Faculté d'informatique - University of Namur

# NDBS

## A simple database system for small computers

Jean-Luc Hainaut

**Draft, unrevised version**

September 1986

University of Namur, Faculté d'Informatique
Rue Grandgagnage, 21 B-5000 Namur, Belgium
jlh@info.fundp.ac.be • http://www.info.fundp.ac.be/libd

3 mars 2011

## Foreword

This text has been recovered from various materials written in the eighties (Mac Word documents) and roughly assembled into a unique FrameMaker document. The result would require in-depth polishing, but, to be frank, I currently have no spare time to carry out this task. Sometimes in the near future perhaps !

J-L Hainaut - March 2011

The current document describes NDBS, a small database management system specifically aimed at helping programmers to write efficient PASCAL programs that work on large and complex data structures. NDBS has been designed and developed at the Institut d'Informatique of the University of Namur.

Besides the introduction, the document comprises four main sections covering the following topics :

Part 1 - what a database is and how to define one

Part 2 - basic database programming

Part 3 - programming with a high-level database language

Part 4 - case studies.

NDBS has been programmed and is maintained by

D. ROSSI (database handler)

L. GOFFINET (schema processor)

E. Van ROSSUM (ADL-PASCAL precompiler)

# Table des matières

8. **DGL : a language for the management of data bases**
9. **LOAD / UNLOAD UTILITIES FOR NDBS DATA BASES**
10. **REPORT GENERATOR FOR NDBS - VERSION 2**
11. **SCREEN GENERATOR FOR NDBS**
12. **SQL CONVERTER FOR NDBS DATA BASES**
13. **NDBS - EVALUATION DE PERFORMANCES**

# 1. INTRODUCTION

## 1.1 What is NDBS ?

NDBS, or Network Database System, is a comprehensive database environment for the rapid development of complex database applications in PASCAL on microcomputers.

The NDBS environment comprises three main components, namely a database handler for PASCAL programs, an interactive data dictionary/schema processor and a high-level language pre-processor for PASCAL programs.

That document describes version 1 of NDBS.

## 1.2 The state of the art in data management on microcomputers

The design of NDBS is based on three general purposes. The system should be **simple** enough to allow anyone to write easily programs that use and manage databases of any complexity. Writing database programs should be at least as simple as using standard sequential files or ISAM packages. The system should be **efficient**, particularly in disk access (I/O), processing time (CPU) and main memory requirement. If needed, several technical parameters can be set to fine tune database management. However, that tuning is optional, since default settings lead to satisfying performances in most situations. The system should be **complete** and provide a comprehensive environment for building simply and efficiently database programs of any complexity.

### 1.2.1 The conventional data management approach

Most application programs typically use a small number of simple files, reading some, and writing into others. Current programming languages offer very primitive file organisations, namely **sequential organisation** and **direct (or random) organisation**. So, once an application has to deal with more complex data in large volume, the programmer will spend considerable effort in writing complex algorithm to manage a set of files dedicated to his application. He generally can rely on such toolkits as ISAM packages, providing him with **indexed files**. Though allowing a higher level of power in file management, that approach still suffers severe drawbacks and limitations as far as large and/or complex programs and data structures are concerned. These drawbacks are the following.

***Low descriptive power***. Complex data structures describe real world objects and their relations. For instance, each customer is characterised by his number, his name, his address, customers send orders, orders specify quantities of products,

products are supplied by suppliers, products and invoices are sent to customers, etc. Traditional file management systems (FMS) cannot ensure a clear, simple and consistent representation of such information since data will be stored in a large number of separate and independent files and no functions are provided to maintain inter-file relationships.

***Low processing power***. The developer is provided with generally low level processing functions such as opening a file, sequentially scanning a file, getting a record by key, etc.

***Complex programming***. Due to the lack of powerful descriptive and processing functions, the management of the data structures mainly relies on the skill of the programmer. The programs include large sections of code doing the data management tasks that are not done by the file management system. Navigating through sophisticated data structures and ensuring correctness of stored data implies building complex algorithms, an activity which is prone to logical errors (that kind of errors that are detected too late, when the invoices have been sent to customers, or when, several weeks later, data integrity appears to be corrupted). Obviously enough, database programming with such tools is beyond the scope of the novice programmer when large database applications are concerned.

***Low performance***. Using traditional file management systems for organising data with many relationships usually leads to poor performance. For instance, getting customer information corresponding to an order is done by accessing the CUSTOMER file through an index, an operation that generally costs at least 3 physical I/O for a realistic number of CUSTOMER records.

***Technical limitation***. Though any information structure can theoretically be mapped onto traditional file structures, some technical constraints prevent the developer from designing clear data structures and straightforward program design. The main limitation is the number of system files a program can open simultaneously (remember that a user file may consist of several system files if it is associated with one or several indices). A typical database structure will be mapped into several dozens of data files, while some operating systems accept no more than 15 files simultaneously open.

***Development costs***. Both the design step and the programming step require highly skilled programmers who master complex data structures and complex algorithms. On the other hand, the lower the development tools, the longer the development time.

***Maintenance costs***. Since a large part of the application programs is devoted to complex data management, these programs are large, tricky and therefore difficult to read. Their maintenance is more difficult and inevitably more costly. A much more severe problem will occur when several application programs use the same database. Without drastic programming discipline (a concept generally unknown among small system developers), many data management functions are duplicated in each application program, increasing the probability of functional discrepancies among them

(leading to data inconsistency), but above all making program maintenance a very hazardous enterprise.

### 1.2.2 The DBMS approach

The traditional solution to the problems mentioned above is using a Database Management System (DBMS) for the management of complex data structures. Two ways are currently followed in the microcomputer market, the relational DBMS technology and the network DBMS technology.

***Relational DBMS*** (ORACLE, INGRES, various SQL products, future OS/2 DB component, dBase, Knowledgeman, etc) present data as a collection of flat files or tables. Powerful interactive query languages (such as the standard-to-be SQL) allow the user to obtain data from several tables without worrying about access path or index. The DBMS includes a sophisticated optimiser that determines the cheapest way to get the data requested by the user according to the access structures available.

***True relational DBMSs*** (ORACLE, INGRES, SQL DBMS) are very large pieces of software (several Mbytes); they need powerful machines featuring large core memory, quick hard disk and high power processor (MC68020, Intel 80386, NS-32032, or that of a minicomputer). Moreover, they are very expensive, ranging from 3,000 to 30,000$. Some relational DBMSs provide the programmer with a programming language interface. A puzzling conclusion comes when examining large application programs that work with a relational interface. They seldom make use of the full power of the DBMS optimizer. A relational database program often looks like the equivalent ISAM program. However, that small improvement has to be paid by a very high and expensive overhead in CPU, main memory and external memory resources. True relational DBMS are not yet adequate tools for developing applications for current small personal systems, on which most of their their functions are spoiled and that cannot provide the power these DBMSs need.

***Pseudo-relational* DBMSs** (dBase, R:Base, Knowledgeman, etc) also present data as a collection of tables. They generally offer a stand-alone development environment allowing a programmer to write fairly large application programs. A table is a file with which several indices can be associated. The programming language offers global functions, operating on whole files and producing new files by extracting, merging, joining, etc, together with low level functions ("get next record") similar to that of ISAM interfaces. There is no optimizing function, so the programmer must explicitly specify the access methods that have to be used in order to get the needed data. A conventional programming language interface is seldom available (some low level routines are available for accessing dBase files from a C program). As far as large application programs are concerned, these DBMSs do not offer many more functions than conventional file management systems.

***Network DBMSs*** use is decreasing on mainframes, where they are progressively replaced by relational DBMSs. On microcomputers however, they surprisingly seem to gain popularity, with products such that MDBS-III, db-VISTA and Mac REFLEX. We shall try to analyse some of the reasons of that phenomenon.

A first reason is the increasing complexity of microcomputer applications, approaching that of mainframe applications, both in business and scientific domains (CAD databases for instance). Such programs need much more sophisticated data structures than several years ago.

Unfortunately, conventional file management systems are unable to cope with such complexity, and true relational DBMSs are inadequate in the current state of small microcomputers, specially as to their resource use.

On the other hand the network DBMSs are much less resource consuming than their relational counterpart, while offering a more powerful and clear expression of complex data structures, together with better performances in disk I/O. A network DBMS generally does not offer query optimization. However, it allows the explicit representation of relations between entities (hence the name *network*, describing the type of schema it allows to define).

These advantages, however, do not come without drawbacks. The main problem is the complexity of the programming interface. The data model, according to which data will be described, is richer and therefore more complex than mere relational tables. Consequently, the number and the complexity of the primitives (procedure) that the programmer must use in order to manage and get data from the database is sometimes very high. The current leading contender offers more than 100 primitives. Finally, the cost and the licencing policy are not always attractive for small developers.

A fourth kind of DBMS is beginning to appear, based on the Entity-Relationship (ER) data modelling. In an ER DBMS, the data are organised according to a model that generalizes the Network model, making it both more powerful and more natural. Moreover, the user interface and the programming interface are as powerful as those of true relational DBMSs.

## 1.3 The NDBS environment

NDBS is a member of Network and ER DBMSs. Its characteristics have been set by analyzing the causes of inadequacy of the DBMSs currently available on microcomputers.

The data model (i.e. the kind of structure according to which the data can be organized), is simple, powerful and natural. Deriving from the current Entity/Relationship approach, of which it is a subset, it can be understood and used by non technical users. A clear distinction is established between the semantic structure of data and their technical parameters. The latter can always be ignored.

The programming interface which is given to the programmer is much simpler than the current state in that domain. The most complex programs can be written by using 12 database functions only. Database functions are made easy to use by their absence of side effect, and the total visibility of the programming objects pertaining to database operation. A database function has no implicit arguments (such as the

static currency registers or indicators of classical DBMSs) that limit or make it difficult to deal with situations in which several objects of the same type are processed in parallel. Moreover, the functions offer a regular interface totally complying with the PASCAL data types and programming rules. As an example, a complete, stand-alone program of "bill-of-material" explosion can be written in 20 statements only (see BOM1ADL example in part 4). That program uses no trick and is a straightforward translation of the natural language expression of the problem. That kind of recursive problems is generally considered as very difficult to program with all the current DBMSs, including the true relational systems.

The technical architecture of NDBS has been kept simple as well. A multi-layer approach allows the easy maintenance and extension of the functions.

However, NDBS is not a toy DBMS. Self-describing data files, compact variable-length data storage, bi-directional record chaining, B-trees indices, LRU buffer management, parametrised storage schemes and various performance tuning facilities are some examples of characteristics that make NDBS efficient for managing complex and large database and that are only found on complex DBMSs. In its current version, NDBS still lacks some functionalities that are generally provided by more powerful DBMS. NDBS is currently a single-user system with no access control; moreover, transaction and recovery management is still very primitive. That option is consistent with the objectives of simplicity, low-cost and efficiency in the context of small personal systems.

NDBS Version 1 includes three main components.

The **database handler** is a set of procedures that give the programmer the database functions to access a database, to find data according to several criteria and to update data.

The **schema processor** is a program that allows the user to give the description of databases, to modify, display and print database descriptions, to generate operational databases from their description. The schema processor is interactive and uses its own database to store the database descriptions; it is therefore a simple but true **data dictionary system** as well. The schema processor is an NDBS program and the data dictionary is an NDBS database.

The high-level language pre-processor allows the programmer to write database programs in a high-level language called ADL-PASCAL (standing for Access Algorithm Description Language embedded in PASCAL). The programs are more compact and clearer than standard programs written with the database handler basic interface. In particular, some kind of optimization is automatically performed by the pre-processor. ADL-PASCAL is close to some pseudo-codes used in program design, and to some 4-th generation languages found on mainframes. The pre-processor transforms an ADL-PASCAL program into a pure PASCAL program making use of the basic database interface. The pre-processor is a NDBS program that uses the data dictionary.

# Part 1

## 2. WHAT A DATABASE IS AND HOW TO DEFINE ONE

### 2.1 What is a database ?

A database is a collection of data which describe a subset of the real world. We shall call that subset the **application domain.** Personnel management, a marketing departement, a chemistry laboratory, a school, a library, an architect office are examples of such application domains.

When analyzing an application domain, we shall structure it in terms of entities, entity attributes and inter-entity relationships. An **entity** is any individual concept that is perceived as being important in the application domain ; an employee, a product, a drug, a student, a borrowing, an invoice, a purchase, an accident, etc. Entities are classified into **entity types**. For instance, EMPLOYEE is the collection of all the employees. The collection of the employees is constantly changing, but the notion of EMPLOYEE entity type is a stable, static concept for the personal management departement.

An **entity attribute** is any property that characterizes all the entities of a given type. Each employee has a number, a name and an address. We therefore associate the attributes NUMBER, NAME and ADDRESS with the EMPLOYEE entity type. Some attributes can play a special role for their entity type. No two employees have the same number, no two department have the same name. We shall say that NUMBER is an **identifier** of the EMPLOYEE entity type (the same for NAME of DEPARTMENT).

In general, entities are not isolated in the application domain. Each employee belongs to a department, each product can be obtained from a supplier, etc. These situations correspond to **relationships** between entities. The collection of similar relationships which bear the same meaning is called a **relationship type**. In the example above, the connections can be described by the BELONGS-TO relationship type between EMPLOYEE and DEPARTMENT, and the IS-SUPPLIED-BY relationship type between PRODUCT and SUPPLIER. In the former example, each employee can be associated with **only one** department, and each department can be associated with **any number** (say **N**) of employees via the BELONGS-TO relationship type. Such a relationship type is called **1-N**, or *one-to-many*, from DEPARTMENT to EMPLOYEE, and, conversely **N-1**, or *many-to-one*, from EMPLOYEE to DEPARTMENT.

That kind of system analysis is based on a subset of the Entity-Relationship modeling approach that is now standard in database design and database description. It is considered as being simple enough to be used by non technical persons, though powerful enough to be adopted for designing very large and complex databases.

Besides the descriptive objective of a database that has been discussed, we should mention the well known and more obvious purpose : a database must be an **efficient and reliable data server** for a large class of needs.

## 2.2 What is a NDBS database ?

A NDBS database is organised according to the descriptive rules explained above. Each entity of the application domain is represented by some sort of data stored on a non volatile, high capacity medium, such as a magnetic disk. Entity attributes are represented by PASCAL data types. Relationships are represented by connection between entity representations. To make things simpler though perhaps less rigorous, we shall call **NDBS entity** (or, from now on simply **entity**) the representation of an application domain entity. A NDBS database therefore contains a variable number of **entities** with their **attribute values** and their **relationships**.

The specification of the entity types, entity attributes and relationship types of a database is called the **schema** of that database. Knowing the schema is all that is needed to write programs that use and modify data in a database.

A NDBS database schema accepts a simple and intuitive graphical representation. Each entity type is represented by a box containing the name of the entity type and the names of its attributes (see figure 1.1). An identifier is specified by underlining the identifying attribute.

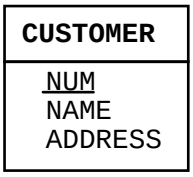


**Figure 2.1 - Figure 1.1** -The entity type CUSTOMER with three attributes NUM, NAME and ADDRESS. Num is an identifier for CUSTOMER.

A relationship type is represented by an arc joining the boxes of the entity types (see figure 1.2). A label gives the name of the relationship type. The 1-N direction is indicated by a small triangle sticked on the arc.

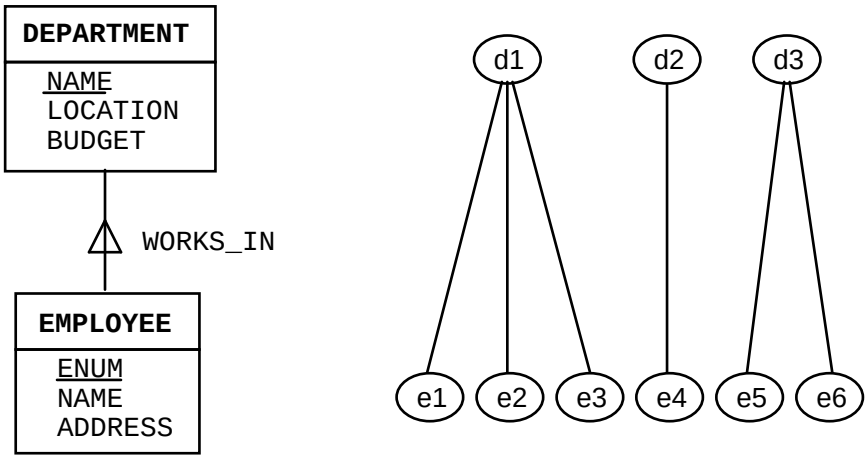


**Figure 2.2 - Figure 1.2** -Representation of the relationship type **WORKS-IN** between entity types DEPARTMENT and EMPLOYEE (left). The 1-N direction is indicated by a triangle which mimics the way the arcs connect each DEPARTMENT entities, such as *d1*, *d2* and *d3*, to EMPLOYEE entities such as *e1* to *e6* (right).

The schema of figure 1.3 is an example of a more complex database that could describe an application domain analyzed as follows.

The application domain is about customers that order quantities of products. The relevant entities can be classified into four types. The **CUSTOMER** entity type describes all the customers that have been registered so far. The **PRODUCT** entity type represents the products that are currently available and that can be bought by customers. The **ORDER** entity type describes all the valid orders that have been placed by customers, and that haven't been satisfied yet. The **LINE** entity type describes items of an order that specify one product in a certain quantity.

The entities are connected to each other through relationships classified as follows. Each ORDER is associated (through **CO**) with the CUSTOMER who placed it. Each LINE is connected (through **OL**) to the ORDER in which it appears and (through **PL**) to the PRODUCT it specifies.

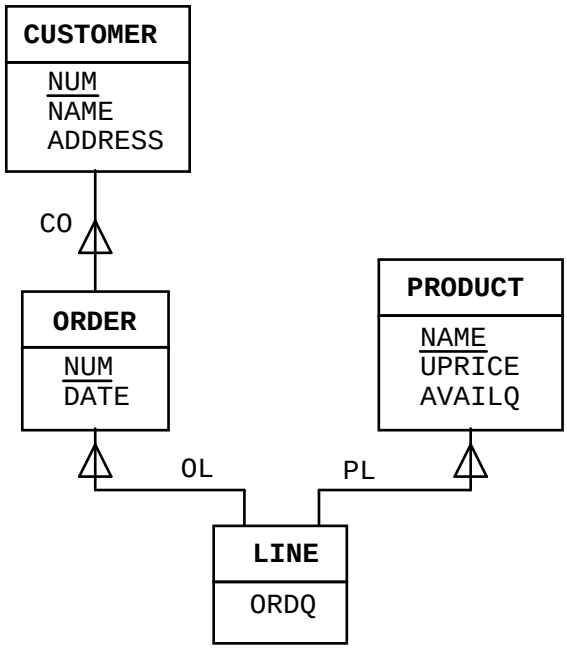


**Figure 2.3 - Figure 1.3** -Schema of a database describing an application domain in which customers order quantities of products.

Each CUSTOMER is characterized by an identifying number (**NUM**), his(her) name (**NAME**) and his(her) address (**ADDRESS**). A PRODUCT is characterized by its name (**NAME**), which is an identifier, its unit price (**UPRICE**) and the available quantity (**AVAILQ**). Each ORDER is characterized by its identifying number (**NUM**) and its date it was received (**DATE**). A LINE is only characterized by the quantity of the product that is requested (**ORDQ**). A LINE entity has no explicit identifier.

## 2.3 The logical components of a NDBS schema

A NDBS schema describes a database, entity types, entity attributes, entity identifiers and relationship types.

A **database** is given a name which is a valid file name in the operating system of the computer. That name cannot have an *extension*. On a MS-DOS or PC-DOS machine, any character string from 1 to 8 characters, without spaces, dots, etc, can be a database name. A database may be available in several versions. These versions are different databases, with the same name, but distinguishable by their version number. Their schemas need not be the same. A database is therefore identified by its name and its version number.

An **entity type** has a name. An entity type name is a valid PASCAL name with a length which is 1 less than the maximum. No two entity types of a schema have the same name.

An **entity attribute** has a name which is a valid PASCAL name. The attributes of an entity type have different names. Different entity types may have attributes with the same name. An entity type need not have attributes. If it has at least one attribute, one of them can be the **entity identifier**. An attribute is defined as a valid PASCAL data type. For now however the schema processor deals with *integers, char, boolean, reals* and *strings* only.

A **relationship type** has a name which is a valid PASCAL name. It is defined between two entity types. These two entity types need not be distinct. For instance, a relationship type named HAS_SONS, and describing the father/son family structure between male persons can be defined between the entity type MAN and itself. Indeed, a man's sons are men too; and a man's father is also a man. Such an entity type is sometimes called *recursive*. It is graphically represented with a loop in which the entity type is included (see example 4.2 in part 4).

In a program, relationships will be used to obtain entities which are logically connected to another one. In the example of figure 1.2, once the program has got a DEPARTMENT entity, it can ask for the EMPLOYEE entities that are linked to it through relationships WORKS_ON. Conversely, starting from an EMPLOYEE entity, the program can ask for the DEPARTMENT entity which is connected to it. The program is said to *use the path* from DEPARTMENT to EMPLOYEE, or the path from EMPLOYEE to DEPARTMENT. As far as the database is concerned, the program *navigates* through the database, following the paths that correspond to the relationship types. A relationship type (say WORKS_IN) offers two **path types**, just like a road linking cities A and B offers two ways of following it : from A to B, and from B to A. A 1-N path type (from DEPARTMENT to EMPLOYEE), and its inverse, N-1 path type (from EMPLOYEE to DEPARTMENT) are associated with a relationship type. A path is directed from its **origin** entity towards its **target** entities.

## 2.4 The physical components of a NDBS database

These concepts should normally not be known by the programmer, since the database schema is all he/she needs to write a program. Moreover, they could change from one version of NDBS to another.

**The novice or hurried reader can skip that section without problems when defining a database and writing programs**.

The concepts are useful, however, for the design of an **efficient** database (remember that a database must be an efficient data server as well; see 1.1), and when a programmer wants to foresee the performance of a given program.

The internal (*physical*) representation of an entity is a byte string called the physical record of the entity, or **entity record**. An entity record contains the attribute values

of the entity, if any. In addition, it includes technical data needed by the NDBS routines to manage the database. Some of these data are the entity type code, sequential access pointers and path pointers. The length of an entity record is obtained as follows (NDBS version 1):

$$\text{entity record length} = \text{total attribute length} + 7 + 6 \times \mathbf{n}_{1N} + 9 \times \mathbf{n}_{N1}$$

where $\mathbf{n}_{N1}$ is the number of 1-N path types the origin of which is the entity type

$\mathbf{n}_{1N}$ is the number of N-1 path types the origin of which is the entity type

**Expression 1.1** - Computing the record length of an entity type

For instance, if we suppose that the total attribute length of CUSTOMER entity type of figure 1.3 is 52, the entity record length is 52 + 7 + 1x6 = 65 bytes. The entity record length of LINE entity type is 2 + 7 + 2x9 = 27 bytes.

The data are stored in an extensible **data file** organised in **pages**. A page is a fixed-length frame, typically 1 Kb long for example, in which entity records are stored, whatever their (entity) type. A page is identified by its page number. The records are stored so that they do not span several pages. So, the entity record length cannot exceed the page length. Page numbers range from 1 to 65,000.

The entity records are stored in the pages according to a **storage scheme** which is specific to each entity type. The records of an entity type are stored in a given **page range**, i.e. in one of the pages ranging from one page number to another page number. The default page range comprises all the pages of the data file. Version 1 of NDBS offers two storage scheme, that is two different ways to choose the page in which a new record will be stored. These are the random storage scheme and the clustered storage scheme.

In the **random storage scheme**, the page is chosen at random in the page range, so that the records are distributed as uniformly as possible in the page range, leaving space between them. In **clustered storage scheme**, the record is stored in the page of the last record that has been accessed to or modified. With that scheme, entity records that are created in burst tend to be stored in contiguous pages (clusters of records of the same type). The sequential access to them will be very efficient.

In both storage schemes, if the page chosen cannot accomodate the record, the next pages are searched for free space. If no space is found in the page range, the search goes on outside. If needed, new pages are appended to the data file. Therefore, the only limit to the extension of the database is the disk space available.

An example of the use of the storage scheme is based on the schema of figure 1.3. Let's suppose that CUSTOMER entity records are stored according to the random scheme, while ORDER and LINE entity records are stored according to the clustered scheme and within the same page ranges. Moreover, let's suppose that registering an ORDER entity follows the natural way described as follows:

*get the CUSTOMER entity or create one*
*create an ORDER entity and connect it to the CUSTOMER entity*
*loop*
*get the PRODUCT entity*
*create a LINE entity*
*and connect it to the ORDER entity and to the PRODUCT entity.*

That procedure will induce the uniform distribution of the CUSTOMER records, the grouping of the ORDER records in the same page (or in neighbouring pages) as that of their CUSTOMER record and the grouping of the LINE records around their PRODUCT record. Asking for the ORDER entities of a CUSTOMER entity, and asking for the LINE entities of a PRODUCT entity , will most often need no physical access to pages (a very costly operation). It is therefore possible to tune the parameters of the database in order to minimize the execution time of the future programs.

It is important to note that these physical parameters affect only the execution time of the programs.

When NDBS reads pages from a data file, it stores them in a specialized area in main memory, called the **buffer**, from where it extracts or modifies the entity records. The contents of one or several pages can be stored in the buffer. When an entity record is asked by the program, NDBS first searches the buffer contents for that entity record. If it is found, the data are transmitted to the program, and no disk access occurs. If it has not been found, the corresponding page is got from the disk and stored in the buffer. The larger the buffer, the higher the probability that it contains the requested record. However, a large buffer needs more memory and more time to be searched and managed than a small one. Note that the database buffer is quite independent of the operating system buffer. The latter can be small as far as the database is concerned.

The last physical component of an NDBS database is the **index**. An index is a hidden data structure that allows a quick access to an entity the identifier of which is given by the program. Indices use B-tree structure to localise an entity record in the data file. This gives much better performance than sequentially scanning the entities of that type as can be seen in table 1.1 below. These results have been calculated for 1 Kbyte pages filled at 75% (which is a realistic rate for active B-tree structure). The table gives the average number of entities of a given type (entity type size) among which is is possible to locate one specific entity in at most 2, 3 ... n disk accesses. Theses values are given for various identifier lengths. For instance, a collection of 2200 entities with a 12 character identifier is expected to require no more than 3 disk accesses to locate one entity. That figure can be lower for some programs and a not too small buffer.

The general expression is as follows :

FR = filling rate

PS = page size

ETS = entity type size

IL = identifier length

DA = number of disk accesses to locate one entity

floor[X] = the greatest integer that is no greater than X

$$ETS = floor[(PSxFR)/(100x(IL+3))]^{DA-1}$$

**Expression 1.2** - Computing the performance of an index

| | IL=7 | IL=12 | IL=17 | IL=22 | IL=27 |
|---|---|---|---|---|---|
| **DA= 2** | 75 | 50 | 37 | 30 | 25 |
| **DA= 3** | 5 625 | 2 500 | 1 370 | 900 | 625 |
| **DA= 4** | 422 000 | 125 000 | 50 500 | 27 000 | 15 500 |
| **DA= 5** | 31 650 000 | 6 250 000 | 1 875 000 | 810 000 | 390 000 |

| | IL=7 | IL=12 | IL=17 | IL=22 | IL=27 |
|---|---|---|---|---|---|
| **DA= 2** | 75 | 50 | 37 | 30 | 25 |
| **DA= 3** | 5 625 | 2 500 | 1 370 | 900 | 625 |
| **DA= 4** | 422 000 | 125 000 | 50 500 | 27 000 | 15 500 |
| **DA= 5** | 31 650 000 | 6 250 000 | 1 875 000 | 810 000 | 390 000 |

**Table 1.1** - Entity type size (ETS) as a function of identifier length (IDL) and the maximum number of disk accesses (DA).

The existence of an index (that is, of an identifier in the schema), has no relation with the storage scheme of the entity type, be it random or clustered. In addition to better performance, there is a visible consequence of the presence of an index. When a sequential access is carried out, the entities come in increasing order of the identifier values. The entities appear to be sorted on these values. That order is automatically maintained by NDBS. If entity records are not indexed, they are scanned by chronological order of their creation.

## 2.5 Physical parameters of an NDBS database

Among the physical components that have been described above, some can be parametrized by the database designer in order to get optimized performance from a

given set of application programs. The parameters that can be set are the page size, the buffer size, the page range and the storage scheme of each entity type. Their description is as follows (NDBS version 1).

- The **page size** is an integral number in the range 512 to 4096 bytes. A multiple of 512 is better. Default value : 1024 bytes.
- The **buffer size** is an integral number of pages defining a memory area ranging from 512 bytes to 64Kbytes. Default value : 8 pages.
- The **page range** of each entity type can be set. Its specification comprises two page numbers P1 and P2 such that $1 \leq P1 \leq 65000$ and $P1 \leq P2 \leq 65000$. If the storage scheme is RANDOM, an explicit page range is mandatory.
- The **storage scheme** of each entity type can be set to CLUSTERED or RANDOM. Default setting : CLUSTERED.

## 2.6 How to define a database

The schema processor is a program that allows a user to define and to modify the components of database schemas and to build databases. Its main functions are as follows.

- Data dictionary management :
  display/add/modify/delete a database, an entity type, an attribute, a relationship type.
- Prints a report of a database schema.
- Initializes a new database with a schema stored in the data dictionary and generates a PASCAL text defining the constants types and variables that are needed to write programs working on the database (see part 4).

# Part 2

## 3. THE BASIC PROGRAMMING INTERFACE

### 3.1 INTRODUCTION

The basic programming interface provides the programmer with a set of PASCAL procedures allowing him to open and close a database, to scan sequentially the entities of a given type, to get entities linked to another one via a path, to access the entity with a given identifier value, to create, modify and delete entities, to connect and disconnect entities. That collection of procedures constitutes the **database handler (DBH)**. These procedures must be included in the application program (in NDBS version 1, the procedures are contained in file NDBS.PAS). Communication between the program and the DBH most often concerns one entity : either the DBH delivers information concerning a requested entity, or the program gives information to modify an entity. The program therefore needs additional data types and variables in order to communicate consistently with the database handler, and particularly to designate an entity. These data types and variables are specific to a particular database schema. They are automatically generated by the schema processor, and must be included in the application program (in NDBS version 1, the file ORDER.TYP contains the declarations needed to work on the ORDER database).

All the examples of this chapter are based on the database schema of Figure 3.1.

### 3.2 THE NEW DATA TYPES AND VARIABLES

Besides the procedures of the database handler, the programmer is provided with new data types and variables that allows him to designate database entities and to transmit attribute values. The important concepts of the programming interface are that of reference, reference variable, entity variable and schema component designators.

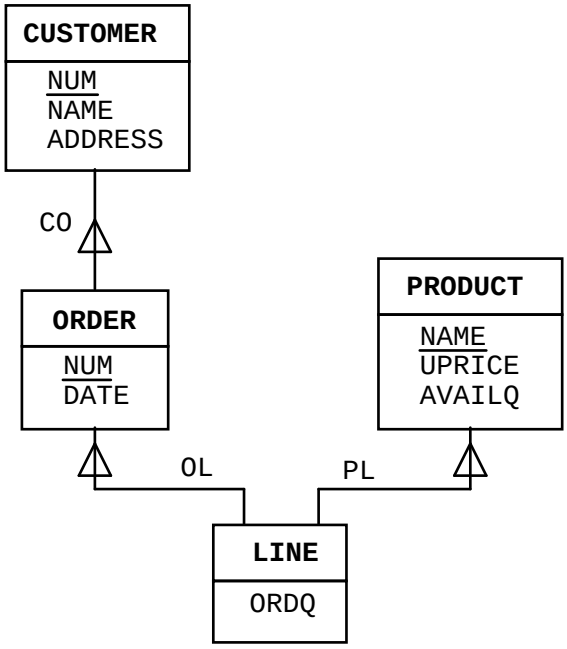


**Figure 3.1 -** The ORDER database. That schema has been designed in chapter, figure 1.3.

**Reference** : a data type of which a value can designate an entity in the database. A **null** reference is a special value telling that no entity is referenced. The reference type is called **DBREF**.

**Reference variable** : a PASCAL variable of type DBREF. Reference variables can be organized in arrays, records. They can be static or dynamic. They cannot be organized in sets nor in files. They must be manipulated by DBH procedures only. In particular, they cannot be read nor written.

A reference variable is undefined (before it is assigned a value), null, or it designates an entity. If an undefined reference variable is transmitted as an input argument to a DBH procedure, unpredictable results can occur. A reference variable can contain the reference of an entity of any type.

*Examples*

```
CUR_CUST : DBREF;
LIST_OF_ENTITIES : array[1..100] of DBREF;
```

**Entity variable** : a PASCAL variable of one of the predefined entity variable types generated by the schema processor. The type name is letter 'T' followed by an entity type name. The TPRODUCT type is dedicated to PRODUCT entity variables. An entity variable contains the reference of an

entity and variables that can accomodate attribute values for entities of the concerned type. An entity variable can be attached to one entity type only. A TPRODUCT variable cannot be used to designate a CUSTOMER entity. Besides the attribute values, no components of an entity variable are visible to the programmer; in particular, the reference part can only be manipulated by DBH procedures. Since an entity variable contains a reference part, that part can be undefined (before it is assigned a value), null or it can designate an entity. If an undefined entity variable is transmitted as an input argument to a DBH procedure, unpredictable results can occur.

Several entity variables may reference the same entity. If that entity is deleted, only the entity variable that has been used for the operation will be set to null. Using the non updated variables leads to the *dangling reference* problem which is well known for PASCAL pointers. In that case however referencing a deleted entity will be detected by the DBH. That behaviour is also true for reference variables.

*Examples*

```
CUS : TCUSTOMER;
LIST_OF_PRODUCTS : array[1..100] of TPRODUCT;
.  .  .
writeln(CUS.NAME);
LIST_OF_PRODUCT[IP].UPRICE := 150;
dbid(CUSTOMER,CUS);
dbdelete(CUSTOMER,CUS);
```

**Database designation** : a database is designated by its name, possibly prefixed with a path designation. That name does not include any extension.

**Entity type designator** : each entity type of a database schema is given a numeric code starting from 1. The transmission of an entity type specification to a DBH procedure is made by that code. For each entity type, the schema processor generates a PASCAL integer constant with that code. Its name is that of the entity type.

*Example*

```
const  CUSTOMER = 1;        {generated by the schema processor}
     .  .  .
dbfirst(CUSTOMER,CUST);     {get the first CUSTOMER in the database}
```

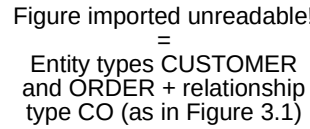


**Figure 3.2 -** Relationship type and path types. A program can use the CO relationships either top down, **following the 1-N *CO* path** and getting the ORDERS of a CUSTOMER, or bottom up, **following the N-1 *-CO* path** and getting the CUSTOMER of an ORDER.

**Path type designator** : Each relationship type of a database schema is given a numeric code starting from 1. By definition that code also designates the 1-N paths related to that relationship type. Negating that code is used to designate the inverse, N-1 paths. The transmission of a path type specification to a DBH procedure is made by that code. For each relationship type, the schema processor generates a PASCAL integer constant with that code. Its name is that of the relationship type. If CO is the name of a relationship type between CUSTOMER and ORDER (see figure 2.2) the code CO (which is also the name of the predefined constant) designates the 1-N paths from CUSTOMER to ORDER and -CO designates the N-1 paths from ORDER to CUSTOMER.

*Example*

```
const  CO = 1;          {generated by the schema processor}
var ORD : TORDER;
    CUS : TCUSTOMER;
.  .  .
dbfpath(ORD,CUS, CO); {get in ORD the first ORDER of CUS customer}
dbfpath(CUS,ORD,-CO); {get in CUS the first (and only) CUSTOMER of ORD order}
```

### DBH return codes

When calling a procedure of the database handler the program is provided with a return code which informs it of how the operation has been

carried out. That code is available in the integer global variable DBSTATUS. In case of emergency situations, the operating system return code is available in the global variable OSSTATUS. The possible values of DBSTATUS are as follows.

0 : all is well; the operation has been carried out correctly

1 : the requested object (database, entity) has not been found

2 : identifier uniqueness violation during a create or modify operation

10 : incorrect entity type code

11 : incorrect relationship type code

30 : incorrect reference value in an input argument

80 : out of disk memory space

90 : incorrect reference found in the database; the database is corrupted

99 : I/O or system error

Return codes 0 , 1 and 2 define normal conditions. They can be checked through the DBH boolean functions `dbfound`, `dbnotfound` and `dbnonunique`. Return codes 10 and 11 probably come from syntactic error in the program (wrong schema component). Return code 30 may be more severe, since it is probably due to a program logic error. Return codes 80, 90 and 99 are not due to the program but are induced by external accidents.

## 3.3 THE ARGUMENTS

Calling a DBH procedure will generally pass arguments. Most of them are common to several procedures, and are therefore described below.

**data-base**

*type* : any string expression containing from 1 to 64 characters;

*meaning* : name of a database;

*examples* : 'ORDER', '\DIR3\PERSONAL', DBNAME

**entity-type**

*type* : any integer expression; generally a predefined integer constant;

*meaning* : the integer value is the numeric code of a valid entity type of the active database; in case of a predefined constant, its name is the name of the entity type it designates;

*examples* (predefined constants) : CUSTOMER, PRODUCT

**path-type**

*type* : any integer expression; generally a predefined integer constant possibly prefixed with a minus sign;

*meaning* : the absolute integer value is the numeric code of a valid relationship type of the active database; in case of a predefined constant, its name is the name of the relationship type it designates; if the value is positive, it designates the 1-N path type; if the value is negative, the designated path type is the N-1 inverse path type;

*examples* : suppose the relationship type CO (CUSTOMER, ORDER);

so,

CO designates the path type from CUSTOMER to ORDER;

- CO designates the path type from ORDER to CUSTOMER.

**ent-var**

*type* : T<entity type name>;

*meaning* : entity variable;

*examples* : CUS (type TCUSTOMER), P (type TPRODUCT)

**targ-ent-var**

*type* : T<entity-type>;

*meaning* : entity variable designating a target entity of a path

**orig-ent-var**

*type* : T<entity-type>;

*meaning* : entity variable designating the origin entity of a path

**ref-var**

*type*: DBREF;

*meaning* : reference variable;

**var**

*type* : either DBREF or entity variable;

## 3.4 THE PROCEDURES OF THE DATABASE HANDLER

The programmer is provided with twelve classes of database procedures or **primitives** :

- opening and closing a database
- sequential scanning of an entity type
- finding an entity by its identifier value
- direct access to an entity
- scanning an access path
- creating, deleting and updating an entity
- updating an access path
- diagnostic functions
- reference and entity variable manipulation
- integrity

All these primitives need not be known nor used for writing comprehensive database management programs. The following kernel is sufficient :

- **dbopen** and **dbclose** : database access;

- **dbfirst** and **dbnext** : sequential access;

- **dbfpath** and **dbnpath** : access by path (through relationships);

- **dbid** : find entity by identifier;

- **dbcreate**, **dbdelete**, **dbmodify**, **dbinsert** and **dbremove** : updating;

### 3.4.1 OPENING AND CLOSING A DATABASE

#### `dbopen(data-base)`

**function**

if the database named **data-base** exists, opens it, makes it available for the program (it becomes the **active** database of the program) and returns dbstatus = 0; if no such database has been found or if any I/O or system error occurred, no database is available for the program and dbstatus is different from 0.

**return codes**

dbstatus = 0 : the database has been opened;

dbstatus = 1 : the database has not been found;

dbstatus = 99 : I/O or system error;

**example**

```
DBNAME := 'ORDER';
dbopen(DBNAME);
if not dbfound then goto ER_OPEN;
```

### dbclose

**function**

close the active database, if any; from now on there is no active database for the program;

**return codes**

dbstatus = 0 : the active database has been closed;

dbstatus = 1 : there was no open database;

dbstatus = 99 : I/O or system error;

**example**

```
dbclose;
```

## 3.4.2 SEQUENTIAL SCANNING OF AN ENTITY TYPE

### dbfirst(entity-type,ent-var)

**function**

finds the first entity of the type **entity-type** in the active database and stores its reference and its attribute values into the variable **ent-var**.

**return codes**

dbstatus = 0 : an entity has been found;

dbstatus = 1 : entity not found; the **entity-type** set is empty;

dbstatus = 10 : **entity-type** has an incorrect value;

dbstatus = 90 : corrupted database

dbstatus = 99 : I/O or system error;

### dbnext(entity-type,ent-var)

**function**

finds the entity of the type **entity-type** that follows the entity designated by **ent-var** in the active database; stores its reference and its attribute values into the variable **ent-var**. If **ent-var** has a null reference, `dbnext` acts as `dbfirst` (*the successor of none is the first one*). The `dbfirst` primitive is therefore redundant.

**return codes**

dbstatus = 0 : an entity has been found;

dbstatus = 1 : entity not found; the **ent-var** entity was the last one of that type;

dbstatus = 10 : **entity-type** has an incorrect value;

dbstatus = 30 : **ent-var** has an incorrect value;

dbstatus = 90 : corrupted database

dbstatus = 99 : I/O or system error;

**example**

```
var PRO : TPRODUCT;
. . .
dbfirst(PRODUCT,PRO);
while dbfound do
begin
    writeln('product name = ',PRO.NAME);
    dbnext(PRODUCT,PRO)
end;
```

## 3.4.3 FINDING AN ENTITY BY ITS IDENTIFIER VALUE

### dbid(entity-type,ent-var)

**function**

finds in the active database the entity of the type **entity-type** that is identified by the indentifier value previously stored in **ent-var**; stores its reference and its attribute values into the variable **ent-var**.

**return codes**

dbstatus = 0 : the entity has been found;

dbstatus = 1 : no entity has been found;

dbstatus = 10 : **entity-type** has an incorrect value;
dbstatus = 90 : corrupted database;
dbstatus = 99 : I/O or system error;

**example**

```
var PRO : TPRODUCT;
    NAM : string[20];
. . .
readln(NAM);
dbid(PRODUCT,PRO);
if dbfound then writeln('price is ',PRO.UPRICE)
else if dbnotfound then writeln('product unknown')
    else goto DB_ERROR;
```

### 3.4.4 DIRECT ACCESS TO AN ENTITY

**`dbdirect(entity-type,ent-var,var)`**

**function**

finds in the active database the entity of the type **entity-t** referenced by the entity variable or the reference variable **ref-var**; stores its reference and its attribute values into the variable **ent-var**. **ent-var** and **var** need not being distinct.

**return codes**

dbstatus = 0 : the entity has been found;
dbstatus = 1 : no entity has been found;
dbstatus = 10 : **entity-type** has an incorrect value;
dbstatus = 30 : **ref-var** has an incorrect value;
dbstatus = 90 : corrupted database;
dbstatus = 99 : I/O or system error;

**example**

```
var PRO : TPRODUCT;
    NAM : string[20];
    PROLIST : array[1..100] of DBREF;
    PRONBR, IP :  0..100;
. . .
for IP := 1 to PRONBR do
begin
```

```
        dbdirect(PRODUCT,PRO,PROLIST[IP]);
        writeln(PRO.NAME)
    end;
```

### 3.4.5 SCANNING AN ACCESS PATH

#### dbfpath(targ-ent-var,orig-ent-var,path-type)

**function**

finds the first entity connected to the entity **orig-ent-var** by the path **path-type** in the active database; stores its reference and its attribute values into the variable **targ-ent-var**. That primitive can be used for both 1-N and N-1 paths.

**return codes**

dbstatus = 0 : the entity has been found;

dbstatus = 1 : no entity has been found; no entity is connected to **orig-ent-var** entity;

dbstatus = 11 : **path-type** has an incorrect value;

dbstatus = 30 : **orig-ent-var** has an incorrect value;

dbstatus = 90 : corrupted database

dbstatus = 99 : I/O or system error;

**example**

```
var CUS : TCUSTOMER;
    ORD : ORDER;
. . .
dbfpath(CUS,ORD,-CO);
if dbfound then writeln('customer nname = ',CUS.NAME);
```

#### dbnpath(targ-ent-var,orig-ent-var,path-type)

**function**

finds the entity that follows the entity **targ-ent-var** among those connected to the entity **orig-ent-var** by the path **path-type** in the active database; stores its reference and its attribute values into the variable **targ-ent-var**. If **targ-ent-var** has an initial null reference, `dbnpath` acts as `dbfpath` (*the successor of none is the first one*). The `dbfpath` primitive is therefore redundant. That primitive can be used for both 1-N and N-1 paths. In the latter case, however, it always returns dbstatus = 1.

**return codes**

dbstatus = 0 : the entity has been found;

dbstatus = 1 : no entity has been found; the **targ-ent-var** entity was the last one;

dbstatus = 11 : **path-type** has an incorrect value;

dbstatus = 30 : **targ-ent-var** or **orig-ent-var** have an incorrect value;

dbstatus = 90 : corrupted database

dbstatus = 99 : I/O or system error;

**example**

```
var CUS : TCUSTOMER;
    ORD : ORDER;
. . .
dbfpath(ORD,CUS,CO);
while dbfound do
begin
    writeln('order number = ',ORD.NUM);
    dbnpath (ORD,CUS,CO);
end;
```

## dbtestpath(targ-ent-var,orig-ent-var,path-type) : boolean

**function**

returns `true` if the entity **targ-ent-var** is in the 1-N path of type **path-type** the origin of which is entity **orig-ent-var**; in other words, returns `true` if entities **targ-ent-var** and **orig-ent-var** are connected by **path-type**; returns `false` otherwise.

**return codes**

dbstatus = 0 : the test has been carried out;

dbstatus = 11 : **path-type** has an incorrect value;

dbstatus = 30 : **targ-ent-var** or **orig-ent-var** have an incorrect value;

dbstatus = 90 : corrupted database

dbstatus = 99 : I/O or system error;

**example**

```
var CUS : TCUSTOMER;
    ORD : ORDER;
. . .
```

```
   dbid(CUSTOMER,CUS);
   dbid(ORDER,ORD);
   if dbtestpath(ORD,CUS,CO) then
      writeln('Order ',ORD.NUM,' placed by customer
',CUS.NUM);
```

### 3.4.6 CREATING, DELETING AND UPDATING AN ENTITY

#### `dbcreate(entity-type,ent-var)`

**function**

inserts in the database an entity of type **entity-type**. The attribute values are obtained from **ent-var**. Stores the reference of the new entity into the variable **ent-var**. If **entity-type** has an identifier, there must be no other entity of that type with the same value for that attribute.

**return codes**

dbstatus = 0 : the entity has been created;

dbstatus = 2 : an entity with the same identifier value already exists; no entity created;

dbstatus = 10 : **entity-type** has an incorrect value;

dbstatus = 90 : corrupted database

dbstatus = 99 : I/O or system error;

**example**

```
var CUS : TCUSTOMER;
. . .
CNUM := CNUM+1;
CUS.NUM := CNUM;
CUS.NAME := CNAME;
CUS.ADDRESS := '';
dbcreate(CUSTOMER,CUS);
case dbstatus of
    0  :  ;
    2  :  goto ERR_ID;
    else  goto PANIC;
end;
```

## dbdelete(entity-type,ent-var)

**function**

erases from the active database the entity designated by **ent-var** of type **entity-type**. The reference part to variable **ent-var** is set to null, but no other components are modified. If the entity to be deleted is a target in some 1-N paths, it is first removed from them. If the entity to be deleted is an origin of some 1-N paths, their target entities are first removed from these paths, and thus made free again.

**return codes**

dbstatus = 0 : the entity has been deleted;

dbstatus = 10 : **entity-type** has an incorrect value;

dbstatus = 30 : **ent-var** has an incorrect value;

dbstatus = 90 : corrupted database

dbstatus = 99 : I/O or system error;

**example**

```
var CUS : TCUSTOMER;
. . .
dbid(CUSTOMER,CUS);
dbdelete(CUSTOMER,CUS);
if dbstatus > 0 then goto ERR_DB;
```

## dbmodify(entity-type,ent-var)

**function**

modifies the entity of type **entity-type** designated by **ent-var**. The attribute values are obtained from **ent-var**. If **entity-type** has an identifier, there must be no other entity of that type with the same value for that attribute.

**return codes**

dbstatus = 0 : the entity has been modified;

dbstatus = 2 : an entity with the same identifier value already exists; no modification;

dbstatus = 10 : **entity-type** has an incorrect value;

dbstatus = 30 : **ent-var** has an incorrect value;

dbstatus = 90 : corrupted database;

dbstatus = 99 : I/O or system error;

**example**

```
var CUS : TCUSTOMER;
. . .
CUS.NUM := CNUM;
dbid(CUSTOMER,CUS)
CUS.ADDRESS := CADDRESS;
dbmodify(CUSTOMER,CUS);
if dbstatus > 0 then goto DB_ERROR;
```

### 3.4.7 UPDATING AN ACCESS PATH

**`dbinsert(targ-ent-var,orig-ent-var,path-type)`**

**function**

connects entity **targ-ent-var** to entity **orig-ent-var**; more precisely, inserts the entity designated by **targ-ent-var** as a target of the 1-N path of type **path-type** with origin designated by **orig-ent-var**. If that entity was already a target in a path of that type, it is first removed from that path.

**return codes**

dbstatus = 0 : the entity has been inserted;
dbstatus = 11 : **path-type** has an incorrect value;
dbstatus = 30 : **targ-ent-var** or **orig-ent-var** have incorrect values;
dbstatus = 90 : corrupted database;
dbstatus = 99 : I/O or system error;

**example**

```
var CUS : TCUSTOMER;
    ORD : TORDER;
. . .
dbid(CUSTOMER,CUS)
dbcreate(ORDER,ORD);
dbinsert(ORD,CUS,CO);
if dbstatus > 0 then goto DB_ERROR;
```

**`dbremove(targ-ent-var,orig-ent-var,path-type)`**

**function**

disconnects entity **targ-ent-var** from entity **orig-ent-var**; more pre-

cisely, removes the entity designated by **targ-ent-var** from the 1-N path of type **path-type** in which it is a target. Parameter **orig-ent-var** is ignored.

**return codes**

dbstatus = 0 : the entity has been removed;
dbstatus = 11 : **path-type** has an incorrect value;
dbstatus = 30 : **targ-ent-var** has an incorrect value
dbstatus = 90 : corrupted database;
dbstatus = 99 : I/O or system error;

**example**

```
var ORD : TORDER
      LIN : TLINE;
. . .
dbid(ORDER,ORD);
dbfpath(LIN,ORD,OL);
dbremove(LIN,ORD,OL);
if dbstatus > 0 then goto DB_ERROR;
```

### 3.4.8 DIAGNOSTIC FUNCTIONS

#### `dbfound : boolean`

returns `true` if the last dbh-procedure called returned **dbstatus = 0**; returns `false` otherwise.

#### `dbnotfound : boolean`

returns `true` if the last dbh-procedure called returned **dbstatus = 1**; returns `false` otherwise.

#### `dbnonunique : boolean`

returns `true` if the last dbh-procedure called returned **dbstatus = 2**; returns `false` otherwise.

#### `dbsevere : boolean`

returns `true` if the last dbh-procedure called returned **dbstatus** > **2**; returns `false` otherwise.

**dbpanic : boolean**

returns `true` if the last dbh-procedure called returned **dbstatus _ 80**; returns `false` otherwise.

### 3.4.9 REFERENCE AND ENTITY VARIABLE MANIPULATION

**dbclear(ent-var)**

has the *ent-var* variable reference no entity (it "becomes `null`"); the attribute values are left unchanged;

**dbcopyatt(ent-var1,ent-var2)**

copies the attribute values of *ent1-var* variable to *ent-var2* variable;

**dbcopyall(ent-var1,ent-var2)**

copies the entity reference and the attribute values of *ent1-var* variable to *ent-var2* variable;

**dbcopyref(var1,var2)**

var1 and var2 are reference variables or entity variables;

copies the reference part of var1 variable to the reference part of var2 variable;

**dbequal(ent-var1,ent-var2) : boolean**

returns `true` if the entity variables *ent-var1* and *ent-var2* designate the same entity in the database or are both null; returns `false` otherwise;

**dbnull(ent-var) : boolean**

returns `true` if the entity variable ent-var is `null`;

### 3.4.10 INTEGRITY

**dbcommit**

puts the database in a save state; unless updates are made in the database, any further software crash will leave the database intact; no programming objects (entity variables for instance) are modified;

(*technically speaking, the contents of the NDBS buffers are rewritten in the database, the database system files are closed then opened; this is not as powerful as a true* ***commit-transaction*** *primitive)*

# Part 3

## 4. THE HIGH LEVEL PROGRAMMING ENVIRONMENT

### 4.1 INTRODUCTION

Comparing the basic programming interface with the high-level programming language ADL-PASCAL is somehow like comparing assembly language programming with PASCAL programming, at least as far as database operations are concerned. Let's first illustrate the point with the example of the PASCAL *for* loop.

Suppose we have to display the list of numbers from 10 to 100. The most standard procedure, that is accepted by all languages, is probably the following :

```
NUMBER := 10;
LAST   := 100;
while NUMBER <= LAST do
begin
     display NUMBER;
     NUMBER := NUMBER + 1
end;
```

Some languages, however, including PASCAL provide a much more concise way to express that procedure. We will therefore rewrite the procedure as follows :

```
for NUMBER := 10 to 100 do
     display NUMBER;
```

The main difference is that we have described the list to be processed instead of explaining the way to build it. The difference is therefore between WHAT we want and HOW we want it to be done.

Let's now work on a more DB-specific example, based on the now well known ORDER database (see schema of figure 2.1). Suppose we want to write a procedure which displays the NUM and ADDRESS values of the CUSTOMER entities for which a NAME value is specified. Here is a pseudo-code version of the procedure :

```
{display NUM and ADDRESS of CUSTOMERS named 'Thomas'}
get first CUSTOMER
while one is found do :
     if his NAME is 'Thomas' then
         display his NUM and ADDRESS
         get next CUSTOMER
```

We have described in great detail **how** entities have to be obtained (get first, while do, get next) and **how** they are selected (if then). The properties the entities have to satisfy in order to be selected are not obvious from the text above. Defining other selection criteria would have led us to other algorithmic structures. For instance, getting the CUSTOMER whose NUM is 123, or getting the ORDERS of a given CUSTOMER sent before a given DATE, require different ways of obtaining the data, namely an *access through an identifier* and an *access through an access path*, as seen in the examples below.

```
{display NUM and ADDRESS of CUSTOMER number N}
set identifier value to N
get CUSTOMER through identifier
if found do
   display his NUM and ADDRESS
```

and

```
{display NUM and DATE of ORDERS sent by CUSTOMER CUS
 before date GIVENDATE}
get first ORDER from CUS via CO
while one is found do
   if its DATE < GIVENDATE then
      display NUM and DATE
      get next ORDER from CUS via CO
```

An expression that is close to the initial English description would have been better. Let's consider the following one.

```
for each CUSTOMER with NAME = 'Thomas' do :
   display his NUM and ADDRESS
```

We have described the set of entities which we want to process, whatever the way the computer will get the entities. The comment header line is no longer necessary and has been dropped. The two other examples suggested could be written in a similar way.

```
for each CUSTOMER with NUM = 123 do :
  display his NUM and ADDRESS
```

and

```
for each ORDER connected to CUS and with DATE < GIVENDATE do :
  display its NUM and DATE
```

This is the way ADL-PASCAL allows the programmer to write procedures.

An ADL-PASCAL program is a PASCAL program that includes some new statements related to database access and updating. These statements can replace most DBH procedure calls, thus producing more concise and readable programs. An ADL-PASCAL is translated by a preprocessor into a pure PASCAL program, including DBH procedure calls, that can be compiled and run. The translation is simple and straightforward, so that mixing ADL-PASCAL statements and DBH procedure calls in the same program can be done without much difficulty. A translated ADL-PASCAL is what we would have written in the basic programming interface. Before explaining precisely how ADL-PASCAL works, let's have a look at the ADL-PASCAL expression of the three procedures developed above.

```
for CUS := CUSTOMER(: NAME = 'Thomas') do
    writeln(CUS.NUM, CUS.ADDRESS)
endfor;

for CUS := CUSTOMER(: NUM = 123) do
    writeln(CUS.NUM, CUS.ADDRESS)
endfor;

for ORD := ORDER((CO: CUS) and (DATE < GIVENDATE)) do
    writeln(ORD.NUM,ORD.DATE)
endfor;
```

## 4.2 ADL-PASCAL PROGRAMMING CONVENTIONS

A line containing an ADL-PASCAL statement begins with a # symbol. An ADL-PASCAL statement can span several lines; only the first one needs the # symbol. Each ADL-PASCAL statement begins on a new line, and doesn't share its lines with any other statement.

There are no new data types nor variables in addition to those which are needed with the basic programming interface. Most ADL-PASCAL statements use entity variables to denote database entities. In some cases, checking the value of DBSTATUS

will be wise since ADL-PASCAL only controls values 0 and 1 in FOR-ENDFOR loops.

There is, however, a new constant, called **null**, that can be assigned to any entity variable, whatever its type, and can be used in the **modify** statement. That constant means ***no entity*** and can be used in ADL-PASCAL statements only (DBH procedure calls excluded).

It should be noted that the preprocessor doesn't know about PASCAL syntax, and will therefore ignore the PASCAL statements. Some errors will be detected at PASCAL compile time only.

The arguments follows the rules detailed in 2.3.

## 4.3 ADL-PASCAL DECLARATIONS

The declaration part of the program or procedure must include a database declaration statement giving the name of the database.

Format :

```
# database  data-base
```

where

`data-base-name` is the name of a database;

Example :

```
# database ORDER
```

ADL-PASCAL makes use of entity variables. They can be declared either the standard way (see chapter 2) or the more explicit and natural way as follows.

Format :

```
# entity-variable-name-list : entity entity-type-name
```

where

`entity-variable-name-list` is a list of one or several names of entity variables;

`entity-type-name` is the name of an entity type;

Examples :

```
#   CUS : entity ORDER;
#   L1, L2, NEWL : entity LINE;
```

## 4.4 OPENING AND CLOSING A DATABASE

Opening a database and closing the active database is done as follows.

Format :

```
# open;
# close;
```

## 4.5 ENTITY SET SPECIFICATION

The main ADL-PASCAL statements specify actions to be carried out on each element of a set of entities. The entities of an entity set are of the same type. The set can include all the entities of that type, or it can include a specific subset only.

The set of all the entities of a given type is designated by the name of that entity type. For instance, **`CUSTOMER`** is an expression which designates the set of CUSTOMER entities.

A subset of entities is built by specifying the condition the entities must satisfy in order to be included in the subset. That *selection condition* is made of one or several elementary conditions. The subset is made up of the entities that satisfy all the elementary conditions.

An elementary condition can be an *attribute condition* or a *relationship condition*. The **attribute condition** specifies what the attribute values of the selected entities must be. Its format is as follows.

```
( : attribute-name  comp-operator  value-designation )
```

where

`attribute-name` is the name of an attribute of the entity type;

`comp-operator` is a comparison operator : `=, <, >, <=, >=, <>`

`value-designation` is a constant, a constant name, a variable name, a record element name or an array element expression;

Examples :

```
( : NAME = 'Smith J.')
```
: (the customers ) whose name is 'Smith J.'

```
( : DATE < DATEOFDAY)
```
: (the orders) the date of which is before DATEOFDAY;

```
( : NUM = LOP[I])
```
: (the product) with number LOP[I];

The **relationship condition** states that the selected entities must be connected to a given entity. Its format is as follows.

```
( relationship-type-name: entity-variable )
```

where

`relationship-type-name` is the name of a relationship type;

`entity-variable` is the name of an entity variable;

It's worth noting that while DBH procedures make use of path types, an ADL-PASCAL relationship condition specifies only the name of a relationship type, which is both simpler and more symmetrical. Path types tell how to get entities, while relationship types are logical properties of the entities.

Examples :

`(CO:CUS)` : (the orders ) connected via CO to customer CUS;
`(CO:ORD)` : (the customer) connected via CO to order ORD;
`(PL:L)` : (the product) connected to line L;

There is, however, a situation in which that syntax is ambiguous, namely when using a **recursive relationship type**. Let's consider the **SON** relationship type between **MAN** and **MAN**. Each relationship links a father with his son. Let **M** be a MAN entity. Does the expression

```
MAN(SON: M)
```

describe the sons of M or the father of M ? The ambiguity is solved by adopting the following conventions :

`MAN(SON: M)` designates the sons of M (as if the 1-N path were used)

`MAN(*SON: M)` designates the father of M (as if the N-1 path were used)

Let's now go back to the selection expression : it is either an elementary condition, or a parenthesized list of elementary conditions linked with *and* logical connectors. A selection expression can include one relationship condition only, but any number of attribute conditions.

Here are some examples of complete entity set expressions, together with their meaning.

`ORDER` designates all the ORDER entities;

`ORDER(: DATE < TODAY)`
all the ORDER entities the DATE of which is before (the content of) TODAY;

`PRODUCT(: NAME = PRONAME)`
the PRODUCT entity with NAME (equal to the content of) PRONAME;

`PRODUCT((: AVAILQ < PRO.AVAILQ) and (: UPRICE > 150))`
all the PRODUCT entities with an AVAILQ which is less than that of entity PRO and with UPRICE more than 150;

`ORDER(CO: CUS)`
all the ORDER entities connected to CUSTOMER CUS;

`ORDER((CO: CUS) and (:DATE < TODAY))`
all the ORDER entities connected to CUSTOMER CUS, with DATE before TODAY;

An entity set can be empty if no entities match the selection condition. Moreover, if the selection expression includes an elementary equality condition on the entity identifier, the entity set will contain 0 or 1 elements.

## 4.6 ENTITY ASSIGNMENT STATEMENT

That statement lets an entity variable reference a database entity. The entity is any one element of a specified entity set. The entity set will generally contain only one element. If it contains more than one, one of them will be selected, the other ones being discarded. If the set is empty, the entity variable is left unchanged. The entity variable can be set to null, thus having its reference part designate no entity. This is obtained by using the constant **null** in the right part of the statement. Please note the difference between **null** and an empty entity set.

Format :

`# entity-variable-name := entity-set-expression`

where

`entity-variable-name` is the name of an entity variable;

`entity-set-expression` is any valid expression of an entity set;

Examples :

```
# CUS := CUSTOMER(: NUM = INUM);
  if dbnotfound then INVALID_INPUT;
# PRO := PRODUCT(PL: LIN);
# ORD := ORDER(: DATE = D1);
# C := CUSTOMER;
# PRO := null;
```

## 4.7 ENTITY *FOR-ENDFOR* STATEMENT

The *entity for-endfor loop* allows the programmer to specify actions to be carried out for each element of an entity set. The current element of the set, that is the element currently being processed in the loop, is designated by an entity variable. That *loop*

*variable* appears in the loop header, the same way as in a standard PASCAL *for* loop.

Format :

```
# for entity-variable-name := entity-set-expression
      sequence
# endfor
```

where

`entity-variable-name` is the name of an entity variable;

`entity-set-expression` is any valid expression of an entity set;

`sequence` is the loop body, that is any sequence of PASCAL or ADL-PASCAL statements;

If the entity set is empty, the loop body is not executed and the entity variable is left unchanged. If the entity set contains at least one element, the entity variable references the last entity that has been processed in the loop.

The following program sections illustrate the use of entity loops.

```
# CUS : entity CUSTOMER;
      . . .
# for CUS := CUSTOMER do
          writeln(CUS.NUM, CUS.NAME);
# endfor;

# PRO : entity PRODUCT;
# LIN : entity LINE;
# ORD : entity ORDER;
. . .
# for PRO := PRODUCT(: AVAILQ < 200) do
      writeln(PRO.NAME);
#     for LIN := LINE((PL: PRO) and (ORDQ > 10)) do
#         ORD := ORDER(OL:LIN);
          writeln(ORD.NUM, LIN.ORDQ);
#     endfor;
# endfor;
```

Statements of the body loop can modify the contents of the database. One situation is particularly delicate and must be dealt with very carefully. That case will occur when modifications are made in the loop body in such a way that **the entity set is modified**, by including some entities or removing others. Such situations are generally prohibited, at least methodologically, with standard *for loops*, since modifying

the variable or the bounds of the *for loop* from within the loop body itself is not allowed. The same is true in general with ADL-PASCAL loops. However, one exception is tolerated, **you may delete the current entity** (in that case, the loop will behave correctly, but will sometimes slow down - see examples in section 5). So the following procedure is allowed :

```
   {a valid example}
# for ORD := ORDER(CO: CUS1) do
#    delete ORDER ORD;
# endfor;
```

In all the other cases, the result can be unexpected or undefined. Here is a surprising example.

```
{a wrong example}
# for ORD := ORDER(CO: CUS1) do
# modify ORDER ORD(CO:CUS2); {connect it to customer
CUS2}
# endfor;
```

The loop body modifies the entity set by removing successively all its elements. In fact, only the first ORDER will be transferred (Can you explain why ? - see section 5).

In the next example, the loop body adds entities to the selected entity set. It is unpredictable whether the new LINES L will be processed by the loop or not. If all of them are processed, the procedure will only end on a *out of disk memory* condition !

```
   {a dangerous example}
# ORD := ORDER(:NUM = Y);
# for LIN := LINE((OL:ORD) and (ORDQ < 100)) do
#    create L := LINE((OL:ORD) and (:ORDQ = 100-LIN.ORDQ));
# endfor;
```

As another example, two embedded for-endfor loops cannot use the same current entity variable :

```
   {a wrong example}
# for ORD := ORDER(CO: CUS1) do
#    for ORD := ORDER(:NUM = X) do
     . . .
#    endfor;
# endfor;
```

Modifying the natural behaviour of a for-endfor loop can be done either by dropping the processing of the remaining elements of the entity set, i.e. closing the loop (**exit**

statement) or by dropping the processing of the current entity and getting the next entity (**next** statement).

Format :

```
    # exit
 or # exit entity-variable-name
    # next
 or # next entity-variable-name
```

where
`entity-variable-name`is the name of an entity variable used as a loop variable;

The simple *exit* and *next* formats control the innermost loop in which the statement is included. The extended formats allow controlling the loop that uses the specified variable. If the statement is in the body of one or several loops embedded in the loop designated by the entity variable, these loops are automatically closed by implicit *exits*.

Displaying the name of the customer whose NUM is an even number can be programmed as follows.

```
# forCUS := CUSTOMER do
     if (CUS.NUM mod 2)=0 then
#       next;
     writeln(CUS.NAME);
# endfor;
```

The following procedure displays the name of the customers that have an order placed before date D. The same result will be obtained by replacing **next CUS** with **exit**.

```
#  for CUS := CUSTOMER do
#    for ORD := ORDER(CO: CUS) do
       if ORD.DATE <= D then
          begin writeln(CUS.NAME);
#           next CUS;
          end;
#    endfor;
#  endfor;
```

## 4.8 DATABASE UPDATING

ADL-PASCAL has three statements for changing the contents of a database. Creating a new entity in the database is done by a ***create*** statement.

Format :

```
# create entity-variable-name := entity-type-name
```

or

```
# create entity-variable-name := entity-type-name (create-
condition)
```

where

`entity-name-variable` is the name of an entity variable;

`entity-type-name` is the name of an entity type of the database;

`create-condition` is an elementary condition, or a list of elementary conditions linked with *and* logical connectors; an elementary condition is a relationship condition or an attribute condition the comparison operator of which is '='; the definition of an elementary condition is similar to the elementary selection condition;

The **create** statement inserts a new entity in the database so that it satisfies the create condition. The new entity is referenced by the entity variable. Attribute values are either stored in the entity variable before the creation and/or specified in the create condition. The three following examples are equivalent.

```
   {create a new CUSTOMER entity}
#  create CUS := CUSTOMER((:NUM = X) and (:NAME = Y)
                     and (:ADDRESS = Z));
   {create a new CUSTOMER entity}
   CUS.NUM := X;
   CUS.NAME := Y;
#  create CUS := CUSTOMER(:ADDRESS = Z);
   {create a new CUSTOMER entity}
   CUS.NUM := X;
   CUS.NAME := Y;
   CUS.ADDRESS := Z;
#  create CUS := CUSTOMER;
```

A relationship condition asks for the connection of the new entity to one or several other existing entities. If ORD is a valid reference of an ORDER entity, and PRO is a valid reference of a PRODUCT entity, the following example creates a new LINE entity and connects it to entity ORD and entity PRO.

```
#  create L := LINE((OL:ORD) and (PL:PRO) and (:ORDQ = XQ));
```

If the operation has not succeeded, the database is not modified, and the contents of the entity variable is set to *null*.

The **delete** statement allows to remove an entity from the database. Its characteristics are those of the **dbdelete** basic primitive.

Format :

```
# delete  entity-type-name entity-variable-name;
```

where

`entity-type-name` is the name of an entity type

`entity-variable-name` is the name of an entity variable.

The example above asks for deleting the CUSTOMER entity referenced by CUS.

```
#  delete CUSTOMER CUS;
```

**Modifying** an entity is made through the **modify** statement. That statement makes it possible not only to change attribute values of the entity, but also to change its connections with other entities. The modify format is as follows.

Format :

```
# modify entity-type-name entity-variable-name
```

or

```
# modify entity-type-name entity-variable-name modify-condition
```

where

`entity-type-name` is the name of an entity type of the database;

`entity-name-variable` is the name of an entity variable;

`modify-condition` is defined the same way as *create-condition* in the create statement, except that the entity-variable-name of a relationship condition can be replaced with the **null** entity constant, standing for *no entity.*

The **modify** statement carries out the necessary changes so that the referenced entity satisfies the modify condition. New attribute values are either stored in the entity variable before the modification and/or specified in the modify condition. Here are some examples (the first two are equivalent).

```
DATE := X;
# modify ORDER ORD;
# modify ORDER ORD(:DATE = X);
# modify PRODUCT PRO((:UPRICE = PRO.UPRICE)
                  and (:AVAILQ = X));
```

A relationship condition that appears in a modify condition asks for the connection of the concerned entity to another existing entity, or the disconnection from an

entity. If the relationship condition specifies an entity variable, the entity concerned is connected to it. If the entity concerned was already connected to another entity, it is first disconnected from it. If the relationship condition specifies the constant **null** instead of an entity variable, and if the entity concerned was connected to another entity, it is disconnected from that entity.

Some examples :

```
# modify LINE L(PL:PRO2);
     {if LINE L is connected to a PRODUCT entity,
      disconnect it;connect it to PRODUCT PRO2}

#  modify ORDER ORD(CO: null);
     {disconnect ORDER ORD from its CUSTOMER}
#  modify ORDER ORD((:NUM = Y) and (CO:CUS2));
     {modify ORDER ORD by changing its NAME and its CUSTOMER}
```

## 4.9 MIXING ADL-PASCAL STATEMENTS AND DBH PROCEDURE CALLS

As the ADL-PASCAL preprocessor ignores the pure PASCAL statements, and as ADL-PASCAL statements are translated into PASCAL statements and DBH procedure calls, nothing can prevent a programmer to mix both programming styles in the same program. Such practice is not only possible, but it is sometimes needed since ADL-PASCAL is not as complete as DBH procedures. The programmer should be careful in that case.

# Part 4

# 5. NDBS CASE STUDIES

## 5.1 The ORDER database

### 5.1.1 Description of the application domain

The proposed database has been analysed in section 1.2 of part 1.

### 5.1.2 Graphical schema of the ORDER database

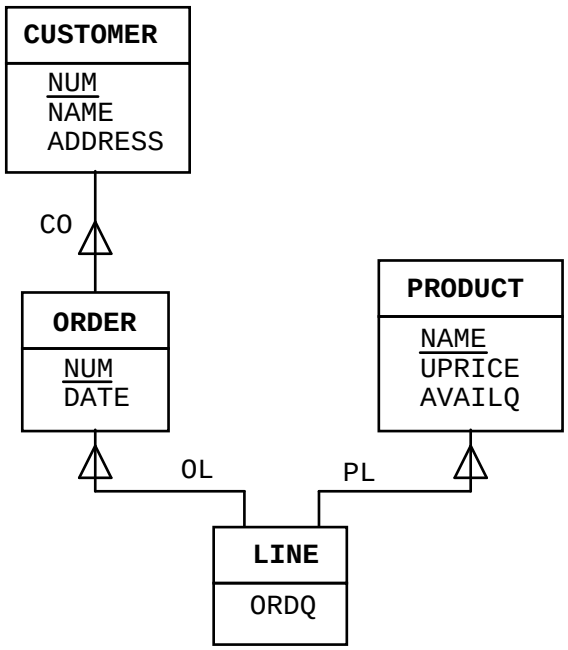


**Figure 5.1 - Figure 4.1** - Schema of the ORDER database

### 5.1.3 The programming environment of the ORDER database

The following definitions are available in the file ORDER.TYP that must be included in any program working in the ORDER database. That file has been automatically generated by the Schema Processor.

```
const  CUSTOMER = …;
       PRODUCT = …;
       ORDER  = …;
       LINE  = …;

       CO  = …;
       OL  = …;
       PL  = …;

type
  TCUSTOMER   = record
             ...
             NUM    :integer;
             NAME    :string[15];
             ADDRESS : string[35]
           end;

  TPRODUCT   =record
             ...
             NAME   :string[20];
             UPRICE :real;
             AVAILQ :integer
           end;

  TORDER   =record
             ...
             NUM: integer;
             DATE:string[6]
           end;

  TLINE   =record
             ...
             ORDQ:integer
           end;

var
  dbstatus : integer;
```

### 5.1.4 SIMPLE SEQUENTIAL PROCESSING

The first program lists the characteristics of all the CUSTOMER entities. It simply scans the CUSTOMER entities in the database and displays the values of NUM, NAME and ADDRESS for each of them.

### The ADL-PASCAL version

```
   program SEQ1ADL;
#  database ORDER;
   var CUS : TCUSTOMER;
   begin
#    open;
#    for CUS := CUSTOMER do
       writeln(CUS.NUM,CUS.NAME,CUS.ADDRESS);
#    endfor;
#    close;
   end.
```

### The basic PASCAL version

```
program SEQ1;
(*$I ORDER.TYP *)
(*$I DBMS.PAS *)
var CUS : TCUSTOMER;
begin
   dbopen('ORDER');
   dbfirst(CUSTOMER,CUS);
   while dbfound do
   begin
     writeln(CUS.NUM,CUS.NAME,CUS.ADDRESS);
     dbnext(CUSTOMER,CUS)
   end;
   dbclose
end.
```

## 5.1.5 SELECTIVE SEQUENTIAL PROCESSING

The program lists the characteristics of all the CUSTOMER entities whose NAME matches the value given by the user at the terminal. Since NAME doesn't identify a CUSTOMER, the basic PASCAL program has to check explicitly the NAME value of each CUSTOMER entity.

### The ADL-PASCAL program

```
   program GET1ADL;
#  database ORDER;
   var CUS : TCUSTOMER;
     NAM : string[15];
   begin
#    open;
     write('Enter customer name : '); readln(NAM);
#    for CUS := CUSTOMER(: NAME = NAM) do
       writeln(CUS.NUM,CUS.ADDRESS);
#    endfor;
#    close
   end.
```

### The basic PASCAL program

```
program GET1;
(*$I ORDER.TYP *)
(*$I DBMS.PAS *)
varCUS : TCUSTOMER;
   NAM : string[15];
begin
   dbopen('ORDER');
   write('Enter customer name : '); readln(NAM);
   dbfirst(CUSTOMER,CUS);
   while dbfound do
   begin
     if CUS.NAME = NAM
     then
       writeln(CUS.NUM,CUS.ADDRESS);
     dbnext(CUSTOMER,CUS)
   end;
   dbclose
end.
```

## 5.1.6 IMMEDIATE ACCESS BASED ON IDENTIFIER

The program displays the characteristics of the CUSTOMER entity (if any) whose number (NUM) matches the value given by the user at the terminal. A NUM value identifies at most one CUSTOMER entity. Notice that the ADL-PASCAL program is quite similar to the selective sequential program GET1ADL (see 4.5), since the programmer doesn't have to worry about the existence of an identifier in the selection condition.

### The ADL-PASCAL program

```
   program GET2ADL;  { first version }
#  database ORDER;
   var CUS : TCUSTOMER;
     NUMBER : integer;
   begin
#    open;
     write('Enter customer number : '); readln(NUMBER);
#    for CUS := CUSTOMER(: NUM = NUMBER) do
       writeln(CUS.NAME, CUS.ADDRESS);
#    endfor;
#    close
   end.
```

Since there is at most one matching CUSTOMER entity, the *for-endfor* loop can be replaced by an *entity assignment* statement as follows :

```
   program GET3ADL;  { second version }
#  database ORDER;
   var CUS : TCUSTOMER;
     NUMBER : integer;
   begin
#    open;
     write('Enter customer number : '); readln(NUMBER);
#    CUS := CUSTOMER(: NUM = NUMBER);
     if dbfound then writeln(CUS.NAME, CUS.ADDRESS);
#    close
   end.
```

### The basic PASCAL program

```
program GET2;
(*$I ORDER.TYP *)
(*$I DBMS.PAS *)
varCUS : TCUSTOMER;
   NUMBER : integer;
begin
   dbopen('ORDER');
   write('Enter customer number : '); readln(NUMBER);
   dbid(CUSTOMER,CUS);
   if dbfound then
        writeln(CUS.NAME,CUS.ADDRESS);
   dbclose
end.
```

### 5.1.7 A 2-LEVEL EMBEDDED ACCESS PROGRAM

The program prints a report giving the name and the total ordered quantity of each PRODUCT entity of the database. For each PRODUCT entity, the program examines all the LINE entities that are connected to it via PL, gets the values of ORDQ of each of them and then sums these values.

The part of the schema that is concerned with that program is as follows. The access paths that will be used are specified by directed arrows (an arc with no origin represents a sequential access).

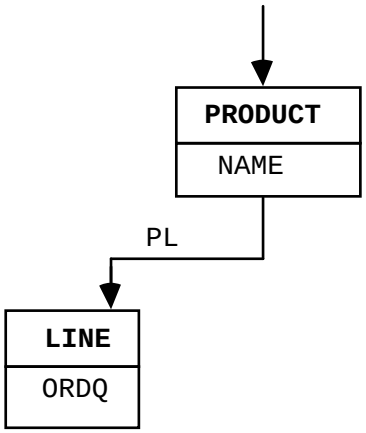


**Figure 5.2 -**

**The ADL-PASCAL program**

```
program SUM1ADL;
#  database ORDER;
   var PRO : TPRODUCT;
     L  : TLINE;
     Q  : real;
begin
#  open;
#    for PRO := PRODUCT do
       Q := 0;
#      for L := LINE(PL: PRO) do
         Q := Q + L.ORDQ;
#      endfor;
       writeln(PRO.NAME,Q);
#    endfor;
#  close;
end.
```

### The basic PASCAL program

```
program SUM1;
(*$I ORDER.TYP *)
(*$I DBMS.PAS *)
var PRO : TPRODUCT;
   L  : TLINE;
   Q  : real;
begin
   dbopen('ORDER');
   dbfirst(PRODUCT,PRO);
   while dbfound do
   begin
      Q := O:
         dbfpath(L,PRO,PL);
         while dbfound do
         begin
            Q := Q + L.ORDQ;
            dbnpath(L,PRO,PL)
         end;
      writeln(PRO.NAME,Q);
      dbnext(PRODUCT,PRO)
   end;
   dbclose
end.
```

## 5.1.8 A 4-LEVEL EMBEDDED ACCESS PROGRAM

That program is a bit more complex since it navigates through all the entity types of the database. Its purpose is to list the names of the PRODUCTS that are currently requested by a CUSTOMER identified by his(her) number, given at the terminal.

The main structure of the algorithm can be paraphrased into the pseudo-program :

*get the specified CUSTOMER*
*for each of his(her) ORDER*
*for each LINE of the current ORDER*
*get the corresponding PRODUCT*
*print its name*

The part of the database schema that is concerned with the program is as follows.

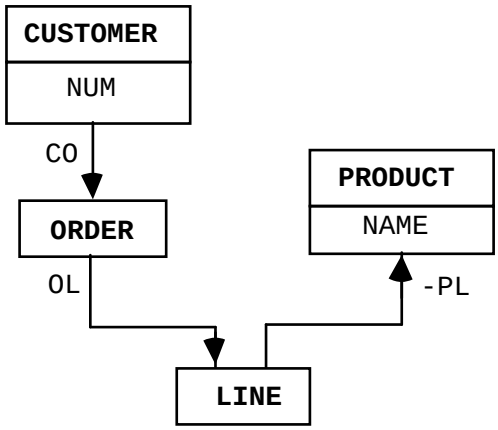


**Figure 5.3 -**

The PL relationship type from LINE to PRODUCT is noted —**PL** since the N-1 path will be used in the basic PASCAL program.

It is important to note that the program is not quite satisfactory since it may print the same PRODUCT name more than once. Can you prove it ? Can you state the general conditions under which such situations can occur ? Can you amend the program so that it avoids that problem ?

## The ADL-PASCAL program

```
   program LIST1ADL;
#  database ORDER;
   varCUS : TCUSTOMER;
      ORD : TORDER;
      L   : TLINE;
      PRO : TPRODUCT;
      NUMBER : integer;
   begin
#     open;
      write('Enter customer number : '); readln(NUMBER);
#     for CUS := CUSTOMER(: NUM = NUMBER) do
#        for ORD := ORDER(CO: CUS) do
#           for L := LINE(OL: ORD) do
#              PRO := PRODUCT(PL: L);
               writeln(PRO.NAME);
#           endfor;
#        endfor;
#     endfor;
#     close
```

```
end.
```

### The basic PASCAL program

```
program LIST1;
(*$I ORDER.TYP *)
(*$I DBMS.PAS *)
varCUS : TCUSTOMER;
   ORD : TORDER;
   L   : TLINE;
   PRO : TPRODUCT;
begin
   dbopen('ORDER');
   write('Enter customer number : '); readln(CUS.NUM);
   dbid(CUSTOMER,CUS);
   if dbfound then
   begin
      dbfpath(ORD,CUS,CO);
      while dbfound do
      begin
         dbfpath(L,ORD,OL);
         while dbfound do
         begin
            dbfpath(PRO,L,-PL);
            if dbfound then
               writeln(PRO.NAME);
            dbnpath(L,ORD,OL)
         end;
         dbnpath(ORD,CUS,CO)
      end;
   end;
   dbclose
end.
```

## 5.1.9 Creating a simple entity

The following procedure creates a CUSTOMER entity from the contents of parameter IC. IC is a PASCAL record containing the components NUM, NAME and ADDRESS of the same types as their counterparts in the CUSTOMER entities. The procedure returns an entity variable referencing the newly created entity together with a diagnostic code. The database is already open.

```
procedure CR_CUSTOMER(IC : IN_CUST; var C : TCUSTOMER;
                      var RCODE : integer);
begin
   C.NUM := IC.NUM;
   C.NAME := IC.NAME;
   C.ADDRESS := IC.ADDRESS;
   dbcreate(CUSTOMER,C);
```

```
    case dbstatus of
      0 :    RCODE := 0;
      10 :   RCODE := 1;
      else     RCODE := 3
    end
  end;
```

### 5.1.10 Creating a complex entity

The procedure creates an ORDER entity together with its subordinate LINES from the contents of parameter IO. IO is a PASCAL record containing the components NUM, DATE, N_OF_LINES and an array of LINE records containing PNAME and QTY. These informations are validated, so that the procedure has neither to check the uniqueness of NUM, nor the existence of PRODUCT entities with NAME = PNAME. The procedure returns an entity variable referencing the newly created ORDER together with a diagnostic code. The database is already open.

```
procedure CR_ORDER(IO : IN_ORDER; var O : TORDER;
                   var RCODE : integer);
var L : TLINE;
   P : TPRODUCT;
   IL : integer;
begin
        {create the ORDER entity}
   O.NUM := IO.NUM;
   O.DATE := IO.DATE;
   dbcreate(ORDER,O);
        {create the LINE entities}
   for IL := 1 to IO.N_OF_LINES do
   begin
        {create a LINE entity}
     L.ORDQ := IO.LINE[IL].QTY;
     dbcreate(LINE,L);
        {connect it to the ORDER entity}
     dbinsert(L,O,OL);
        {connect it to the PRODUCT entity}
     P.NAME := IO.LINE[IL].PNAME;
     dbid(PRODUCT,P);
     dbinsert(L,P,PL)
   end
end;
```

### 5.1.11 Deleting a complex entity

The procedure removes from the database the ORDER entity identified by NUM = IO together with its subordinate LINES. The database is already open and IO is a valid order number. The main idea would be to get the ORDER entity, then to loop

on each LINE entity which is connected to it and to delete it and finally to delete the ORDER entity :

```
procedure DEL_ORD(IO : integer);
var O : TORDER;
   L : TLINE;
begin
   O.NUM := IO;  {get the ORDER entity}
   dbid(ORDER,O);
   dbfpath(L,O,OL);  {get the first LINE entity}
   while dbfound do
   begin
      dbdelete(LINE,L); {delete it}
      dbnpath(L,O,OL)  {get the next LINE entity}  {*}
   end;
   dbdelete(ORDER,O)  {delete the ORDER entity}
end;
```

However, the reader should be aware that the basic idea could be wrong since asking (via *dbnpath*, see * above} for the entity that follows the entity that has just been deleted makes no sense (it's just like sawing the branch on which you sit). Once the first LINE of an ORDER has been deleted, the next one has become the first one, so we have to get the first one once again. The incriminated statement can be replaced by the following :

```
dbfpath(L,O,OL)  {get the first LINE entity}  {*}
```

Coming back to the definition of the `dbnpath` primitive, we can observe that asking for the entity that follows *none*, is interpreted as asking for the first one. So the first program **do behave correctly** , and can be kept as such. We just suggest a slight modification :

```
dbnpath(L,O,OL)  {get the first LINE entity}  {*}
```

Since a normal, well-behaved, loop on the LINE entities solves the problem correctly, an ADL-PASCAL program can also be designed as follows.

```
procedure DEL_ORD1(IO : integer);
var
#  O : entity ORDER;
#  L : entity LINE;
begin
#  O := ORDER(:NUM = IO);
#  for L := LINE(OL: O) do
```

```
#    delete LINE L;
# endfor;
# delete ORDER O;
end;
```

### 5.1.12 Transferring LINE entities from a PRODUCT to another one

Suppose all the LINE entities that should have been connected to PRODUCT with NAME = N1 have been erroneously connected to PRODUCT with NAME = N2. The following procedure corrects that situation. That problem contains a programming trap similar to that of procedure DEL_ORD. Once a LINE has been transferred to the new PRODUCT, we cannot ask for the next one. The transfer has made the second LINE the first one for the wrong PRODUCT. Thus, we must repeatedly ask for the first LINE of the wrong PRODUCT.

That program is a good example of the use of several entity variables of the same type.

```
procedure TRSF_LIN(N1,N2 : string20);
var P1, P2 : TPRODUCT;
   L : TLINE;
begin
   P2.NAME := N2;   {get the wrong PRODUCT entity}
   dbid(PRODUCT,P2);
   P1.NAME := N1;      {get the correct PRODUCT entity}
   dbid(PRODUCT,P1);
   dbfpath(L,P2,PL);   {get the first LINE entity}
   while dbfound do
   begin
      dbinsert(L,P1,PL); {reconnect it}
      dbfpath(L,P2,PL);  {get the first LINE entity}
   end;
end;
```

In that case, however, the procedure cannot be entirely written in ADL-PASCAL (why ?). The following program is a little more compact :

```
procedure TRSF_LIN(N1,N2 : string20);
var
# P1, P2 : entity PRODUCT;
# L : entity LINE;
begin
# P2 := PRODUCT(:NAME = N2);   {get the wrong PRODUCT entity}
# P1 := PRODUCT(:NAME = N1);   {get the correct PRODUCT
entity}
   dbfpath(L,P2,PL);     {get the first LINE entity}
   while dbfound do
   begin
      dbinsert(L,P1,PL);   {reconnect it}
```

```
      dbfpath(L,P2,PL);    {get the first LINE entity}
    end;
  end;
```

A specific trick could allow us to write a seemingly more elegant ADL-PASCAL program. Replacing `dbfpath(L,P2,PL)` by the equivalent sequence `dbclear(L)`; `dbnpath(L,P2,PL)` in TRSF_LIN gives us a traditional loop structure which can be translated into ADL-PASCAL as follows.

```
  procedure TRSF_LIN(N1,N2 : string[20]);
    {example of a dirty program}
  var
  #  P1, P2 : entity PRODUCT;
  #  L : entity LINE;
  begin
  #  P2 := PRODUCT(:NAME = N2);   {get the wrong PRODUCT entity}
  #  P1 := PRODUCT(:NAME = N1);   {get the correct PRODUCT
entity}
  #  for L := LINE(PL: P2) do
  #    modify LINE L(PL:P1);
  #    L := null;      {get the first LINE entity}
  #  endfor;
  end;
```

Though leading to a simpler program, such a practice cannot be recommended since the loop variable is explicitly modified by the loop body.

## 5.2 The BILL-OF-MATERIAL (BOM) database

### 5.2.1 Description of the application domain

The second database concerns the description of a collection of machine parts in such a way that a part can be made up of other simpler parts, called its sub-parts. We admit that a part enters into at most one other part, called its super-part. Moreover, a part cannot go into itself, neither directely nor indirectely.

The database schema includes only one entity type, namely **PART**. A PART entity is characterized by its name (**PNAME**), which is an identifier, the number of such parts that enter into the super-part (**QTITY**) and the weight of one such part (**WEIGHT**).

The super-part / sub-part relationships are represented by the **PSP** relationship type.

### 5.2.2 Graphical schema of the BOM database

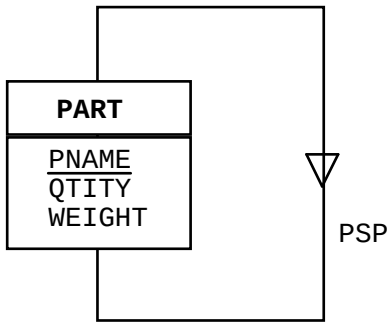


**Figure 5.4 - Figure 4.2** - Schema of the BOM database.

### 5.2.3 The programming environment of the BOM database

The following definitions are available in the file BOM.TYP that must be included in any program working in the BOM database. That file has been automatically generated by the Schema Processor.

```
const  PART = …;
       PSP  = …;
type
       TPART   = record
                 ...
                 PNAME  :string[35];
                 QTITY  :integer;
                 WEIGHT : real;
               end;
```

### 5.2.4 PART EXPLOSION

The schema is basically *recursive* and naturally induces some recursive procedures such as the following.

Let's design a program that displays the characteristics of a given part together with all its direct and indirect sub-parts. We will define a procedure, called EXPLODE, that displays the characteristics of a part, then applies itself to each sub-part of that part.

The schema components that will be used are shown in the following figure.

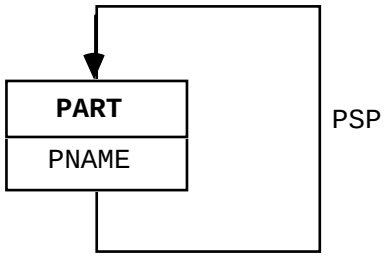


**Figure 5.5 -**

## The ADL-PASCAL program

```
   program BOM1ADL;
#  database BOM;
   var P : TPART;
      N : string[35];

   procedure EXPLODE(var P : TPART);
   var SP : TPART;
   begin
      writeln(P.PNAME);
#     for SP := PART(PSP: P) do
         EXPLODE(SP);
#     endfor;
   end;

   begin
#     open;
      write('Enter name of the part to explode : '); readln(N);
#     P := PART(: PNAME = N);
      if dbfound then EXPLODE(P);
#     close;
   end.
```

The display would be made nicer by indenting the description of each part according to its level of decomposition.

## The basic PASCAL program

```
   program BOM1;
   (*$I BOM.TYP *)
   (*$I DBMS.PAS *)
```

```
   var P : TPART;
     N : integer;

   procedure EXPLODE(var P : TPART);
   var SP : TPART;
   begin
     writeln(P.PNAME);
     dbfpath(SP,P,PSP);
     while dbfound do
     begin
       EXPLODE(SP);
       dbnpath(SP,P,PSP)
     end
   end;

   begin
     dbopen('BOM');
     write('Enter id-number of the part to explode : ');
readln(P.NAME);
     dbid(PART,P);
     if dbfound then EXPLODE(P);
     dbclose;
   end.
```

### 5.2.5 COMPUTING THE WEIGHT OF A PART

A similar structure can be used to solve the problem of computing the weight of a part, knowing the weight and the number of each of its sub-parts, and so on recursively. The PWEIGHT procedure updates the database so that the WEIGHT of each super-part depending on a given part P is computed as the sum of the weight of all its sub-parts.

```
   program WEIGHTADL;
#  database BOM;
   var P : TPART;{the root of the part decomposition}
     N : string[35];{the name of the root part}

   function PWEIGHT(var P : TPART): real;
   var SP : TPART;{one of the components of part P}
     PW : real;{the weight of part P}
   begin
     PW := 0
#    for SP := PART(PSP: P) do
       PW := PW + SP.QTITY*PWEIGHT(SP);
#    endfor;
     if PW > 0 then{update part P if it has at least one
component}
     begin
       P.WEIGHT := PW;
```

```
       dbmodify(PART,P)
     end;
     PWEIGHT := PW{return the weight of part P}
   end;

   begin
#    open;
     write('Enter name of the part to update : '); readln(N);
#    P := PART(: PNAME = N);
     if dbfound then writeln('Weight of part ',N,' =
',PWEIGHT(P));
#    close;
   end.
```

A basic PASCAL version can easily be derived from that program.

## 5.3 THE SOLID MODELLING DATABASE

### 5.3.1 Description of the aplication domain

A solid is made up of volumes. A volume can be either elementary or made up of several other volumes. An elementary volume is made up of faces. A face is defined by the polygon that delimits it. A polygon is a coplanar chain of edges. An edge is a line. A line is defined by two points. A point is defined by its coordinates X and Y.

### 5.3.2 Graphical schema of the BOM database

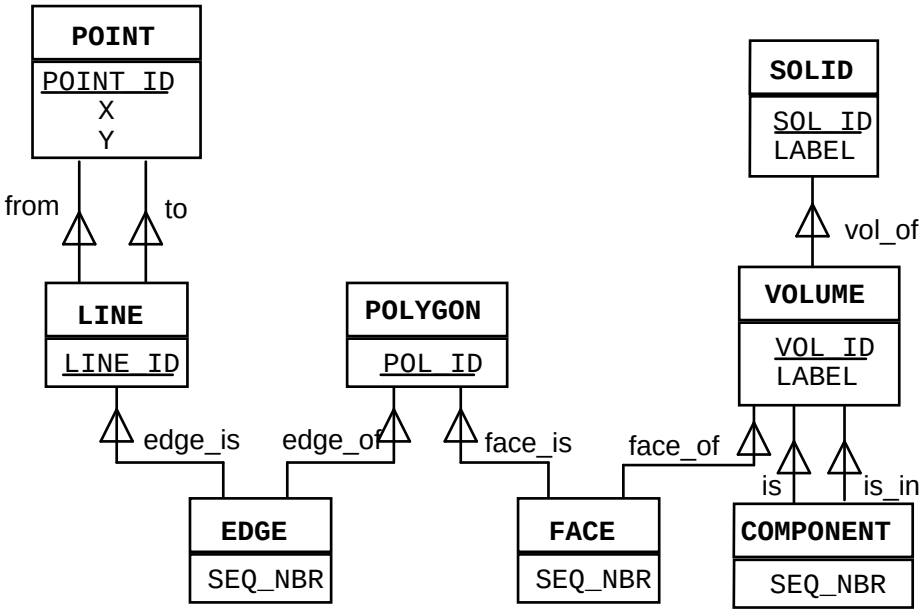


**Figure 5.6 - Figure 4.3** - Schema of the SOLID database.

## 5.4 A SIMPLIFIED INFORMATION RESOURCE DICTIONARY

### 5.4.1 Description of the application domain

The database describes the activities of a EDP application development center of a large company. The activities are organized in projects. A project consists in building databases and applications in order to meet the needs of users. Databases and applications are developped and maintained by teams of programmers. A database is described by its schema, in terms of entity types, relationship types between two entity types, attributes (some of them can be made up of other, more elementary, attributes). An application is a set of programs solving a problem. A program can work on a database. A program is an assembly of procedures calling each other. A procedure executes actions (create, read, modify, etc) on entities. A programmer is a member of a development team (formely, he could have been a member of other teams). A procedure is created and maintained by a programmer. A user uses programs and databases.

Such a database is called a **Data Dictionary**, or more and more often **Information Resource Dictionary (IRD)**. Actual IRD are much more precise and comprehen-

sive than the PROJECT database. For instance, a widespread IRD (IDD of Cullinet) comprises more than 150 entity types.

On the other hand, a database that describes other databases (generally including itself!), is sometimes called a **meta database**.

The NDBS environment includes a schema processor that uses a small data dictionary, which is a subset of the IRD database. The schema of that data dictionary includes the following entity types : DATABASE, ENTITY_TYPE, REL_TYPE and ATTRIBUTE.

### 5.4.2 Graphical schema of the IRD database

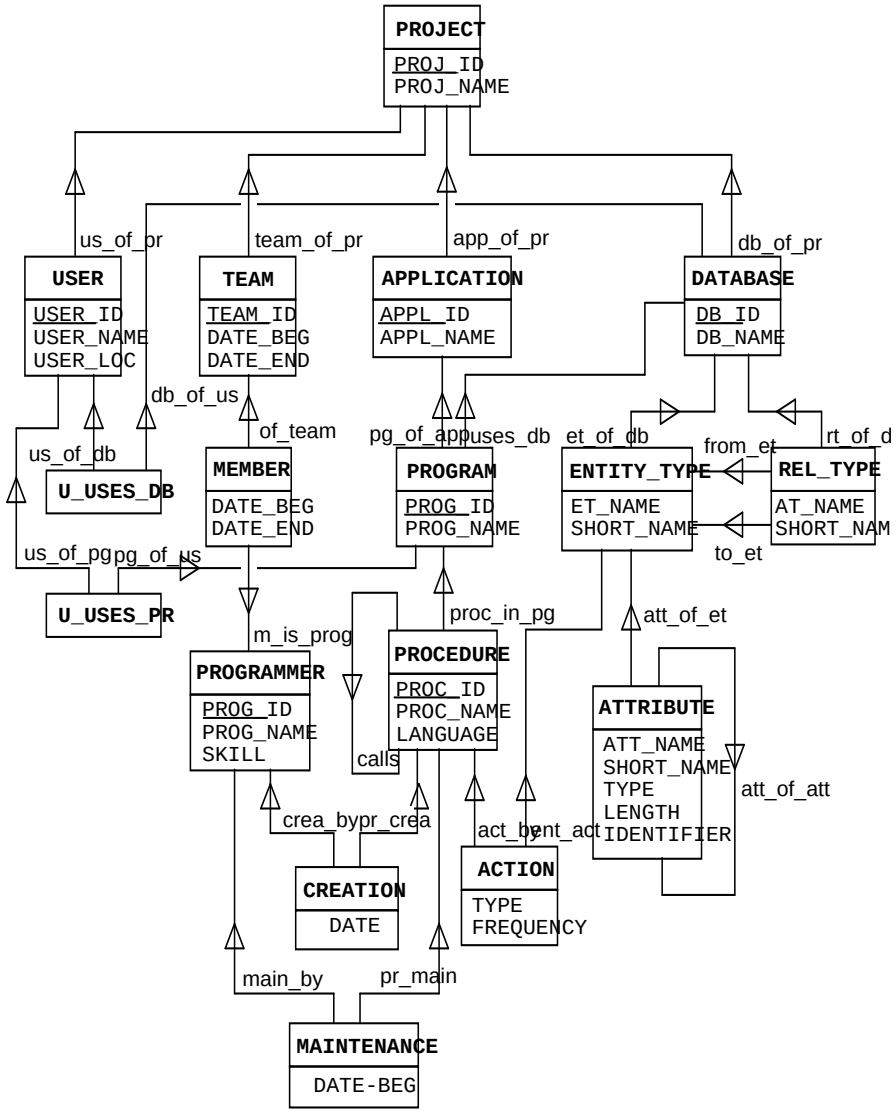


**Figure 5.7 - Figure 4.4** - Schema of the IRD database. Four sub-domains are visible : users, programmers, programs and data. [Sorry, the import process failed! ]

## 5.5 Simulation of the `dbid` primitive

This is a model procedure that simulates the **dbid** primitive for the ORDER database. It can be adapted easily for any database when the version of NDBS in use lacks that primitive. The trick is of course not recommanded for large databases.

```
procedure DBID(ETYPE:integer; var EV);
type ENTITY = array[1..100] of byte;
var  TEMP : array[1..100] of byte ;  {common entity variable}
     CUS : TCUSTOMER absolute TEMP;
     PRO : TPRODUCT absolute TEMP;
     ORD : TORDER absolute TEMP;
var  EVAR : ENTITY absolute EV;{represents parameter EV}
     ELEN : integer;{length of the current entity variable}
     EFOUND : boolean;{has the entity been found ?}
     CNUM :   integer;{CUSTOMER id value}
     PNAME :  string[20]; {PRODUCT id value}
     ONUM :   integer; {ORDER id value}
begin
   EFOUND := false;
   case ETYPE of
      CUSTOMER:begin move(EVAR,CUS,sizeof(CUS));
             CNUM := CUS.NUM;
             dbfirst(CUSTOMER,CUS);
             while dbfound and not EFOUND do
               ifCUS.NUM = CNUM then EFOUND := true
                else dbnext(CUSTOMER,CUS);
             if EFOUND then
                move(CUS,EVAR,sizeof(CUS))
          end;
      PRODUCT:begin move(EVAR,PRO,sizeof(PRO));
             PNAME := PRO.NAME;
             dbfirst(PRODUCT,PRO);
             while dbfound and not EFOUND do
               if PRO.NAME = PNAME then EFOUND := true
                else dbnext(PRODUCT,PRO);
             if EFOUND then
                move(PRO,EVAR,sizeof(PRO))
          end;
      ORDER : begin move(EVAR,ORD,sizeof(ORD));
             ONUM := ORD.NUM;
             dbfirst(ORDER,ORD);
             while dbfound and not EFOUND do
               if ORD.NUM = ONUM then EFOUND := true
                else dbnext(ORDER,ORD);
             if EFOUND then
                move(ORD,EVAR,sizeof(ORD))
          end;
      else dbstatus := 10;
   end;
end;
```

# Part 6

# 6. DDL: a Data Definition Language for NDBS Data Bases

## 6.1 Objective

NDBS-DDL is a data definition language that allows a user to define the schema of an NDBS data base. The NDBS-DDL compiler is a processor that loads the description of a data base expressed by a DDL text in the data dictionary, and that initializes a NDBS data base according this description.

The compiler checks the correctness of the data base description, gives explicit diagnostic when errors are encountered, load the description in the data dictionary, generates an empty data base (the *.DTB file), and generates the PASCAL text defining the constants, types and variables that describe the data structures for application programs (the *.TYP file).

## 6.2 Definition of the NDBS-DDL language

### 6.2.1 Meta syntax[1]

| | |
|---|---|
| symbol | natural language description |
| *symbol* | subexpression still to be defined (meta term) |
| **symbol** | constant symbol of the language |
| \| | separates alternatives |
| [ *symbol* ] | sub-expression *symbol* is optional |
| *symbols* * | expression *symbol* must appear at least once and possibly several times |

### 6.2.2 Syntax of NDBS-DDL

*schema* ::= *database* *entity-type** [*rel-type**] [*physical-spec*] **end**
*database* ::= **database** *dbname*
*entity-type* ::= **entity-type** *etname* [*identifier* ] [*attributes*]
*identifier* ::= **identifier** *idname* [**sorted**]

1. The grammar of a language is described by the syntactic and semantic rules that govern the way of building valid sentences in this language. Expressing the syntactic rules needs in turn a specific language. Since the latter is a *language to describe languages*, it is called a **meta-language**. The meta-language used here is a variant of the classical Backus-Naur form (BNF).

*attributes* ::= **begin** attribute* **end**
*rel-type* ::= **rel-type** *rtname role1 role2*
*role1* ::= **between one** *etname-list*
*role2* ::= **and** *connectivity2 etname*
*etname-list* ::= *etname* [ **,** *etname* * ]
*connectivity2* ::= **one** | **many**
*attribute* ::= *attname* **:** *attribute-type*
*attribute-type* ::= *integer-type* | *string-type* | *real-type* | *character-type* | *boolean-type* |
*byte-type* |*array-type* | *record-type* | *text-type*
*integer-type* ::= **integer**
*string-type* ::= **string[***integer***]**
*real-type* ::= **real**
*character-type* ::= **char**
*boolean-type* ::= **boolean**
*byte-type* ::= **byte**
*array-type* ::= **array[1..***integer***] of** *attribute-type*
*record-type* ::= **record** *attribute** **end**
*text-type* ::= **txt**
*physical-spec* ::= **physical-spec** [*buffer* ] [*storage-scheme**]
*buffer* ::= **buffer** *integer*
*storage-scheme* ::= **storage** *etname* **clustered** | **storage** *etname* **random** *range*
*range* ::= *integer1 integer2*
*integer,integer1,integer2* ::= integer number
*dbname* ::= data base name
*etname* ::= entity type name
*rtname* ::= relationship type name
*idname* ::= name of the identifier of an entity type
*attname* ::= attribute name

### 6.2.3 Short description

- **Comments**

A line the first character of which is ***** is ignored. It can thus contain any comment. Leading blank characters are ignored.

- **Schema description**

*schema* ::= *database*
*entity-type**
[*rel-type**]
[*physical-spec*]
**end**

The description of the schema consists of the data base declaration, one or more entity type descriptions, zero, one or more relationship type descriptions and the physical parameters specification.

- **Data base declaration**

*database* ::= **database** *dbname*

Declares a data base with name *dbname*.

- **Entity type description**

*entity-type* ::= **entity-type** *etname*
[*identifier* ]
[*attributes*]
*identifier* ::= **identifier** *idname* [**sorted**]
*attributes* ::= **begin**
attribute*
**end**

The description of the entity type *etname* consists of the declaration of its identifier (optional), and of its attributes (optional). The identifier clause specifies the attribute *idname* which is of one of the following types : integer, real, string, byte or character. It cannot be a component of a record or an array attribute. It cannot be a text. An identifier can be declared *sorted*. In that case, a sequential scanning of the entities will supply them in ascending order of the identifier values. Otherwise, the order will be time-sequenced (FIFO).

- **Attribute description**

*attribute* ::= *attname* **:** *attribute-type* **;**
*attribute-type* ::= *integer-type* | *string-type* | *real-type* | *character-type* | *byte-type* |
*boolean-type* | *array-type* | *record-type* | *text-type*
*integer-type* ::= **integer**
*string-type* ::= **string[***integer***]**
*real-type* ::= **real**

*character-type* ::= **char**
*byte-type* ::= **byte**
*boolean-type* ::= **boolean**
*array-type* ::= **array[1..***integer***] of** *attribute-type*
*record-type* ::= **record**
*attribute**
**end**
t*ext-type* ::= **txt**

The description of an attribute follows the PASCAL syntax, except that the semi-colon symbol (';') is omitted. A text attribute cannot be organized as an array nor can it be a component of a record structure.

- **Relationship type description**

*rel-type* ::= **rel-type** *rtname role1 role2*
*role1* ::= **between one** *etname-list*
*role2* ::= **and** *connectivity2 etname*
*etname-list* ::= *etname* [ **,** *etname* * ]
*connectivity2* ::= **one** | **many**

The relationship type is declared in such a way that the paths are *one-to-many* or *one-to-one* from *role1* to *role2*.

- **Physical specifications**

*physical-spec* ::= **physical-spec**
[*buffer* ]
[*storage-scheme**]
*buffer* ::= **buffer** *integer*
*storage-scheme* ::= **storage** *etname* **clustered** | **storage** *etname* **random** *range*
*range* ::= *integer1 integer2*

The (optional) last section of the schema description is devoted to the physical description of the data base. The *buffer* clause allows for the specification of the buffer size (in pages). The default value is 8 pages. The storage scheme of each entity type can be specified either as *clustered* or *random*. In the latter case, the page range must be specified. If the storage scheme specification of an entity type is omitted, its storage scheme is *clustered*.

- **Remark**

The following clauses will be written one per line and one line per clause :

**database** *dbname*
**end** (from *schema*)
**entity-type** *etname*
**identifier** *idname*

**begin** and **end** (from *attributes*)
**rel-type** *rtname* **between one***etname* **and** *connectivity2etname*
*attname* **: integer**
*attname* **: string[***integer***]**
*attname* **: real**
*attname* **: char**
*attname* **: byte**
*attname* **: boolean**
*attname* **: array[1..***integer***] of** *attribute-type* (if *attribute-type* is not *record-type*)
*attname* **: array[1..***integer***] of record**
*attname* **: record**
*attname* **: text**
**buffer** *integer*
**storage** *etname* **clustered**
**storage** *etname* **random** *range*
**end** (from *record-type*)

## 6.3 The NDBS-DDL versions

NDBS is currently available in two versions, namely version 0 and version 2 (version 1 is for internal use only). NDBS V.0 is a tiny version specifically aimed at educational use. This subset doesn't include the integrated data dictionary, multi-base programming, index management, identifier checking, text management, one-to-one relationship types nor multi-origin relationship types. The DDL specified above corresponds to version 2. The DDL V.0 language is submitted to the following restrictions :

- *identifier* clause : no **sorted** option
- *role1* clause : no **one** keyword
- *role2* clause : no **one** | **many** keyword
- no *text* clause
- the *etname-list* clause develops to *etname* only.

## 6.4 Examples of data base DDL schemas

### 6.4.1 An order/invoice management data base

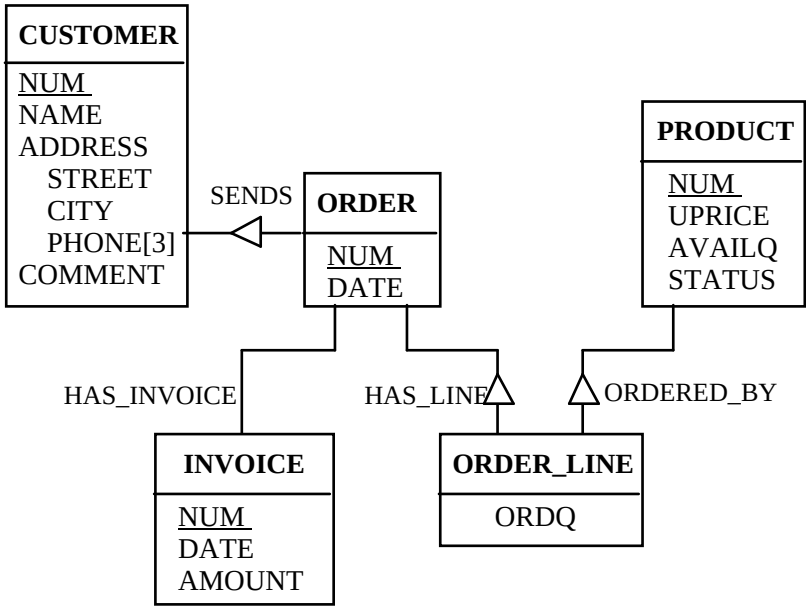


**Figure 6.1 - Figure 1 -** A revised version of the ORDER data base.

```
database ORDER

   entity-type CUSTOMER
      identifier NUM
      begin
         NUM : integer
         NAME : string[15]
         ADDRESS : record
            STREET : string[20]
            CITY : string[30]
            PHONE : array[1..3] of string[10]
         end
         COMMENT : txt
      end

   entity-type ORDER
      identifier NUM
      begin
         NUM : integer
         DATE : string[6]
      end
```

```
    entity-type ORDER_LINE
       begin
          ORDQ : integer
       end

    entity-type PRODUCT
       identifier NAME sorted
       begin
          NAME : string[20]
          UPRICE : real
          AVAILQ : integer
          STATUS : char
       end

    entity-type INVOICE
       identifier NUM
       begin
          NUM : integer
          DATE : string[6]
          AMOUNT : real
       end

    rel-type SENDS between one CUSTOMER and many ORDER
    rel-type HAS_LINE between one ORDER and many ORDER_LINE
    rel-type ORDERED_BY between one PRODUCT and many
ORDER_LINE
    rel-type HAS_INVOICE between one ORDER and one INVOICE

physical-spec
    buffer 30
    storage CUSTOMER random 1 1000
    storage ORDER clustered
    storage LINE clustered
    storage INVOICE clustered
    storage PRODUCT random 250 750
end
```

### 6.4.2 An office automation data base (extract)

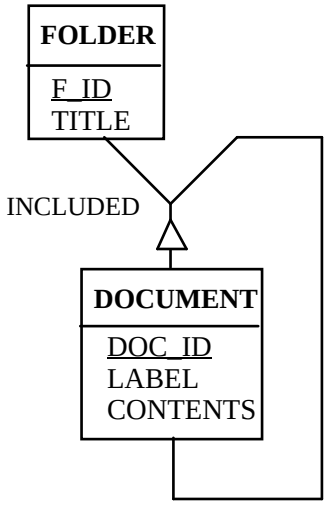


**Figure 6.2 - Figure 2 -** The interpretation of the OFFICE schema is as follows : a document is included either in a folder or in another document; a document is identified by its DOC_ID.

```
database OFFICE

   entity-type FOLDER
      identifier FOLD_ID
      begin
        F_ID :  integer
        TITLE : string[25]
      end

   entity-type DOCUMENT
      identifier DOC_ID
      begin
         DOC_ID :   integer
         LABEL :    string[50]
         CONTENTS : txt
      end

   rel-type INCLUDED between one FOLDER,DOCUMENT and many
DOCUMENT

end
```

## 6.5 Building a data base

### 6.5.1 Building a data base with NDBS V0.2

The user is provided with two ways of defining a new data base, namely through the interactive data dictionary system (DIC) and the DDL compiler (DDL).

#### Defining a data base with the data dictionary system

The data dictionary system comprises a data dictionary and a management program for this dictionary. The data dictionary is an NDBS data base with name DIC.DTB. DB schemas can be stored in this data base; they can be updated, retrieved and processed. The management program is a PASCAL-NDBS program (with name DIC.COM) that allows interactive input, updating, consulting and processing of data base schemas. When a schema is considered complete and consistent, a data base can be initialized according to the data structures specified. This initialization consists of a formated, empty data base (such as ORDER.DTB), and of a text file (named, say, ORDER.TYP) containing PASCAL data types and constants that will be included in any PASCAL-NDBS program in order to gain acces to the data base.

To define a new data base, you need a disk with the following files :

- the schema processor (DIC.COM and DIC.000)
- the data dictionary (DIC.DTB); you may find it better to define a new one by copying the empty data dictionary : *copy DIC.VID DIC.DTB*
- the message file (DIC.MSG)

Then run the DIC management program and define your data base.

When the schema is OK, ask for the generation of the DB/table. The output will be :

- an empty data base (such as ORDER.DTB)
- a PASCAL text file containing the constants and types describing to the new data base (ORDER.TYP); that file will be included in any program using the ORDER data base.

If you intend to write programs working on the data dictionary, you will use the DIC.TYP PASCAL definition file.

#### Defining a data base with the DDL compiler

The DDL compiler is an NDBS processor that loads a data base schema into the data dictionary and that optionally generates an empty data base and the text file containing the PASCAL declarations.

You need the following files :

- the DDL compiler (DDL.COM and DDL.000)
- the data dictionary (DIC.DTB); you may find it better to define a new one by copying the empty data dictionary : *copy DIC.VID DIC.DTB*
- a DDL text that defines the schema your data base.

The DDL text (let's call it INVOICE.DDL) can be processed three ways :

1. checks its syntax;
   calling : **DDL INVOICE.DDL 0**
2. same as 1 + loads the description in the data dictionary if it is correct;
   calling : **DDL INVOICE.DDL 1**
3. same as 2 + generates an empty data base (*.DTB) and the constant/type file (*.TYP)
   calling : **DDL INVOICE.DDL 2**

Note : after loading the schema in the data dictionary (processing 2), the DIC processor can be used as well, to modify, report and generate the data base files, as described above.

If the DDL text defines a data base that already exists in the data dictionary, the old description is replaced, the old *.TYP file is replaced and the old *.DTB file is renamed *.BAK.

The error messages appear on the screen. Be aware that a first error may induce several other errors in the following of the text. Some of them may seem odd if the compiler has lacked some information. Therefore, correct the first errors before trying to understand the ones that puzzle you.

### 6.5.2 Building a data base with NDBS V2

One of the main differences between version 0 and version 2 is that any NDBS V.2 data base includes a data dictionary. Therefore, there is no need for a dedicated data dictionary.

NDBS V.2 users are provided with two programs for initializing a data base, namely DDL and INIT_DB. The DDL program behaves as in version 0, except that the data dictionary is in the data base itself. It is aimed at simple users with classic needs. The INIT_DB program allows more sophisticated activities that can be required by advanced users.

#### Compiling a DDL schema with DDL

According to this procedure, the commands are the same as in version 0. A DDL text, such as INVOICE.DDL, can be processed three ways :

1. checking of its syntax;
   calling : **ddl INVOICE.DDL 0**
2. same as 1 + loading of the description in the data dictionary part of a new data base if it is correct;
   calling : **ddl INVOICE.DDL 1**
3. same as 2 + initializing the data base and the constant/type file (*.TYP)
   calling : **ddl INVOICE.DDL 2**

*Note* : the DDL program is a batch procedure (DDL.BAT) that makes use of the INIT_DB program. The equivalence is as follows :

```
ddl INVOICE.DDL 0    =    init_db INVOICE.DDL C
ddl INVOICE.DDL 1    =    init_db INVOICE.DDL DL
ddl INVOICE.DDL 2    =    init_db INVOICE.DDL DLG
```

## Data bases and data dictionaries : a deeper look

For ordinary data base users, a data base comprises two parts (Figure 3), a schema part (e.g. ORDER) and data part (e.g. ORDER data). The role of the schema part is to explain the DBMS how to interpret the data in order to retrieve and manage them.

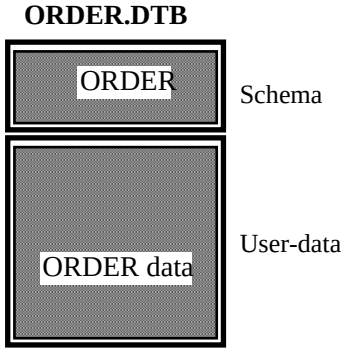


**Figure 6.3 - Figure 3 -** The traditional user's view of a data base.

As already mentionned, an NDBS V2 data base includes a data dictionary in which one can store E-R schemas of any complexity [2]. The data dictionary is therefore a part of the user data base. A data base comprises a data dictionary (DD) and a user part, containing, for instance, ORDER data. As far as the schemas are concerned, an NDBS data base comprises two suschemas, one that describes the data dictionary (in terms of SCHEMA, ENTITY_TYPE, REL_TYPE, ATTRIBUTE, IDENTIFIER,

etc), and one that describes the user data (in terms of CUSTOMER, ORDER, PRODUCT, etc). As far as the data are concerned, an NDBS data base comprises two parts, one that is made up of the description of the user schema (among others), in terms of ENTITY_TYPES entities with name 'CUSTOMER', 'ORDER', etc, and the other which is made up of user data, in terms of CUSTOMER entities with name 'Smith', 'Dupont', etc. Moreover, since the data dictionary is a part of the data base, it is described by an NDBS schema called the meta-schema[3] of the data base. This meta-schema (DD-schema) can be stored in the data dictionary itself, making the latter auto-descriptive. These parts are described in Figure 4. Let's recall that this organization need not be understood by simple users, that perceive the simplified components of Figure 3 only.

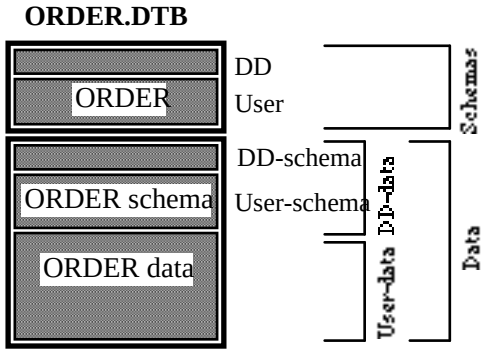


**Figure 6.4 - Figure 4 -** Organization of most current data base systems. The schema part (Schemas) specifies the structure of the user data (Schemas.User) and of the data dictionary data Schemas.DD). The data part (Data) comprises the user data (User-data) and the data dictionary data (DD-data). The latter (DD-data) may be made up of data describing the user schema (User-schema) and of data describing the data dictionary itself (DD-schema). Some of these parts may be empty at some times.

This organization makes it possible for a program to access both schema descriptions and user data with the same access primitives[4].

2. This data dictionary is a subset of the TRAMIS specification data base. TRAMIS is a DB design tool developed at the University of Namur (Belgium). See document *TRAMIS - a transformation-based database CASE tool*, October 1990 for more informations. Nevertheless, only schemas complying with the NDBS model will be processed correctly by INIT_DB.

3. I.e. a schema that describes data base schemas, included itself. Moreover, data dictionaries are sometimes called *meta data bases*.

4. The same way an SQL program can use both system catalog tables and user tables.

## The INIT_DB processor

This program allows users process DDL texts, load their contents in a data dictionary, initialize data bases, generate the DDL text of an existing data base, etc.

Its functions are called for by specifying one or several function codes. Using INIT_DB without arguments produces a help text but has no other effect.

The functions of INIT_DB are the following.

- Checking the correctness of a DDL text (option **C**, for check). The result is a list of diagnostic messages (Figure 5).
  Calling : **init_db ORDER.DDL C**

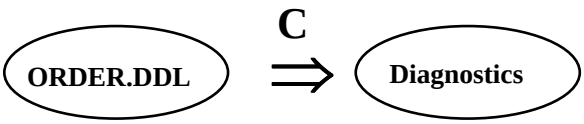


**Figure 6.5 - Figure 5 -** Checking of a DDL text. No data base is involved in the process.

- Building an empty data directory (option **D**, for data dictionary). The result is a data base whose only schema is that of the data directory (Figure 6). It contains no user schema and no data.
  Calling : **init_db ORDER.DTB D**

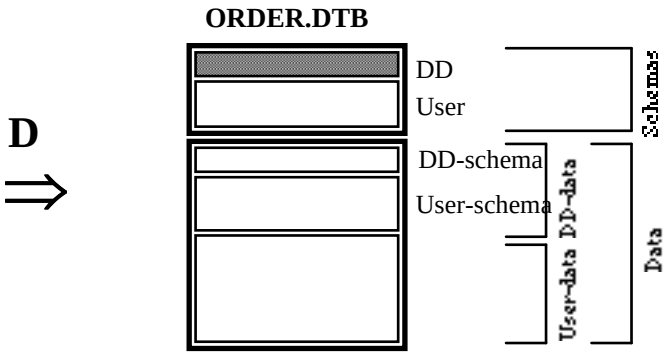


**Figure 6.6 - Figure 6 -** Building a minimal data base, i.e. an empty data dictionary.

- Loading the data describing the DD-schema (i.e. the meta-meta-data) (option **M**, for meta-meta-data). The other data base parts are left unchanged (Figure

7). This function will be rarely used and is provided for completeness purpose only.

Calling : **init_db ORDER.DTB M**

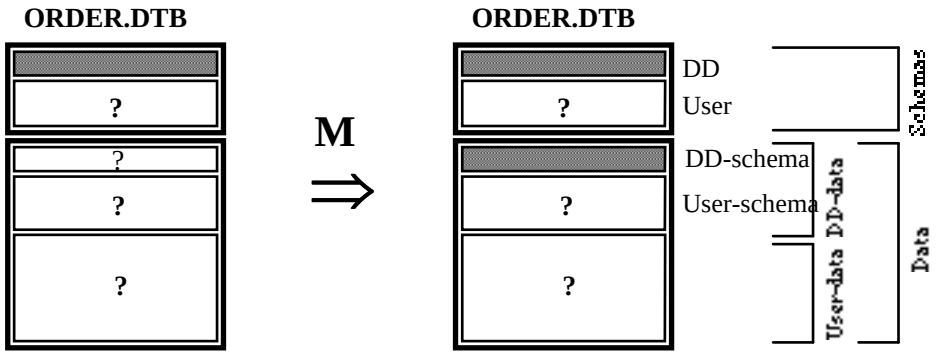


**Figure 6.7 - Figure 7 -** Loading in the data dictionary the data describing the data dictionary itself.

- Loading in the data dictionary the schema described in a DDL text (option **L**, for loading). A check operation is carried out (a diagnostic message list is supplied as well). The other data base parts are left unchanged (Figure 8). Note that this schema may be in contradiction with the current state of the data base[5].

  Calling : **init_db ORDER.DTB L**

5. To be precise, the DBMS doesn't use these data directly, but a more efficient, compiled version of them, called the internal schema tables. These tables are built from these data by the G function and are stored in the data base (in the *Schemas* part).

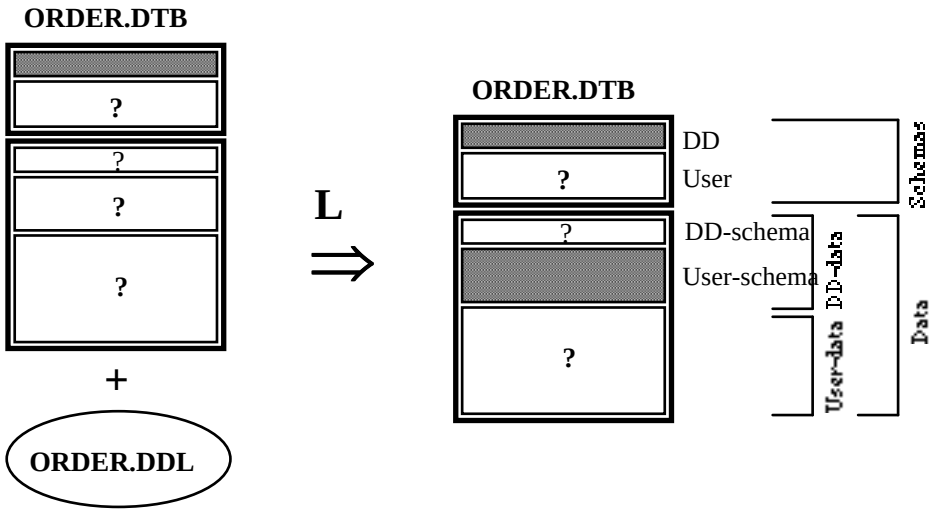


**Figure 6.8 - Figure 8 -** Loading the contents of a DDL text in the data dictionary of a data base. The schema of this data base is left unchanged.

- Initializing the data base according to the user schema stored in the data dictionary and generating the PASCAL types/constants text file (option **G**, for generate). From now on, the data must comply with this new schema (and with that of the DD as well). The old data are erased (Figure 9). Should the old data base contains some user data, a backup version is provided. Note that the schema name that is mentioned in the User schema must be the same as that of the data base. If it is not the case, the data base is renamed accordingly. For instance, a schema that describes an INVOICE data base and that is stored in the ORDER.DTB data base will lead to the definition of an INVOICE.DTB data base.

  Calling : **init_db ORDER.DTB G**

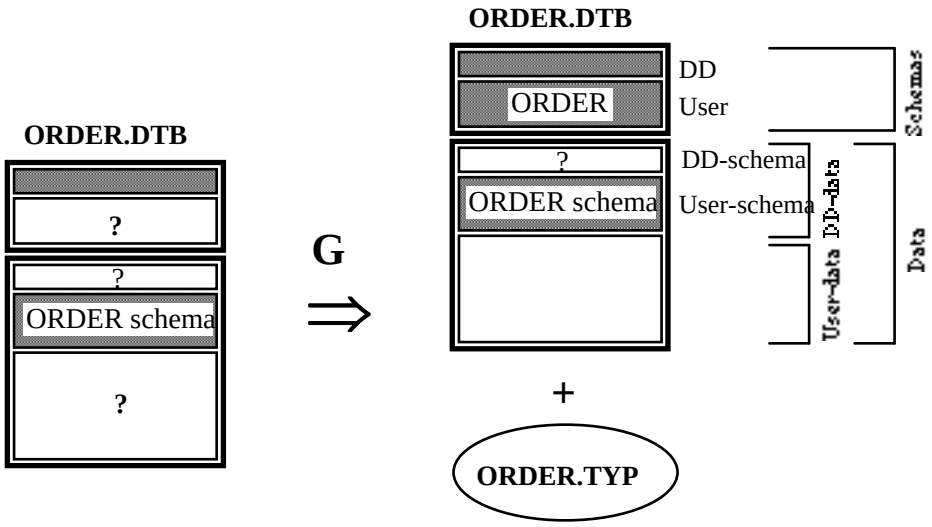


**Figure 6.9 - Figure 9 -** The data base is initialized according to the schema stored in the data dictionary. The user data that it would have contained are erased.

- Erasing the data stored in the data dictionary (option **E**, for erase). This is a way to recover the data base space used by these meta-data when they are of no use (Figure 10). The other parts of the data base are left unchanged.
  Calling : **init_db ORDER.DTB E**

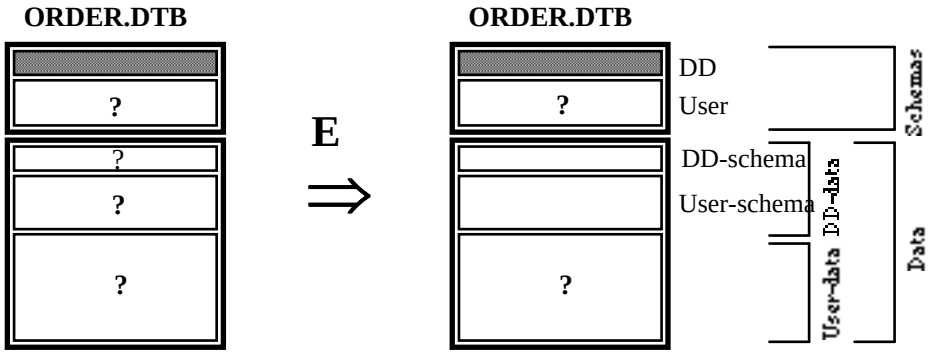


**Figure 6.10 - Figure 10 -** Erasing the data dictionary contents. This reduces the space overhead of the data base but prohibits the use of all the processors that use these data.

- Unloading the contents of the data dictionary (option **U**, for unload). The result is a DDL text describing the user schema (Figure 11). The data base is left unchanged.
  Calling : **init_db ORDER.DTB U**

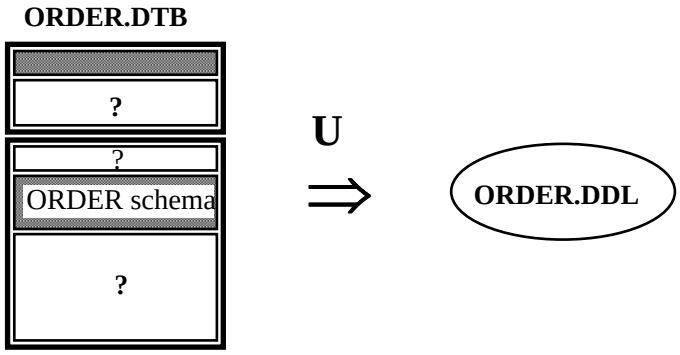


**Figure 6.11 - Figure 11 -** Generating (unloading) a DDL text that represents the contents of the data dictionary.

- Building a new data base from a DDL text (Figure 12). This result can be obtained by combining several primitive functions :
- Building a data dictionary (option D);
- Loading the DDL text contents in the dictionary (option L);
- Initializing and generating the data base and the PASCAL text (option G).
  Calling : **init_db ORDER.DDL DLG**

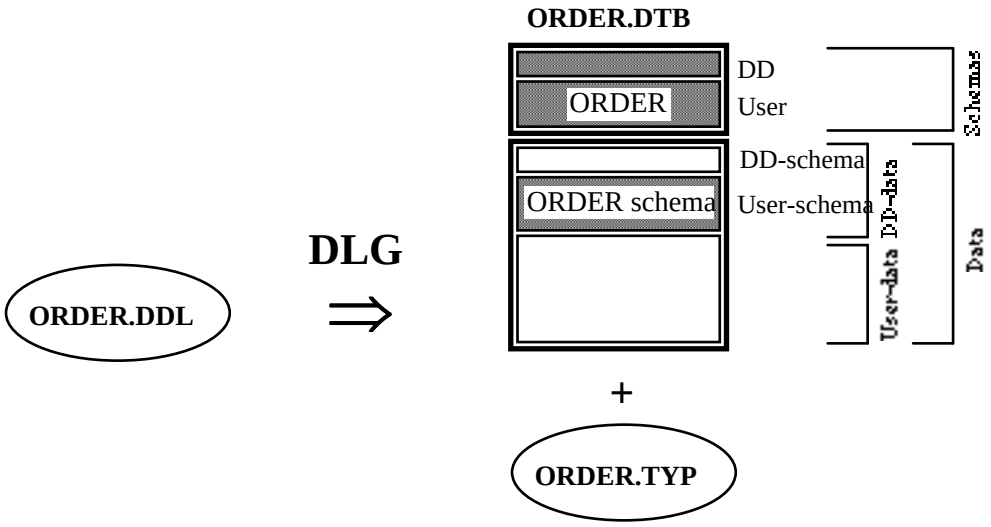


**Figure 6.12 - Figure 12 -** Combining primitive functions to define a new data base from a DDL text.

From the unsophisticated user point of view, the whole process appears as in Figure 13.

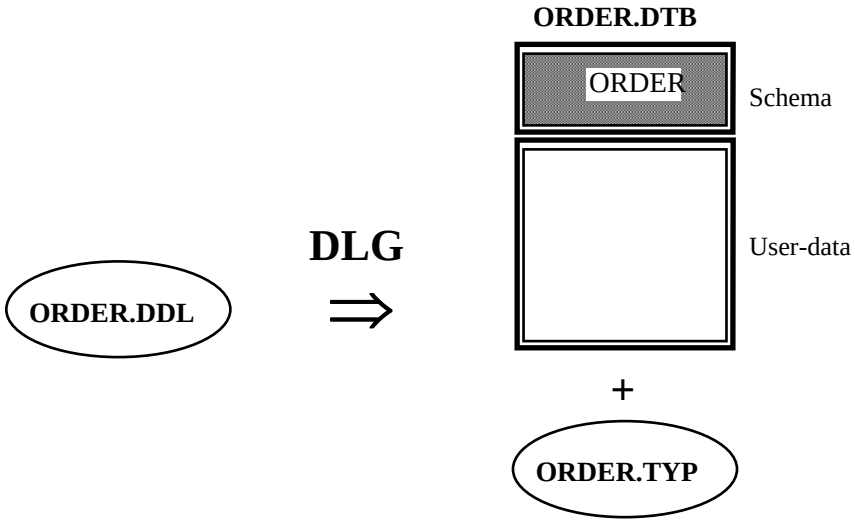


**Figure 6.13 - Figure 13 -** The unsophisticated user's view of the process described in Figure 12..

## Remark on the arguments of INIT_DB

All the **db-init** functions need an argument that specifies the DDL schema or the data base to be processed. When examining the argument, the processor ignores the extension of the name. The kind of file that it will process depends on the function. According to this rule, the following commands are equivalent :

**db-init ORDER.DTB L**

**db-init ORDER.DDL L**

**db-init ORDER L**

## Combined operations

The last example given above shows that INIT_DB can execute several primitive functions in a single run. The rule of combining function codes are simple :

- the functions are executed in the order in which their code are given,
- when needed, a primitive function will get the first argument of the command line, whatever its extension,
- any failure stops the process. For instance, the following sequences will never terminate correctly :

DLEU
DLEG
DG

while the following can be executed completely :

GU
DLGUE
DLGE
MG
CDLG

# Part 7

# 7. AN INPUT PROCESSOR GENERATOR FOR NDBS

## 7.1 FUNCTIONS OF THE GENERATOR

The input processor generator produces an interactive data entry processor for an NDBS data base. A data entry processor allows the user create entities in the data base without resorting to PASCAL programming. The input process can be organised in group of entities called *complex objects*, defined by the users by a specification written in a *complex object definition language* (NDBS-CODL).

### 7.1.1 Complex objects

A **complex object type** is defined as a root entity type together with zero, one or several subordinate entity types. Entity type C can be a subordinate entity type of P if a relationship type R exists from E to S. P is said the *parent* of C via R, and C is said a *child* of P via R. C can be a child of P more than once, but the relationship types R must be different. For instance, ORDER can be a subordinate of CUSTOMER, LINE can be a subordinate of ORDER and PRODUCT. A subordinate entity type itself can have subordinates. For instance, CUSTOMER can have the subordinate ORDER, which in turn can have the subordinate LINE. A complex object type is defined as a tree of entity types that can be created as a whole. If one define a complex object type as follows,

CUSTOMER with his ORDERs (via CO)

any CUSTOMER entity, together with its ORDER entities associated with it via relationships CO makes up a **complex object**.

### 7.1.2 Creating a complex object

Creating a complex object such as ORDER as described above, means creating a CUSTOMER entity and any number of ORDER entities. Any entity type can be the root of no more than one complex object type but can be a subordinate in any number of complex object type. As already mentionned, an entity type can appear more than once in a complex object type.

An input processor can obtain data from users for creating instances from a given set of complex object types. After having obtained data for an entity, the processor automatically asks for data for its possible children. Data for an entity consist of attribute values for it and identifier values for entities to which it must be linked (its parent excepted). Creating a LINE entity as a subordinate of an ORDER entity requires values for attributes *, * and * of PRODUCT, so that the LINE entity can be connected to a PRODUCT entity.

## 7.2 The Complex Object Definition Language (NDBS-CODL)

A CODL text consists in the specification of the data base and of a list of complex object type. The tree structure of the complex object type definition is specified with level numbers *à la* COBOL.

General structure of a CODL text :

```
db data base name
    complex object type 1 specification
    complex object type 2 specification
    complex object type N specification
end
```

The specification of a complex object type is as follows :

**1** root entity type name
    **2** child entity type name **via** relationship type name (if any)
        **3** grand-child entity type name **via** relationship type name (if any)
            and so on ...
        and so on ...
    and so on ...

*Conventions* : one line for each root or child specification. The same is true for the DB specification and for the END statement. Any spaces befor the first word of a line is ignored. Any sequence of spaces is reduced to only one.

## 7.3 EXAMPLES

### 7.3.1 Example 1

```
db ORDER
    1 CUSTOMER
    1 PRODUCT
end
```

That text specifies two complex object types consisting of their root only. The input processor will ask the user to input attribute values for CUSTOMER entities or for

PRODUCT entities. It will not be possible to create ORDER nor LINE entities with that input processor.

### 7.3.2 Example 2

```
db ORDER
    1 CUSTOMER
    1 ORDER
        2 LINE via OL
    1 PRODUCT
end
```

Three complex object types are presented to the user, namely CUSTOMER, ORDER and PRODUCT. CUSTOMER and PRODUCT are as above, while ORDER complex objets are made up of one ORDER entity with its accompanying LINE entities. For the latters, the user will be asked for values of *, * and NUM of PRODUCT.

### 7.3.3 Example 3

```
db ORDER
    1 CUSTOMER
        2 ORDER via CO
            3 LINE via OL
    1 ORDER
        2 LINE via OL
    1 LINE
    1 PRODUCT
end
```

This text defines a complex object type for each entity type of the data base schema. The LINE entity type appears in three complex object types, so that the user is provided with three ways to create a LINE entity. Let's observe that creating a complex object LINE requires the specification of values for *, *, NUM of ORDER and NUM of PRODUCT.

# Part 8

# 8. DGL : A DATA BASE MANAGEMENT LANGUAGE FOR END-USERS

## 8.1 Objective

This function provides the users of an NDBS data base with simple means to store, update and delete entities without resorting to traditional language programming (e.g. NDBS/PASCAL). The entities to be created, updated or deleted are described through a Data manaGement Language[6] (DGL). A DGL text is interpreted by a DGL processor that carries out the operations.

Conversely, a DGL generator that can produce a DGL expression of (a part of) the contents of a data base is proposed as well.

## 8.2 DGL : a language for the management of data bases

### 8.2.1 Principle

A DGL text is a sequence of sections, each one describing an entity to create, an entity to modify or an entity to delete. An entity to create is described by a list of attribute values and/or a list of entities with which the new entity must be associated. An entity to modify is described by its identification, the new entity values and/or the new entities to which it is associated. An entity to delete is described by its identification.

### 8.2.2 Identification of an entity

Specifying an entity to modify, to delete, or to associate with, needs some information to identify it. In some cases, the entity type has an attribute that identifies its entities, such as C_NUMBER for entity type CUSTOMER. However, according to the NDBS data model, an entity type need not have an identifier (*entity identity* principle). These entities have no explicit properties to identify them[7], and therefore cannot be referenced in a DML section.

6. The acronym DML would have been preferable provided it had not been used already (Data Manipulation Language).
7. In fact, there is one stable internal property that identifies all the entities independently of their types, namely their **data base reference** (or DBKey). However, this information is for internal use only, and is not known by the users.

Since any entity may be referenced, whether its type has an identifier or none, entities are given an **external identifier**, independent of their explicit properties (attribute values and associated entities), and that will be used during the updating of the data base. The entities currently stored in the data base have distinct external identifier values, whatever their type. When creating an entity, the user must provide an external identifier value that doesn't exist in the data base yet. Further references to this entity will be made through this unique value. The choice of the external identifier values is up to the user. Any 1-to-9 character string (with ASCII code greater than 32) is a valid value.

### 8.2.3 Data base Management Language specification

The DGL language comprises two statements : the entity statement and the delete statement. Creating and modifying an entity are specified by an **entity statement** while deleting an entity is asked for with a **delete statement**. A DGL text comprises any number (including zero) of entity or delete statements, in any order.

### 8.2.4 The *entity* statement

The entity statement is used to build create and modify sections. The syntax is identical in both cases. Such a section correspond to a creation when the external identifier value has not be associated with an entity yet. It corresponds to a modification when such an entity already exists[8].

## Syntax

| | | |
|---|---|---|
| entity-section | ::= | **entity** entity-type-name<br>id-spec<br>[att-rel-spec]$^{0..*}$<br>end |
| entity-type-name | ::= | *entity type name* |
| id-spec | ::= | **id** id-value |
| id-value | ::= | *string of 1 to 9 characters* |
| att-rel-spec | ::= | att-value-spec \| rel-spec |
| att-value-spec | ::= | att-design att-value |
| att-design | ::= | elem-design \| repeat-design \| compound- |

8. Using the same syntax for both operations allows multiple processing of the same DGL text without problem. When processed the first time, a create entity section define a new entity. When processed another time, it only update the previously created entity. The same can be said of the delete statement : deleting an entity more than once doesn't change the contents of the data base. Therefore, the data base contents is not changed by further processing of the same DGL text. This property is called idempotence.

| | | |
|---|---|---|
| design | | |
| elem-design | ::= | att-name |
| repeat-design | ::= | att-design **[** index **]** |
| compound-design | ::= | att-design **.** att-name |
| att-name | ::= | *attribute name* |
| att-value | ::= | integer\|real\|character\|string\|boolean\|byte\|text |
| integer | ::= | *decimal integral number (limited to 16-bit integers)* |
| real | ::= | number\|number.number\|.number\|number**E**number |
| character | ::= | *one character* |
| string *attribute* | ::= | *character string (from the second position following designation until the end of line)* |
| boolean | ::= | **F** \| **T** |
| byte | ::= | *decimal integral number from 0 to 255* |
| text | ::= | marking-symbol text-value marking-symbol |
| index | ::= | *integral number _ 1* |
| number | ::= | *decimal integral number* \| **-** *decimal integral number* |
| marking-symbol | ::= | *any character used to enclose a text value* |
| text-value | ::= | *any character string (marking symbol exluded)* |
| | | |
| rel-spec | ::= | rel-type-name [id-value] |

## Short description

### Entity section

```
entity-section   ::= entity entity-type-name
                     id-spec
                     [att-rel-spec]0..*
                     end
entity-type-name ::= entity type name
id-spec          ::= id id-value
id-value         ::= string of 1 to 9 characters
att-rel-spec     ::= att-value-spec | rel-spec
att-value-spec   ::= att-design att-value
rel-spec         ::= rel-type-name id-value
```

The description of an entity consists of its type name, its external identifier value, and a possibly empty list of pair *attribute designation* / *attribute value* and of pair *relationship type name* / *entity external identifier value*. The order of the clauses `att-rel-spec` doesn't matter.

### Examples

**entity** CUSTOMER
  **id** CLI123
  NUM 2541
  NAME Mercier A.
  ADDRESS 34, rue des Charmes, Nevers, France
**end**

**entity** ORDER
  **id** O123
  DATE 10/12/90
  CO CLI123
**end**

**entity** LINE
  **id** L1234
  CL O123
  PL WOOD23
  ORDQ 12
**end**

## Attribute designation

```
att-design       ::= elem-design  |  repeat-design  |  com-
pound-design
elem-design      ::= att-name
repeat-design    ::= att-design [ index ]
compound-design  ::= att-design . att-name
att-name         ::= attribute name
index            ::= integral number _ 1
```

This term designates an entity attribute by specifying its name, its index value (array type) and its parent attribute (record type). The notation follows the PASCAL syntax.

### Examples

NUM 2541
ACCOUNT[2] 00-00034561/98
ADDRESS.CITY London
ADDRESS.PHONE[1] 54/239864
ADDRESS.PHONE[3] 54/239867

## Attribute value

```
att-value        ::= integer|real|character|string|boo-
lean|byte|text
integer          ::= decimal integral number (limited to
16-bit integers)
real             ::= number|number.number|.number|numberE-
number
character        ::= one character
string           ::= character string (from the second
position following attribute designation until the end of
line)
boolean          ::= F | T
byte             ::= decimal integral number from 0 to 255
text             ::= marking-symbol  text-value  marking-
symbol
number           ::= decimal integral number | - decimal
integral number
marking-symbol   ::= any character used to enclose a text
value
text-value       ::= any character string (marking symbol
exluded)
```

One and only one space (or separator ':' or '=') is required between the attribute designation expression and the value expression. A string value is terminated by the end of line (this one excluded). A text value is enclosed with a pair of identical characters (marking symbols). This character is chosen freely for each text value and must appear as the first character of the expression. The character chosen must not appear in the text value.

### Examples

NUM 12345
AMOUNT 12.5
WEIGHT 12E8
CODE X
NAME Smith C. J.
LIABLE T
COMMENT $This project can be considered as the first one that takes into account ...
...

Therefore, I propose to give a positive answer to this proposal$

### Relationship specification

```
rel-spec          ::= rel-type-name [id-value]
```

The concerned entity must be declared the target of relationship type `rel-type-name`. The type of the entity referenced by `id-value` must be declared an origin of the relationship type `rel-type-name`. If the `id-value` argument is omitted, the current relationship (if any) is deleted and is not replaced.

### Remarks

The clauses ***entity*** *entity-name*, *id-spec*, *att-rel-spec*, ***end*** must be written on a new line. Each of these clauses must be written on a single line, except for a text attribute value, that can span several lines. In this case, the end of lines are included in the text value.

Any line begining with symbol '*' is ignored (it may be used as a comment), except in a text value expression. Leading spaces are ignored in any line, except in a text value expression.

## Semantics

1. If the database doesn't contain any entity with external identifier **`id-value`**, the entity section asks for the creation of an entity of type **`entity-type-name`**, the attribute values of which are specified in the clauses **`att-value-spec`**, and that is connected with the entities specified in the clauses **`rel-spec`**.
   **Note** : if the entity type has been given an explicit identifier (through an NDBS attribute), the create operation may fail due to the presence in the data base of an entity with the same identifier value. In this case, the external identifier value is assigned to this entity. This is the way to assign an external identifier value to an existing entity.

**Errors** : the entity type name is unknown (the creation fails)
the attribute name is unknown (the attribute value is not assigned)

the attribute value is of the wrong type (the attribute value is not assigned)
the relationship type name is unknown (the relationship is not created)
the external identifier value of the referenced entity to be associated doesn't exist (the relationship is not modified nor created, but the operation is retried after completing interpretation of the DGL text)
the referenced entity to be associated is not of the correct type (the relationship is not created)
the referenced entity to be associated doesn't exist (the relationship is not modified nor created)
unexpected end of text (the processing is stopped).

2. If the database already contains an entity with type **`entity-type-name`** and with external identifier **`id-value`**, the entity section asks for the modification of this entity. The clauses **`att-value-spec`** gives the new attribute values while the clauses **`rel-spec`** specifies the new relationships. If a relationship of the specified type already axists, it is replaced with the new one.

**Errors** : the entity type name is unknown (the modification fails)
the entity has not been found in the data base (the modification fails)
the attribute name is unknown (the attribute value is not assigned)
the attribute value is of the wrong type (the attribute value is not modified)
the relationship type name is unknown (the relationship is not created)
the external identifier value of the referenced entity to be associated doesn't exist (the relationship is not modified nor created, but the operation is retried after completing interpretation of the DGL text)
the referenced entity to be associated is not of the correct type (the relationship is not created)
the referenced entity to be associated doesn't exist (the relationship is not modified nor created)
identifier violation (the attribute value is not modified)
unexpected end of text (the processing is stopped).

In short, when analysing an entity description, a DGL processor works as follows,

- the entity type has an explicit identifier attribute
  - *the id-value is already assigned to an entity* :
    the load processor modifies the attribute values and relationships of the referenced entity;
  - *the id-value has not been assignedyet* :
    - *an explicit identifier value is in the LUN text* :
      - *the data base contains no entity with that identifier value* :
        a new entity is stored in the data base and it is

assigned this id-value for future use;

- *the data base already contains an entity with that identifier value* (identifier violation) :

the creation fails, but the id-value is assigned to this entity for future use;

- *no explicit identifier value is in the LUN text* :

the creation fails;

- *the entity type has no explicit identifier attribute* :
  - *the id-value has already been assigned to an entity* :

the load processor modifies the attribute values and relationships of the referenced entity;

  - *the id-value has not been assigned to an entity yet* :

a new entity is stored in the data base and is assigned this id-value for future use;

### 8.2.5 The *delete* statement

A delete section asks for the deletion of an entity.

## Syntax

```
delete section   ::= delete entity-type-name
                         id-spec
                     end
entity-type-name ::= entity type name
id-spec          ::= id id-value
id-value         ::= string of 1 to 9 characters
```

## Short description

The delete section specifies the type and the external identifier value of an entity to delete.

**Example :**

**delete** CUSTOMER
  **id** CLI123
**end**

## Semantics

If the data base contains an entity with type **`entity-type-name`** and with external indentifier value **`id-value`**, this entity is deleted.

**Errors** : the entity type name is unknown (the deletion fails)
the external indentifier value is unknown (the deletion fails)
the entity has not been found in the data base (the deletion fails)
unexpected end of text (the processing is stopped).

## 8.3 The DGL programs of an NDBS data base

### 8.3.1 Organization

Any NDBS data base can be complemented with two DGL processing programs, namely a DGL processor and a DGL generator. The DGL processor interprets a DGL text in order to carry out the create, update and delete operations it specifies. The DGL generator produces the descriptive DGL text of (a part of) the contents of a data base.

### 8.3.2 The DGL processor

This program accepts DGL texts that are aimed at managing a specific NDBS data base. It proposes two functions, namely syntax checking and data updating.

A. **Syntax checking** : the syntax of the input DGL text is checked. However, no access is made to the data base. Therefore, only type level and syntax level checking are made : entity type names, relationship type names, attribute names, value types and DGL syntax. In particular, whether the operation will be a creation or a modification is unknown. In addition, the validity of the **`id-values`** is not verified. The processor produces a report on the errors found in the DGL text.
B. **Database updating** : the syntax of the input DGL text is checked as in A, and the updating of the data base contents is carried out. The processor produces a report on the errors found in the DGL text.

### 8.3.3 The translation data base

Updating the database is a process that needs special informations about the external identifier values of the concerned entities. In particular, since the entities can be referenced through these values only, a mapping mechanism between external identifier and internal entity references (DBKeys) must be provided. This mechanism is available as a **translation data base** that gives the DBKey of the entity the external identifier of which is known. This data base is used by the DGL processor only, and

is by no means accessible by other programs. The translation data base is updated by create and delete operations.

The translation data base contains the external identifiers that have been given in a DGL text. Several DGL texts can use the same translation data base, provided they use the same external identifiers to designate the same entities (let's remind that these identifier values can be chosen freely by users).

An important problem will occur inevitably when entities already exist in the data base. The problem is that it is not possible to designate an entity that has not yet been processed by the DGL processor since it has been assigned no external identifiers. Two solutions are proposed.

1. The DGL generator program (see 3.5) produces a DGL text describing the contents of the data base. It associates with each entity an external identifier that is derived from the DBKey of this entity (in fact it is the easiest way for it to generate systematically such identifiers). In adition, the DGL generator can be asked to build a translation data base for these external identifier values. Using these external identifier is somewhat ackward, since they appear as arbitrary character strings that are not known at first. The correspondance between these identifiers and the identified entities can be found in the output of the DGL text produced by the generator.

   **Example** :

   **entity** ORDER
   **id** 4122/3
   CC 4120/13
   ORDQ 30
   **end**

2. The entity statement has a particular behaviour when a create operation fails due to identifier violation. In such a case, the operation is cancelled, but is accompanied with a side effect, that is an update of the translation data base : the mentioned external identifier value is stored in the latter. Therefore, we are provided with means to declare an external identifier for an existing entity, the NDBS identifier of which is known. Note that this procedure makes it possible to assign several external identifier values to the same entity. Managing the consistency of these multiple identifiers when deleting entities occurs is up to the user.

   **Example** :

   **entity** CUSTOMER
   **id** CLI123
   NUM 30256

**end**

* CLI123 is not assigned yet, but there exists an entity with NUM value equal to 30256;
* therefore, the create operation fails; however, external identifier CLI123 is assigned to this
* entity, which may now be referenced in the following

**entity** ORDER
**id** O234
CC CLI123
DATE 10/3/91
**end**

### 8.3.4 The DGL generator

This program creates a DGL text that describes attribute values and relationships of the entities of selected types in an NDBS data bases. The concerned attributes and relationships can be selected. The external identifier values are generated as the ASCII representation of entity data base keys (entity physical addresses), as explained in section 3.3. If required, the generator builds a translation data base for the selected entities.

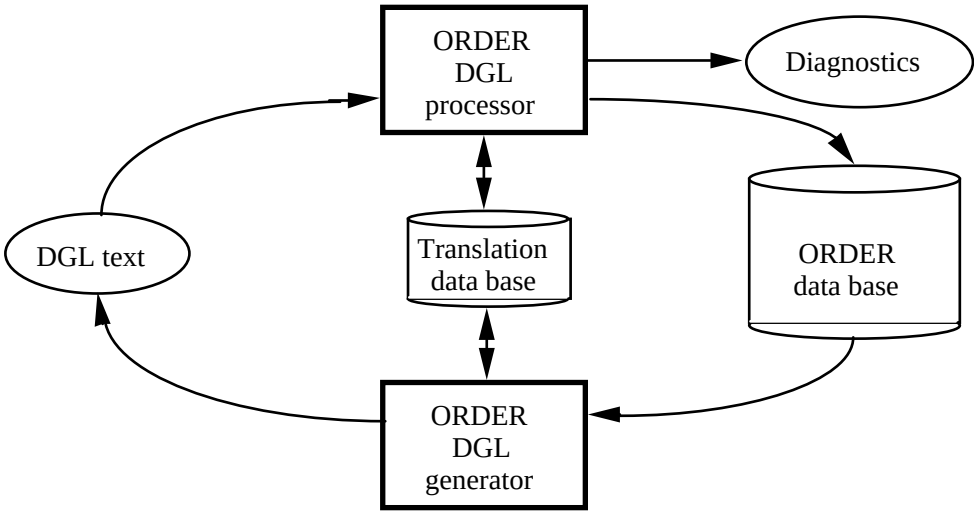


**Figure 8.1 - Figure 1** - Information flows of DGL processes on the ORDER data base. The DGL programs process the data of the data base.

### 8.3.5 DGL programs builder

This program produces the DGL processor and DGL generators for an NDBS data base. This builder creates these DGL programs thanks to the contents of the data dictionary of the target data base.

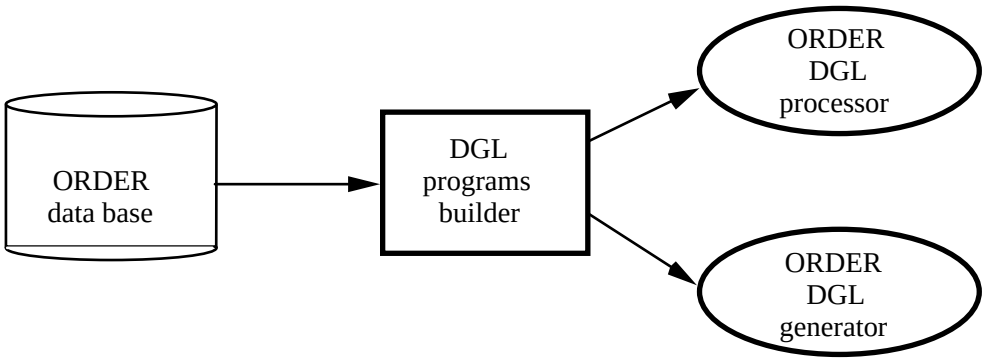


**Figure 8.2 - Figure 2 -** Building the DGL programs for the ORDER data base. The DGL builder processes the data dictionary of the data base.

#### Generation of a DGL generator

The user is asked for :

- the *name of the data base*
- the *name of the DGL generator* that will be generated
- the *names of the selected entity types*
  - for each entity type selected,
    - *the attributes to be described*
    - for each of them, *a possible new name*;
  - *the relationship type to be described*
    - for each of them, *a possible new name*.

#### Generation of a DGL processor

The user is asked for :

- the *name of the data base*

- the *name of the DGL processor* that will be generated

# Part 9

# 9. LOAD / UNLOAD UTILITIES FOR NDBS DATA BASES

## 9.1 Objective

This function allows users of a source NDBS data base to unload data under an ASCII format and to reload these data later on, in a target data base. This unload/ load function can be used to build save and compact backup versions of selected parts of a data base or to export and import data with external data bases. Since the schemas of these data bases can be different, this function also provides users with a mean to restructure the schema of a data base by adding or removing attributes and relationship types, or by changing the format of the attributes.

The extracted data are organized according to a specific Load/UNload (LUN) format.

## 9.2 The LUN format

### 9.2.1 The sections of a LUN text

A Load/Unload formated text (LUN text) is made up of two sections, namely the *structure section* and the *data section*. The structure section declares the type,the attributes and the relationship types of the entities that are described in the data section. The data section comprises a list of entity descriptions in terms of attribute values and associated entities.

### 9.2.2 The *structure section* of a LUN text

The structure section comprises at least one entity type description. Each of these descriptions specifies the name of the entity type (prefixed with "*") followed by clauses describing some of its attributes and relationship types which it is a target of. This name and each attribute and relationship clause are terminated with an end-of-line. The structure section is terminated with a line containing only '*'.

### 9.2.3 The *attribute* clauses

An attribute clause mentions the name and the value type of an attribute. The first attribute clause is mandatory and describes a virtual attribute which is the data base key (DBKey) of the described entity. Its name is ID-E (an invalid PASCAL name) and no explicit value type is declared.

The other attribute clauses are optional. Each of them describes a terminal attribute, i.e. an attribute that has one elementary and simple value (or which is neither compound nor multivalued). If an attribute is compound, only its components can be specified, if an attribute is multivalued, only its simple values can be described.

Consider for instance the following entity type description, expressed in DDL :

```
entity-type CUSTOMER
begin
  NUM : integer
  NAME : string[25]
  CHR_NAME : array[1..2] of string[20]
  ADDRESS : record
      NUM : integer
      STREET : string[25]
      CITY string[35]
      PHONE : array[1..3] of string[12]
  end
end
```

An attribute clause describes any of the following elementary and simple values :

```
NUM
NAME
CHR_NAME[1]
CHR_NAME[2]
ADDRESS.NUM
ADDRESS.STREET
ADDRESS.CITY
ADDRESS.PHONE[1]
ADDRESS.PHONE[2]
ADDRESS.PHONE[3]
```

An attribute clause specifies the attribute designation and the value type according to the PASCAL syntax. The attributes of entity type CUSTOMER defined above can be described be any subset of the following attribute clauses :

```
NUM integer
NAME string[25]
CHR_NAME[1] string[20]
CHR_NAME[2] string[20]
ADDRESS.NUM integer
ADDRESS.STREET string[25]
ADDRESS.CITY string[35]
ADDRESS.PHONE[1] string[12]
ADDRESS.PHONE[2] string[12]
ADDRESS.PHONE[3] string[12]
```

### 9.2.4 The *relationship* clauses

A relationship clause specifies the name of a relationship type in which the described entity type is the target. Its type is declared by the word **rel**. The clause **CO ref** used in the description of the ORDER entity type specifies the CO relationship type. The relationship clauses are optional.

### 9.2.5 Example of the description of an entity type

The following text describes the ORDER entity type.

```
*ORDER
ID
NUM integer
DATE string[6]
CO ref
```

### 9.2.6 The *data section* of a LUN text

This section directly follows the structure section. It is made up of as many **entity paragraphs** as there are entities to describe. Each entity paragraph begins with the sequence number of the entity type, i.e. the rank of the description of the entity type in the structure section. Then, the paragraph comprises as many value lines as there are attribute and relationship clauses in the structure section for the entities of this type, in the corresponding order. Each line is made up of an attribute value for attribute clauses and of an entity DBKey value[9] for relationship clauses. This latter value designates the origin entity of a relationship in which the described entity is the target. A value is in ASCII format and terminates with an end-of-line (this one excluded)[10]. A text value is represented in two lines : the first line specifies the exact number of characters of the value, the following line contains the text value and is terminated with an end-of-line (that is not part of the text value).

A missing value is represented by an empty line.

The last entity paragraph is followed by a line that contains only the character "*".

9. The fact that entities are designated by their data base keys is completely irrelevant. In fact, this reference could have been any other means by which the origin entity can be identified. In particular, this value can be changed by hand to any arbitrary value, provided this value identifies the concerned entity everywhere it is referenced. Moreover, in the target database, the entity would generally have another data base key.
10. Consequently, the end-of-line character cannot be a part of a value, except for text attributes.

### 9.2.7 Example of a LUN text

```
*CUSTOMER                       Description of entity type CUSTOMER (#1)
ID-E                                  DBKey
NUM integer                           attribute NUM
NAME string[25]                       attribute NAME
ADDRESS.NUM integer                   component NUM of ADDRESS
ADDRESS.CITY string[30]               component CITY of ADDRESS
ADDRESS.PHONE[1] string[12]           component PHONE[1] of ADDRESS
ADDRESS.PHONE[2] string[12]           component PHONE[2] of ADDRESS
COMMENT text                              attribute COMMENT
*ORDER                          Description of entity type ORDER (#2)
ID-E                                  DBKey
NUM integer                           attribute NUM
DATE string[6]                        attribute DATE
CO rel                                relationship type CO
*LINE                           Description of entity type LINE (#3)
ID-E                                  DBKey
ORDQ real                             attribute ORDQ
CL rel                                relationship type CL
PL rel                                relationship type PL
*                                     end of structure section
*1                              Description of a CUSTOMER entity
175/1                                 DBKey value
123                                   NUM value
Dupont, A.                            NAME value
24                                    ADDRESS.NUM value
Paris                                 ADDRESS.CITY value
1/6578849                             ADDRESS.PHONE[1] value
1/6578850                             ADDRESS.PHONE[1] value
1250                                  the COMMENT text value
                                          comprises 1250 characters
This a text that constitutes an example
of a long string of characters ...
... these are the last of the 1250
characters.
*1                              Description of a CUSTOMER entity
175/2
231
Mercier
Bruxelles
01/2341876
                                      ADDRESS.PHONE[2] value missing
                                        COMMENT value missing
*1                              Description of a CUSTOMER entity
177/4
539
BERTILLET, Michel
```

```
13
Lyon
                                        ADDRESS.PHONE[1] value missing
                                        ADDRESS.PHONE[2] value missing
                                            COMMENT value missing
*2                                  Description of an ORDER entity
335/11                                      DBKEY value
3452                                        NUM value
8/6/90                                      DATE value
177/4                                       CUSTOMER entity DBKey
*3                                  Description of a LINE entity
335/12                                      DBKey
12.5                                        ORDQ value
335/11                                      ORDER entity DBKey
6345/2                                      PRODUCT entity DBKey
*3                                  Description of a LINE entity
335/13
500
335/11
                                        PRODUCT entity unknown
*                                   end of data
```

## 9.3 The UNLOAD processors

Any number of UNLOAD processors can be associated with an NDBS data base. Each processor generates a LUN text in which a subset of attributes and relationship types are described for entities of selected types.

Moreover, the names of the entity types, attributes and relationship types can be different from those in the origin schema. Changing the names can be useful for migrating data toward another data base with a different schema.

For instance, the entity type defined as follows,

```
entity-type CUSTOMER
begin
  NUM : integer
  NAME : string[25]
  CHR_NAME : array[1..2] of string[20]
  ADDRESS : record
      NUM : integer
      STREET : string[25]
      CITY string[35]
      PHONE : array[1..3] of string[12]
  end
end
```

would be given the following standard LUN description :

```
*CUSTOMER
NUM integer
NAME string[25]
CHR_NAME[1] string[20]
CHR_NAME[2] string[20]
ADDRESS.NUM integer
ADDRESS.STREET string[25]
ADDRESS.CITY string[35]
ADDRESS.PHONE[1] string[12]
ADDRESS.PHONE[2] string[12]
ADDRESS.PHONE[3] string[12]
```

However, the following description could have been proposed instead :

```
*CLIENT
ID.NUM integer
ID.NAME string[25]
ID.FIRST_CHR_NAME string[20]
ID.SECOND_CHR_NAME string[20]
ADD_NUM integer
ADD_STREET string[25]
ADD_CITY string[35]
ADD_PHONE[1] string[12]
ADD_PHONE[2] string[12]
ADD_PHONE[3] string[12]
```

This description would allow migrating the CUSTOMER data into a data base comprising the following CLIENT entity type :

```
entity-type CLIENT
begin
  ID record
     NUM : integer
     NAME : string[25]
     FIRST-CHR_NAME : string[20]
     SECOND-CHR_NAME : string[20]
  end
  ADD_NUM : integer
  ADD_STREET : string[25]
  ADD_CITY string[35]
  ADD_PHONE : array[1..3] of string[12]
end
```

Note that changing the names concerns the structure sections only, the data sections being the same in both cases.

By default, an unload processor working on the ORDER data base will produce a LUN text with file name *ORDER.LUN*. This default name can be changed by the user.

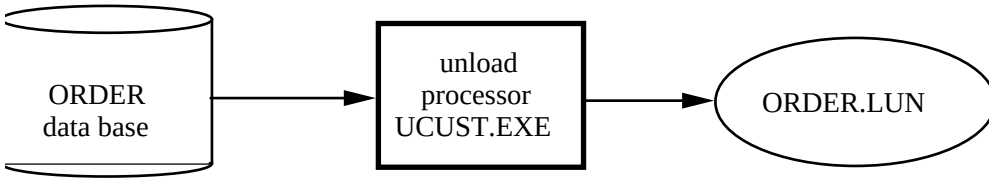


**Figure 9.1 - Figure 1 -** Unloading data from the ORDER data base with the unload processor UCUST.EXE. The LUN output text has been given the default name ORDER.LUN.

## 9.4 The LOAD processors

There is generally only one LOAD processor associated with an NDBS data base. A LOAD processor can store entity data (attribute values and relationships) for the entity types of the schema. It can modify already stored entities as well.

Creating a relationship between entities requires identifying both of them, despite the fact that an entity type need not have an explicit identifier (i.e. an attribute the values of which are distinct among all the entities of that type). The LUN format makes use of an arbitrary external identifier to reference entities : when created, an entity is given an identifying value (value of the virtual attribute ID-E). Incidently, because an unloader cannot invent truly arbitrary values, these values happens to derive from the DBKey of the concerned entity. However, any other means could have been used.

To allow further references to the entities created, a **translation data base** associates the DBKEY of the entity in the target data base (there are no reason for old and new DBKeys being the same). This translation data base is maintained by the loaders and its use is in principle transparent to the user. If incremental loading is carried out, it is adviced to keep the same translation data base. The translation data base is also used by the DGL programs[11], so that the LUN and DGL programs can work synchronously.

11. The Data Base Management Language (or DGL in short) programs are utilities that allows end-users to enter, modify and delete data in an NDBS data base. The DGL principles are explained in the document *NDBS project - NDBS Data base management for end-users*. This document presents the problem of entity identifier and how the translation data base mechanism can solve it.

When analysing an entity paragraph, a load processor works as follows,

- *the entity type has an explicit identifier attribute* :
  - *the ID-E value is already in the translation data base* :
    the load processor modifies the attribute values and relationships of the referenced entity;
  - *the ID-E value is not in the translation data base yet* :
    - *an explicit identifier value is in the LUN text* :
      - *the data base contains no entity with that identifier value* :
        a new entity is stored in the data base and its ID-E value is stored in the translation data base for future use;
      - *the data base already contains an entity with that identifier value* (identifier violation) :
        the creation fails, but the ID-E value is stored in the translation data base for future use;
    - *no explicit identifier value is in the LUN text* :
      the creation fails;
- *the entity type has no explicit identifier attribute* :
  - *the ID-E is already in the translation data base* :
    the load processor modifies the attribute values and relationships of the referenced entity;
  - *the ID-E is not in the translation data base yet* :
    a new entity is stored in the data base and its ID-E value is stored in the translation data base for future use;

The errors that may happen when creating an entity are the following :

- the entity type name is unknown
- the attribute name is unknown
- the relationship type name is unknown
- the DBKey of the referenced entity to be associated is unknown until now (the relationship is not modified nor created, but the operation is retried after completing interpretation of the LUN text)
- the referenced entity to be associated is not of the correct type
- the referenced entity to be associated doesn't exist
- unexpected end of text .

The errors that may happen when modifying an entity are the following :

- the entity type name is unknown
- the entity has not been found in the data base
- the attribute name is unknown
- the relationship type name is unknown
- the DBKey of the referenced entity to be associated is unknown until now (the relationship is not modified nor created, but the operation is retried after completing interpretation of the LUN text)
- the referenced entity to be associated is not of the correct type
- the referenced entity to be associated doesn't exist
- identifier violation
- unexpected end of text .

The structure section of the LUN text to load must mention entity type, attribute and relationship type names that are in the schema. Data corresponding to any unknown construct is discarded. However, these unknown structures are reported in a diagnostic file.

The load processor accepts type conversion in attribute values :

- changing string lengths (truncature may occur);
- converting a numeric (integer and real) value into a string value (truncature may occur);
- converting an integer value into a real value;
- converting a real value into an integer value (loss of decimal figures may occur)
- converting a character value into a string value;
- converting a boolean value into a character or string value;
- converting a byte value into an integer, real, character or string value;
- converting a string value into a numeric value (unconvertible characters may be discarded).

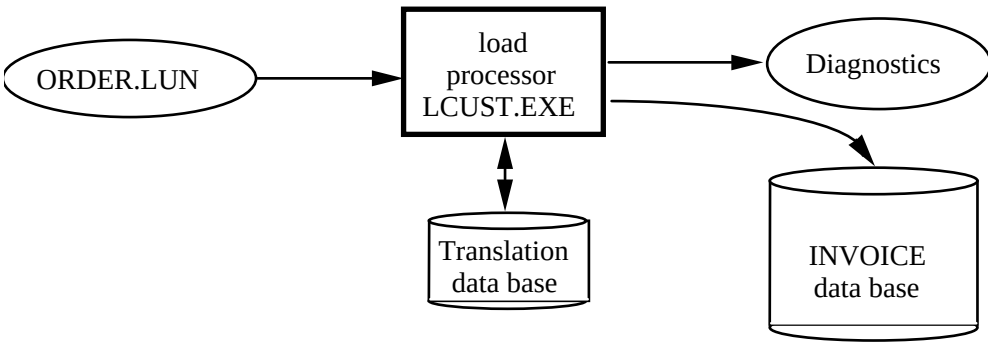


**Figure 9.2 - Figure 2 -** The load processor LCUST stores the data described in ORDER.LUN into the INVOICE data base. This processor makes use of and maintains the translation data base in order to convert the previous entity DBKeys into the new ones.

## 9.5 The LUN processors builder

This program produces LOAD and UNLOAD processors for an NDBS data base thanks to the analysis of the data dictionary of this data base. Note that producing processors for migrating data from data base A to data base B requires building an *unload processor for A* and a *load processor for B.*

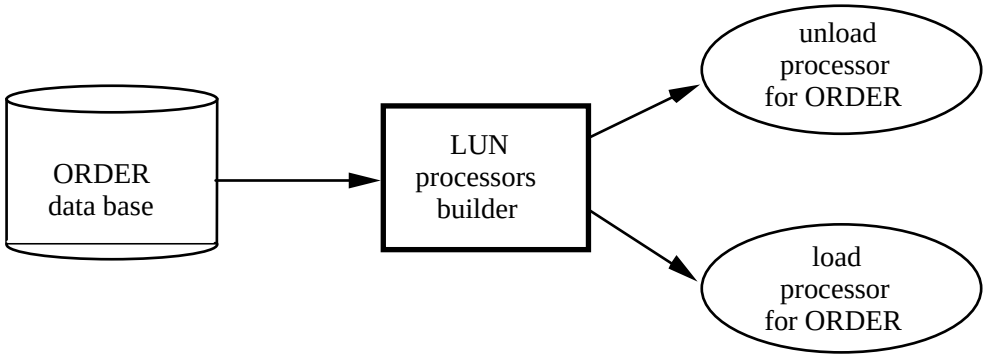


**Figure 9.3 - Figure 3 -** The LUN.EXE processors builder analyses the data dictionay of the ORDER data base in order to generate LOAD and UNLOAD processors for that data base.

### Generation of an unload processor

The user is asked for :

- the *name of the origin data base*
- the *name of the LOAD processor* that will be generated
- the *names of the selected entity types*
  - for each entity type selected,
    - *the attributes to be unloaded*
    - for each of them, *a possible new name*;
  - *the relationship type to be unloaded*
    - for each of them, *a possible new name*.

## Generation of a load processor

The user is asked for :

- the *name of the target data base*
- the *name of the UNLOAD processor* that will be generated

# Part 10

## 10. REPORT GENERATOR FOR NDBS - VERSION 2

### 10.1 FUNCTIONS OF THE REPORT GENERATOR

The description of a report is a text written the NDBS-RDL language. The description is processed by an ad'hoc compiler that checks it against the schema of the data base contained in the NDBS data dictionary. The description can be built by means of any text editor. If the report description is complete and correct, the compiler produces a PASCAL program the execution of which will print the report. The generated program asks the user if the report has to be printed on the screen, on the printer or in a file. It also asks the user for the unknown values mentioned in the report description.

### 10.2 GRAMMAR OF THE REPORT DEFINITION LANGUAGE (NDBS-RDL)

**Meta syntax**

| | |
|---|---|
| symbol | natural language description |
| *symbol* | subexpression still to be defined (meta term) |
| **symbol** | constant symbol of the language |
| \| | separates alternatives |
| [ *symbol* ] | subexpression *symbol* is optional |
| *symbols* * | expression *symbol* must appear at least once and possibly several times |
| [ *symbol* *] | any number (including 0) of *symbol* |

**Grammar**

| | |
|---|---|
| *report* | ::= **db** *dbname statement** **end** |
| *statement* | ::= *et* \|*w* \| *l* \| *pg* \| *let* |
| *et* | ::= **for** *var-name* **:=** *entity-seq* [ *if* ] *statement** **end** |
| *entity-seq* | ::= *etname* \| *etname* **(:***idname* **=** *value***)** \| *etname***(***pathname* **:** *var-name***)** |
| *if* | ::= **if** *attname compop value* |
| *w* | ::= **write** *value-list* \| **writeline** *value-list* |

*l* ::= **line** [*integer*]
*page* ::= **page**
*let* ::= **let** *var-name* := **$(**character string**)** | *value*
*pathname* ::= *rtname* | ***** *rtname*
*compop* ::= < | > | <= | >= | = | <>
*value-list* ::= *value* *
*value* ::= *num-const* | *str-const* | *var-name***.***attname* | *var-name*
*num-const* ::= numeric constant
*str-const* ::= ' character string '
*integer* ::= integer number
*dbname* ::= data base name
*etname* ::= entity type name
*rtname* ::= relationship type name
*idname* ::= name of the identifier of an entity type
*attname* ::= attribute name

## Short description

### Report description

*report* ::= **db** *dbname statement** **end**

The first statement of the report specifies the data base to report from and the operations (statements) that have to be carried out to produce the report.

### Entity bloc

*et* ::= **for** *var-name* **:=** *entity-seq* [ *if* ] *statement** **end**

Describes a subset of entities together with operations to be carried out on each of them (called the current entity for the statements). If no entities are selected, the execution of the entity bloc has no effect. When an entity is under consideration (it is evaluated for selection or it is processed within the *statement** list), it is called the *current entity* and it can be referred to by the name *var-name*. The current entity is *knoxn* in *entity-seq*, in *if* and in *statement**. It is *unknown* outside its entity bloc.

### Entity subset

*entity-seq* ::= *etname* format 1
*etname* **(:***idname* **=** *value***)** format 2

*etname* **(***pathname* **:** *var-name***)** format 3

*value* ::= *num-const* | *str-cont* | *var-name***.***attname* | *var-name*

*pathname* ::= *rtname* | ***** *rtname*

Describes an ordered subset of entities of type *etname*. In the first format, all the entities are selected. In the second format, the entity with the identifier value specified by *value* is selected (if any). The *value* can be a numeric or string (between ' and ') constant, an attribute value from a known entity, or the value of an internal variable. In the third format, the selected entities are linked via the path *pathname* to the current entity of the block in which the statement is located. The path is designated by the relationship type on which it is defined. In case of a non-recursive relationship type, the ***** prefix is ignored. If the relationship type is recursive, the ***** prefix specifies the N-1 path, while its absence designates the 1-N path.

## Entity filter

*if* ::= **if** *attname compop value*

*value* ::= *num-const* | *str-cont* | *var-name***.***attname* | *variable*

*compop* ::= < | > | <= | >= | = | <>

This optional statement directly follows an *et* statement. It limits the selected entities to those which satisfy the condition *attname compop value*. The condition states that a selected entity must have a value for attribute *attname* which satisfies the relation *compop value*, where *compop* is <, >, <=, >=, = or <>. The *value* can be a numeric or string (between ' and ') constant, an attribute value from a known entity, or the value of an internal variable.

## Write statement

*w* ::= **write** *value-list* | **writeline** *value-list*

*value-list*::= *value* *

*value* ::= *num-const* | *str-cont* | *var-name***.***attname* | *variable*

This statement asks for the writing of a list of values. The values can be numeric constants, string constants (between ' and '), attribute value of a known entity or value of an internal variable. The **write** statement issues no carriage return. The **writeline** statement skip to next line after writing the value list.

**Skip lines**

*l* ::= **line** [*integer*]

Skip to next line. If an integer argument is specified, the operation is executed the specified number of times.

**Skip page**

*page* ::= **page**

Go to the top of next page

**Assignment**

*let* ::= **let** *var-name* := **$ (**character string**)** | *value*

assign to the internal variable *var-name* the value *value* or get a value from the user. In the latter case, the user is prompted with the message represented by the *character string*.

## 10.3 EXAMPLES BASED ON THE *ORDER* DATA BASE

1. List of the PRODUCTS

```
db ORDER
    for P := PRODUCT
        writeline P.NAME  P.UPRICE  P.AVAILQ
    end
end
```

2. Information about CUSTOMER n° 1234

```
db ORDER
    for C := CUSTOMER(: NUM = 1234)
        writeline 'Name : '  C.NAME  ' - Address : '
C.ADDRESS
    end
end
```

3. Information about CUSTOMER n° X (to be provided by the user at execution time)

```
db ORDER
   let   N := $('Number of the customer : ')
   for C := CUSTOMER(: NUM = N)
      writeline 'Name : '  C.NAME  ' - Address : '
C.ADDRESS
   end
end
```

4. List of the ORDERS received before date X (to be provided by the user at execution time)

```
db ORDER
   let   D := $('Date maximum : ')
   for O := ORDER
      if O.DATE < D
      write
      for C := CUSTOMER(CO: O)
         writeline 'Order'  O.NUM  ' - Date : '
O.DATE 'Customer '  C.NAME
      end
      line 3
   end
end
```

5. List of the ORDERS from customer X received before date Y (X and Y are to be provided by the user at execution time); print details about the ordered quantities of products; skip page after each order.

```
db ORDER
   let N := $('Name of the customer : ')
   let D := $('Date maximum : ')
   for C := CUSTOMER(: NUM = N)
      line  4
      writeline   'Name : ' C.NAME ' - Address : '
C.ADDRESS
      for O := ORDER(CO: C)
         if  O.DATE < D
         writeline   '       Order num : ' O.NUM ' -
Date : ' O.DATE
         for L := LINE(OL: O)
```

```
                    for P := PRODUCT(PL: L)
                        writeline   '            ' L.ORDQ '
of  ' P.UPRICE
                    end
                end
                page
            end
        end
    end
```

5. Build a table of the LINES with their PRODUCT and the ORDER and CUSTOMER they belong to.

```
    db  ORDER
        page
        writeline '   QTY            PRODUCT
ORDER                CUSTOMER'
        writeline ' ---------      -----------------
-------------            --------------------'
        for L := LINE
            write  '|' L.ORDQ '|'
            for P := PRODUCT(PO: L)
                write  P.NAME '|'
                for O := ORDER(OL: P)
                    write O.NUM '|'
                    for C := CUSTOMER(CO: O)
                        write   C.NUM '|'
                    end
                end
            end
            line  2
        end
        page
end
```

# Part 11

# 11. SCREEN GENERATOR FOR NDBS

## 11.1 FUNCTIONS OF THE GENERATOR

The screen generator produces a collection of PASCAL procedures that allows for the quick writing of interactive programs dealing with data of NDBS data bases. These procedures offers three functions for each entity type of an NDBS schema : initialisation of the attribute values, displaying the attribute values and obtaining the attribute value from the user. The procedures are automatically generated from the contents of the NDBS data dictionary (DIC). However, the layout of the display and input screens can be designed by the user.

## 11.2 DESCRIPTION OF THE PROCEDURES

The programmer is provided with two main procedure sets, namely INIT_*etname*, and SCR_*etname*. The first set initialises the attribute values of entity variable of type T*etname*. The second set allows for the display and input of attribute values of entity variables of type T*etname*.

**INIT_*etname*(var V: T*etname*)**

Input

none

Output

V : entity variable;

Function

initialises the attributes of V with the following default values void string for strings, space for character, 0 for numeric;

Example

```
INIT_CUSTOMER(C);
```

Example definition of INIT_CUSTOMER

```
procedure INIT_CUSTOMER(var C : TCUSTOMER);
begin
    C.NUM := 0;
    C.NAME := '';
    C.ADDRESS.CITY := '';
    C.ADDRESS.PHONE[1] := '';
    C.ADDRESS.PHONE[2] := '';
    C.ADDRESS.PHONE[3] := '';
end;
```

### SCR_*etname*(var V: T*etname*;X,Y: integer; OP: integer; var S: integer)

input

V : entity variable containing initial attribute values to be displayed or modified;

X,Y : screen coordinates of the upper left corner of the screen window for the dialog;

X is in [1..80-WW], where WW is the window width;

Y is in [1..25-WH], where WH is the window height;

OP : function required to be performed;

OP = 0 : display only

OP = 1 : display and get new values

OP = 2 : clear screen window

output

V : entity variable containing the (possibly) modified attribute values, according to S value;

S : state of C;

S = 0 : C has not been modified

S = 1 : C has be modified

functions

if OP = 0, the attribute values of V are displayed in a window screen at position X,Y; control is then returned to the calling procedure;

if OP = 1, the attribute values of V are displayed in a window screen at position X,Y;for each value, the user is invited to modify it; he can replace the current value by typing another value or leave it unchanged by depressing the ENTER key; when all the attribute values have been displayed and (possibly) modified, the user is proposed with three conclusions :

either leave the screen and cancel the modifications he could have

done (answer 'C' for Cancel); the calling procedure is provided with V unchanged and S = 1;

or leave the screen and send the new values to the calling procedure (answer 'S' for Send), The calling procedure is provided with V modified and S = 1;

or ask for retrying the screen display and modification (answer 'R' for Retry);

if OP = 2, the screen window defined by the procedure and located at position X,Y is cleared.

Examples

```
       SCR_CUSTOMER(CUS,1,1,2,S);  { clear the screen }
       SCR_ORDER(O,1,10,2,S);      { clear the screen }
       CUS.NUM := X;               { get a CUSTOMER entity
}
       dbid(CUSTOMER,CUS);
       SCR_CUSTOMER(CUS,1,1,0,S);  {    display    CUSTOMER
attribute values }
       if dbfound then begin
         S := 1;
         while S > 0 do begin      { get and create ORDER
entities }
           INIT_ORDER(O);          { - clear attribute val-
ues in O }
           SCR_ORDER(O,1,10,1,S);  { - get attribute val-
ues from the user }
           if S = 1 then begin     { - if not finished, }
               dbcreate(ORDER,ORD);{      create an ORDER
entity }
               dbinsert(O,CUS,CO); {  attach it to the CUS-
TOMER entity }
           end;
         end;
       end;
```

Example definition of SCR_CUSTOMER

```
procedure SCR_CUSTOMER( var C : TCUSTOMER;
                        X,Y : integer;
                        OP : integer;
                        var S : integer);
var
   C0 : TCUSTOMER;
   STATE : char; {'C'= return without modification, 'S' : return the
modified values, 'R' : keep displaying the same screen}
```

```
   I : integer;

label 999;

procedure FRAME;     (* draw the passive part of the screen
*)
begin
   SCR_BOX(X+2,Y+1,X+60,Y+12);
   gotoxy(X+8,Y+3);write('Customer number : ');
   gotoxy(X+8,Y+4);write('Customer name   : ');
   gotoxy(X+10,T+5);write('C(ancel),S(end),R(etry): ');
end

begin
   if (OP < 0) or (OP > 2) then begin SCR_OPERROR(CUS-
TOMER,OP); goto 999; end;
   if OP = 2 then begin     (* clear screen window *)
       for I := 1 to 12 do begin
          gotoxy(X+2,Y+I);write(' ':59);
       end;
   end
   else begin
       FRAME;
       dbcopyatt(CUSTOMER,C,C0);
       repeat
          SCR_INTEGER(X+21,Y+3,C0.NUM,OP);
          SCR_STRING(X+21,Y+4,C0.NAME,15,OP);
          if OP = 1 then begin
              STATE := 'R';
              repeat
                 SCR_STRING(X+49,Y+5,STATE,1,1);
                STATE := UpCase(STATE);
              until STATE in {'C','S','R'};
          end
          else
              STATE := 'C';
       until STATE in {'C','S'};
       if STATE = 'S' then dbcopyatt(CUSTOMER,C0,C);
       id STATE = 'C' then S := 0 else STATE := 1;
   end;
999: ;
end;
```

## 3. AUXILIARY FUNCTIONS

The SCR_ functions make use of lower-level procedures. Some of them are described below.

### SCR_OPERROR(ETtype: integer; OP: integer)

Input

ETtype : numeric code of an entity type;
OP : value of parameter OP received by a SCR_*etname* procedure;

Output

screen

Function

displays on the screen an error message stating that the value <OP> has been received by procedure SCR_*etname* dedicated to entity type ETtype.

Example

```
SCR_OPERROR(CUSTOMER,OP);
```

### SCR_BOX(X1,Y1,X2,Y2: integer)

Input

X1,Y1,X2,Y2 : screen coordinates of a rectangle; 1<=X1<X2<=80; 1<=Y1<Y2<=25

Output

screen

Function

Draws on the screen a rectangle the diagonal of which is (X1,Y1)-(X2,Y2);

Example

```
SCR_BOX(X,Y,X+40,Y+12);
```

**SCR_INTEGER(X,Y: integer; var V: integer; OP: integer)**

Input

X,Y : screen coordinates of a character position; 1_X_80; 1_Y_25;

V : integer value to be displayed;

OP : function to be performed,

OP = 0 : displays the value of V;

OP = 1 : displays the value of V and wait for a new value;

Output

V : new integer value obtained from the user;

Function

displays the value of V at screen location (X,Y); if OP = 1, waits for the user giving a new value and returns it in V. The value is left unchanged if the user only depresses the ENTER key when being provided with the initial value;

Examples

```
SCR_INTEGER(20,5,I,0);
SCR_INTEGER(X+5,Y+2,T[I],S);
```

**SCR_STRING(X,Y: integer;var V: string[*]; L: integer; OP: integer)**

Input

X,Y : screen coordinates of a character position; 1_X_80; 1_Y_25;

V : integer value to be displayed;

L : integer value specifying the maximum length of V values;

OP : function to be performed,

OP = 0 : displays the value of V;

OP = 1 : displays the value of V and wait for a new value;

Output

V : new integer value obtained from the user;

Function

displays the value of V at screen location (X,Y); if OP = 1, waits for the user giving a new value and returns it in V. The new value length cannot exceed L. The value is left unchanged if the user only depresses the ENTER key when being provided with the initial value.

Example

```
SCR_STRING(5,12,C.NAME,20,1)
```

# Part 12

## 12. SQL CONVERTER FOR NDBS DATA BASES

### 12.1 Objective

This NDBS subsystem allows for the conversion of NDBS data bases into relational structures and data. Non relational NDBS constructs are converted in order to express all the semantics of the entity-relationship schema. Let us mention, for instance:

- relationship types,
- entity type without attributes,
- entity type without identifier,
- compound attributes,
- multivalued attributes.

This converter generates SQL statements for creating and loading an SQL data base from an NDBS data base. The converter outputs two kinds of ASCII files. The first file contains SQL statements for creating tables and unique indices, and the other files contain SQL statements for inserting data lines into the SQL tables.

### 12.2 Schema translation principles

The NDBS schema to be translated is submitted to some structural restrictions that should keep the resulting relational structures readable :

- the components of a compound attribute cannot be compound,
- the components of a multivalued compound attribute cannot be multivalued.

An NDBS schema is translated according to the following rules.

| | | |
|---|---|---|
| **Entity type E** | → | Table with name E |
| • With an identifier | → | / |
| • Without identifier, with attributes and origin of no relationship type | → | / |
| • Without identifier, without attributes, origin of no relationship type, and target of relationship types | → | / |

- Without identifier, without attributes, origin of no relationship type

  target of no relationship types → add an identifying DBKey column with name **ID_**E

- Without identifier

  and origin of some relationship types → add an identifying DBKey column with name **ID_**E

Elementary, simple **attribute** A → column A

Elementary, multivalued **attribute** A[N] → N columns with names Ai, i = [1..N]

Compound, simple **attribute** → replace by components before generating;
Compound, multivalued **attribute**
A[N].(C1,C2,..,Cj,..) → each component Cj is translated into N columns with name Ci_j, i = [1..N] ;

**Identifier** A of E → primary key (A) + unique index on E(A);

An identifying DBKey column has been added → primary key (comumn) + unique index on this column;

For each identifying columns → declare it "not null"

One-to-many **relationship type** R(A,B) → column in B with name R and with the type of the identifying column of A;
foreign key (R) + index on B(R);

One-to-one **relationship type** R → column in B with name R and with the type of the identifying column of A;
foreign key (R) + unique index on B(R);

if B has no identifier, add a primary key (R) to B;

Additional name building rules :

- indices on table E are named XE1, XE2, etc;
- hyphens ("-") are translated into underscores ("_");
- the name length is limited to M characters (depending on the DBMS);
- no two columns of a table can have the same name; should this occur, the user will be asked for a new name.

## 12.3 Data translation

Each NDBS entity of type E is translated into an **insert into E values ( ... )** SQL statement.

The data of entities of type E are collected into a single file with name E.rda (for **r**elational **da**ta); if the name E is longer than 8, a new name is asked for to the user. If the file already exists, the user is asked for its replacing.

## 12.4 Example

The following SQL statements are generated from a variant of the ORDER data base.

### 12.4.1 SQL table and indices structures

```
create tableCUSTOMER (
          NUM smallint not null,
          NAME char(25),
          ADNUM smallint,
          ADCITY char(30)
          ADPHONE1 char(12),
          ADPHONE2 char(12),
          primary key (NUM)
          )
create unique index XCUSTOMER1 on CUSTOMER(NUM)

create table ORDER (
             NUM smallint not null,
             DATE char(6),
             CC smallint,
             primary key (NUM),
             foreign key (CC) references CUSTOMER
             )
create unique index XORDER1 on ORDER(NUM)
```

```
  create index XORDER2 on ORDER(CC)

  create tableLINE (
              ORDQ float,
              CL smallint,
              PL char(10),
              foreign key (CL) references ORDER,
              foreign key (PL) references PRODUCT
  )
  create index XLINE1 on LINE(CL)
  create index XLINE2 on LINE(PL)

  create tablePRODUCT (
              NUM char(10) not null,
              NAME char(30),
              PRICE float,
              QOH float,
              primary key (NUM)
              )
  create unique index XPRODUCT1 on PRODUCT(NUM)
```

### 12.4.2 SQL data insert statements

```
insert into CUSTOMER values (
              123,
              "Dupont, A.",
              24,
              "Paris",
              "1/6578849"
              "1/6578850"
              )

insert into LINE values (
              12.5,
              557
              "PAS1765"
              )

insert into LINE values (
              500
              557
              "QEB298"
              )
```

## 12.5 Organization of the SQL converter

Two phases must be distinguished, namely the generation phase and the conversion phase.

Through the **generation phase**, the SQL converter produces two programs for an NDBS data base D (Figure 1) :

- The SQL program that defines the SQL data base D' corresponding to the schema of D.
- A PASCAL/NDBS program that reads the contents of the NDBS data base D and converts it into SQL programs containing data insert statements.

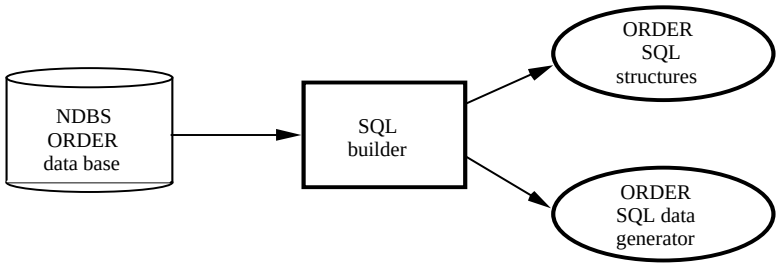


**Figure 12.1 -** Through the generation phase, a data base definition SQL program, and a PASCAL NDBS data generator program are built. The latter program must then be compiled.

The **conversion phase** corresponds to actually building and loading the SQL data base D'. It includes the following activities :

- building the SQL data base D'. Normally, this process should be activated only once;

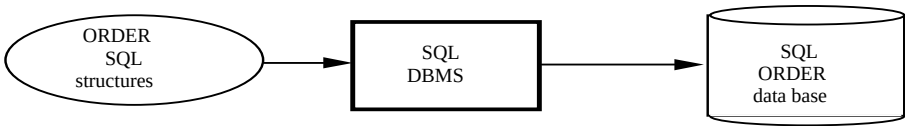


**Figure 12.2 -** Building the SQL data base.

- extracting the contents of the NDBS data base D and producing data insert SQL programs;

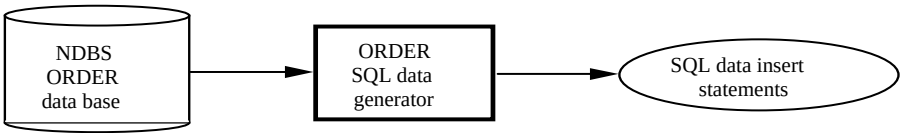


**Figure 12.3 - Figure 3 -** Extracting the data from the NDBS data base.

- submitting these data insert SQL programs to the SQL DBMS in order to load the specified data into the data base D'. These last two activities are generally carried out repeatedly.

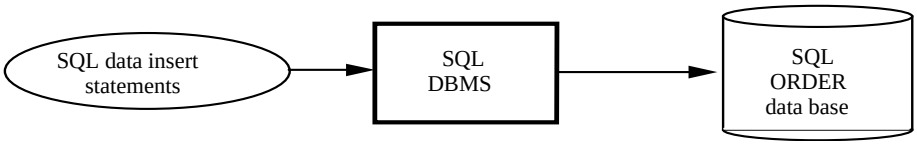


**Figure 12.4 -** Loading the data into the SQL data base.

# Part 13

## 13. NDBS - EVALUATION DE PERFORMANCES

### 13.1 Schema de test

```
database ADDITION
     entity-type CENT
       identifier VALEUR
       begin
         VALEUR : integer
       end

     entity-type DIX
       begin
         VALEUR : integer
       end

     entity-type ENTIER
       begin
         VALEUR : integer
       end

     entity-type PLUS

     rel-type CENT_DIX from CENT to DIX

     rel-type DIX_ENTIER from DIX to ENTIER

     rel-type OP1 from ENTIER to PLUS
     rel-type OP2 from ENTIER to PLUS
     rel-type RESULTAT from ENTIER to PLUS

physical-spec
     buffer 16
     storage CENT clustered
     storage DIX clustered
     storage ENTIER clustered
     storage PLUS clustered
end
```

### 13.2 APPLICATIONS

### 13.2.1 Programme ADDLOAD

*Description*

Chargement de données dans la BD ADDITION.
Nombre d'entités :

- CENT : 5 (VALEUR de 0 à 400)
- DIX : 50 (VALEUR de 0 à 490)
- ENTIER : 500 (VALEUR de 0 à 499)
- PLUS : 2500(VALEUR = E1 x E2 pour E1,E2 dans [0..49] )

*Algorithme*

```
1.0TIMER
1.1for I := 0 to 400 step 100
     create C := CENT (: VALEUR = I)
   endfor
1.2for C := CENT
     for I := C.VALEUR to C.VALEUR + 90 step 10
        create D := DIX (:VALEUR = I)
        modify D (CENT_DIX: C)
   endfor C
1.3for C := CENT
     for D := DIX (CENT_DIX: C)
        for I := D.VALEUR to D.VALEUR + 9
           create E := ENTIER (: VALEUR = I)
           modify E (DIX_ENTIER: D)
   endfor C
1.4TIMER

2.0TIMER
2.1for E1.VALEUR := 0 to 49
     READENTIER(E1)
     for E2.VALEUR := 0 to 49
        READENTIER(E2)
        create P := PLUS
        modify P (OP1: E1)
        modify P (OP2: E2)
        E3.VALEUR := E1.VALEUR + E2.VALEUR; READENTIER(E3)
        modify P (RESULTAT: E3)
2.2TIMER
```

Remarque. L'algorithme de la fonction "READENTIER(E)" est conçu

comme suit :

```
0. I := E.VALEUR
1. C := 1# CENT (: VALEUR <= I-99)
2. D := 1# DIX (CENT_DIX: C) and (: VALEUR <= I-9))
3. E := ENTIER (DIX_ENTIER: D) and (VALEUR = I))
```

*Analyse*

Partie 1

Création d'un cluster de 5 entités CENT
Création de 5 clusters de 10 entités DIX
Création de 50 clusters de 10 entités ENTIER

Bilan :

dbcreate : 555
dbinsert : 550
dbfirst : 2
dbnext : 8
dbfpath : 5
dbnpath : 45

Nombre de transactions : 555

Constitution d'une transaction

1 dbcreate
+ 0.99 dbinsert
+ 0.004 dbfirst
+ 0.014 dbnext
+ 0.009 dbfpath
+ 0.08 dbnpath

Partie 2

Création de 2500 entités PLUS, en 50 clusters de 50 entités via OP2, mais éloignés de leur origine

Bilan de la fonction READENTIER (exécutée 5050 fois)

dbfirst : 1
dbfpath : 2
dbnpath : 6.5

Bilan de la partie 2
dbcreate : 2500
dbinsert : 7500
dbfirst : 5050
dbfpath : 10.100
dbnpath : 32.825

Nombre de transactions : 2500

Constitution d'une transaction
1 dbcreate
+ 3 dbinsert
+ 2.02 dbfirst
+ 4.04 dbfpath
+ 13.13 dbnpath

### 13.2.2 Programme ADDLIST

*Description*

Lecture des entités de la BD ADDITION chargée par ADDLOAD.

*Algorithme*

```
1.0TIMER
1.1for C := CENT
     for D := DIX (CENT_DIX: C)
        for E := ENTIER (DIX_ENTIER: D)
   endfor C
1.2TIMER

2.0TIMER
2.1for E := ENTIER
     for P := PLUS (OP1: E)
        E2 := ENTIER (OP2: P)
        E3 := ENTIER (RESULTAT: P)
   endfor C
```

```
2.2TIMER
```

*Analyse*

Partie 1

Bilan
dbfirst : 1
dbnext : 4
dbfpath : 55
dbnpath : 495

Nombre d'accès logiques : 555

Partie 2

Bilan
dbfirst : 1
dbnext : 499
dbfpath (1-N) : 50
dbnpath : 2450
dbfpath (N-1) : 5000

Nombre d'accès logiques : 8000

## 13.3 RESULTATS

*Conditions*

Matériel : TOSHIBA 1100+ (8086, 7.14 Mhz, 640K de MC)
Mémoire secondaire : RAMDisk de 340 K
Logiciel : MS-DOS 3.22, TURBO-PASCAL 3
dans CONFIG.SYS : BUFFER = 20
Mesures : temps d'exécution :
mesuré par l'horloge interne (en 100ème de sec.), dbopen et dbclose exclus;
taux de transactions :
TPS, nombre de transactions par seconde (programme

ADDLOAD)

ALPS, nombre d'accès logiques par seconde (programme ADDLIST)

*Mesures du programme ADDLOAD*

Partie 1

temps d'exécution : 36 sec.

taux de transactions : 15.15 TPS

Partie 2

temps d'exécution : 480 sec

taux de transactions : 5.2 TPS

*Mesures du programme ADDLIST*

Partie 1

temps d'exécution : 3.2 sec

taux de transactions : 173 ALPS

Partie 2

temps d'exécution : 40 sec

taux de transactions : 200 ALPS