commands will be introduced in Chapter 12.

The *other commands* are heavily system-dependent. They include, for example, commands that prepare and execute other SQL commands in a host program, store and execute other interactive SQL commands, clarify the data base description or insert remarks into the catalogue, as well as operator commands that provide information about the state of the system, but which can also be issued from SQL, and the EXPLAIN command which supplies information about the actual access paths.

These will be discussed shortly in Chapter 13.

Data Definition Commands

10.1. General Characteristics

With the help of SQL data definition commands

- tables,
- views, and
- indices on the tables

can be created (CREATE TABLE/VIEW/INDEX), deleted (DROP), or modified (ALTER).

In addition, depending upon where the data base is physically located on the disk, and how independently its logical and physical parts can be handled, it may be possible to define table areas (TABLESPACE), data base (sub)areas (DBSPACE), and collections (COLLECTION) as well.

While all SQL-based database management systems¹ include commands to create or drop tables, views, and indices, many systems do not allow their modification.

The definition of the smaller, independent parts of the data base and their storage allocation are very strongly system-dependent. Therefore this book does not deal with these topics. In general, these commands can be given only by the data base administrator.

Only authorized users can create, drop, or modify any object. The data base administrator possesses always all of these authorities. In addition, some users can obtain explicit permission for a given activity. This can be granted to them by name, due to membership in a given group, or as part of the entire user's group (PUBLIC), or they may possess total authority over a defined part of the data base.

Generally, the creator of an object is authorized to drop or modify it. He can also define views, indices, primary and foreign keys on it. Other users — except the data base administrator — can perform these operations only if the object's owner permits them (either granted to them by name or to everyone as a member of the group PUBLIC).

¹ Further, for brevity, they will be called simply SQL system or SQL systems.

10.2. Table Creation (CREATE TABLE Command)

To create a table, its name and the exact definition of its columns must be given. The area where it should be stored can be specified also, as well as who must be considered as its creator, that is to say, who will be its owner. (In general, the creator and the owner are the same, but — as already mentioned — the data base administrator can create tables for other users, too.) This is significant particularly when the table creation right is limited. For those lacking this authority, the data base administrator must create the tables, based, naturally, upon their logical definition. Primary and foreign keys can also be defined during creation. If optional parameters are not given in the CREATE TABLE command, then the system uses automatically rational default options. (E.g., the owner of the object will be its creator; there are no foreign keys, etc.)²

The general form of the table creating command can be seen in Fig. 10.1.

The name of the command is CREATE TABLE. The table's name must be specified and unique throughout the entire data base. In multi-user systems, the table's name consists of two parts. The first part is the table owner's name. This is the user who created the table or in whose name — typically when he lacked authority — the data base administrator completed it. If the creator's name is not given in the command, then the system will assume by default that the table's creator / owner will be the issuer of the CREATE TABLE command. The second component of the name is that name given by the creator. It follows from the above, that any number of users can have, e.g., a table named CLASSES, but each of them can have just one.

The table's column names can also be freely chosen by the user. He should be mindful only not to use identical column names within the same table. Along with the column name, how data are represented in the columns has to be specified.

Every respectable data base management system supports the following data types. (In the commands they can usually be abbreviated.)

- CHARACTER (*n*); a character string consisting of exactly *n* characters (in general $n < 256$),
- INTEGER; a binary integer number stored in 4 bytes,
- DECIMAL (*p,q*); a *p* digit decimal number, having *q* digits after the decimal point. Generally $p \leq 15$, (in some newer systems $p \leq 31$).

Besides the above-mentioned, most systems recognize the following data types also:

- VARCHAR (*n*); a variable length character string consisting of maximum *n* characters. Its application is especially expedient when the useful length of the stored text varies greatly. Generally $n \leq 254$,

² It should be noted that SQL does not demand itself even the primary key. So relations can be created without a primary key. [In DB2 they are first so created. The primary key can be defined only after unique indices are defined on the corresponding key field(s).] Naturally, a data model including relations without a primary key is relational only in name. Yet formally with the help of SQL (regrettably), it is possible to work with it. But in this case, the real benefits of the relational model can be utilized only to a limited extent.

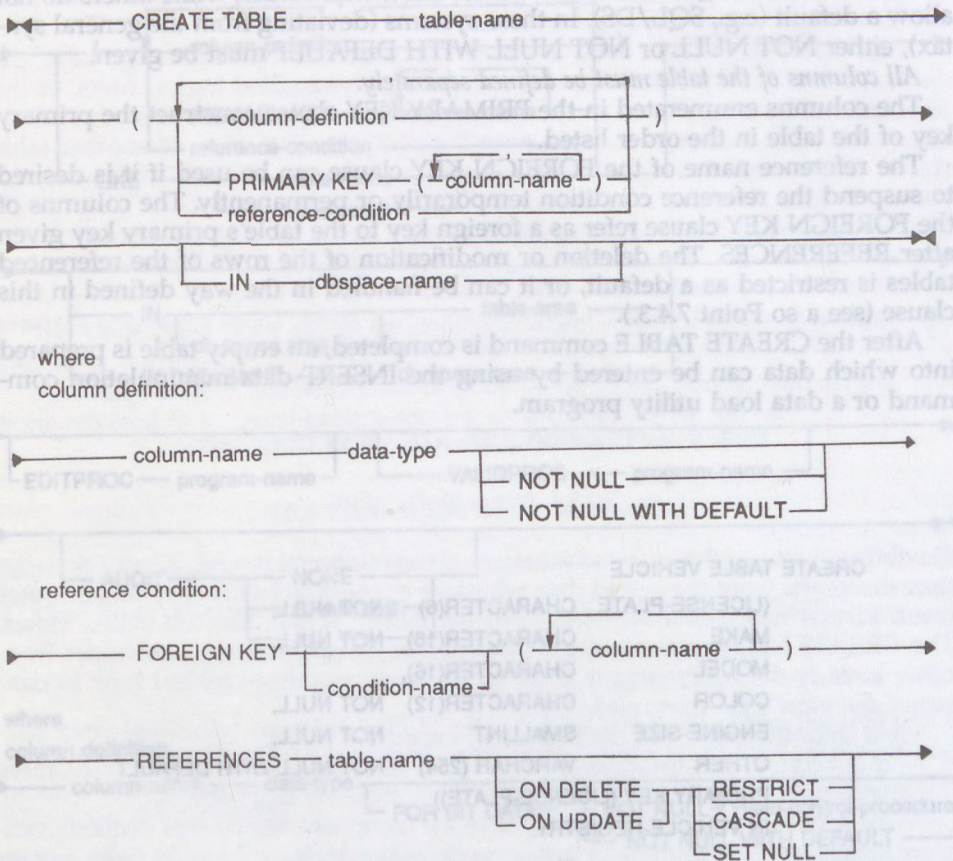


Fig. 10.1. General form of the table creation command

SMALLINT; an integer number stored in two bytes,
 FLOAT / REAL; a floating point number stored in 4/8 bytes,
 NUMERIC (n); a n digit integer stored on one digit in one byte.

Some more-advanced systems permit the declaration of DATE and TIME type columns for storing and calculating date/duration, very long (32 KB) character columns, as well as graphics symbols and special characters represented in 2 bytes.

The declared data type determines the useful meaning of the bit pattern stored in a column.

If the NOT NULL clause is given in the column definition, then this column *must* get a value during data entry in any case. If this clause is not given and no data is entered into this column, then its content will be NULL, or if the clause NOT NULL WITH DEFAULT was given, then the default option for the column (usually zero for numeric, space for character data types). There are systems (e.g.,

earlier versions of SQL/400) which do not allow null values, while others do not allow a default (e.g., SQL/DS). In these systems (deviating from the general syntax), either NOT NULL or NOT NULL WITH DEFAULT must be given.

All columns of the table must be defined separately.

The columns enumerated in the PRIMARY KEY clause construct the primary key of the table in the order listed.

The reference name of the FOREIGN KEY clause can be used if it is desired to suspend the reference condition temporarily or permanently. The columns of the FOREIGN KEY clause refer as a foreign key to the table's primary key given after REFERENCES. The deletion or modification of the rows of the referenced tables is restricted as a default, or it can be handled in the way defined in this clause (see a so Point 7.4.3.).

After the CREATE TABLE command is completed, an empty table is prepared into which data can be entered by using the INSERT data manipulation command or a data load utility program.

CREATE TABLE VEHICLE

```
(LICENSE-PLATE CHARACTER(6) NOT NULL,
MAKE CHARACTER(16) NOT NULL,
MODEL CHARACTER(16),
COLOR CHARACTER(12) NOT NULL,
ENGINE-SIZE SMALLINT NOT NULL,
OTHER VARCHAR (254) NOT NULL WITH DEFAULT
PRIMARY KEY (LICENSE-PLATE))
IN VEHICLE-REGISTRY.
```

a)

CREATE TABLE V&O

```
(LICENSE-PLATE CHAR (6) NOT NULL,
CAR-OWNER CHAR (30) NOT NULL,
OWNERSHIP-START DATE,
OWNERSHIP-END DATE
PRIMARY KEY (LICENSE-PLATE, CAR-OWNER)
FOREIGN KEY CAR (LICENSE-PLATE)
REFERENCES VEHICLE)
IN VEHICLE-REGISTRY
```

b)

Fig. 10.2. (A possible) definition of the V&O and VEHICLE tables seen in Figs 7.1 and 7.2. The syntax, seen in Fig. 10.1, makes possible reference to only one table. That is why in the V&O table the definition of the CAR-OWNER as a foreign key can be realized only with the alteration of the table (see Section 10.6.)

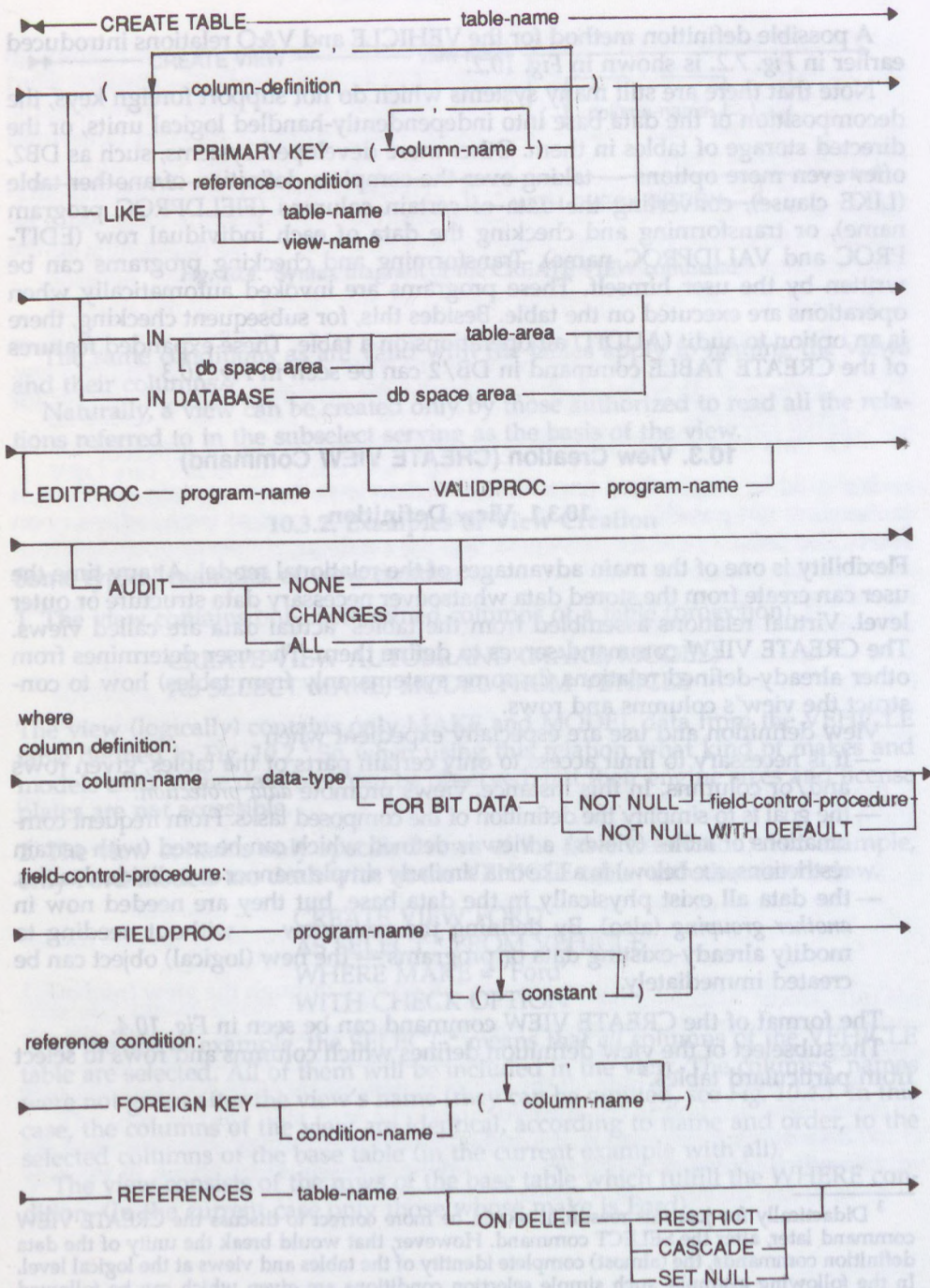


Fig. 10.3. Syntax diagram of the CREATE TABLE command in DB/2

A possible definition method for the VEHICLE and V&O relations introduced earlier in Fig. 7.2. is shown in Fig. 10.2.

Note that there are still many systems which do not support foreign keys, the decomposition of the data base into independently-handled logical units, or the directed storage of tables in them. Other more developed systems, such as DB2, offer even more options — taking over the complete definition of another table (LIKE clause), converting the data of certain columns (FIELDPROC program name), or transforming and checking the data of each individual row (EDITPROC and VALIDPROC name). Transforming and checking programs can be written by the user himself. These programs are invoked automatically when operations are executed on the table. Besides this, for subsequent checking, there is an option to audit (AUDIT) all operations on a table. These expanded features of the CREATE TABLE command in DB/2 can be seen in Fig. 10.3.

10.3. View Creation (CREATE VIEW Command)

10.3.1. View Definition

Flexibility is one of the main advantages of the relational model. At any time the user can create from the stored data whatsoever necessary data structure or outer level. Virtual relations assembled from the tables' actual data are called views. The CREATE VIEW command serves to define them. The user determines from other already-defined relations (in some systems only from tables) how to construct the view's columns and rows.

View definition and use are especially expedient when

- it is necessary to limit access to only certain parts of the tables, given rows and/or columns. In this instance, views promote *data protection*.
- the goal is to simplify the definition of the composed tasks. From frequent combinations of tables (views), a view is defined which can be used (with certain restrictions, see below) in a like or a similarly simple manner as individual tables.
- the data all exist physically in the data base, but they are needed now in *another grouping* (also). By defining the new view — without needing to modify already-existing data or programs — the new (logical) object can be created immediately.

The format of the CREATE VIEW command can be seen in Fig. 10.4.

The subselect of the view definition defines which columns and rows to select from particular tables.³

³ Didactically for just this reason, it would be more correct to discuss the CREATE VIEW command later, after the SELECT command. However, that would break the unity of the data definition commands, the (almost) complete identity of the tables and views at the logical level. In the following example, such simple selection conditions are given which can be followed without exact knowledge of the SELECT command. Insofar as this will cause a problem to the reader, he should first read Section 11.2. and then return to this part.

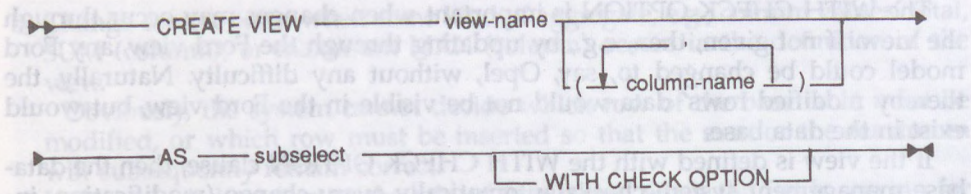


Fig. 10.4. Syntax diagram of the CREATE VIEW command

The same conditions as are valid with the tables apply to naming the views and their columns.

Naturally, a view can be created only by those authorized to read all the relations referred to in the subselect serving as the basis of the view.

10.3.2. Examples of View Creation

Some typical examples of view creation:

1. The view contains only determined columns of a table (projection).

```
CREATE VIEW AUTOBRAND (MAKE, MODEL)
AS SELECT MAKE, MODEL FROM VEHICLE
```

The view (logically) contains only MAKE and MODEL data from the VEHICLE table defined in Fig. 10.2.⁴ So when using this relation what kind of makes and models exist in the data base can be observed, but their engine sizes and license plates are not accessible.

2. The view contains only specified rows of the table (restriction). For example, only Ford models are dealt with in the VEHICLE table with the view below.

```
CREATE VIEW FORD
AS SELECT * FROM VEHICLE
WHERE MAKE = 'Ford'
WITH CHECK OPTION
```

In the above example, the SELECT * means that *all* columns of the VEHICLE table are selected. All of them will be included in the view. The columns' names were not given after the view's name (they can be omitted, see Fig. 10.4.). In this case, the columns of the view are identical, according to name and order, to the selected columns of the base table (in the current example with all).

The view consists of the rows of the base table which fulfill the WHERE condition. (In the current case only those whose make is Ford).

⁴ Naturally, there are no data in the view physically. The data are contained actually in the basic table(s) of the view, i.e., the table(s) on which the view is defined.

The WITH CHECK OPTION is important when changes may occur through the view. If not given, then, e.g., by updating through the Ford view, any Ford model could be changed to, say, Opel, without any difficulty. Naturally, the thereby modified rows' data would not be visible in the Ford view, but would exist in the data base.

If the view is defined with the WITH CHECK OPTION clause, then the database management system checks automatically every change (modification, insertion). Those changes which would result in a row not corresponding to the view definition will be rejected, accompanied by an error message.⁵

3. Similarly

- Only specific columns (those enumerated in the SELECT list) and rows (WHERE condition is also given) can be selected through the defined view.
- Aggregates can be selected, for example, the group total, maximum, etc. for given columns.
- Or data constructed from the join of several tables can be selected.

It should be emphasized once more that the views do not contain any data. It is therefore not possible to define primary or foreign keys or even indices upon them. The definition of the view goes into the catalogue. In every operation concerning the view, the database management system first selects the corresponding data from the base table(s) according to the definition of the view, and afterwards it completes the requested operation with these data.

10.3.3. Restrictions

If the base tables, the tables upon which the views were built, are modified, then the result of the change will be reflected immediately in the views.

What happens then if the 'data' of a view should be changed?

Several scenarios are possible:

- The change can be executed. The data of the base table changes.
- The change cannot be executed because its result would contradict the definition of the view and the WITH CHECK OPTION is given.
- The system does not allow any kind of change through the view (*read-only*).

In the following, some criteria will be enumerated which result in the so-defined view changes not being executable at all or only in a limited way.

a) If a view does not include all the columns defined by NOT NULL in the base table, then insertion is not allowed.

This restriction is obvious. If insertion were made through the view, then the value of columns not occurring in its definition would be NULL in the new rows or the default option instead (NOT NULL WITH DEFAULT).

⁵ Unfortunately, in many implementations this option is not yet available (e.g., neither in SQL/400 nor SQL/DS release 2.). Thus in these systems it is illusory to use the views for data protection; rather it is prudent only to permit reading.

- b) Change cannot occur when some kind of aggregate, e.g., column values' total, SUM (column), or maximum MAX (column) occurs in the definition of the view.

Obviously, the system cannot decide which row of the base table must be modified, or which row must be inserted so that the sum or the maximum will subsequently remain correct.

- c) If the view is based upon the join of two tables, when the fields serving for the unification of the two tables are not unique in *both* tables, then it is not possible to decide unambiguously, in case of a deletion or modification through the view, which row or rows to delete from or modify in the base tables.

For example, with the join of the VEHICLE and V&O tables in Fig. 10.2. a view is created containing the LICENSE-PLATE, CAR-OWNER, and MAKE columns. If from this view the row, where CAR-OWNER=Walter Mitty is deleted, then it is not unambiguous what should happen to the data of the V&O and VEHICLE tables (see Figs 7.1. and 7.2.).

Obviously, the rows with owner Walter Mitty and license plate numbers DY 9680 and V 24034 will be deleted from the V&O. However, is it necessary to delete from VEHICLE both license plates or only that one where the present car owner was deleted? If so, then how can the data base management system realize this? If both must be deleted, then it modifies further the V&O, because in row CAR-OWNER = Bob Jones 1 the deleted column value is a foreign key.

Since it is not possible to solve unambiguously these problems from the data structure, it is not permitted to modify views based upon join. From a technical aspect, most SQL systems operate with a stronger restriction. They consider as read-only all views based upon the join of two or more tables and also those in which the selection includes grouping (GROUP BY, see Point 11.2.2.).

Another typical restriction is that the rows cannot be selected in an ordered way to the view. (This is understandable because the view is a relation as well where rows do not have a preferred order.)⁶

10.4. Index Creation (CREATE INDEX Command)

The user cannot specify the data access path in the relational data model. He/she cannot refer to the indices in any data access command, even when he/she knows that they exist. The system decides itself whether and which indices to use from among the existing ones.

The user and the data base administrator have the option solely to create indices for columns and column combinations which may thereby be expedient to process. If access occurs through a view, then the SQL will use the indices of the base table(s).

⁶ For exactness, it should be noted that the select used for the definition of the view must not include UNION either. However, it follow from the syntax that in the subselect the union operation cannot be used (see Point 11.2.2.).

When creating an index (Fig. 10.5.), the index's name must be specified, as well as on which table and which of its columns to build upon. Besides this, it may be specified that the value of the index column(s) should be unique (UNIQUE) within the table whereas the default also allows duplicates, ascending (default value, ASC), descending, or clustering (CLUSTER). It can be specified to the system in the FREEPAGE and PCTFREE clauses how much free storage to reserve for the insertion of subsequent expansion.

The primary key can always be considered as an ascending, unique index. Some systems create automatically the indices when defining the primary key; in others the index must first be defined.

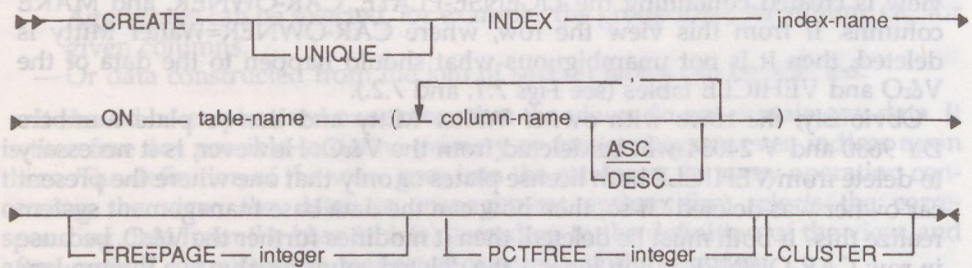


Fig. 10.5. Syntax diagram of the CREATE INDEX command

In any case, as seen in Fig. 10.6, it is expedient to create indices to the V&O table defined in Fig. 10.2.b. However, whether indices are defined for the columns OWNERSHIP-START and -END depends upon the intended applications. Yet it does not matter whether a wise decision was made when the data base was created since indices can be created subsequently, too, or if found unnecessary, deleted.

The index serves always to access a table. Any number of indices can be defined on a table. In general, one index may consist of a maximum of 16 columns, and the

```

CREATE UNIQUE INDEX LP-OWN ON V&O
      (LICENSE-PLATE, CAR-OWNER)

```

```

CREATE INDEX LP ON V&O (LICENSE-PLATE) CLUSTER

```

```

CREATE INDEX OWN ON V&O (CAR-OWNER)

```

Fig. 10.6. Indices defined for the V&O table (Fig. 10.2.b)

(Assume that the V&O table is accessed most frequently based upon the license plate; so that should be the clustering index)

combined length of the columns is also limited (usually between 120–254 bytes). Indices may be defined at any time, so they can be created subsequently for tables already loaded with data. In this case, a unique index is created only when the existing data satisfy this requirement. If two rows having identical index values occur, then the system sends an error message and does not create the index. The system checks also the uniqueness of all so-declared index values when executing any modifications. If such an operation would result in a duplicate index value, then the system rejects it and issues an error message.

In general, columns which are used as an index or part of any index should be defined with NOT NULL clause. Nevertheless, if it is defined for a duplicate key, then the null values appear at the front in an ascending and at the end in a descending index.

The CREATE INDEX command creates a B^+ tree (see Point 4.7.5.) for the existing values of the table specified by the command and generates additional commands which update the table when modified.

10.5. Dropping of Table, View, and Index (DROP Command)

An already-existing table, view, or index can be dropped with the DROP command. The syntax diagram of the command can be seen in Fig. 10.7. (Note that the diagram is incomplete since many other items, for example, synonyms, table spaces, or database areas, can be dropped as well with the DROP command.)

The DROP command deletes existing objects from the catalogue. Dropping a table deletes automatically all data of the table, as well as all views, indices, and foreign keys built on it. Naturally, those previously compiled programs which contain SQL commands and worked with the dropped objects will not operate. Programs are similarly effected when views are deleted. However, if only an index used for access is deleted, then — because the access path does not occur explicitly in any command — during the next execution of the program, a new access path will be built and no error will occur.

If a new index is defined for the table, then programs dealing with this table and compiled before the creation of the new index will use it only after recompilation. So when generating a new index, the efficiency of programs including already-existing, compiled SQL commands is not improved. These programs need recompilation.

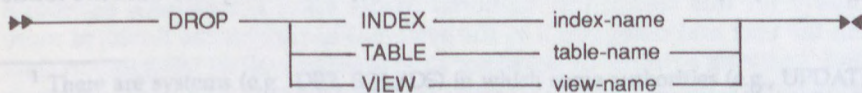


Fig. 10.7. Command to drop a table, view, or index

10.6. Table Modification (ALTER TABLE Command)

Some more advanced systems permit also a certain degree of modification of already-existing tables. This means usually their extension with one or more columns at the end of the rows, or the definition, reactivation, or temporary or permanent deactivation of the primary and foreign keys.

The newly-added column's value in the already-existing rows will be NULL or the default value, depending upon the column's definition.

It is usually not allowed to delete an already-existing column or change its data type (e.g., integer instead of decimal).

To create primary and foreign keys upon the already-existing data is possible only if they satisfy the conditions. Suspending these keys — especially the foreign key — may be necessary in case of a reference loop if a value must be given to the foreign key. An example of this is the following (every column definition is NOT NULL):

```
EMPLOYEES (EMPLOYEE-CODE, EMPLOYEE-NAME, SECTION-CODE)  
PRIMARY KEY (EMPLOYEE-CODE)  
FOREIGN KEY (SECTION-CODE) REFERS TO SECTIONS
```

```
SECTIONS (SECTION-CODE, SECTION-NAME, SUPERVISOR-CODE)  
PRIMARY KEY (SECTION-CODE)  
FOREIGN KEY (SUPERVISOR-CODE) REFERS TO EMPLOYEES
```

Not a single row can be entered into the two so-defined empty tables. An EMPLOYEE row cannot be entered because it does not have an already-existing SECTION-CODE in the SECTIONS table. However, only rows having a SUPERVISOR-CODE occurring in the still empty EMPLOYEES table could be entered into the SECTIONS table.

This contradiction can be solved by lifting the foreign key reference (at least during the loading of the sections) and after data loading, redefining it. Naturally, absent foreign key checking, contradictory or incorrect data may have been entered. That must first be checked and corrected, if need be, since it is a prerequisite for restoring foreign key references.

In some systems the ALTER command includes the option to modify the size of the free space left within the index and data areas for subsequent expansion, as well as modifying the adjustment of the concurrent-access regulating locks (see Section 12.4.).

Many simpler data base management systems do not recognize the ALTER command. If table expansion (albeit retaining its original name) is necessary in these systems, then the data are saved, the original table is dropped (two relations cannot occur with an identical name), a new table is created which deviates from the original only by the additional new column(s), and finally the data are loaded into it.

Data Manipulation Commands

11.1. Introduction

Data manipulation commands enable data to be selected (SELECT), and rows to be inserted (INSERT), modified (UPDATE), or deleted (DELETE) from already existing tables and views.

A command means always the execution of specific data manipulation operations on one or more explicitly-given tables or views (e.g., insertion to the VEHICLE table) concerning 0, 1, or more rows of it, depending upon the parameters given in the further clauses of the command.

The command can be executed only by those possessing authority to the given table(s) and/or view(s) to execute the given command. In most systems the authority is needed not only for the table/view itself, but for the operations, too. So, for example, it is possible that A is authorized to read the V&O table (has a SELECT privilege), but cannot modify anything in it (has no DELETE, INSERT, or UPDATE privilege), insofar as executing any operations on the data of the VEHICLE table.¹

In general, the data base administrator can execute any operation on every relation (with the exception of direct modifications in the catalogue). The owner of the given object is also authorized to do this in any of his tables/views. The other users (except in the simplest personal computer systems) can execute only those operations on the table/view for which they have received permission from its owner. The GRANT (see Section 12.2.) and REVOKE (see Section 12.3.) commands serve to grant and revoke privileges and authorities.

The data manipulation commands (DML) can be issued interactively or embedded into a host language as well. For some commands the two formats are slightly different. Here the interactive form will be introduced. The deviation of the embedded form of the DML commands from the former will be introduced in Chapter 14.

¹ There are systems (e.g., DB2, SQL/DS) in which some authorities (e.g., UPDATE) can be regulated at the column level, too. In this case, the user can modify only the defined columns of the table/view. In other systems this is solved by views. The user gets an UPDATE privilege not for the table, but for the view which consists only of the modifiable columns.

The system transforms the syntactically-correct data manipulation commands into a sequence of relational operations. Afterwards it checks whether the user has the right to execute all of them. If yes, then based upon available information, the system determines (according to it) the optimal command execution sequence and access paths and generates the commands necessary for that for the operating system.

So, for example, for the execution of a command which selects all data about Ford-made vehicles from the VEHICLE table and supplies them in ascending order by license plate, any of the following three methods can be efficient. The data determines which will likely be the optimal.

- a) All vehicle records are read in physical order; the Ford-made ones are selected and stored in a temporary table. This will be sorted by license plate. If there is neither a make nor license plate index, then this method will be the fastest.
- b) If there is a license plate index, but none for the make, then the license plate index is examined, and from this only the Ford-made records are selected.

Generally, in this case, it is the fastest method since neither a temporary table nor sort is needed. Yet if most of the records in the table fulfill the selection conditions, then it can happen that method a) is more expedient. Sorting the selected data may require less time than frequent paging during sequential reading via the indices. Naturally, if the license plate index is clustering, then method b) is more efficient in any case.

- c) If there is a make index, but none for the license plate, then the Fords will be selected by this index and stored temporarily, and the rows of the temporary table sorted by license plate. In this case, this solution is almost always the best. Similar to point b), there are exceptional cases when the selection criteria are highly unselective; most vehicles are Fords.

If there is an index for both the license plate and the make, then whether method b) or c) is more efficient depends in part upon which index is clustering and how selective the search condition is. If the make index is clustering, then it is sure that method c) should be chosen. However, if the proportion of Fords surpasses a certain percentage, then method b) will be better. If none of the indices cluster, then generally method c) will be the most efficient.

It should be clear from the above how difficult it is to define the (likely) optimal access path even in a simple operation, and how many factors the user must take into account when attempting to solve this task in a probably correct manner. This type of analysis is (or should be) made when working with network systems. Naturally, when dealing with small data bases, the time differences are insignificant (in absolute value), but with a large data volume, they can be enormous. Absent sound analysis, data base operation becomes futile.

Particularly for this reason the relational model takes access path definition away from the user and delegates it to the system. That solves the problem more rapidly and, ignoring extreme cases, better or at least as well as most expert data base administrators, not to mention the average programmer or non-programmer user.

The system sends a message about the successful (or unsuccessful) completion of the command to the user, displays the requested data, or executes the modifications in the data base.

In multi-user systems, the system also ensures automatically that if, during the execution of a command, another application would like to work with the same data, then the two uses remain consistent. So, for example, if one user would like to modify all rows of a table (He can do it with one single UPDATE command.), then another user cannot modify anything in this table until the first modification has been completed for all rows; its result has not been committed in the data base. These issues are discussed in more detail in connection with locks in Section 12.4.

11.2. Selection (SELECT Command)

11.2.1. The Operation of the Full Select and Subselect

The SELECT command selects from one or more tables/views data satisfying defined conditions. It is a composite command. Almost all relational operations defined in Section 7.6. and their (permitted) combinations can be executed with one single SELECT command.

The result of the command is always a temporary table which, however, based upon the definition in Section 7.2., cannot be considered as a relation.

As default, the result of an interactive SELECT will be displayed. If required, it can be printed or redirected to another sequential file.

The user can view the selected data rendered in any order and any form forming union with the result of other, similar subselects. The SELECT command containing the union and/or the sort is called *full select*.

The options of the subselect are slightly limited. They make it possible to: view rows or a part of them selected from one or more relations, create a view or select rows for insertion into another relation, select rows of a relation which can be deleted or modified, and in case of nested SELECTS, define the selectable rows of the outer SELECT.

The following, if not stated otherwise, is valid both for the subselect and the full select.

11.2.2. The Basic Form of the Select Command (Subselect)

The basic form of the select command is the *subselect* which is as follows (Fig. 11.1.):

SELECT how, what

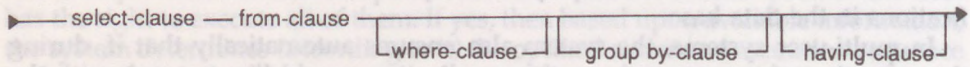
FROM whence

WHERE to which it is true

GROUP BY according to this grouping

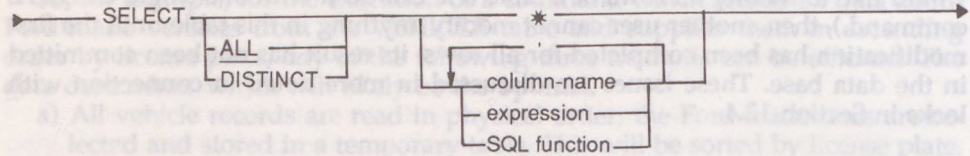
HAVING group selection condition

subselect:

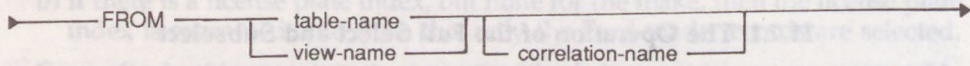


where

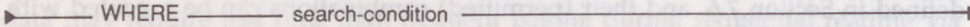
select-clause:



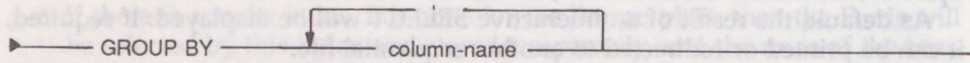
from-clause:



where-clause:



group-by-clause:



having-clause:

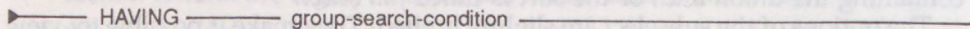


Fig. 11.1. Syntax diagram of the SELECT command (subselect)

The main characteristics of the individual clauses are the following²:

how:

ALL,

Every row satisfying the condition is selected. This is the default value.

² For didactic reasons, it is considered more clear if the concepts are introduced first comprehensively in order to give the reader an overview of the entire structure of the SELECT command and examples are given later. However, those who feel that they can understand this part more easily only by examples should turn to Point 11.2.4., and after studying the examples found there, return here.

DISTINCT

If among the rows satisfying the condition, there are identical ones (for example, because there is a projection in the SELECT, not all the columns are retained from the selected rows), then only one of them is kept.

what:

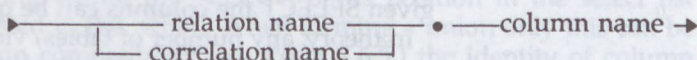
*

All the columns are selected from the tables and/or views given in the FROM clause, in the order of the columns of the relations which facilitate selection. If several relations occur in the FROM clause, then the order of the relations will be the same as in the FROM clause, and within this their column order will be valid. Generally, this is used in interactive work even when not all columns are needed because it is simpler to type a * symbol than list the columns. If possible, avoid it in a program because if a table is changed (e.g., a new column is added with ALTER), then it can cause an error.

column name

The command contains also the projection operation. Only the here-listed columns are selected from the result rows. The column names must be separated by commas.

The column name must be unambiguous within the select command. Thus if the selection takes place from several tables/views, then ambiguous column names must be made unambiguous. It generally means that the column names must be qualified by the name of the corresponding table/view or by the correlation name given in the FROM clause. The usual form is:

**expression**

Constant or the result of permitted arithmetic or string operations completed on columns and/or constants. E.g., AMOUNT*PERCENT/100, one/hundredth of the product of the existing (numeric) *amount* and *percent* column values at the given row are put at the correct column of the result row.

SQL function

A special form of expressions.³ They can be used to represent the values of a given column in another form (e.g., truncating decimals), or aggregate column values [e.g., in the selected column the maximum of the column values, MAX (column name)].

³ In the usual syntax SQL functions are regarded as expressions. However, considering the special use and importance of these functions, in this chapter they are discussed separately. Their possible use is shown separately in the syntax diagram as well as in the text.

In functions, as well as in expressions, ambiguous column names must be qualified.

The columns specified explicitly in place of 'what' or implicitly with a * are called *select list*. The order of the columns in the result table is identical to the column order of the select list.

whence

table name/view name

The columns given in 'what' have to be selected from this relation.

When several tables are given, then the rows are selected from their cross-product. In practice, that means most of the time a natural join. The join column is defined in the WHERE condition.

To fulfill JOIN compatibility (see Point 7.6.7.) distinct column names are ensured by qualification. The identity of the JOIN column names is realized with the explicit enumeration of the columns serving to join in the WHERE condition which can be considered implicitly as a renaming to a common name (see example in Point 11.2.4.b.).

The table/view name must be unambiguous within the entire data base. This is usually achieved by qualifying the relation name with the name of its owner (user identifier). If the SELECT command is issued by the owner of the table, then this qualification can be dropped.

A short correlation name consisting usually of 1-2 characters makes references to the table/view easier. Within the given SELECT the columns can be qualified by it.

In theory, any number of tables/views can be joined, but, in practice, no more than three are usually given since the resultant cross-product produces a truly large temporary table.

to which it is true

The definition of 'what' and 'whence' must always be given. If the WHERE clause 'to which it is true' is not given, then all the rows of the 'whence' relation(s) are selected, otherwise only those satisfying the selection condition given after the WHERE keyword. In its most simplest form this is a

column-name-1 operand comparand,

where the comparand is another column name, expression, or SQL function, while the operand is an arithmetic operation (e.g. :=, < etc.).

All these conditions can be combined together with AND, OR, and/or NOT logical operations depending upon particular needs.

group by

This is used to select aggregates. The rows selected earlier based upon the condition are grouped together according to the values of this column(s) and to every column value the aggregate value belonging to it is given. (E.g., selecting from VEHICLE, the grouping is by MAKE, and for every make the minimum and maximum engine sizes of the different models are given. (See the example of Point 11.2.4.)

If aggregates occur in the select list, but there is no GROUP BY clause, then the result will be one row. The aggregate concerns all of the selected rows.

grouping condition

Only those groups occur in the final result of the SELECT which satisfy all the thereby given conditions. (E.g., the minimum and maximum can be gotten only when the minimum value of the group is greater than 70, see example c) of Point 11.2.4.)

11.2.3. The Full Select

The full select is used only to select data for reading. The result rows of the subselect can be retrieved in ascending (ASC, default value) or descending (DESC) order according to the values of any selected column, and the union of the several selections' result can be formed, too.

The syntax diagram of the full select is shown in Fig. 11.2.

The column-identifier is the unique (if necessary, qualified) name of that column, serving for sorting within the SELECT, or its position in the select list among the columns listed in the 'what'. When forming a union only this can be used, because for union compatibility (see Point 7.6.5.) the identity of column names is realized implicitly, by the position of the columns in the subselect list. Naturally, the user must ensure that the same number of columns will be selected in all subselects and in that order that columns' data standing at the same position in the select list represent the same type of attribute. (It is generally sufficient

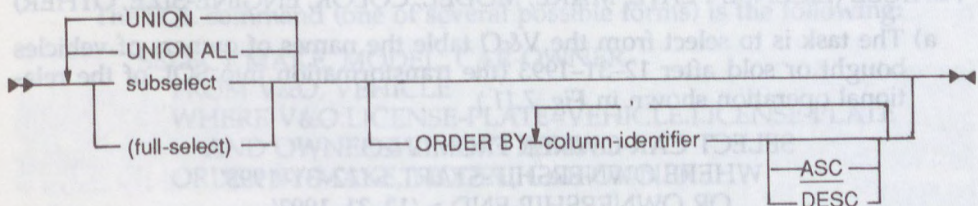


Fig. 11.2. Syntax diagram of the full select

if only character or numeric type data are found in the same places of the select lists in all relations of the union because the system can convert varying length character or different types of numeric representations.)

In case of a UNION, if identical rows result from the union, then only one of them will occur in the result, whereas all of them are gotten when using UNION ALL.

When giving DISTINCT in the subselect or UNION in the full select, the system filters out duplicate rows. To do this, after the selection is made, the result has to be sorted even if an explicit request for that (ORDER BY clause) has not been given. In many cases, this can be very time-consuming. For this reason — if all other conditions are the same — SELECT ALL or UNION ALL is more efficient in general than SELECT DISTINCT or UNION.

It is also evident that *it is not possible to define the access path* in the SELECT command. This is handled by the system itself, and obviously it is able to find a more efficient path if suitable indices have been created.

For this reason, if in the foreseeable selections a column occurs often in the WHERE clause, then it is usually expedient to define an index on it. Using the indices the system can select easily those rows satisfying all of the conditions concerning that column, and it will deal further only with these rows instead of all of them.

It is also expedient to define indices for columns used often in ORDER BY clauses. Doing so, it is possible that the rows are read immediately according to this index, and those not satisfying the further conditions will be bypassed. The final result therefore need not be sorted.

The same is valid if the selection is made through a view. However, in this case the index must be created on the column of the base table(s) of the view.

11.2.4. Examples of Selections

Some simple examples of the application of the SELECT command are given below. The examples concern tables introduced at the relational model shown in Figs 7.1. and 7.2., and created with the definition in Fig. 10.2. To spare the user turning back to the original definitions, the columns of the two tables are given once more.

V&O (LICENSE-PLATE, CAR-OWNER, OWNERSHIP-START, OWNERSHIP-END)

VEHICLE (LICENSE-PLATE, MAKE, MODEL, COLOR, ENGINE-SIZE, OTHER)

- a) The task is to select from the V&O table the names of owners of vehicles bought or sold after 12-31-1993 (the transformation into SQL of the relational operation shown in Fig. 7.11.).

```
SELECT CAR-OWNER FROM V&O
WHERE OWNERSHIP-START > '12-31-1993'
OR OWNERSHIP-END > '12-31-1993'
```


Because the OWNERSHIP-START and END columns were defined as DATE types (see Fig. 10.2.), the date is given as a character string in the comparison. Constant character strings are generally delimited by apostrophes (') in SQL.

The command contains besides the mandatory select list ('what') and the FROM clause ('whence'), only the WHERE clause. The selection condition consists of two conditions connected by OR. The second operands are (constant) expressions.

If the same person bought or sold a vehicle several times after the given date, then he/she appears in the result several times. If desired, this can be eliminated by giving SELECT DISTINCT.

If the owners' names are to be retrieved in ascending order, then the ORDER BY CAR-OWNER clause should be added.

The following SQL commands would produce also the same result:

```
SELECT CAR-OWNER FROM V&O
WHERE OWNERSHIP-START > '12-31-1993'
UNION ALL
SELECT CAR-OWNER FROM V&O
WHERE OWNERSHIP-END > '12-31-1993'
ORDER BY 1
```

(The OR condition of the WHERE restriction can be transformed into a UNION.)

If the UNION operand were used instead of UNION ALL, then the names of the selected CAR-OWNERS would appear only once in the result of the query.

Note that it is conceivable that in response to the two differently-defined queries, the SQL system will generate different access paths (It is not desirable, but regrettably it can happen occasionally.). Thus the response time will depend on the explicit form of the command.

- b) The task is to find out the make and model of all cars and the name of their current owner (the transformation into SQL of the relational operation shown in Fig. 7.12.).

Since the owner name occurs in the V&O table, and the make and model are in the VEHICLE table, the selection must be completed from the join of the two tables. The join is based upon the LICENSE-PLATE columns. This name is identical in both tables, so it must be qualified.

The SQL command (one of several possible forms) is the following:

```
SELECT MAKE, MODEL, CAR-OWNER
FROM V&O, VEHICLE
WHERE V&O.LICENSE-PLATE=VEHICLE.LICENSE-PLATE
AND OWNERSHIP-END IS NULL
ORDER BY MAKE, MODEL, CAR-OWNER
```

It can be seen that the join column need not be in the select list. The join can be executed without it. The condition OWNERSHIP-END IS NULL en-

sures that among the former and current owners of the same vehicle, only the current one is selected, since for the former owners no sale date occurs in this column, to wit, its value is not null.

- c) The task is to select the minimum and maximum engine sizes of different makes' vehicles (the transformation into SQL of the relational operation shown in Fig. 7.13.). Cars whose maximum engine size is less than 70 cubic inches should be excluded. The results are to be in descending order based upon the value of the maximum engine size.

Since an aggregate is now formed, and the same make's vehicles form one group, the GROUP BY clause must be given as well. However, the above must also be completed with the HAVING clause since not all groups must be taken into account. All of the data exist in the VEHICLE table or can be generated from it. Therefore only that table is specified in the from clause. The query can be posed by the following SQL command:

```
SELECT MAKE, MAX(ENGINE-SIZE), MIN(ENGINE-SIZE).
FROM VEHICLE
GROUP BY MAKE
HAVING MAX(ENGINE-SIZE) >= 70
ORDER BY 2
```

11.2.5. SQL Functions

Built-in functions make the usage of SQL systems very convenient. They consist of two groups:

- column functions, and
- scalar functions.

The *column functions* serve to form aggregates. If the GROUP BY clause is used, the

- average, AVG (argument)
- count, COUNT (*)
- maximum, MAX (argument)
- minimum, MIN (argument) or
- sum, SUM (argument)

of a column, an expression, or an SQL function occurring in the aggregate is given to identical groups or to selected groups.

The argument of the average and sum functions can include only columns defined as numeric, or numeric constants, or SQL functions that produce a numeric result.

Minimum and maximum are obtained according to the internal representation (generally EBCDIC code).

The argument of the AVG, SUM, and COUNT functions can be supplemented with the DISTINCT keyword. In this case, when forming an aggregate, the system reckons only with the different values. In that case, the column name must be specified at the COUNT function as well.

If in a selected row, the value of any column occurring in the argument is null value, then it is excluded from the aggregate. The COUNT (*) function is an ex-

ception; all the selected rows are reckoned with in its result. It is therefore important to exercise care since if COLUMN1 may include a null value as well, then it is not certain that the selection of AVG(COLUMN1) will produce the same result as SUM (COLUMN1)/COUNT(*).

If no row satisfies the WHERE condition, then the result column function's value will be the null value, except for the COUNT function which actually takes the 0 numeric value.

Scalar functions transform the value of the argument column, expression, or SQL function into another form. The type of argument depends upon the function. If illegal conversion is attempted, then the command halts and an error message is issued at interpretation or execution time.

The most important scalar functions are the following:

DECIMAL (*argument*, *p*, *q*)

The numeric *argument* is converted into a '*p*' digit decimal number, having '*q*' digits after the decimal point.

DIGITS (*argument*)

The numeric *argument* is transformed into character format. Only this function makes it possible to compare the content of character and numeric columns.

FLOAT (*argument*), INTEGER (*argument*)

The numeric value of the argument is transformed into floating point or binary form.

SUBSTR (*argument*, *begin*, *length*)

It makes a *length* long substring from the *argument* character string, starting from the *begin* position.

In systems allowing the declaration of DATE and TIME type columns, functions permitting operations on such variables can be very useful.

These include, for example, the MONTH, DAY, and YEAR functions, which produce the month, day, and year part of the date-type argument, or the use of, e.g., the DATE+2MONTHS expression whose value is the DATE's value extended by two months. (E.g., in case of DATE='12-31-1991' resulting in '02-29-1992', but in case of DATE='12-31-1990' its result is '02-28-1991'.)

Based on the EMPLOYEES table shown in Fig. 7.21., for example, to find out how many employees work in a given section, the answer can be obtained with the following select command:

```
SELECT S-CODE, COUNT(*) FROM EMPLOYEES
GROUP BY S-CODE
```

However, this query ignores those sections in which no one works currently, though it would be preferable to include them as well with COUNT(*)=0. In the example discussed in Point 11.2.6. by the EXISTS, it is shown how once this query is extended to a full select, they appear as well in the result.

For the same reason, in response to the query 'How many sections does the company have?' it is not certain that the

SELECT COUNT(*) FROM SECTIONS
 and
 SELECT COUNT(S-CODE,DISTINCT) FROM EMPLOYEES

commands would produce the same result.

The first query produces all sections existing in principle, while the second results in only those actually functioning ones possessing already (or still) at least one employee.

11.2.6. Further Selection Conditions

The truly great advantage of the SELECT command is that the correct formulation of select conditions enables to define very complicated conditions which are to be valid for the selected rows.

It is not planned to list the full range of the various conditions. Furthermore, they are system-dependent. Only the most frequently-used options are introduced here. Some of them (e.g., BETWEEN and IN) only simplify or give a clearer representation of the selection conditions.

Selection conditions specifiable in the WHERE clause and supported by most systems are presented in Fig. 11.3. (Some systems, e.g., DB2, SQL/DS permit wider choice than this; others, mainly PC-based, do not allow even the use of all of the above.)

Among the parameters seen in the figure, the host variable will be defined when inserting the SQL commands into the program. The most frequent value of the special register is USER. This means the userid of the user issuing the command. Time period refers to DATE type variables or operations executed on them.

The simplest condition (essentially the upper route of the figure) was introduced earlier by the general definition of the SELECT command.

BETWEEN

The BETWEEN option simplifies and makes more understandable the definition of the condition.

The *expression_1* BETWEEN *expression_2* AND *expression_3* condition is equivalent to the

expression_1 >= *expression_2* AND *expression_1* <= *expression_3*

(that is to say, to the $expression_2 \leq expression_1 \leq expression_3$ mathematical) condition.

When specifying NOT, the opposite of the above-mentioned must be valid.

IS NULL

If a column contains the null value, then neither the *column-name*=0 nor *column-name*='' condition is met. The characteristic feature of the null value is that its value is unknown. So it cannot be said whether it is null or space, or it is not.

The special condition

column-name IS NULL

or

column-name IS NOT NULL

validates (or negates) the presence of the null value.

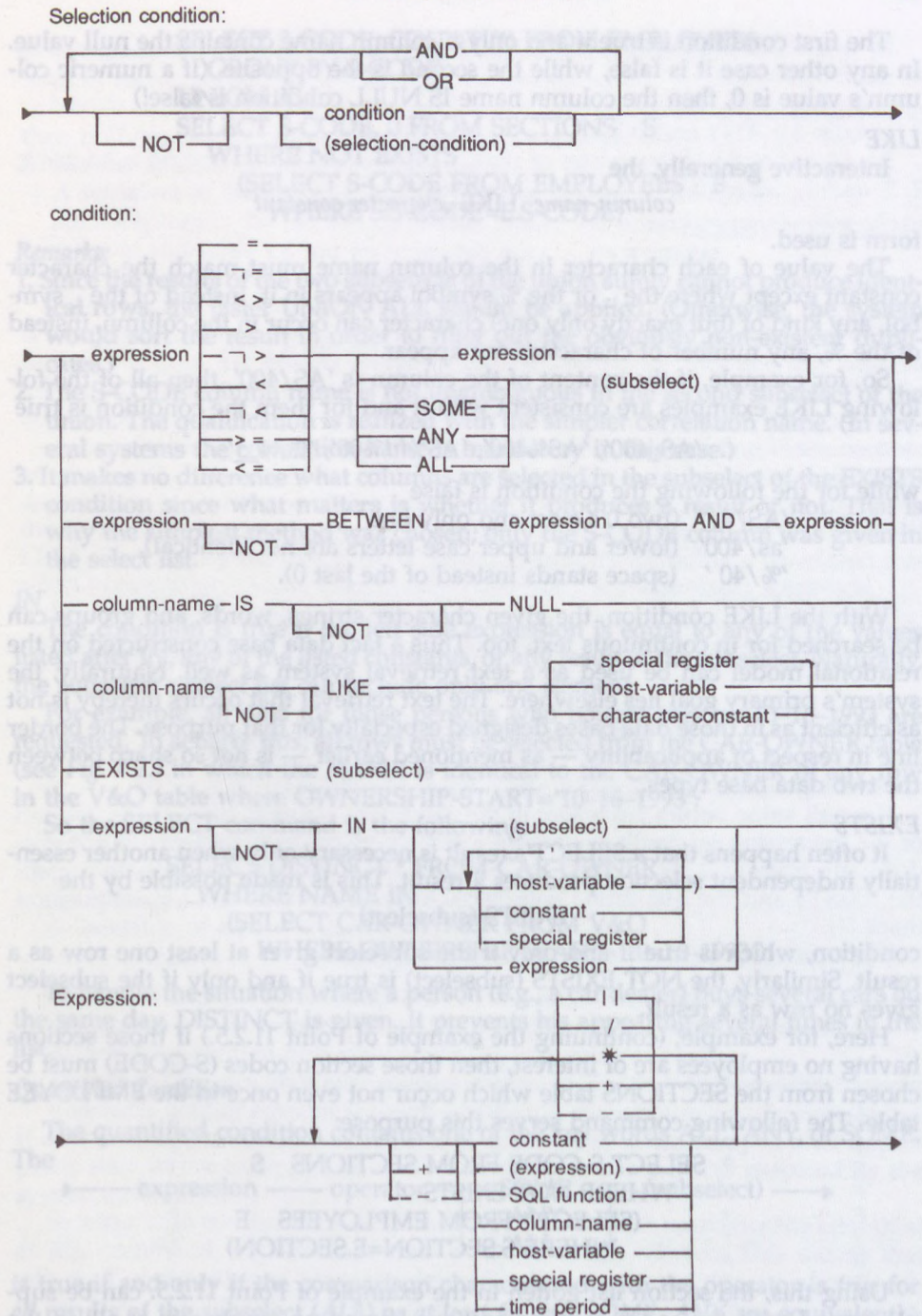


Fig. 11.3. Syntax diagram for selection conditions

The first condition is true if and only if column name contains the null value. In any other case it is false, while the second is the opposite. (If a numeric column's value is 0, then the column name IS NULL condition is false!)

LIKE

Interactive generally, the

column-name LIKE *character-constant*

form is used.

The value of each character in the column name must match the character constant except where the `_` or the `%` symbol appears in it. Instead of the `_` symbol, any kind of (but exactly only one) character can occur in the column, instead of the `%`, any number of characters can appear.

So, for example, if the content of the column is 'AS/400', then all of the following LIKE examples are consistent with it and for them the condition is true

'AS_400', 'AS%400', 'AS%' '%00', '%/4_'

while for the following the condition is false

'AS/4_' (two 0's exist and only one `_`),

'as/400' (lower and upper case letters are not identical),

'%/40' (space stands instead of the last 0).

With the LIKE condition, the given character strings, words, and groups can be searched for in continuous text, too. Thus a fact data base constructed on the relational model can be used as a text retrieval system as well. Naturally, the system's primary goal lies elsewhere. The text retrieval that occurs thereby is not as efficient as in those data bases designed especially for that purpose. The border line in respect of applicability — as mentioned earlier — is not so sharp between the two data base types.

EXISTS

It often happens that a SELECT's result is necessary only when another essentially independent selection produces a result. This is made possible by the

EXISTS (subselect)

condition, which is true if and only if the subselect gives at least one row as a result. Similarly, the NOT EXISTS (subselect) is true if and only if the subselect gives no row as a result.

Here, for example, (continuing the example of Point 11.2.5.) if those sections having no employees are of interest, then those section codes (S-CODE) must be chosen from the SECTIONS table which occur not even once in the EMPLOYEE table. The following command serves this purpose:

```
SELECT S-CODE FROM SECTIONS  S
WHERE NOT EXISTS
  (SELECT * FROM EMPLOYEES  E
   WHERE S.SECTION=E.SECTION)
```

Using this, the section list gotten in the example of Point 11.2.5. can be supplemented with the codes of those sections having no employees at all, in the following way:


```

SELECT S-CODE, COUNT (*) FROM EMPLOYEES
  GROUP BY S-CODE
UNION ALL
SELECT S-CODE, 0 FROM SECTIONS  S
  WHERE NOT EXISTS
    (SELECT S-CODE FROM EMPLOYEES  E
     WHERE S.S-CODE=E.S-CODE)

```

Remarks:

1. Since the results of the two subselects of the union surely cannot produce identical rows, the faster UNION ALL should be applied. (Otherwise, the system would sort the result in order to filter out the obviously non-existent duplicates.)
2. The S-CODE column name is not unambiguous in the second subselect of the union. The qualification is realized with the simpler correlation name. (In several systems the correlation name is mandatory in this case.)
3. It makes no difference what columns are selected in the subselect of the EXISTS condition since what matters is whether it produces a result or not. That is why the simplest method was chosen; only the S-CODE column was given in the select list.

IN

The condition is true if the left side expression is equal to any of the values listed after IN, any of the listed constants, or any result of the subselect. Naturally, the subselect's select list can include only one element.

For example, all data for those persons who bought cars on 10-16-1993 are requested. To achieve this, all rows must be selected from the CAR-OWNER table (see Fig. 7.2.) in which the NAME is identical to the CAR-OWNER of any row in the V&O table where OWNERSHIP-START='10-16-1993'.

So the SELECT command is the following:

```

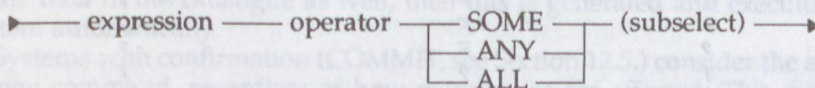
SELECT DISTINCT * FROM CAR-OWNER
  WHERE NAME IN
    (SELECT CAR-OWNER FROM V&O
     WHERE OWNERSHIP-START='10-16-1993')

```

To handle the situation where a person (e.g., a car dealer) buys several cars on the same day, DISTINCT is given. It prevents his appearing several times in the list.

Quantified Condition

The quantified condition contains one of the key words ALL, ANY, or SOME.



is true if and only if the comparison characterized with the operator is *true* for *all* results of the subselect (ALL) or *at least for one* (SOME, ANY are equivalent).

By quantified subselect, if there is no result row, then the condition according to the definition is

- true for ALL, and
- false for ANY and SOME.

So, for example, if the values of the C1 column in the rows of table T1 are 1, 2, 3, and 4, and in the T2 table's C2 column are 2 and 3, then

a) SELECT C1 FROM T1

WHERE C1 = ANY (SELECT C2 FROM T2)

its result: two rows with values C1=2 and 3

b) SELECT C1 FROM T1

WHERE C1 > ANY (SELECT C2 FROM T2)

its result: 3, 4

c) SELECT C1 FROM T1

WHERE C1 > ALL (SELECT C2 FROM T2)

its result: 4

d) SELECT C1 FROM T1

WHERE C1 > ALL (SELECT C2 FROM T2
WHERE C2 < 0)

its result: 1, 2, 3, 4

(The subselect gives no result; the condition will be true for all C1 values.)

11.2.7. Remarks to the Selection Conditions

a) *Handling NULL Values*

If a null value occurs anywhere in the condition, whether because the column or the expression originally contained a null value or the subselect did not produce any rows as a result, then the result of the condition is indefinite (except the EXISTS and the quantified subselect). This will be indicated by a question mark (?).

The expanded truth-table can be seen in Fig. 11.4.

P	Q	P	AND	Q	P	OR	Q
T	T		T			T	
T	F		F			F	
T	?		?			?	
F	T		F			T	
F	F		F			F	
F	?		F			?	
?	T		?			T	
?	F		F			F	
?	?		?			?	

Fig. 11.4. Truth table. The two conditions: P and Q
T = true, F = false, ? = not defined

P and Q are the two components of the composite condition. Both of them can be true (T), false (F), or undefined (?).

Watch out for NOT $?$ =?. If it is not known whether something is true or false, then nothing can be asserted about its opposite as well.

b) *Number of Result Columns in the Subselect*

A subselect in the WHERE condition can select only one column.

An exception is the EXISTS condition, where the select list can consist of any number of columns.

c) *Correlated and Non-Correlated Subselects*

If the result of the subselect is independent of which row of the outer select is compared in the condition, then it is non-correlated; otherwise, it is correlated. In the first case, the system evaluates the subselect once only and this result is used in the comparisons of all rows of the outer SELECT. All examples of the quantified conditions in Point 11.2.6. are non-correlated.

A correlated subselect, for example, is that subselect defining those sections employing 0 employees (the EXISTS example of Point 11.2.6.) because its value depends upon which row of the outer SELECT is examined. The system completes the correlated subselects over and over again for every new row of the outer SELECT. That is why the execution of select commands including a correlated subselect is essentially slower than similar ones using non-correlated subselects.

Efficiency must always be taken into account when composite queries are formulated.

11.3. Alteration Commands

11.3.1. The Common Features of the Alteration Commands

Alteration commands change the content of tables and views.

They can result in the

- deletion of an existing row(s) (DELETE),
- insertion of a new row(s) (INSERT), or
- modification of one or more columns of an existing row(s) (UPDATE).

The command can be executed only by the data base administrator, the owner of the table/view, or a user granted authority for this relation by its owner.

Altering the content of a view alters the base table as well.

The alteration of a table also appears automatically in all the views based on it.

Change is neither possible through a read-only view nor in a system catalogue. If alteration of the data base structure or data makes it necessary to alter appropriate data in the catalogue as well, then this is generated and executed by the system automatically.

Systems with confirmation (COMMIT, see Section 12.5.) consider the alteration as one command, regardless of how many rows are effected. This means that either all of the selected rows are deleted, selected, or modified, or none at all. In the latter case, the user gets an error message. This may happen, for example, during a cross-reference when due to a restrictedly deletable foreign key, none of the WHERE-selected rows may be deleted or updated.

The same can also occur when several rows are entered into a data base with one INSERT command, and the entry of a row or some rows is prohibited by the system due to the absence of a mandatory value in some columns or the insertion of a duplicate value into a column defined as a unique index.

The most important characteristics of the interactive SQL alteration commands are introduced next. Their slightly different embedded versions are discussed in Chapter 14.

11.3.2. Deletion (DELETE Command)

The DELETE command deletes a row(s) from a table or a view.

Its syntax diagram can be seen in Fig. 11.5. The selection conditions can be any of those given in Point 11.2.6. (Another form of the command, used only in programs, is introduced in Point 14.7.4.)

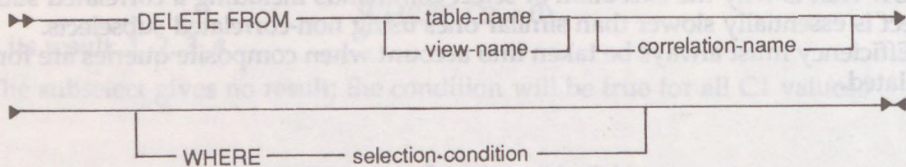


Fig. 11.5. Syntax diagram of the DELETE command
(see Fig. 11.3. for further details of the select conditions)

If the WHERE clause is not given, then the command deletes all the rows of the table/view; otherwise only those satisfying the WHERE condition.

The table/view remains even after the deletion of all rows. The DROP command serves to drop it.

If another table refers to the table with a foreign key, then its rows are changed in the way given in the foreign key definition. The rows possessing the same foreign key value as the deleted key are deleted also from the referred table (ON DELETE CASCADE), change to the null value (SET NULL), or it is not possible to execute the original DELETE command because the delete reference concerns a value with a restricted condition.

The command

```
DELETE FROM V&O
WHERE OWNERSHIP-END < '01-01-1970'
```

deletes from the V&O table all data of all cars which were sold prior to January 1, 1970.

Since no other table refers to the V&O table, these records will be deleted from the data base in any case, assuming that the user who issued the DELETE command was authorized to execute it on the V&O table.

If the select command contains a subselect, then this cannot refer to the erasable relation. The reason is that in this case the result would depend on the deletion order of the rows.

This will be shown in the following erroneous DELETE command. (Such a command is not even accepted by the SQL interpreter; it generates a syntax error.)

The user would like to delete the smallest engine size vehicles from the VEHICLE table. He tries to do this with the following command, which would formally fit the syntax without the above-mentioned supplement:

```
DELETE FROM VEHICLE
WHERE ENGINE-SIZE =
(SELECT MIN(ENGINE-SIZE) FROM VEHICLE)
```

Since nothing can be presumed about the order of the rows of the table, it cannot be predicted in which order the command will pass through the rows of the VEHICLE table. Now two extreme cases will be examined where the execution will occur

- a) according to the engine size in descending order, or
- b) in ascending order.

The subselect is correlated. At execution to examine a row of the VEHICLE table, the system evaluates over and over again the actual minimum chosen by the subselect. The row will be deleted or not based upon this value. In case a) this means until the end of the table, the smallest engine size, is not reached; no row is deleted. The command deletes actually only the row or rows with the smallest engine size value.

However, in case b) the first row examined will be deleted immediately because its engine size value is equivalent to the minimum. The second row will be compared with the thus-modified table's minimal engine size which is naturally equivalent to it. So this will be deleted, too. This procedure will continue until all the rows of the table have been deleted.

If the rows of the VEHICLE table are unordered by engine size (as is probably true in practice), then it is impossible to predict the command's result, to wit, which rows would be deleted. Especially for this reason, the system does not allow such a condition. Naturally, a subselect can be given in which the selection does not come from the erasable table.

So, for example, when deletion of all the ownerless vehicles from the table is desired, then it can be achieved with the following command:

```
DELETE FROM VEHICLE V
WHERE V.LICENSE-PLATE IN
(SELECT LICENSE-PLATE FROM V&O
WHERE CAR-OWNER='NONE')
OR NOT EXISTS
(SELECT LICENSE-PLATE FROM V&O VO
WHERE VO.LICENSE-PLATE=V.LICENSE-PLATE
AND OWNERSHIP-END IS NULL)
```


Vehicles without a current owner can be stored in two ways in the V&O table: 'NONE' is the CAR-OWNER column's value or the OWNERSHIP-END column has a value in every row of the given LICENSE-PLATE.)

11.3.3. Data Entry (INSERT Command)

The INSERT command adds a row(s) into a table or a view.

If no checking clause was specified (WITH CHECK OPTION, see Point 10.3.1.) in the CREATE VIEW command, then even such a row not matching the conditions of the view might be written to the base table.

Format and integrity are checked at insertion. Into a numerically-defined column only numeric data can be entered, and only character values into a character-defined column. (The system transforms among various representations.) If a field of the input data cannot fit in the appropriate column (except for the decimal digits), then the command will not be executed; the row will not be entered into the table.

The same applies when a unique index has been defined (or a primary key) for a column or column combination, and there is a request to enter an already-existing value into it, or a foreign key value during cross-references, but it does not exist in the referred table.

The syntax diagram of the INSERT command can be seen in Fig. 11.6. With the VALUES format, one row, with the subselect, all the rows selected by it, are entered into the table/view.

If no columns are listed, then data are entered into every column of the table/view. If listed, then data will get only into the listed ones. The other columns will contain the null or the default value.

The order of the values given after VALUES, as well as the column order of the subselect's select list must be identical to the column order given explicitly

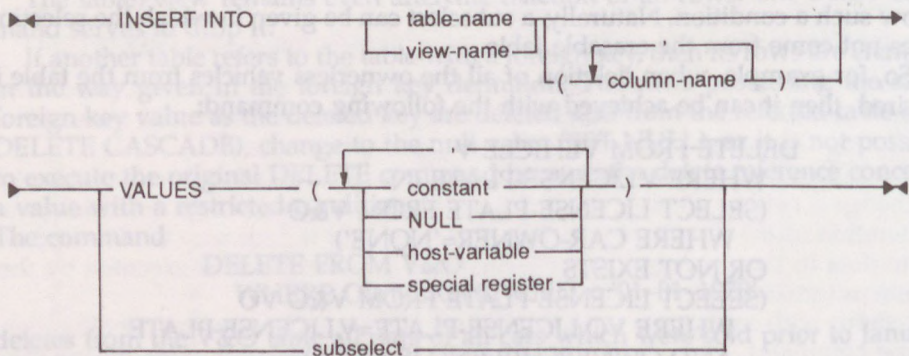


Fig. 11.6. Syntax diagram of the INSERT command
(see Fig. 11.3. for further details of the select conditions and expressions)

in INSERT, or if absent, then to the column order of the CREATE TABLE command.

For similar reasons as the DELETE, here neither can the subselect refer to that table/view in which insertion is to be made.

The owner of a new car can be entered into the V&O table, for example, using either of the following commands:

```
INSERT INTO V&O
VALUES ('ICH-972', 'Random House', '01-31-1993', NULL)
```

or

```
INSERT INTO V&O(LICENSE-PLATE,CAR-OWNER,OWNERSHIP-START)
VALUES ('ICH-972', 'Random House', '01-31-1993')
```

Naturally, if a car with license plate 'ICH-972' does not exist in the VEHICLE table, similarly 'Random House' in CAR-OWNER, or this key combination already exists in the V&O table, then this row cannot be entered due to the integrity conditions.

11.3.4. Modification (UPDATE Command)

It modifies the value(s) of a defined column (or columns) in one or more rows of a given table or view. The rows to be modified are selected by the WHERE condition and the columns with the SET clause. If no WHERE clause is specified, then all the rows are modified; otherwise only those which match the WHERE condition. So it can happen in a particular case that no rows are modified.

Modifying a column used in a *cross reference*

- in the *referred table* the new value must be unique and not null. If the modification is restricted, then no reference to the former value can remain in the referring table either. If the modify condition was not restricted, then the foreign key value in the corresponding rows of the referring table will be modified according to the update reference condition. It changes either to the new or to the null value.
- in the *referring table* a primary key for the new value in the referred table must exist.

If these conditions are not satisfied, then the command will not be executed. When an error is detected, an automatic ROLLBACK will be executed. This means that the original values will be restored to all rows modified by this command before the detection of the error.

The syntax diagram of the UPDATE command can be seen in Fig. 11.7. (Another form of the command, usable only in programs, is introduced in Point 14.7.4.) For example, if the occupation of Walter Mitty changes, then the CAR-OWNER table can be modified thus:

```
UPDATE CAR-OWNER
SET OCCUPATION='daydreamer'
WHERE CAR-OWNER='Walter Mitty'
```

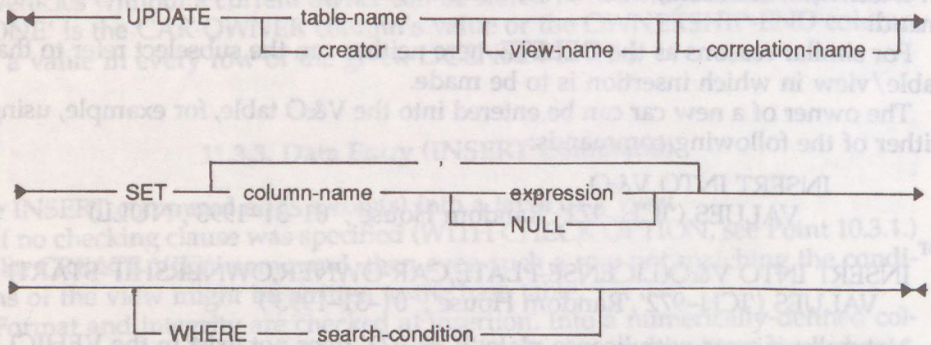



Fig. 11.7. Syntax diagram of the UPDATE command
(see Fig. 11.3. for further details of the select conditions and expressions)

If mandatory checking was not specified at view definition, then during the modification through the view, data could get into the table which did not exist in the modified view.

If the system allows automatic field control procedures (e.g., DB2, SQL/DS Release 3), then this will be activated by each INSERT command and by all UPDATES which change any field controlled by these procedures. When the field value does not fulfill these requirements, then the command will not be executed.

Control Commands

12.1. Introduction

With control commands

- the *owner* of a table/view can permit various uses (GRANT),
- the owner can revoke the earlier-granted privileges (REVOKE),
- the user when working with different objects can deny them to other users (LOCK),
- it is possible to mark a command group consisting of several steps and finalize their result only if all of them are executed (COMMIT), else the original condition is restored (ROLLBACK),
- for completeness, the WHENEVER command, useable only from within a program, is also mentioned (see Section 14.6.); it generates a program-branch, depending on the result of the SQL command,
- besides this — mainly in mainframe SQL systems — the data base administrator can authorize certain users to physically change the structure of the data base (e.g., they are allowed to create tables), or he can revoke these authorities (special forms of GRANT and REVOKE).

12.2. Granting Authority and Privileges (GRANT Command)

The GRANT command

- gives privileges to manipulate the data of the listed tables, and/or views,
- permits the execution of SQL programs, and
- authorizes changing the condition of the data base (e.g., creating or dropping of independent data base areas and tables, etc.). (For users listed in the TO clause of the command, the last feature is strongly system-dependent, so it is not dealt with here.)

The syntax diagram of the GRANT permitting the privileges can be seen in Fig. 12.1.

The ALL option gives all privileges on the listed tables/views to the users listed by name, or to everyone (PUBLIC). If an itemized list is given, then only

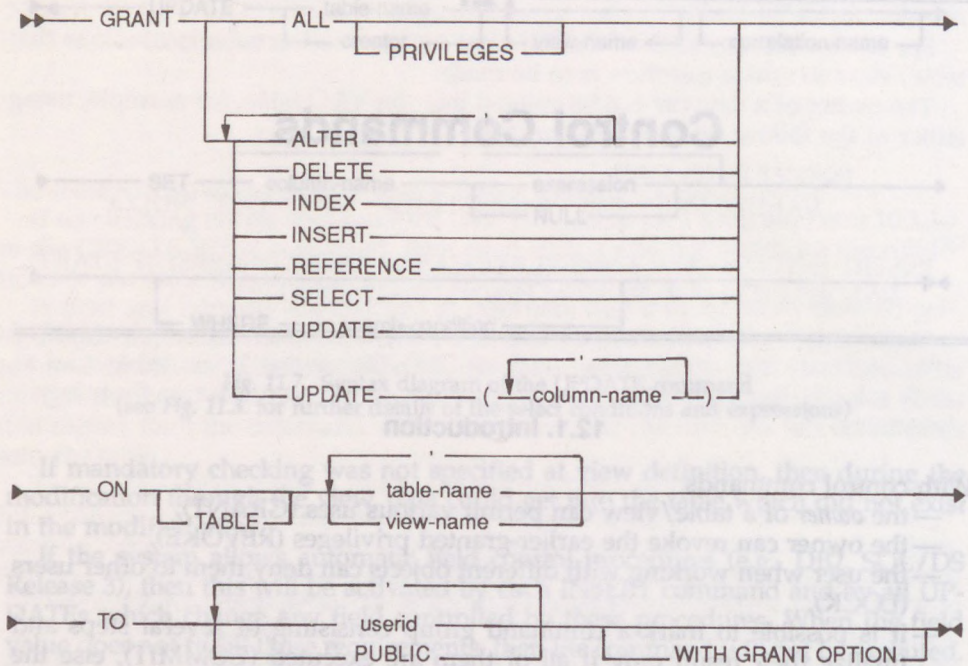


Fig. 12.1. Syntax diagram of the GRANT command

those privileges are granted. If the WITH GRANT OPTION clause is also specified, then the authorized user can pass on the so-granted privileges to anyone else, as well as revoke what was granted by him.

Usually the data base administrator possesses all the privileges specified in the GRANT command without the permission of the owner, but he cannot pass them on to others, in place of the owner, unless he has received explicit permission by a GRANT WITH GRANT OPTION to the table/view.

It goes without saying that, except for the data base administrator, the data manipulation and ALTER TABLE commands can be used by others only with the permission of the owner. It is a normal safety and basic data protection requirement. However, it may not be so obvious why a special privilege is needed to create an index or to refer with a foreign key. Imagine what would happen if someone without informing the owner, created unique indices on the table, even though the owner's application did not require them, or referred to it with a restricted deletion. It would be impossible, even for the owner, to enter a row with a duplicate key, or to delete a row superfluous to him, but referred by someone else. In addition, in a system possessing good safety measures, it is possible that he would be unable to drop disruptive indices, or the redundant error-triggering row could not be deleted since no permission had been given by the owner of the index or the referring table.

12.3. Revoking Authority (REVOKE Command)

The privileges given by the GRANT command can be withdrawn with the REVOKE command. Its form is essentially the same as that of the GRANT command, except that before the user list, instead of TO, FROM is the mandatory keyword.

If an authorization has been received from several sources (e.g., PUBLIC or according to name), then even if one of them has been revoked, by virtue of the others it is still possible to carry out the authorized operation on the table/view.

12.4. Temporary Locking of Objects

12.4.1. Lock Handling

Typically several users work simultaneously on large data bases, often with the same data. There is no problem if each user would like only to read these data. After all, just by looking, nothing would be changed.

However, when any of them would like to change some part of the data viewed by the others, the problem arises of what should the others see or change if they would also like to make changes. The original or the already-modified data? If changes are requested simultaneously, which should occur first? Should the modifications occur in parallel by columns on the same row? If the modification affects several rows of a table, then in this case, one user working simultaneously at one part of the rows would get the old, and another at another part the modified data. Therefore it is very probable that he would draw false conclusions. For example, one command increases every employees' salary by \$ 100 (UPDATE EMPLOYEE SET SALARY = SALARY + 100), while another simultaneously collects salary statistics from the same table (SELECT AVG(SALARY), ... FROM EMPLOYEE GROUP BY ...).

Different and various results would be obtained depending on the exact moment when the two commands overlapped, that is, how many members of the group had already received a salary increase at the moment of group creation. Obviously, this cannot be allowed; the data base management system therefore provides in part automatic, in part user-specified protection against these kinds of inconsistencies which may arise from simultaneous access.

Locks serve this purpose.

Two types of locks can be created:

- *shared lock*:

- anyone can read the so-locked data, but no one can modify them.

- *exclusive lock*:

- no other user can access, read, or change the so-locked data.

The data base management system can lock automatically

- data base areas,

- data pages, and

- rows.

It is obvious that a more-refined locking mechanism generates a bigger administrative overhead to the system, yet it hinders parallel work less.

Besides automatic lock handling, the LOCK TABLE command provides the user with the option to block others from using the table that interests him in a shared or exclusive way. However, the command should be used only when it is really critical not to allow others to read or change the table when working with it.

The user-issued lock is valid until the confirmation of his modifications, or until his next COMMIT (see Sec. 12.5.) or ROLLBACK command (see Section 12.6.).

The system locks automatically at the specified lock levels (area, page, row) as follows:

- During a read anyone else can read it. Modification is possible except when repeatable read has been requested. That means that the repetition of the original read command should produce the same result. In this case, until the user (program) who gave a read command ends viewing (in the program, processing) the selected data, no one else can modify them. This method can be useful when, based upon the viewed data, a decision requiring a rather long time needs to be made, and it must be ensured that the relevant data will not change in the time period between obtaining the information and decision-making. (E.g., when completing a multiple item order during an inventory, or when a group reservation confirmation involving several transfers is requested in a seat reservation system.).
- During modification it locks exclusively at the corresponding lock level.

Especially for this reason, users should hasten to finalize the effect of the alteration command by giving a COMMIT or ROLLBACK command as soon as possible. Or (above all, in an interactive environment) the completion of every command results automatically in the execution of a COMMIT, or in case of an error, a ROLLBACK command.

If access is attempted to an object just locked by another user, then depending upon the system and its operation parameters:

- The user must wait until the object is released, and afterwards the command is executed.
- The user waits for a specified time and if the object is not released by that time, then an error message is received. If released, then the command is executed.
- An error message is received immediately, and the command must be repeated.

12.4.2. Deadlock

Unfortunately, applying exclusive locks can produce *deadlock*, in other words, stalemate in the data base. In this case, two or more users or processes are waiting for reserved resources which, according to standard operating procedures, will never be accessible to them. They hinder each other in such a way that none of them can advance. A typical example of this is when two users work simultaneously. The first exclusively locks the A set of the rows (records), the second the B. If the first user needs now from B one or more records for the processing of

A, while the other to complete the processing of B needs a record from A, then both of them have to wait. So neither of them can advance; the system has become unworkable for each.

Deadlock occurs not only in data bases. Many types of methods exist to hinder its emergence. However, these are not truly applied in data base management systems since they would retard processing. Most systems consider deadlock a necessary evil and notice when it occurs. Their response is to force one process to terminate and restore its data to a previous, well-defined state, thereby releasing the blocked resources which were hindering progress and resolving the deadlock.

12.5. Alteration Confirmation (COMMIT Command)

When modifications are completed in a data base, they are considered temporary by the system until either confirmed by an explicitly or implicitly-given COMMIT command, or a ROLLBACK command restores the original starting state. The rationale is that it has no sense, e.g., to write a mistaken row to the data base in which, for example, numeric data cannot fit into the appropriate column. In this case, it is more expedient to repeat the entry of the entire row with valid data. The same applies if an alter command affects several rows and this command cannot be fully executed. The situation can be made clearer by restoring to the pre-command, unambiguously-defined state, instead of trying to find out which rows have already been altered correctly and which not.

As mentioned earlier, above all, in interactive work the default is to confirm alterations following the error-free execution of each command. However, it can happen that the processing result should be confirmed only once all steps of a multi-part transaction have been executed error-free. For example, a bank remittance can be considered as complete only when the sum has been deducted from the payer's account (an UPDATE command), credited to the payee's account (another UPDATE command), and the bank commission due to the money transfer has been recorded (an INSERT command). In these cases, the user can suspend the automatic confirmation following each command and request confirmation himself by giving a COMMIT command when he deems it fitting to confirm *all* the alterations from the prior confirmation point (the previous COMMIT, ROLLBACK, or process start). In this case, it is said that the transactions run under commitment control. In the above-mentioned example, the two UPDATE and the INSERT commands create one logical work unit. The starting COMMIT must be given before the first UPDATE; the confirming one after the error-free execution of the INSERT.

During Commitment Control selected, but not yet confirmed, data cannot be modified by anyone, and modified data are not even visible. As with table locking, this option should be exploited only when unfailingly necessary, and even then only as briefly as possible.

In order for the COMMIT and ROLLBACK (introduced in the next point) commands to operate, the system maintains a *log* about every modification. In it is entered every alteration — its method, pre-change and/or post-change state. The

confirmation or restoration to the original state is drawn from this log. If some kind of hardware or software malfunction occurs at the exact moment of confirmation/restoration, and consequently it cannot be completed, then the system at the time of restart automatically restores the correct unambiguous state, based upon the logged data.

12.6. Restoration of the Pre-Command State (ROLLBACK Command)

If for any event a command cannot be executed *fully*, then the SQL notices this, interrupts the command, and restores the original state prior to the start of the command. This is called automatic ROLLBACK. It is possible because the original value of every row modified by a command is recorded in a log file.

In addition to automatic restoration, if a command sequence is executed under Commitment Control, then the user has the option of giving an explicit ROLLBACK command at any point of the sequence. (For example, it is noticed that some of the input data were given falsely, or the modification does not match the given checking conditions.) In this case, the system restores the state at the time of the last COMMIT or ROLLBACK (or at the beginning of the task). The logging of new transactions begins anew from there.

Note that most PC-based data base management systems do not possess such confirmation and restoration capabilities due to the large storage capacity and time requirement needed for continuous logging. In those systems, the COMMIT and ROLLBACK commands cannot be used.

There are systems where it is necessary to specify also the WORK keyword in the command.

In them the format is either COMMIT WORK, or ROLLBACK WORK.

13.

Further SQL Commands

13.1. Introduction

The data definition and data manipulation commands introduced in Chapter 10 and 11 are available in all SQL systems and the control commands of Chapter 12 can be applied also in the more advanced systems. The interactive use of the SQL language and SQL data bases is fostered to a really large degree by those commands which feature storing a command in parameterized form and using it repeatedly once prepared, formatting simply a selection's result, supplying information about the current state of the data base, listing or adjusting the actual parameters of the user's own 'data base machine', showing the access path generated by the system for the execution of a selection command, displaying interactive help about the use of the commands, and stopping the actual command, etc.

The commands discussed in this chapter are not as general and standardized as the earlier-mentioned ones. To wit, a portion of them even cannot be found in many systems (mainly in the smaller ones), or, if available, then under a different name, and/or (somehow) they may serve different purposes. Particularly for this reason, below are introduced only some of the further SQL facilities available within the bigger commercial SQL systems. As was also mentioned, attempts have been made to introduce commands independent of the system; however, most of the examples are based on Big Blue, the IBM SQL systems.

13.2. Operator Commands

Operator commands serve to control the entire data base management system and provide information about its state. However, some of them are in reality a series of commands or program-invoking commands which can be issued only by the central operator. Among these belong, for example, those for starting, stopping, or archiving the entire system, as well as inserting new disk units for data storage.

When starting the SQL system, for example, the maximum number of concurrent users, the buffer area for I/O operations, and the character set used by the

system can be defined. When stopping, the operator can decide whether the system should stop immediately (resulting in a ROLLBACK of the commands in use for currently working users) or simply prohibit the start of new operations and stop the system only after currently executing programs finish, and whether to archive the current contents of the log.

Besides these commands, there are some operator commands which can be issued also by the data base administrator or even by any user. The most important among them are the SHOW and COUNTER commands which provide information about the current status of the system (SHOW) or summarize its previous activity (COUNTER)

The SHOW command, among other things, indicates

- how many users are working on the system and what they are doing just now,
- what are the physical characteristics of the data base, what storage is free and which sections are already reserved,
- which locks are currently valid, who waits, and for what kind of lock to be lifted, and
- how full is the log.

The COUNTER command reports, among others, that following the last initialization of the various counters

- how many SQL commands have the applications used to turn to the data base,
- how many transactions were committed and how many were rolled back to their original state,
- how many times did users have to wait due to locks, and how many times did deadlock therefore occur, and
- the total number of page read/writes operations.

13.3. The Storage and Execution of Stored Commands

It greatly aids interactive work when repeatedly-used commands can be stored, and at some later time started and rerun with a simple RUN or START type command. The task can be made particularly flexible if these commands can accept parameters as well.

Individual commands can be stored with the STORE and re-executed with the START command. They can be deleted with the ERASE command while the STORE ... REPLACE option commands overwrites them. The LIST SQL command lists the text of all (*) stored command or only the given one(s). Command sequences can be stored in a special table, in the ROUTINE table.

Its mandatory columns are:

- the name of the stored command sequence. The command can be activated by this name,
- the exact specification of each step of the command, and
- the order of their execution.

Besides the above-mentioned, remarks can be written about every command.

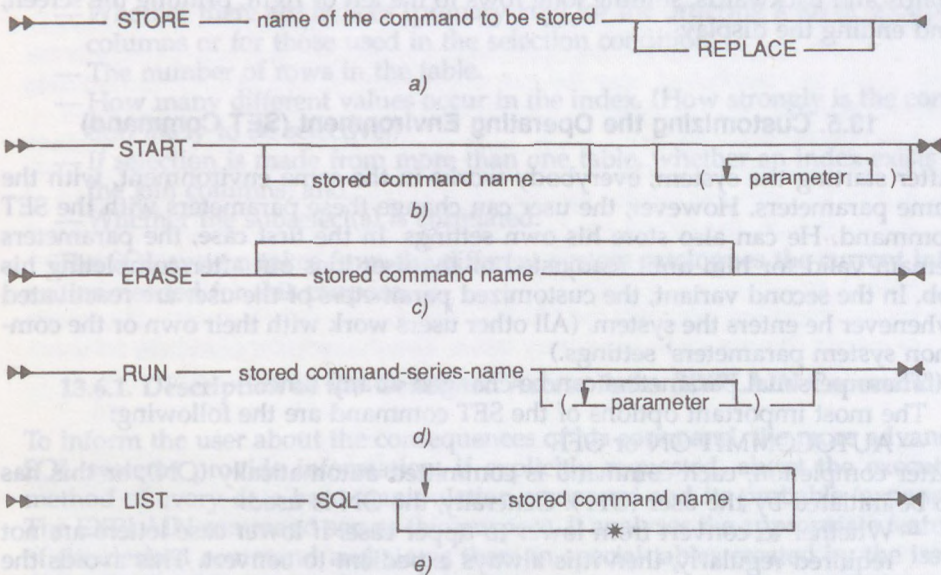


Fig. 13.1. Syntax diagram of commands handling the stored commands.

- a) storing a command, b) starting and executing a stored command,
 c) deletion of a stored command, d) starting and executing stored command routines,
 e) displaying the text of a stored command

The ROUTINE table is created, filled up, or modified with basic SQL commands. The execution of the command sequences stored in the ROUTINE table can be initiated with the RUN command. The actual run time parameters have to be given at invocation.

The syntax diagram of SQL commands is shown in Fig. 13.1.

13.4. Changing the Format of the Results (FORMAT Command)

The SELECT command displays the result in the form given in the select list. However, this format is often not suitable. It would be desirable to view the result rows and header in another format, another grouping, without re-executing the command. The FORMAT command serves this purpose, making possible

- naming the full list,
- changing the appearance of the header, the width, decimals, and leading zeroes in any column,
- excluding or including columns,
- grouping columns to provide group totals,
- changing the representation of the null value, and
- defining the column separator.

Besides this, separate commands serve screen operations such as paging forwards and backwards, shifting long rows to the left or right, printing the screen, and ending the display.

13.5. Customizing the Operating Environment (SET Command)

After starting the system, everybody works in the same environment, with the same parameters. However, the user can change these parameters with the SET command. He can also store his own settings. In the first case, the parameters remain valid for him until readjusted or the user logs out after completing his job. In the second variant, the customized parameters of the user are reactivated whenever he enters the system. (All other users work with their own or the common system parameters' settings.)

These personal parameters can be changed at any time.

The most important options of the SET command are the following:

— AUTOCOMMIT ON or OFF.

After completion, each command is committed automatically (ON), or this has to be initiated by the user (OFF). Generally, the ON is used.

— Whether to convert from lower to upper case. If lower case letters are not required regularly, then it is always expedient to convert. This avoids the problem how to search for 'book' if it is not known whether the input text contains it as 'BOOK', 'book', or 'Book'. If not converted, then all three forms are different.

— Whether the data being currently displayed can be changed (Repeatable Read, see Section 12.4.).

If not necessary, do not adhere to this since it hinders simultaneous use.

— The standard representation of decimals (the number of digits and decimal places).

— How to represent the null value in the output,

— The column separator, and

— The length of variable length columns on the display.

— (The last four parameters can also be specified in the FORMAT command. However, in that case, they only change the form of the actual SELECT command while those given in the SET are valid for every SELECT for which a different format is not given.)

— The actual values of the parameters can be obtained with the LIST SET * or the LIST SET *parameter-name* command.

13.6. Optimal Access Path

To access the data, the SQL establishes itself the optimal access path based upon the available information. Interactive and dynamic SQL (introduced in the following chapter) determine the path just before the execution of the command. For the static commands of the programs this is done at precompilation time.

The definition of the access path is influenced by:

- Whether there are indices (primary key) for the ordering and/or grouping columns or for those used in the selection condition.
- The number of rows in the table.
- How many different values occur in the index. (How strongly is the condition likely to be selective.)
- If selection is made from more than one table, whether an index exists for the join columns, and
- Whether the subselect(s) is correlated.

The SQL system takes from the different system catalogues the current information needed for this purpose.

13.6.1. Description of the Generated Access Paths (EXPLAIN Command)

To inform the user about the consequences of his command, the more advanced SQL systems provide information, if explicitly requested, about the execution method of every data base manipulation command and its probable 'expenses'. The EXPLAIN command serves this purpose. It analyses the appropriate features of the desired command and stores them in special tables created by the issuer of the EXPLAIN command.

The most important of these tables is the PLAN_TABLE. After analyzing a command, the EXPLAIN command stores the following in this table: what is the order of the processing of the tables/views, what indices are used, how is the join realized, and what internal sorting steps are necessary.

The join can be

- a nested loop. For every row of the first table the corresponding rows of the second table are searched, or
- the tables sorted by the join columns are matched.

The access and selection method of the rows can be

- serial, according to physical storage,
- indexed directly, based upon specified values, or
- sequential, based upon indices.

Automatic sorting may be necessary according to the specification of the ORDER BY, GROUP BY or DISTINCT clauses.

After decomposition the command generally consists of several steps. A separate row from each one will be stored in the PLAN_TABLE. The values reflect the condition valid at the time of issuing the EXPLAIN command. If access paths change, then the processing order may vary in the interactive or dynamic SQL, as well as in the recompiled programs.

Besides the processing order, the EXPLAIN command stores information in other tables about:

- the probable processing time (generally, this is only a relative measure; it can be used only to compare the response time of different queries),
- how selective would the conditions likely be for certain columns, and whether, in that case, should an index be used,

- at nested subselects whether the internal subselect is correlated, and
- how frequently would the different subselects likely be executed.

Based upon the data of the analysis tables supplied by the EXPLAIN command, it can be established how the execution time of complex SQL commands might be reduced:

- by creating new indices,
- by reformulating subselects,
- in special cases, by forcing the use of very selective indices (At the time of interpretation, the others are disabled or deleted, so the processor is forced to use the remaining one(s). The restored indices are not used by this program.), or
- by decomposing explicitly the SELECT command, namely storing the results of certain commands in temporary tables, and the creation of indices on them.

13.6.2. Some Remarks about Efficiency

The following orders of magnitude are applicable for middle and big mainframe systems. They are smaller by one order of magnitude for personal computers.

Here are some generally useful guidelines to consider when working with a large data base:

- If the table consists of 10,000 rows, then the user *has to start* considering efficiency.
- If the table consists of 100,000 rows, then efficiency *must be taken into account*.
- If 1,000 rows need to be *ordered* after the SELECT, then the efficiency of ORDER BY must be considered.
- *Joining even smaller tables*, the SELECT's efficiency must be addressed.

Some further advice for efficient operation:

- It is critical to observe with clustering indices whether the clustering still persists. If not, then the table should be unloaded and reloaded according to the corresponding order of the clustering index.
- To reduce wait time due to locks, after all data definition, as well as GRANT, REVOKE, and catalogue modification commands, a COMMIT command should be issued as soon as possible.
- The catalogue indices should be reorganized from time to time.
- Do not look at the catalogue table in a Repeatable Read because it may hinder others from creating a new object.
- If need be, restrict the number of users and programs using concurrently the data base.
- If possible, do not begin the maintenance of large tables during 'rush hour traffic'.

13.6.3. The Automatic Use of Indices

Generally, indices essentially accelerate the selection of rows satisfying the given condition. An exception is when a significant part of the table has been selected, and the result need not be sorted.

Especially for this reason, an index should be prepared for every table. Unique indices for the identification of their rows, the entities, should be defined as well. (It is achieved in any case by the primary key.)

The table should be loaded in the order in which it is processed most frequently, and sufficient space should be reserved for further expansion to avoid table reorganization.

Redundant indices should not be created. Unless it is unfailingly necessary, do not create indices

- for often-modified columns,
- with many duplicates (value > 1000), or
- for sporadic searches.

(If occasionally there is a large search according to a column, then create temporarily an index for this event and drop it afterwards.)

SQL's ability to use indices for generating the optimal access path should not be hindered.

The utilization of the indices is ensured if

- numeric conversion is not needed for comparison,
- the character string length is identical with the comparand literal,
- the LIKE pattern does not start with the % or _ symbol,
- the ORDER BY clause is given to the OR and IN conditions, or
- forms are avoided where SQL bypasses automatically indices.

In general, SQL does not use indices in the following cases:

Numeric conversion

SQL does not use indices if the accuracy of the comparand is greater than that of the column.

For example, if *A* and *B* are SMALLINT types,
then instead of ... WHERE *A* < 1.1E1 AND *B* > 2.0
it is faster

... WHERE *A* < 11 AND *B* > 2

String comparison

SQL does not use indices if in the SQL program the length of the character type host-variable (see Section 14.3.) or constant is greater than that of the column.

For example, if *C* CHAR(4) and *D* CHAR(6),
then instead of ... WHERE *C*='0030' AND *C* > : *D*
it is faster *D1* = *D*
where ... WHERE *C*='0030' AND *C* > : *D1*
D1 CHAR(4)

LIKE pattern

If the LIKE pattern begins with the % or _ symbol, then SQL is unable to use an index.

It slows down selection particularly from large tables. In this case, it can be more efficient to summarize the possible values in an IN (...) condition if not too many of them exist.

IN instead of OR

SQL does not use indices with the OR condition. If the conditions concern the same column, then the IN should be used; it enables the system to work with an index. For example,

```
SELECT * FROM V&O
WHERE LICENSE-PLATE='CYX462' OR LICENSE-PLATE='V24034'.
```

Instead of that, it is better

```
SELECT * FROM V&O
WHERE LICENSE-PLATE IN ('CYX462', 'V24034').
```

UNION instead of OR

If the conditions connected with the OR concern different columns, then it is more expedient to use UNION (if there is an index to the column).

```
SELECT * FROM V&O
WHERE LICENSE-PLATE='CYX462' OR CAR-OWNER='Walter Mitty'.
```

Instead of that, it is better

```
SELECT * FROM V&O
WHERE LICENSE-PLATE='CYX462'
UNION
SELECT * FROM V&O
WHERE CAR-OWNER='Walter Mitty'.
```

Using the ORDER BY Clause

If there is an index on the column that occurs within the OR condition, then SQL accesses it based upon the index, if this is the first column of the order (ORDER BY).

If an ORDER BY is given, even if unnecessary, then the selection will be based upon the index.

If no such index has been defined, then do not use it since SQL will then create a temporary index to the SELECT.

13.7. Other Commands

It is not possible to satisfy everyone's demand for completeness, but here follows a list of some useful SQL commands which — if available — are expedient to use regularly.

CANCEL

It can be given interactively at any time, even when executing another command. It terminates the current command and initiates an automatic ROLL-BACK. It is particularly useful, if it is realized, after issuing a command whose long run is underway, that it is not in the least good or a different execution would be preferable.

HELP

It provides interactive help for SQL use.

It contains as tables:

- the names of topics accessible in the help,
- the syntax of and user's guide to SQL commands,
- the syntax diagrams of interactive SQL commands, and
- SQL messages and codes.

PRINT

It prints the results of the SELECT command appearing on the screen.

COMMENT ON

Comments about the contents can be written into the system catalogue of tables and columns.

Embedding SQL Commands in Programs

14.1. The Creation of SQL Programs

SQL language can be used not only as a self-contained interactive database management language, but also built into different programming languages. The most common host language is COBOL, but it is also used frequently in PL/1. In certain operating systems it is possible to include SQL commands in C, RPG, Assembler, or even FORTRAN programs.

The processing of the source program containing SQL commands is as follows:

1. The corresponding language preprocessor or precompiler recognizes SQL commands embedded in the source language, checks them syntactically, and if they are error-free, transforms them into commands conforming to the syntax of the host language. This usually means a comment to inform the user and a call to a suitably parameterized subroutine. The command defines which subroutine must be retrieved while the parameters are constructed by the preprocessor based upon the command's parameters. To enable the precompiler to complete its job, information must be provided to it in the source program on the one hand, and when calling it on the other hand.
2. The source program is compiled, linked, and edited. The linkage-editor program binds the SQL routines necessary for execution into the loadable program.
3. To execute the program, the permission of the program creator is needed (GRANT RUN). In addition, either the creator of the program or the user who runs it needs the privilege to perform the translated executable SQL commands. If SQL commands are generated dynamically at run time, then the system checks the right of the user who is running the program.

The main steps of SQL program creation are summarized in Fig. 14.1.

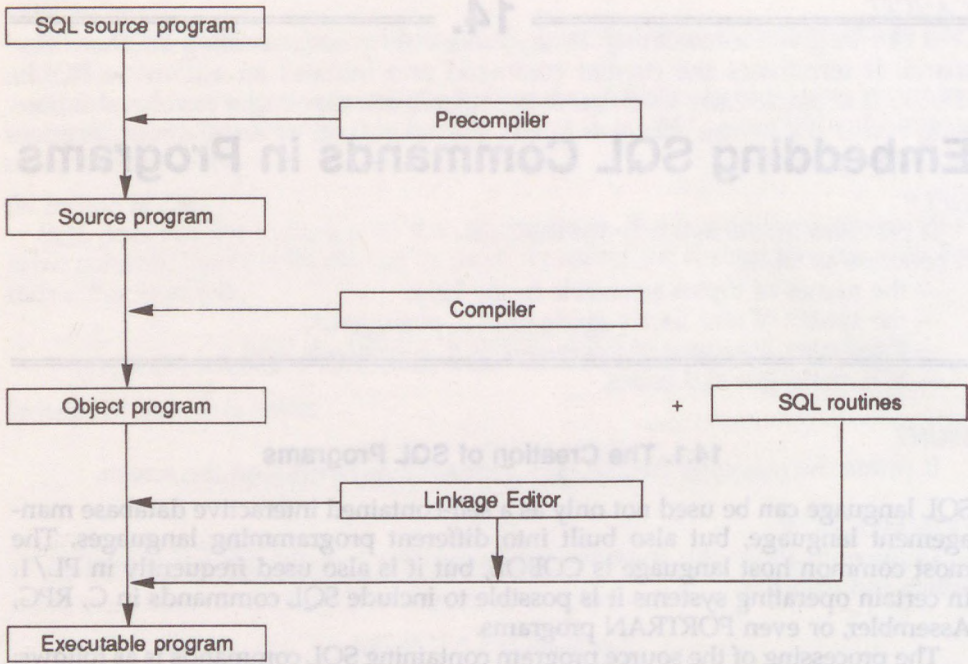


Fig. 14.1. Main steps of a SQL program's creation

14.2. SQL Command Types

In Chapters 10–13 the main commands of the self-contained SQL language were introduced. This chapter deals with those commands which can be invoked from programs. With the exception of the other commands, almost all of the interactive commands can be issued in unchanged form from programs. However, in practice, mostly only the data manipulation commands, plus the COMMIT, ROLLBACK, and LOCK commands, are used in programs. The data definition commands — insofar as needed are usually only for the creation, dropping, or perhaps altering of an object — are simpler and more convenient to give interactively than constructing an entire program around them based on a one-time application. There is at most one exception, when the partial result has to be handled in a program, and a table is therefore created and stored temporarily. However, this table must be dropped before the end of the program or the job, because otherwise an error message due to an identical table name would be issued when the program is repeatedly run.

From a program or interactive SQL, the most important identically callable SQL commands are the following:

ALTER ...
COMMENT
COMMIT

GRANT
INSERT
LOCK

CREATE ...	REVOKE
DROP ...	ROLLBACK
EXPLAIN	

From a program or interactive SQL as well, either in a similar or slightly different given form, are the following:

SELECT or SELECT ... INTO
UPDATE or UPDATE ... WHERE CURRENT OF CURSOR
DELETE or DELETE ... WHERE CURRENT OF CURSOR

Finally, the following SQL commands can be used only from a program:

BEGIN DECLARE SECTION	FETCH
CLOSE	INCLUDE
DECLARE	OPEN
DESCRIBE	PREPARE
END DECLARE SECTION	PUT
EXECUTE	WHENEVER
EXECUTE IMMEDIATE	

To use SQL commands in a program

- SQL commands,
 - host variables, into which SQL imports values while reading in the program (output from the database) and from which SQL exports values while writing (input into the data base),
 - a communications area, in which the success or the reason for any accidental error during command execution is indicated, and
 - commands which enable the handling of predictable or extraordinary situations during the execution of SQL commands
- are embedded.

The embedded SQL commands can be grouped in various ways.

One possible grouping is the following:

- *Executable commands*. They generate actual data base approaches at program compilation time.
- *Dynamic commands*. They prepare and execute SQL commands given at run time, or
- *Non-executable commands*. They supply information for the compilation or execution of commands not directly connected to the data base.

The first group of commands have interactive equivalents while the second and the third command groups can be given only from programs. Another possible grouping is

- static or
- dynamic SQL.

With *static SQL*, it is already known when writing the program just what kind of command will be issued. At most the system connects at run time the value of the actual parameters to the host variables of the program.

The access path is created by the compiler at compilation time. This remains — independent from the changes in the data base — until the program is recompiled or the generated access path for some reason (e.g., an index is dropped) can no longer be used.

With *dynamic SQL*, at the time of compiling the command is not yet known or certain important parameters are unknown (e.g., the table name), without which the access path cannot be generated in advance. The exact form of the command emerges at run time. For this reason, every time the program is executed the system determines dynamically the data base's access method.

The third grouping distinguishes data manipulation commands according to whether they require cursor use.

Traditional programming languages can process only one record at a time whereas SQL manipulation commands can address several rows or records at one time. In this case, a cursor is necessary to show with which record the host language should deal among the records provided by SQL. Cursor-related operations are dealt with in Points 14.7.3. and 14.7.4.

14.3. Recognizing SQL Commands

To enable the precompiler to recognize SQL commands in the source program, the start and end of every SQL command must be marked. The precompiler considers everything between them as an SQL command. It checks syntactically, interprets, and transforms it into host language commands. However, whatever is outside of the marks will be left unchanged. That, as well as precompiler-generated commands, will be processed by the source language compiler.

If the precompiler detects a syntax error during compilation, then it indicates it. In this case, it does not even generate source code for the language compiler. It also indicates whether the program is attempting to execute a SQL command for which the creator lacks authority or privilege. However, in contrast, a ready to run program is compiled because it is conceivable that the creator will receive authority prior to the run, or not the creator's authority, but perhaps the user's will be checked. However, in this case, there is a notation in the compiled program which warns the database management system that it needs to check every appropriate authority during execution of the command at run time.

The syntax to mark the SQL commands depends upon the host language. The start of the command is marked in COBOL, PL/1 (and in most other host languages also) with the character string

EXEC SQL

The end of a SQL command is marked in COBOL by

END-EXEC.

(If followed by another SQL command, then the period is required.)

In PL/1 the semicolon (;) marks it.

Aside from this and some other trivial differences, the use of SQL is the same from all host languages. As COBOL is the most widespread host language in practice, examples in this chapter refer to it.

To copy already-written SQL commands from outside into a program, only the INCLUDE SQL command may be used. It can be embedded in a program as follows:

```
EXEC SQL
```

```
    INCLUDE the file or member name containing the SQL commands to be  
    copied
```

```
END-EXEC.
```

The definition of the SQL communication area (see Section 14.5.) is almost always copied into the program in this way.

14.4. Host Variables

A program would be inflexible if SQL commands always dealt with the same data. Host variables allow statically-given commands to be executed with defined data, with the actual values of the host variable at run time.

The host variables useable in SQL commands and also in other parts of the program must be declared in the source language program according to its syntax. In most implementations, this statement must be put between the

```
BEGIN DECLARE SECTION
```

and the

```
END DECLARE SECTION
```

SQL commands.

Their use in the program is similar to that of other variables. However, within SQL commands a colon (:) must be written before the names of the host variables. This informs the SQL precompiler that the name does not possess equivalence in the data base; it is declared in the host program.

Host variable can be used

- in the WHERE clause as an operand,
- in the INTO clause of the SELECT command (see Point 14.7.2.),
- as a value in the SELECT list,
- in the SET clause of the UPDATE command, and
- in the VALUE clause of the INSERT command.

The INSERT, DELETE, and UPDATE commands have a form totally identical to the corresponding interactive command because they insert the command result into the data base. The SELECT command retrieves from the data base to the program. To store these values, host variables must be declared in the program. For this reason, the interactive form of the SELECT command differs somewhat from the embedded version. The screen-displayed result is passed to the SQL program according to the INTO clause. This follows directly the SELECT list. It lists the host variables, in identical order with the select list, into which the values of elements in the list are transferred.

For example, the

```
EXEC SQL
  SELECT MAKE,MODEL,ENGINE-SIZE
    INTO :P-MAKE,:P-MODEL,:P-ENGINE-SIZE
    WHERE LICENSE-PLATE='CYX462'
END-EXEC.
```

command puts the make ('Mitsubishi'), model ('Colt GT'), and engine size (87) values into the listed program variables of the car with license plate 'CYX462'.¹

Here follow some examples of host variables in data manipulation commands which are fully identical with the interactive SQL:

```
a)      EXEC SQL
        DELETE FROM VEHICLE
          WHERE MODEL=:MOD-DEL
        END-EXEC.
```

During the execution of the command, if the content of MOD-DEL is 'Escort' (obtained as direct input or by some other method), then all the 'Escort' model cars are deleted from the VEHICLE table.

```
b)      EXEC SQL
        UPDATE V&O
          SET CAR-OWNERSHIP-END=CURRENT DATE
          WHERE LICENSE-PLATE=:SOLD-CAR
            CAR-OWNER=:SELLER
        END-EXEC.
```

Before the execution of the command, the program asks, e.g., from the screen, the sold vehicles' license plate and the name of its former owner. The CURRENT DATE is a special reserved SQL word containing the current date.

Generally, the host variables appearing in a program are related to the data of a given column. In this case, the declaration of the host variable must be compatible with the column definition.

SQL transforms numeric data, but indicates overflow errors.

During insertion into a character type column, its content must fit fully into the column. In other cases, during comparison it justifies to identical length (truncates) or during insertion fills up with space.

The handling of the *null value* must be solved in the program also. For this purpose, *indicators* which must be declared as two bytes long integer variables (PIC S999 COMP-4 in COBOL or FIXED BINARY (15) in PL/1) are connected to the host variables.

It is given in the SQL command in the form :host-variable:indicator.

The indicator serves error-handling. If no indicator is given, then errors during execution will be indicated by SQLCODE in the communication area at the end

¹ The names of the program variables can be anything which conforms with the host language syntax. However, for readability, it is expedient to refer in the name to the column with which it is connected.

of the operation (see Section 14.5.). In case of an indicator, certain errors do not generate a negative SQLCODE. Instead, the indicator will differ from 0 at the end of the operation. So, for example, in case of a SELECT INTO command, if the host variable value were a null value, then it would be shown by a -1 indicator value.

So, for example, after the execution of the

```
EXEC SQL
  SELECT MAX(ENGINE-SIZE) INTO:P-ENGINE-SIZE:IND-
  ENGINE-SIZE
  FROM VEHICLE
  WHERE OWNERSHIP-END='12-24-1992'
END-EXEC.
```

command, if a car(s) was sold on December 24, 1992, then the value of the P-ENGINE-SIZE becomes the maximum engine size sold on that date, and the IND-ENGINE-SIZE becomes 0. The SQLCODE value indicating the correctness of the executed operation will be 0 as well.

On the other hand, if no cars were sold on the given date, then the MAX(ENGINE-SIZE) in the SELECT will be the null value. This value is not interpreted directly in the program. It will be shown by an IND-ENGINE-SIZE= -1. The value of P-ENGINE-SIZE can be used in the program only after checking that indicator IND-ENGINE-SIZE=0.

In this case, the SQLCODE value is also 0. The null value is handled by the assignment of the indicator variable. (It does not matter to the system whether it is actually done in the program.)

Naturally, if an indicator has not been assigned to :P-ENGINE-SIZE, then reading null value in it results in negative SQLCODE, accompanied by an error message.

If a character column is truncated at retrieval, then the indicator contains the original length of the column, while the truncated value is entered into the host variable. No error condition is raised. Only a warning is issued in the program.

14.5. Communication Area (SQLCA)

The SQL Communication Area is declared in a program and can be copied expediently with the INCLUDE SQLCA command.

The most important field of the SQLCA area is the SQLCODE variable. It indicates the successful or unsuccessful execution of the prior SQL command. Its value is

- 0 after a normal execution,
- negative in case of an error, or
- positive when detecting a particular event which, however, is not erroneous (by all means). (For example, it has not found the requested record, condition NOT FOUND.)

The negative SQLCODE does not in itself effect program execution. So if it is not checked after the execution of each SQL command, then the program may continue defectively with wrong values.

Further important fields in SQLCA are:

- a brief memo describing the nature of the error,
- the number of records entered, deleted, or modified by the command, and
- warnings about some situations that may cause errors. (E.g., the retrieved column has been truncated, the number of columns of the SELECT and INTO lists do not agree, there is no WHERE condition in the DELETE or UPDATE command, or there was a null value when calculating AVG, SUM, MAX, or MIN.)

The names of the individual fields and the exact division of the SQLCA area depend upon the database management system; moreover, in the same system to a certain degree even upon the host language.

14.6. Handling Particular Events (WHENEVER Command)

SQL command execution can be checked by examining the SQLCODE value after every operation. If it is zero, then the program is continued. If not, then the proper error-handling routine is executed (or the program is terminated). However, this method complicates greatly SQL program writing. It is simpler to use instead the WHENEVER SQL command for handling particular events.

The syntax diagram of the command can be seen in Fig. 14.2.

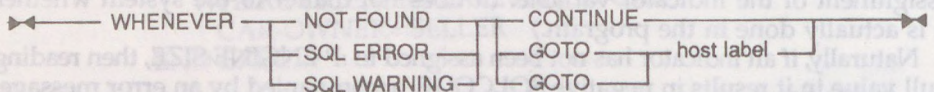


Fig. 14.2. Syntax diagram of the WHENEVER command

WHENEVER is not an executable command. It orders the SQL preprocessor and the compiler to generate a program branch, depending on the result of the SQL operation. The program continues as defined in the WHENEVER command. The WHENEVER effects all SQL statements written after it (not executed!) in the program, and is valid until the next WHENEVER handling the same situation.

Possible versions for the result:

- NOT FOUND: there is no row satisfying the condition, or when processing with a cursor (see Point 14.7.3.), there are no more rows. SQLCODE = 100.
- SQLERROR: any non-handled error. (E.g., the program has no authority for the requested operation, or the result is the null value, but there is no indicator or overflow, etc.) SQLCODE < 0.
- SQLWARNING: a warning about an unusual situation, but not one likely to cause a serious error. (For example, a column was truncated when retrieved by the program; the error has already been

indicated by the indicator, etc.) The SQLWARN0='W' value indicates it generally within the SQLCA communications area or SQLCODE>0 (but $\neq 100$ which means NOT FOUND).

After detecting the special event (NOT FOUND, SQLERROR, or SQLWARNING), the program resumes from one of the following locations, depending upon the conditions specified in the current WHENEVER command.

- CONTINUE: from the next command. If WHENEVER has not been given for the result, then this is the default value.
- GOTO or GO TO: the program continues from the label given in this clause (despite all the curses of mainly theoretical programmers who promote the absoluteness of structured programming!).
- STOP: the program stops. This entails an implicit ROLLBACK as well.

Some Practical Advices:

1. If there is any SQL command in the error-handling procedure (where the program branches in case of SQLERROR.), then it should not be forgotten to issue the command

WHENEVER SQLERROR CONTINUE

at the beginning. Otherwise, if a SQL error occurred there as well, then the program would fall into an infinite loop.

2. If a SQL command requiring rather custom error-handling (e.g., for dynamic commands) is in use, then it may be expedient to separate it from the common SQLERROR branch. A separate error-handling routine is executed on it after the individual examination and analysis of SQLCODE $\neq 0$. In this case, the command

WHENEVER SQLERROR CONTINUE

should be given before the command, followed by the

WHENEVER SQLERROR GOTO common error handling

command (similarly for SQLWARNING as well) after the SQL command.

If this were neglected, then following an error or warning, common error handling would be executed after this command as well. In addition, after the SQL commands which follow, the processing also would continue with the next command without any error handling, instead of using the standard error-handling declared in the program.

An example of the WHENEVER command use is shown in Fig. 14.3.

In case of error, the program branches always to the label *not-good*. At the end of cursor processing, or if there is no row fulfilling the condition, then the program resumes from the *does-not-exist* or *no-more* label, depending upon where the erroneous commands are located in the defined program.


```

EXEC SQL
    WHENEVER NOT FOUND GOTO no-more
END-EXEC.

EXEC SQL
    WHENEVER SQLERROR GOTO not-good
END-EXEC.

...
EXEC SQL
    ... here in case of NOT FOUND branches to no-more
END-EXEC.

...
EXEC SQL
    WHENEVER NOT FOUND GOTO does-not-exist
END-EXEC.

EXEC SQL
    ... here in case of NOT FOUND branches to does-not-exist
END-EXEC.

```

Fig. 14.3. An example of the use of the WHENEVER command

- in case of NOT FOUND, depending upon where it stands in the program, the program branches to a label named *no-more* or to *does-not-exist*,
- in case of SQLERROR, it branches always to the label named *not-good*,
- in case of SQLWARNING, based on the default, the program continues with the next command

The CONTINUE default value is valid for warning since no WHENEVER command has been given for it.

14.7. Static SQL Commands

14.7.1. General Features

Static SQL commands are used if it is already known when writing the program what command will be given, with what clauses, as well as what tables will be referred.

The program can work with the actual values selected or use variable data for update into the data base, or use them in the conditions of the WHERE clause.

However, the command can be changed during execution, too, for example, 'Where the results of SELECT should go.' and 'What must be the value of the clause INSERT or WHERE.'

This can be handled by host variables.

Data definition commands (CREATE, DROP, GRANT etc.) can be given statically from the program, but since table names cannot vary, it is much simpler to execute them interactively (or perhaps with a stored command).

In general, statically-used commands are the

- SELECT,
- INSERT,
- DELETE, and
- UPDATE

commands. If the command refers to several rows, then

- cursor definition and
- cursor use commands are used to assign the actual row.

These are the following:

- DECLARE CURSOR,
- OPEN,
- CLOSE,
- FETCH, and
- PUT.

14.7.2. Selecting a Single Row (SELECT INTO Command)

If it is certain that the SELECT command will result in at most one row, then the SELECT INTO format can be used.

For its individual clauses, the same is true as for interactive selects (see Section 11.2.). The only difference is that the columns of the (single) selected row are put in order into the host variables defined after INTO.

Indicators can also be defined for the host variables.

The column types and their appropriate host variable types must match.

Conversion and justification is done automatically by SQL.

A warning (SQLWARN0='W') will be given if

- a character field has been truncated. The indicator contains the original length;
- null value(s) has not been taken into account in a group function,
- the numbers of columns and host variables do not match, or
- there was a modification in the time arithmetic because of month end or leap year (e.g., March 31+1 month=April 30!).

If no cursor is needed, then the formats of static and interactive commands are identical. The only difference is that in a static program command, instead of an expression, a host variable can stand wherever the former can be given according to the syntax, while with interactive commands naturally this is not possible.

14.7.3. Cursor Use

A cursor is needed when the SELECT command results in several rows which must then be processed one by one.

To be able to use it,

- the cursor must first be defined (DEFINE CURSOR),
- then it has to be opened (OPEN),
- next it must be moved to refer to the row to be processed (FETCH), and
- after processing of the last row is completed, it must be closed (CLOSE).

Before using it, the cursor must be declared in the procedure section of the program as shown in Fig. 14.4.a).

The name of the declared cursor is unique in the program.

Any number of cursors can exist in a program. The declaration of the cursor does not result in real data selection and transfer. This takes place after executing OPEN, when the cursor is opened. The data selected by the SELECT command are then read into a temporary table. So the SELECT command for cursor definition cannot contain an INTO clause.

The rows of the temporary table formed as a result of the OPEN are sequentially retrieved by the FETCH command.²

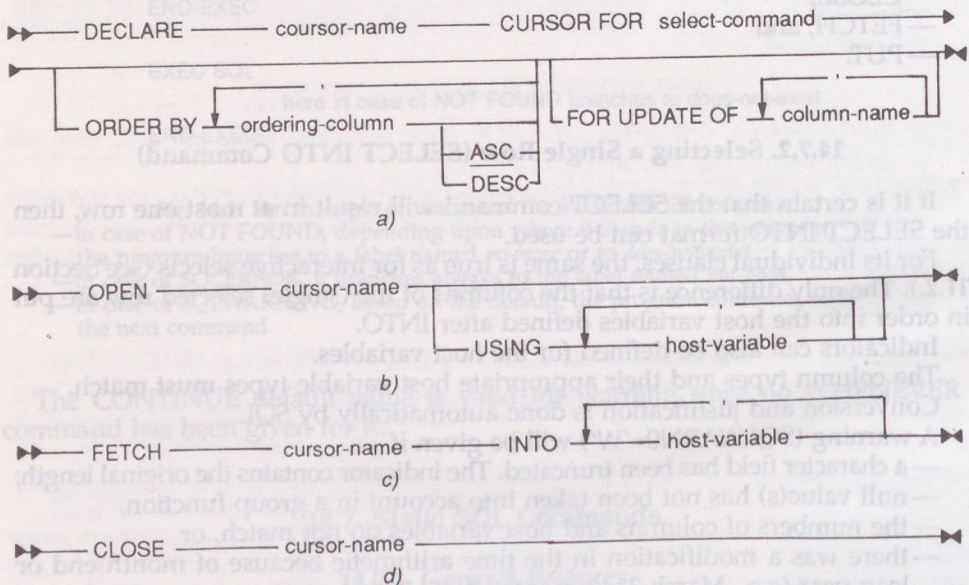


Fig. 14.4. Syntax diagram of the most important cursor manipulation commands
a) cursor declaration, b) opening the cursor, c) moving the cursor, d) closing the cursor

² This table is automatically handled by the SQL. Neither its name, size, nor other parameters have to be declared either in the program or in its run environment. Regardless, when its rows are processed, the table always indicates the status existing at the moment of opening the cursor. So in case of concurrent processes, if it is desired to block another user from changing the table during processing or looking at the incompletely modified table(s), then it must be locked from others with the LOCK TABLE command, or by defining Repeatable Read (the result must be identical after rereading) or running under Commitment Control.


```

* HOST VARIABLES DECLARATION
EXEC SQL
  BEGIN DECLARE SECTION.
END-EXEC.
77 P-LICENSE-PLATE PIC X(6).
77 P-CAR-OWNER PIC X(30).
77 P-OWNERSHIP-START PIC X(10).
77 P-OWNERSHIP-END PIC X(10).
77 IND-OWNERSHIP-START PIC S9(4).
77 IND-OWNERSHIP-END PIC S9(4).
EXEC SQL
  END DECLARE SECTION.
END-EXEC.

* CURSOR DECLARATION
EXEC SQL
  DECLARE CAR CURSOR FOR
    SELECT CAR-OWNER, OWNERSHIP-START, OWNERSHIP-END
    FROM V&O
    WHERE LICENSE-PLATE = :P-LICENSE-PLATE
END-EXEC.

* RETRIEVE THE REQUESTED LICENSE-PLATE INTO P-LICENSE-PLATE.

* SQL ERROR-HANDLING PROCEDURE
EXEC SQL
  WHENEVER SQLERROR GOTO ERROR-HANDLING.
END-EXEC.

* CURSOR-OPENING
EXEC SQL
  OPEN CAR
END-EXEC.

* RETRIEVE ROWS OF V&O
PERFORM READ-IN UNTIL SQLCODE=100

* ALL OF THE OWNERS FOUND CURSOR CLOSING
EXEC SQL
  CLOSE CAR
END-EXEC.

READ-IN
* PROCESSING THE ROWS OF V&O
EXEC SQL
  FETCH CAR INTO :P-CAR-OWNER, :P-OWNERSHIP-START
  :IND-OWNERSHIP-START, :P-OWNERSHIP-END :IND-OWNERSHIP-END
END-EXEC.
* BRANCH TO PROCESS PRIOR OR CURRENT CAR-OWNER
IF IND-OWNERSHIP-END = -1 THEN PERFORM CURRENT-OWNER
ELSE PERFORM FORMER-OWNER.

CURRENT-OWNER.

FORMER-OWNER.

ERROR-HANDLING.
* SQL ERROR-HANDLING

```

Fig. 14.5. Processing with cursor

From the V&O table all the rows of a car with a given license plate are processed.

Processing differs with the present and former owners.

Branching is made upon the NULL value, shown by the OWNERSHIP-END indicator

This moves the cursor one row forward in the temporary table, and inserts the values of the selected columns of the so-assigned row into the host variables defined in the `FETCH` command.

The end of the table is indicated by `SQLCODE=100` after the `FETCH` command (the occurrence of the `NOT FOUND` condition in `WHenever`). Next the cursor must be closed (`CLOSE`) to release any locked rows of the temporary table.

To modify based on the cursor, namely modification of the row on which the cursor currently stands, the clause `UPDATE FOR` must also be included in the declaration. In this case, the columns of the actual row specified can be modified by the cursor-related `UPDATE` command (see Point 14.7.4.).

A program excerpt which reads in from the `V&O` table all the rows of a car with a given license plate, and works further with them depending upon whether they refer to a former or the current owner, is shown in Fig. 14.5. The latter is ascertained by checking the indicator assigned to the `OWNERSHIP-END` column. This is negative for the current owner, since in that case the column value is null. No indicator was defined for the `CAR-OWNER` column because it cannot contain null value (see command `CREATE TABLE V&O` in Fig. 10.2.).

14.7.4. Modification based on Cursor Position

To modify or delete all of the rows satisfying a selection condition, the `DELETE` and `UPDATE` commands can be used from the program as well. They are totally identical in form and effect to their interactive versions.

However, it often happens that various operations must first be executed with the selected rows and depending on their results, some rows will be deleted while others will be changed in various ways (or left unchanged). The `DELETE` and `UPDATE` commands serving this purpose can only be used from the program and work with the positioned cursor (Fig. 14.6.).

To use them, the cursor must be defined first. Next the so-selected rows are read into a temporary table by opening the cursor. Its rows are retrieved with the `FETCH` command, just as detailed in Point 14.7.3. To delete or modify the so-assigned row, the appropriate command is given with the `WHERE CURRENT OF` clause. Unlike the normal `DELETE` and `UPDATE` where any number of rows can be changed, the positioned command deletes or modifies always one row only.

After a delete operation the position of the cursor is undefined. It must be set to the next row with a `FETCH` command. After `UPDATE`, the cursor points to the modified row.

The handling of integrity conditions and locks used with cursor-positioned modifying commands is similar to that with the normal `WHERE` condition.

For example, as with the latter, certain restrictions are valid when selecting based on the subselect (see Point 11.3.2.). Similarly, defining the `SELECT` command included in the cursor definition is not totally free of limits even when modifying based upon the cursor's position. However, its detailed discussion is well beyond the scope of this book.

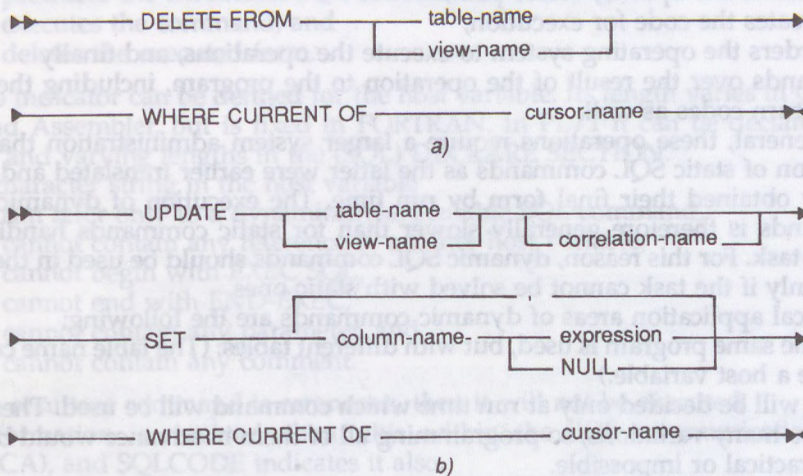


Fig. 14.6. Syntax diagram of positioned cursor commands
a) Deletion of the current row, b) Update of the current row

Note that in most systems COMMIT and ROLLBACK close the cursor. Thus if either is given during, rather than at the end of the cursor's processing, then processing will resume from the first row. To avoid this, some systems recognize also the HOLD clause of these commands. Using it, the cursor remains at the row where it pointed when the COMMIT HOLD or ROLLBACK HOLD command was issued. Processing continues from that or from the subsequent row by giving the FETCH command.

14.8. Dynamic SQL Commands in the Program

14.8.1. The Features of Dynamic Commands

Sometimes it is not possible when writing a program to formulate the database related requests with such precision that they can be built into an executable program using static SQL commands already fixed at compilation time. In this case, dynamic SQL commands which can be formulated during the run of the program are used.

At program run time the data base management system

- checks and interprets the SQL statements syntactically,
- checks whether the user executing the program is authorized for the operations included in it, and
- in case of an error, issues an error message and passes control back to the program.

Otherwise, it

- defines the optimal access path,
- creates the code for execution,
- orders the operating system to execute the operations, and finally
- hands over the result of the operation to the program, including the SQL return codes as well.

In general, these operations require a larger system administration than the execution of static SQL commands as the latter were earlier translated and have already obtained their final form by run time. The execution of dynamic SQL commands is therefore generally slower than for static commands handling a similar task. For this reason, dynamic SQL commands should be used in the program only if the task cannot be solved with static ones.

Typical application areas of dynamic commands are the following:

- The same program is used, but with different tables. (The table name cannot be a host variable.)
- It will be decided only at run time which command will be used. There are too many variations, so programming all of them in advance would be impractical or impossible.
- The operators (types) and/or their number in the WHERE clause are not known in advance.
- It may be undesirable to stop the run because of an integrity error. The ability to continue or give other commands is ensured by the dynamic entry of ALTER ... KEY or DROP ... KEY or INDEX.

Not every SQL command can be formed dynamically; some can be processed only by the precompiler. From among the more well-known SQL commands, the following cannot be dynamically formed:

BEGIN DECLARE SECTION	EXECUTE IMMEDIATE
CLOSE	FETCH
CONNECT	INCLUDE
DECLARE CURSOR	OPEN
DESCRIBE	PREPARE
END DECLARE SECTION	PUT
EXECUTE	WHENEVER

14.8.2. Immediate Execution of Dynamic Commands (EXECUTE IMMEDIATE Command)

Those dynamic SQL program commands containing no parameters, namely those which must be interpreted at once word by word as the program receives them, can be executed by the EXECUTE IMMEDIATE command. Its form is:

►► — EXECUTE IMMEDIATE — host variable — ◄◄

where the actual content of the host variable will be the SQL command to be executed.

The EXECUTE IMMEDIATE command

- produces the executable SQL command from the given character string,
- executes the command, and
- deletes the executed form.

No indicator can be defined for the host variable. Its length varies in COBOL, C, and Assembler, but is fixed in FORTRAN. In PL/1 it can be declared with fixed and varying lengths in the (SQL) DECLARE SECTION.

The character string in the host variable

- can refer only to a dynamically-executable SQL command,
- cannot contain any reference to another host variable,
- cannot begin with EXEC SQL,
- cannot end with END-EXEC,
- cannot contain any parameter, and
- cannot contain any comment.

If the resultant command is erroneous, then it will not be executed.

Information is received about this within the SQL communications area (SQLCA), and SQLCODE indicates it also.

After each of its executions, the EXECUTE IMMEDIATE command generates again the code necessary for the execution of the SQL command in the host variable. So to execute it several times with the same host value, it would be more efficient to produce the SQL command once in a way to be discussed in the following point, in a form remaining constant throughout run time, with the ability to always run unchanged this code sequence.

The EXECUTE IMMEDIATE command permits the entry of non-select type interactive SQL commands into any part of the application program, whether to test the program or to prevent program errors due to faulty data.

An example of this is shown in Fig. 14.7. (see next page).

A non-select type SQL command is read into the host variable named COMMAND-TEXT (for example: DELETE FROM CAR-OWNER WHERE NAME== 'XYZ'). It is executed and SQLCODE is evaluated directly. If an erroneous command is given, then the EXEC-IMMED-ERROR procedure is called. This procedure searches for and destroys the source of the error using ad hoc, error code-dependent, interactive commands. The work continues with the interactive commands until 'END' is read into COMMAND-TEXT.

14.8.3. Preparation and Execution of Dynamic Commands (PREPARE and EXECUTE Command)

The dynamic SQL commands containing parameters — namely, when interpreting, instead of making a word for word translation of the command, some data values come from program variables or are sent there — must first be prepared for the run with the PREPARE command³. The EXECUTE command executes the

³ The capability of SQL commands which can be prepared by PREPARE is restricted. In some languages broader services are available by the use of the DESCRIBE command. For example, the number of parameters also can dynamically change.


```

EXEC SQL
  BEGIN DECLARE SECTION
END-EXEC
77      COMMAND-TEXT      PIC X(80).

EXEC SQL
  END DECLARE SECTION.
END-EXEC.

* EXECUTE NON SELECT TYPE INTERACTIVE COMMANDS
* READ THE SQL COMMAND
  ACCEPT COMMAND-TEXT.
  PERFORM INT-SQL UNTIL COMMAND-TEXT = 'END'.

INT-SQL.
  EXEC-SQL
    WHENEVER SQLERROR CONTINUE
  END-EXEC.
  EXEC SQL
    EXECUTE IMMEDIATE COMMAND-TEXT
  END-EXEC.
  IF SQLCODE NOT = 0 PERFORM EXEC-IMMED-ERROR.
  EXEC SQL
    WHENEVER SQLERROR ...
  END-EXEC.
* RESTORE ORIGINAL ERROR HANDLING OF THE PROGRAM
  ACCEPT COMMAND-TEXT.
  INTSQL-END.

EXEC-IMMED-ERROR.
* ERROR HANDLING BASED ON SQL CODE.
  DISPLAY "INCORRECT COMMAND-TEXT", SQLCODE.
  DISPLAY COMMAND-TEXT.

```

Fig. 14.7. Embedding interactive SQL options into an applications program

so-prepared command. Since the PREPARE-defined code remains during the run, the same command can be rerun several times using varying values by changing the parameters, but without regenerating SQL code.

The syntax diagram of the most often used forms of the PREPARE and EXECUTE commands can be seen in Fig. 14.8. The command name variable is handled by SQL itself; it must not be declared in the program. After executing a PREPARE command, the SQLCODE value should always be examined to learn whether an executable SQL command was actually produced!

The places of the positioned parameters are indicated by a question mark (?) within the command. When executing, the *n*th parameter in the command definition is connected with the *n*th parameter of the USING clause.

In the program the use of non-select type, dynamic SQL commands containing parameters is the following:

- The host variable containing the text of the command is defined.
- A dynamic command is created from it by PREPARE. (Its name is discretionary and must not be declared.)

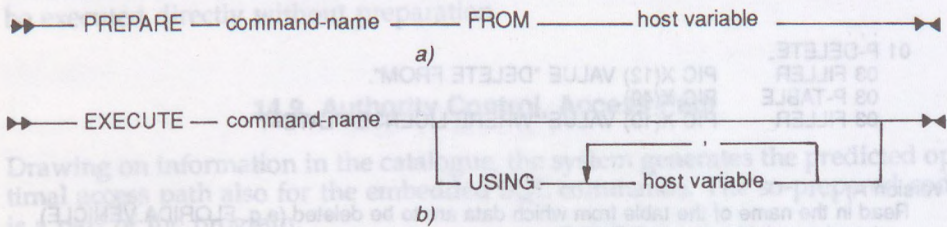


Fig. 14.8. Embedding dynamic SQL commands with parameters into a program

a) Syntax diagram of the creation of the dynamic command,

b) Syntax diagram of the execution of the prepared command

(The syntax diagram is not complete. It includes only the most frequently-used form.)

- The so-named command is executed by `EXECUTE`.
- The parameters are replaced by the host variables in accordance with their position in the command definition and in the `USING` clause.
- The `SQLCODE` value is checked after the execution of both the `PREPARE` and `EXECUTE` commands.

A processing loop where both the owner of the table to be processed and the value of the data to be deleted are parameters is illustrated in Fig. 14.9.

If (the owner of) the table were not changed, then there would be no need for a dynamic command. The row to be deleted could be identified by a host variable in a static command.

The above method is applied, for example, when several users execute various operations with the *same* program, but each of them uses his *own tables'* data.

In version *a*) of the figure, processing concerns the same county throughout. (The county is the owner of the table. Its name qualifies the `VEHICLE` table.) It is therefore sufficient to read it once. In version *b*), any county's data can be executed; that is why the command has two parameters.

In select-type dynamic commands no `INTO` clause is allowed. The selected row(s) is always handled with the cursor.

The steps of preparing and processing a command are:

- The input SQL command is entered into a host variable.
- It is prepared for execution (`PREPARE`).
- A cursor is declared to it (`DECLARE`).
- The cursor is opened (`OPEN`). If the command contains parameters, then their actual values must be passed to them here using the host variables, defined in the `USING` clause.
- In the loop the rows are read one by one into the host variables (`FETCH`).
- The rows are processed individually.
- After processing the last row, the cursor is closed (`CLOSE`).
- After every SQL command, SQL return codes are handled.

Parameters can also be used in the command (with certain restrictions).

The preparation and processing of a dynamic select command is shown in Fig. 14.10. Although the command does not contain parameters directly, due to


```

01 P-DELETE.
03 FILLER      PIC X(12) VALUE "DELETE FROM".
03 P-TABLE     PIC X(49).
03 FILLER      PIC X(19) VALUE "WHERE LICENSE-PLATE=?".

```

version A) . . .

Read in the name of the table from which data are to be deleted (e.g. FLORIDA.VEHICLE) into the variable named P-TABLE

PREPARE COMMAND-TEXT FROM :P-DELETE

in a loop:

read license-plate into LICENSE-PLATE

EXECUTE COMMAND-TEXT USING :LICENSE-PLATE.

or

version B):

MOVE "?" TO P-TABLE.

PREPARE COMMAND-TEXT FROM :P-DELETE

in loop:

read license-plate and the table-name into LICENSE-PLATE

EXECUTE COMMAND-TEXT USING :LICENSE-PLATE.

Fig. 14.9. Preparing and executing a dynamic SQL command. In the command the owner of the table (as well as the name of the registering county) and the license plate of the deletable vehicle are changed
EXEC SQL and END-EXEC, as well as the checking of SQLCODE, are omitted for clarity

```

01 SELECT-TEXT . . . VALUE
   "SELECT LICENSE-PLATE, MODEL, MAKE, CAR-OWNER FROM V&O VO,
- VEHICLE V WHERE V:LICENSE-PLATE = VO:LICENSE-PLATE AND
- OWNERSHIP-END BETWEEN '1992-12-01' AND '1992-12-31' "

```

```

WHENEVER . . .
PREPARE SELECTION FROM :SELECT-TEXT.
DECLARE READ-CURSOR FOR SELECTION.
OPEN READ-CURSOR.

```

```

PERFORM PROCESS-DATA UNTIL . . .

```

```

CLOSE READ-CURSOR.

```

PROCESS-DATA SECTION.

FETCH READ-CURSOR INTO . . . (host variables with indicators)

processing. . .

Fig. 14.10. Creation and execution of a dynamic SELECT command
The command retrieves for processing the data of cars sold between December 1, 1992 and December 31, 1992, ordered by date of sale, and within this, license plate

the select result connections must be established with host variables; so it cannot be executed directly without preparation.

14.9. Authority Control, Access Path

Drawing on information in the catalogue, the system generates the predicted optimal access path also for the embedded SQL commands. The so-prepared code is a part of the program.

With static SQL commands, the access path is defined by the system at compilation time. The optimization is carried out on the basis of the options available at that time. If changes occur later in the data base allowing more efficient access paths (for example, a new index is created, or conditions change due to increased table sizes, enabling a more selective solution), then they can be used by the program only after recompilation. However, if any access paths built into the program come to an end (for example, an index was deleted), but data are still accessible via another path, then at run time, the system generates automatically a new optimal access path based on current conditions.

(Naturally, if changes occur in the data base which make data access impossible (for example, a referred table is dropped), then commands applying to it will not be executable. This is indicated by a negative SQLCODE. It depends upon the program's error handling routine how it will proceed in that case.)

Generally, with static commands, the creator of the program is authorized to execute the operation. He can grant it to others with the GRANT RUN command which permits the run of the program. There are systems (e.g., SQL/400) which allow the execution of static operations whether the creator of the program or its user has specific authority.

If the creator of the program has not yet received the requisite authority at compilation time, or something has changed since the compilation in the authority of tables/views used by the program, then SQL marks the code containing the access path. During the program's next run, it checks the authorities and consequently either restores the original status or postpones further the use of the access path, making it dependent on the fulfillment of authority conditions.

For dynamic SQL operations, just as with interactive ones, both access path generation and authority checking always take place during the program's run, immediately before execution. The user of the program must possess the authority to execute the dynamically-defined SQL operation. If he does not, then the operation will not be completed. A negative SQL code indicates this, but still allows the program to run further. Naturally, the user is responsible for how the program proceeds further in this case, to wit, how it ensures the making up or substitution of the operations which were not executed for data protection reasons, or how it evades them.

Summarizing the above, data protection and integrity-ensuring mechanisms provide the same comprehensive safety when accessing the data base through a program as during the use of interactive operations.

APPENDIX

A) For Further Reading

This book is intended as a general, introductory summary for professionals who are already involved or would like to be involved in data bases. Therefore no references have been given for particular topics or specific points. At the same time, as the author has already written some books and read even more, he knows from personal experience that no single book can include everything that the author wanted to say, nor that the reader would like to know about the given topic. For this reason, here follows a listing of some books which can provide useful help to the reader who would like to acquaint himself with data base tasks in more detail or from other aspects.

This book deals for the most part with the relational model. It was often emphasized that in this model the ordering of the rows has no significance. Bearing this in mind, the following list is not a ranking list; it is only a result relation, ORDERED BY the name of the authors.

Bernstein, P.A. – Hadzilacos, V. – Goodman, D.:

Concurrency Control and Recovery in Database Systems, Addison-Wesley Publishers, Reading, MA, 1987

Burch, J.G. – Grudnitski, G.:

Information Systems: Theory and Practice, J. Wiley, New York, NY, 1986

Ceri, S. – Pelagatti, G.:

Distributed Databases: Principles and Systems, McGraw-Hill, New York, NY, 1984

Codd, E.F.:

The Relational Model for Database Management, Version 2, Addison-Wesley Publishers, Reading, MA, 1990

Curtis, G.:

Business Information Systems, Addison-Wesley Publishers, Reading, MA, 1989

Date, C.J.:

Relational Database: Selected Writings, Addison-Wesley Publishers, Reading, MA, 1986

Date, C.J.:

Relational Database Writings 1985–89, Addison-Wesley Publishers, Reading, MA, 1990

Date, C.J.:

An Introduction to Database Systems, Fifth Edition, Addison-Wesley Publishers, Reading, MA, 1990

Elmasri, R. – Navathe, S.B.:

Fundamentals of Database Systems, Benjamin/Cummings, Redwood City, CA, 1989

Gaydasch, A.:

Effective Database Management, Prentice-Hall, Englewood Cliffs, NJ, 1988

Halassy B.:

Adatmodellezés a rendszertervezésben, (Data Modelling in Systems Analysis) SZÁMALK, Budapest, 1983

- Hicks, J.O. Jr:
Management Information Systems, A User Perspective, Second Edition, West Publishers, St. Paul, MN, 1987
- McNurlin, B.C. – Sprague, R.H.:
Information Systems Management in Practice, Prentice-Hall, Englewoods Cliffs, NJ, 1989
- Merret, T.H.:
Relational Information Systems, Reston Publishers, Reston, VA, 1984
- Nijssen, G.M. – Halpin, T.A.:
Conceptual Schema and Relational Database Design — A fact oriented approach, Prentice-Hall, Englewood Cliffs, NJ, 1989
- Quittner P. – Kotsis D.:
Számítástechnika rendszerszervezőknek, (Computing Techniques for Systems Analysts) Akadémiai Kiadó, Budapest, 1989
- Smith, P.D. – Barnes, G.M.:
Files and Databases: An Introduction, Addison-Wesley Publishers, Reading, MA, 1987

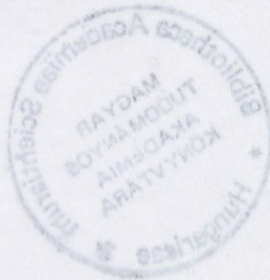
B) Index

Bold and underlined numbers indicate the starting page of the detailed description of the given topic.

- access methods 73
- access path 31, 33, 47, 65, 73, 115, 130, 138, 172,
223, 225, 228, 258, 285
- access time 68
- addressing 73
- administrator user 25
- ALTER TABLE 226
- alternate key 150
- application programmer 25
- application software 21
- area 129
- assignment 168
- authority 194, 215, 227, 243, 285
-
- B⁺ tree 93
- binary tree 114
- block 69
-
- CANCEL command 263
- candidate key 150
- cardinality 146
- cascade deletion 154
- catalogue 38, 53, 192
- clustering 72, 75, 97, 98, 140, 156, 224
- column 143, 193
- column catalogue 193
- column function 236
- commands 1
- COMMIT command 253
- communication area 271
- composite key 83, 96
- computing center 26
- conceptual level 31, 33, 43, 59, 127
- connecting records 125
- consistency 54
- control commands 214, 249
- CREATE INDEX command 223
- CREATE TABLE command 216
- CREATE VIEW command 220
- cross product 160
- cursor use 268, 275, 276
- customization 258
- cylinder 67
-
- data 19
- database 29
- database administrator 22, 44, 39, 40, 55, 65, 72,
115, 130, 132, 175, 187, 215, 250
- database architecture 31
- data condensing 99, 133
- data definition commands 213, 215
- data definition language 34, 127
-
- data dictionary 38, 53, 192
- data fragmentation 51
- data independence 30, 42
- data item 69
- data manipulation commands 213, 227
- data manipulation language 35, 43, 116,
128, 135
- data model 31, 59
- data models 1
- data security 39, 191
- DBTG proposal 127
- deadlock 55, 252
- decision maker user 25
- decomposition 181
- degree 144
- delete anomaly 179, 183
- DELETE command 244, 278
- deletion 154
- dependency 153, 176, 181, 184, 195
- difference 67
- disk 163
- distributed database 40, 46, 50, 196
- direct access 90, 94, 98
- distributed system 46
- document-type database 29, 201, 240
- double linked list 79
- DROP command 225
- duplicate indices 84, 224
- dynamic SQL commands 268, 279
-
- efficiency 139, 156, 172, 228, 234, 243, 260
- embedding SQL command
in programs 265
- encoding 101, 103
- EXECUTE command 282
- EXECUTE IMMEDIATE
command 280
- EXISTS 240
- EXPLAIN command 259
- extend 166
- external level 31, 33, 59, 127
-
- fact-type data base 29
- FETCH command 276
- field 69, 130
- first normal form 178
- foreign key 149, 151, 218, 244
- FORMAT command 257
- fourth normal form 190
- fragmentation 51
- full select 229, 233
- functional dependency 179, 190

- GRANT command 249, 285
- grouping 167, 170, 233, 236
- handling particular events 272
- hardware 20
- hashing 98
- head selection 68
- HELP command 263
- hierarchical data base 115, 117
- hierarchical model 59, 109
- higher normalization forms 188
- hit ratio 206
- host language 23, 36, 265
- host variables 269
- Huffmann code 100
- IDMS 128, 135
- IMS 109, 115
- IN 241, 262
- index 81, 95, 102, 156, 223, 234, 260
- index and key catalogues 194
- index area 89
- index table 82, 83, 88, 93, 202
- indexed-sequential organization 88
- indicator 270
- Information Center 26
- information infrastructure 17
- information system 17
- information technology 18
- inner level 31, 33, 59, 65, 115, 227
- insert anomaly 181, 183, 184
- INSERT command 246
- insertion 153
- insertion of a new record 91
- integrated data base 39
- integrity 148
- internal level 31, 33, 59, 65, 115, 127
- intersection 163
- join 164, 168, 232, 235
- key: 69, 74-76, 95, 115, 148
- LIKE 240, 261
- list structure 76, 132
- lock 251
- log 253
- many to many relationship 63, 123
- member 131
- menu oriented user 25
- merge 105
- modification anomaly 179, 183, 184
- network data model 59, 123
- non-programmer user 25
- normal form, 1NF 178
- normal form, 2NF 179
- normal form, 3 NF 184
- normal forms, higher 188
- normalization 175, 188
- null-value 147, 154, 166, 217, 236, 238, 242, 270
- one to one relationship 63
- one to many relationship 63, 114
- operating principle 21
- operating system 20, 23
- operator commands 255
- ORDER BY 262
- order key 69
- other catalogues 194
- outer join 166
- overflow 89, 91
- owner 131
- page 69, 73
- personal computers 45
- physical level 31, 33, 59, 65, 115
- physical record 70
- physical storage structure 65
- pointer 76, 80, 139, 141
- positioning time 68
- precompiling 37, 265, 268
- PREPARE command 282
- preprocessing 37, 265, 268
- primary data area 89
- primary key 69, 83, 147, 149, 150, 218, 224
- privilege 195, 249
- programming language interface 36
- projection 159, 168, 221
- quantified select 241
- read-only view 222, 243
- record 69, 130
- reference condition 153, 247
- referenced relation 151, 247
- referring relation 151, 247
- relational model 60, 143, 198
- relational operations 157, 191
- relationships 62
- renaming 158
- reorganization 69, 72, 75, 92
- response time 18, 61
- restricted deletion 154, 247
- restriction 158, 168, 221
- report generator 37
- REVOKE command 251
- ring 79
- ROLLBACK command 254
- rotation time 68
- row 143
- search key 69
- scalar function 237
- schema 34, 115, 127

- screen formatter 37
- second normal form 180
- secondary keys 69
- security 39, 75, 191
- segment 115
- SELECT command 229
- SELECT INTO command 275
- select list 232
- self-contained systems 23, 26, 36
- sequential access/organization 75
- sequential indices 93
- serial access/organization 74, 80
- set 131
- SET command 258
- setting to null value 154
- singular set 131
- software 20
- sorting 101
- SQL 209
- SQLCA 271, 273
- SQLCODE 271, 273, 283
- SQL command types 213, 266
- SQL functions 236
- static SQL commands 267, 274
- storage structure 33, 65, 141
- subschema 34, 115, 127, 134
- subselect 229, 241
- summarize 167
- synchronization 55
- synonym 98
- syntax diagram 210
- syntax notation 210
- table catalogue 193
- table 147, 193, 216
- text retrieval systems 29, 201, 240
- thesaurus 205
- third normal form 184
- throughput 18
- track 67
- transparency 51
- tree structure 109
- truth table 242
- union 161, 233, 262
- update anomaly 181, 183
- UPDATE command 247, 278
- user 24, 60
- utilities 38
- variable length 100
- view 147, 190, 220
- WHENEVER command 272
- WHERE clause 232, 238, 244, 247



3058, —

One of the main application areas of today's computers is information technology. Data bases and database management systems are used for the organization and arrangement of data into useful information, that is, for information management. Databases have become common in every field of economy, science and technical domain. Many tasks are regularly solved with the help of database software on mainframes and PCs as well. Information revolution has changed our life just like industrial revolution did two centuries ago.

Due to this general use a great many misbeliefs and false hopes have emerged in this field. Some regard a simple departmental payroll file as their "database" and do not understand the importance of a good database management system. Others are expecting the purchase of such a software to be a panacea which will provide an immediate solution even in the most complicated decision-making processes.

This book presents the characteristics of modern data bases and database management systems. How data must be stored, what relationships can be created between them, and what software methods are available to store data, keep them up-to-date, and retrieve them quickly, and at the same time ensure their security, accuracy, integrity, and unanimity.

Its aim is to present a realistic view of this topic, to give practical guidance to both database managers and users alike on how to manage their data and deal with information systems.

PÁL
QUITTER



A PRACTICAL APPROACH
TO DATABASE SYSTEMS

659969

