THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

Pyramide. A physical engine for the management of entity-relationship databases

Rossi, Didier

Award date:
1989

Awarding institution:
University of Namur

Link to publication

PYRAMIDE

a physical engine for the management
of entity-relationship databases

D. ROSSI

MEMOIRE

Présenté par

Didier ROSSI

en vue de l'obtention du titre de

**Licencié et Maître en Sciences
Option Informatique**

(Promoteur : Professeur J-L Hainaut)
Année académique 88-89

Mai 1989

Institut d'Informatique des Facultés Universitaires de Namur

ABSTRACT

The subject of this dissertation is the physical support of an Entity-Relationship model layer. I have developed a "network" Kernel Data Base Management System (DBMS), the data model of which is based on a subset of the entity-relationship model. This DBMS has been designed while keeping in mind the specific needs of the upper layer's model to be supported, thus allowing further extensions. Another dissertation is aimed at realising a true Entity-Relationship layer on top of the 'PYRAMIDE' (PhYsical engine foR the mAnageMent of entIty-relationship DatabasEs). Besides, I have also developed a build-in Data Dictionary (DD) facility. This DD could be used as a basis to offer high-level user facilities for dynamic schema generation, which is not often possible in existing database systems.

RESUME

Le sujet de ce mémoire est le support physique d'une couche du modèle Entité-Association. J'ai développé un noyau de système de gestion de base de données (SGBD), dont le modèle de données est basé sur un sous ensemble du modèle Entité-Association. Ce SGBD a été conçu en gardant à l'esprit les besoins spécifiques du modèle de la couche supérieure, permettant ainsi des extensions futures. Un autre mémoire a pour but de réaliser une vraie couche Entité-Association au dessus de 'PYRAMIDE' (moteur physique pour la gestion de bases de données Entité-Association). A côté de cela, j'ai également développé une fonctionnalité de Dictionnaire de Données (DD) incluse dans le système. Ce DD pourrait être utilisé comme base pour offrir des fonctionnalités de haut niveau pour la génération dynamique de schémas par l'utilisateur; ce qui n'est pas souvent possible dans les systèmes de base de données existants.

THANKS

It is a bit difficult to thank the persons who have more or less contibuted to a work as one is always afraid to forget someone. Nevertheless, it is really a pleasure for me to have the opportunity to thank the people I remember for their help during my studies.

- I want first of all to thank my parents and my wife. Without them I wouldn't have been able to complete my studies. They have always been by my side when I needed them, while staying away when it was necessary.

- Then, I want to thank all the database research group members at Forschungzentrum Informatik (**FZI**) of Karlsruhe (West Germany) where I have accomplished my last year traineeship. I learned a lot while working on the DAMOKLES - DBMS (UNIBASE project). I will never forget how Doctor Klaus **Dittrich**, Michael **Ranft** and Angelika **Kotz** have welcomed myself, and how helpful and nice the other members of the group have always been to me. Besides the technical abilities I have learned there, I've also earned very good friends.

- I cannot forget the teachers of the **Institut Saint Berthuin** of Malonne who gave me a very rich educational background during my secondary studies. The quality of their teaching and the basic principles they inculcate to their students are very useful in every day life. May I thank more especially professors **De Boel, Leroy, Richelle** and the director mister **Debources**.

- And last but not least, I want to thank all the academic staff of the **Institut d'Informatique** at the Facultés Notre-Dame de la Paix of Namur for the exceptional framework they provide to their students. The well-known quality of the education I have received here is far more useful than only some technical knowledges. And I think I will understand even more things with the time going on. May I thank more especially Professor Jean-Luc **Hainaut** whom I have been working with for three years, and who still helped me a lot for this dissertation.

TABLE OF CONTENTS

Section A :

INTRODUCTION

In the recent past, major advances have been made in the area of Software Engineering Environments (SEEs) and Computer Aided Design/Manufacturing (CAD/CAM). These advances are evident in many application areas such as computer aided drafting systems for instance. But when CAD/CAM application systems were first introduced, they each operated independently and data was prepared manually as input to the individual programs. Gradually, engineers recognized production inefficiencies due to gaps in the data flow, and sought to bridge these gaps in a variety of ways. As Data Base Systems (DBSs) had shown many advantages in administrative and economical fields, it was sensible to also try and use them in engineering fields.

But as these new application's domains of DBSs differ from the usual domains by their specific demands, it is not surprising that the current Data Base Management Systems (DBMSs) on the market do not adequately support these non standard application's domains. Numerous attemps are being made to use existing DBMSs for storing and manipulating CAD/CAM data; however, many features of these data are troublesome for current DBMSs. One of the reasons is that the currently available data models are not designed to efficiently handle complex information structures. Also, designers and engineers need additional capabilities which conventional data management systems do not offer.

It is then time to develop data models which are aimed at supporting such special domains, so that they allow to define particular data structures needed in these non standard applications. The new concepts that are to be found in such data models are for instance, modelisation and manipulation of complex structured Entities, of versions of Entities and generalized n-n Relationships.

In order for a DBS that offers such a particular data model to keep acceptable performances, one must also develop new techniques that support efficiently these new concepts.

The subject of this dissertation is the physical support of an Entity-Relationship model layer. I have developed a "network" Kernel

Data Base Management System, the data model of which is based on a subset of the entity-relationship model. This DBMS has been designed while keeping in mind the specific needs of the upper layer's model to be supported, thus allowing further extensions. Another dissertation is aimed at realising a true Entity-Relationship layer on top of the 'PYRAMIDE' (PhYsical engine foR the manAgeMent of entIty-relationship DatabasEs). That dissertation should be presented by Feraille Patrick and Tomasi Matthias in september of this year. Besides, I have also developed a build-in Data Dictionary (DD) facility. This DD could be used as a basis to offer high-level user facilities for dynamic schema generation, which is not often possible in existing database systems.

This document is made up of five main parts. We have first studied the specific requirements of such a DBS. This study is driven from the researches made by the Database Research Group at the "Forschungzentrum Informatik" of Karlsruhe (West Germany). Then we propose a specification for the DBMS that fulfills our specific goals. A thorough description of the PYRAMIDE programming interface follows, before we explain the physical structure of PYRAMIDE and discuss some implementation aspects. Finally, we conclude this dissertation by evaluating the current environment and giving some ideas for further developments.

Section B :

Requirements Analysis

1.   The State of the Art

This chapter tries to summarize the researches that have been done or that are still under way at FZI-Karlsruhe. The reader needing further development on the subject can consult the literature given at the end of this dissertation. Much of the information are driven mainly from the Damokles and the Damascus projects.

1.1 The Situation in current CAD/CAM systems

One can predict an even larger trend toward CAD/CAM in the future in the industrialized nations. And even though the database is often the central part of such systems, it has traditionally received the least attention. The majority of CAD/CAM systems are based on a customized file system with no generally accepted format to which other modules can interface. Data exchange between modules requires tedious conversion. The reasons are manifold. First of all, today's commercial DBMSs don't adequately support technical problem domains, and it is not clear at all that general purpose DBMSs can achieve the same efficiency that is possible with a special purpose file structure. Secondly, CAD/CAM systems are usually designed and implemented by engineers who are not database experts, and DB experts have traditionally ignored technical problem domains.

The earliest applications of CAD have been in mechanical engineering, where the goals of the CAD process are the design of a mechanical part and the planning of its manufacturing process. But among the engineering applications that have received considerable attention in the recent past by database experts, the design of very large-scale integrated circuits (VLSI) is one of the most popular one. Here, the availability of automated tools relying on an integrated database supported by a suitable DBMS, is an essential precondition for an economical design of customer specific chips. Today's VLSI design is highly automated and a complete CAD tools support is often offered.

But in most case, the designer doesn't access the DB directly. It remains invisible for him and he only expects high performance regarding the CAD tools. But since many of these tools are fairly time consuming because of a high algorithmic complexity (simulators), special care must be given that they are not further slowed down by a large amount of I/O operations caused by the DBMS. Particularily regarding read accesses, since they tend to outnumber the write accesses. Several megabytes of data volume should be read in a few minutes, where the designers seem to tolerate much larger delays when writing the same volume to secondary storage.

1.2 The Situation in current SEEs

Software development is today regarded as an economic activity to which productivity and engineering principles have to be applied. In the last years, the hardware costs have been decreasing while software costs were constantly increasing. This phenomenon is known as the so-called "software crisis". This arises mainly from the fact that softwares are often badly specified, with a bad architecture and are poorly documented.

The costs for the development of a software can be divided in analysis of the needs (10%), fonctional specifications (10%), design (15%), encoding (20%), unitary tests (25%) and integration tests (20%). So that's 45% of the costs to correct (tests) what has been badly specified (10%). What's more, the development costs represent only 33% of the total cost; the larger part (67%) being for the maintenance of the software [Lams87].

To deal with this "software crisis", Integrated Software Engineering Environments have been developed aimed at increasing the productivity of the software development process, and the quality of the final product. The main emphasis has been put on the analysis of the development process, the automation of some steps which seemed fairly straightforward and the integration of tools. So that today, Software Engineers have got at their disposal, a large quantity of methods and tools (such as compilers and editors) that have already

achieved a high degree of sophistication and that can be used during the different phases of the Software Life Cycle (SLC).

Nevertheless, most of the solutions are of an isolated nature and an integrated method covering all phases of the SLC is still missing. Various 'tool boxes' are offered, but the total integration of tools and the covering of the entire SLC can only be achieved with the help of a data management component. Often, tools from two boxes or sometimes even from the same boxe cannot easily be combined, because of the mismatch between desired inputs and produced outputs, or between the underlying methodologies. In today's available SEEs, the data management is often build directly on a particular Operating System (OS), enhanced with special mechanisms -as for instance Version Control- needed by some tools. This solution leads to the well-known problems of high redundancy, lack of data independancy, etc.

But over and above the high data volume to be managed, standardization of existing and new tools is one of the prime motives for the desire to introduce DBSs into SEEs. DBMSs can have an integrating effect because they impose rigid standards on the data organization. The integration must be based on the SLC that structures the software development process into phases, and which may differ from one company to another, or even from one project to another. That is, the DBS must not impose one peculiar model, but has to support different ones.

1.3 Conclusion

Data base concepts such as data modelling and manipulation, data independancy, data security, data integration, consistency control, access control and multi-users support seem to be very attractive also for technical applications such as SEEs or CAD/CAM. But as we will see, conventional business and administrative DBMSs cannot adequately support the special needs of technical fields. Especially, the classical data models, and the "short transaction" and consistency concepts, lead to severe performance problems when applied in technical fields. That is, in order to keep acceptable performances

in a DBS applied to SEEs, one has to develop a completely new DBMS taylored to technical applications.

2. Engineering applications seen as particular Data Base applications

On one hand, Engineering Applications were in the past often based on file-systems. But these systems, as we will see, are not able to fully support the main requirements of these peculiar applications. That is, special tools had to be developed to meet these requirements.

On the other hand, in economical and administrative fields, DBMSs have been of great help. But these conventional systems are inadequate for Engineering applications. The principal weaknesses come from the traditional data model concepts such as consistency and short transactions. But other characteristics of Data Bases (DB) should also benefit Engineering applications.

2.1 Particularities of CAD/CAM applications

In VLSI design, custom design falls into **full-custom design** (where a chip is designed from scratch according to given specifications), and **semi-custom design** (by employing pre-defined structures). In full-custom design, the design cost is high while optimal performance can be achieved. In semi-custom design, one can do : - **Cell-oriented design**, where the designer has to select particular cells (circuits already designed) from a library and to properly interconnect them before evaluating the overall result;

- **Gate array design**, which is very much like cell-oriented design, but where the cells exist already physically (as a master chip) thus lowering design and production costs, but reducing design flexibility;

- **Design based on regular structures**, where cells of a single type are arranged in a regular pattern being prefabricated and which then requires only a few physical additions or alterations.

Furthermore, one can use several techniques depending on the design phase. The design can proceed in a **Constructive fashion** with interactive tools (such as graphic editors), or in an **Automatic fashion** where tools algorithmically generate the result. **Semi automatic** methods requiring human intervention only at critical points, may also be used.

The design process is often presented as a number of (possibly iterated) phases : System level design - Register level design - Logic design - Transistor level design - Geometrical layout design. These phases being themselves decomposed into the same sequence of steps : Specification - Overall design - Detail design - Verification. The specification of a phase (except the first) is always derived from the result of the preceding one, and the three design methods differ mainly by the input information that each of the phases requires.

Each phase produces results called **representations** (in a fixed number for a given methodology), which taken in their totality, form a complete description of a circuit and its properties. Some examples of representations are **functional descriptions** on various levels, **circuit models** (in the form of text, program code or tables) to be used for simulation and test data generation, **manufacturing data** for controlling chip production...

Variants (or alternatives) arise when a result produced in one phase has more than one successor during the next phase (e.g. different circuit diagrams). Also, as one can abandon a design and return to preceeding results, a **design history** must be maintained in the form of a sequence of **revisions** that reflect the chronological evolution of the design process. To arrive at the overall design description, the designer has to choose a **configuration** : that is, one revision from each variant and one variant from each representation (see figures B.1 and B.2).
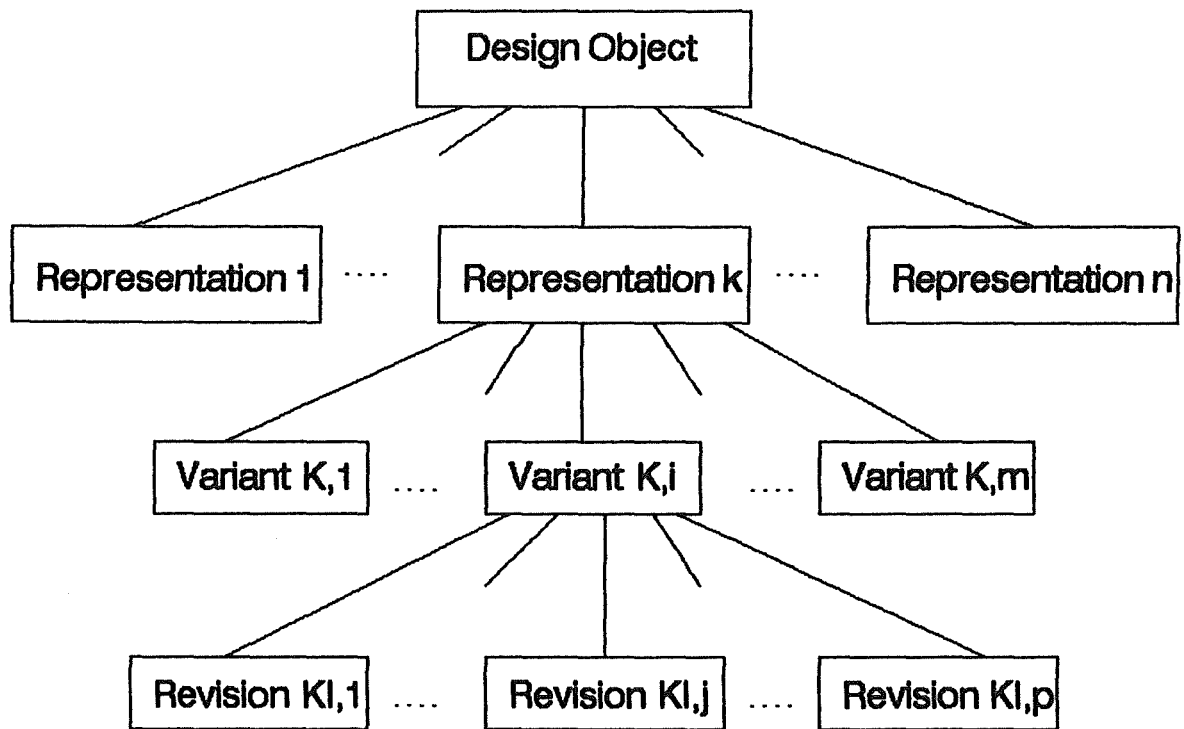
Figure B.1 : Global structure of a design object.
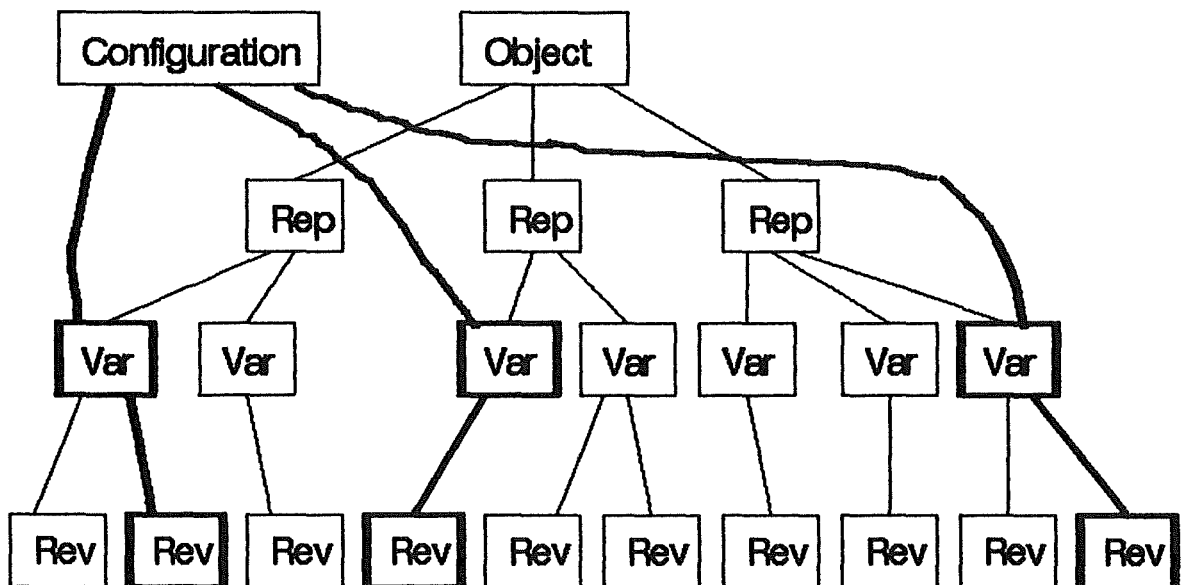
Figure B.2 : A design object instance and a configuration

The design process in mechanical engineering is very much like VLSI design. The different construction methods are :

- **Original design**, when a novel solution is needed, the components being already known or needing to be constructed as well;

- **Adaptive design**, when an existing solution has to be modified (functionality and component arrangement);

- **Variant design**, where functionality and components are preserved, being only modified in fine shape, material properties and scale;

- **Principle design**, being variant design restricted to scaling.

The designer can also strive for an economical and fast production by using to the largest extent possible standardized parts within the company or the industry, that are kept in the inventory.

Here again, the design process can be decomposed in (possibly iterated) phases : List of requirements - Definition of the function - Definition of the physical principle - Shape design - Detailling - Manufacture planning. Each phase including computing and drawing as major activities, tools based on models emphasizing the geometric properties of technical Objects are offered. They include metrics, tolerances and surface properties, and permit the computation of side or section views. A number of approaches for the description of the geometry exist which are thoroughly explained in [Kemp87].

Like in VLSI design, an object has a number of representations (one or more per phase), and is decomposed into smaller Objects (often predesigned) that can be obtained from archives and then be used after suitable parametrization. As the designer is an indirect user of the database (interacting with it through various tools), performance with regard to data retrieval is of prime importance.

Unlike VLSI design, mechanical products exhibit little regularity in their composition of smaller Objects. Also, the version problem is less demanding as the representations are not all part of the final object, but rather play the role of intermediate information. The concept of variant is also somewhat different as an object hasn't got

variants, but may rather be a variant of some other design object. Revisions are still needed, but mainly in order to trace the design history. And as a consequence of this, the concept of configuration is not much relevant in this field.

2.2 Particularities of Software Engineering Environments

There are close similarities between software and VLSI design. In both domains the design object is a large and complex structure composed of numerous smaller objects and each phase of the Life Cycle constructs results giving different aspects of the global object. The life cycle of software development is often presented as : Requirements analysis - Functional specification - Overall design - Component design - Component implementation - Component test - System integration and test - Installation - Maintenance [Lams87]. But this crude organizational model is usually refined by imposing local models on the various phases, giving like in VLSI design, a design procedure plan that determines the exact order (including alternatives) in which the tools are to be applied. Phase-specific tools manipulate the representations within one phase, where transformation tools bridge the gap between successive phases. Further global activities (e.g. project management) cannot be associated with a single phase, but must also be supported.

These phases can be decomposed into (sub-) phases and so on; so that the development process can be represented statically by a treelike structure of units of work. The tools used during the different phases in SEEs transform or analyse so-called **documents**. These documents are part of the software product being developed, or they represent intermediate results of the development process. The resulting documents of a phase are also called **representations** of the software product (source or object code for instance). The communication between work units is done as the output document of the preceeding unit becomes the input for the following unit.

But the dynamic development process does not follow the linear static structure. There are many iterations during which, a unit of

work will be redone. The reasons are that for instance, the result of a work unit is no more necessary, as it doesn't lead to a solution of the problem. In that case, the development process must go back to find a better result (**trial and error**). Or the development method is so, that there are iterative units of work; that is, the result of the preceeding iteration will be employed in the next one. So we get a hierarchy of documents where each document is refined into a quantity of documents of the same type (stepwise refinement). Or another reason can be that the solution found from a document has other alternative solutions (variants).

This is another similarity to VLSI design : **variants** and **revisions** for representations; usually covered under the standard term **versions**. Variants are design alternatives within a given representation that usually share the same module interface while differing in its implementation. Revisions reflect the design history of a given representation or variant in chronological order. The version control ensures the user has the capability to manipulate and access different versions, that given their usually huge amount, must be stored and handled in an efficient manner.

Therefore, another important concept here, is also the one of **configuration**. That is, a particular composition of the system where one variant and in turn one revision of each component representation has been chosen. Here again, the DBMS must offer facilities for representing, manipulating and accessing configurations in an efficient manner. As a difference to VLSI design, one must note that here the number of components in a configuration is much higher (that is, VLSI circuits exhibit regular properties where software products do not), and older versions are relevant over long periods of time due to maintenance requirements.

In order to cut down development costs, existing software modules should be reused whenever possible. So that like in VLSI design, libraries play also an important role in software design. But here again, the number of library elements is larger than in VLSI design because of the lack of regularity in design objects. Also the need

for availability being much longer than in VLSI design (because of the life expectancy), archiving is an essential aspect of software engineering and should be closely integrated with the rest of DBMS's activities.

3. Advantages of DBMS support

The main advantage offered by a DBMS in comparison to the files system approach is the integration of data and tools [Ditt86-b]. In usual files systems, each tool uses its own separate files and individual formats. Therefore, transformation programs are needed to interface the different tools. Furthermore, each time a new tool is created or modified, this set of programs has to be modified or extended. And of course, this separate storage of data for each tool leads to the well-known problems of redundancy and inconsistencies between the different copies of the data.

A Homogeneous standardized interface to realize *data integration* is therefore clearly desirable. As all users and application programs are provided with a stable common interface, tool construction becomes easier and more comfortable. The DBMS "view mechanism" can also support the necessary adaptation to the individual format each program requires.

When designing an application, the concepts of a certain area of interest (the so-called "perceived world") have to be mapped to data structures and operators supported by the DBMS (DB design) [Hain86-a]. This mapping depends strongly on the *data model concepts* the DBMS offers (structuring capabilities, generic operators and implicit consistency constraints). This design process results in a database schema serving as the structural skeleton for the actual database. The concepts of a data model enable the user to express the semantics of an application in higher level terms than bytes. Thus the design process is much more user-friendly and the result is self-documenting by means of the database schema.

DBSs offer *data independancy* on the physical as well as on the logical level. That is, an application ignores the physical storage structures and storage media actually being used by the DBMS. Thus for intance, storage characteristics can be changed in order to improve access performance without modifying the application programs. Information about the logical structure of the data is kept in the DBMS, while each application program can use its individual view; thus, permitting structures for new purposes to be created without modifying any of the previously existing programs.

DBMSs also offer different mechanisms (as logging and recovery management) in order to insure *data security* against loss or damage in case of hardware, software or operating errors.

As the framework for structural consistency defined by the data model is generally not sufficient to insure data consistency (that is the database represents a complete and correct image reflecting the world of interest), *consistency control* enables the user to specify additional explicit consistency constraints which are checked automatically.

DBMSs perform *access control* in order to insure the privacy of sensitive data contained in the database that has to be hidden from unauthorized users. A user's view of the database can be restricted to a subset of data or operations allowed on these data.

4. Specific DB requirements of Engineering applications and existing DBMS technology

4.1 A specific data model

The data model described herein is the one developed at FZI-Karlsruhe (West-Germany) in the framework of the DAMOKLES-DBMS, and is only given as one example of what could be the data model supported by the upper layer running on top of PYRAMIDE.

4.1.1 The concept of structured objects

The basis for the development of a new DBS is its data model. It describes the functional DBMS interface as it is represented to the outside world and determines the facilities for data storage, update and retrieval. The data model excludes all aspects of the physical data organisation (data independency), focusing only on its logical properties.   The principal requirement for the data model of an engineering DBS is that it provides a structural object-orientation. That is, each object of the real world should be built onto one database object, independantly of its internal structure [Ditt86-a].

Engineering applications often manipulate objects of complex structures built out of simpler objects in a hierarchical or netlike manner.   This reflects the two typical design methodologies of decomposing higher-level objects into a set of component objects (stepwise refinement), or of using already existing objects in building complexer ones (down-up).

An object is an entity of the so-called "perceived world" and it is described by its properties.   Objects with the same properties build up an object type.   But as objects in engineering applications are structured objects, they are not only described by descriptive properties, but also by structural properties to express that an object is composed of sub-objects (that can be in their turn also structured objects). The designed object itself is entirely described by the set of all representations and the nodes of the structural tree are often also structured.

Instead of structured objects, one can also talk about objects hierarchies that are not necessarily tree-like (each object has at most one father), but can also be organised as networks (an object can have more than one father). That is, hierarchies can overlap.

Descriptive properties can be described with attributes that build a correspondance between simple type value(s) and an element of the object type. A (combination of) attribute(s) can constitute a key.   In addition to basic simple types, there should also be

constructors that enable the user to define new types (by enumeration, union or intersection).

Structural properties describe the composition of an object into inter-related sub-objects (structured object). Sub-objects can also be structured objects, so that we have an objects hierarchy. Also an object can be part of many structured objects of the same type or different types (overlapping); and a structured object can comprise sub-objects of the same type as his (recursion).

In addition to the concepts for data modelling, we must also have new operators that can deal with such structured objects. These are for instance : creation of objects (and sub-objects), insertion and removal of sub-objects in/from structured objects, reading and writing of attribute values, copying and deleting of objects, navigational search in order to find all sub-objects and relationships belonging to a structured object and reversely, and associative access on basis of attribute values. Objects can also have a surrogate key (given by the system) which is system-wide unique and can never be changed. In that case, navigating operations can give the keys of the found objects as a result [Damo88] and [Härt87].

Long fields (strings of Bytes of any length) with highly efficient, file-like access operations are also required to deal with large data volumes whose structure remains unknown to the DBMS.

4.1.2 Relationships

Relationships are associations of objects in which an object plays a specific role. Relationships can also have attributes and build up relationship types. Consistency constraints can also be formulated by using the cardinality constraints. In an object-oriented data model, we should also have n-n relationships between structured objects (inter documents relationships), and between sub-objects of a structured object (intra document relationships). These relationships should of course be provided with efficient navigation operations along them.

4.1.3 Versions and Configurations

In addition to the data representing the results of the various design steps themselves, **design management information** has to be maintained.   For each design object there are a number of representations; that is, various forms of describing the object at different levels of detail, different methods of description, or emphasizing different aspects.   These representations are the inputs and outputs of the different steps in the design process.   Each representation comprises a number of <u>variants</u> (alternative approaches); and each variant shows a design history consisting of a linear sequence of design states, that is <u>revisions</u> (timed modifications).

In order to be able to manage revisions and variants, the data model must allow the modelling of <u>versions</u> of a structured object. That is, each version belongs to a <u>generic object</u> that as well as its versions, can have descriptive and structural properties.   Each version inherits the properties of its generic object, while different versions differ from themselves by the values given to particular attributes (deltas).   The versions are ordered in a linear, tree-like or acyclic graph <u>(versions-graph)</u>.   And finally, as versions are also objects, they can particip in relationships, have attributes and sub-objects, as well as versions.

The data model must also include operations that allow to navigate in a versions-graph, find the generic object of a given version, create and delete versions.

Another problem closely tight to the versions management is the management of <u>configurations</u>.   The user must be able to define a configuration; that is, he chooses for each object, which version he will be working with.

4.1.4 Discussion

The data model exposed previously could be supported by special relationships (under-object and version relationships). But there are at least two reasons not to do so [Ditt86-b] :

- Structured objects and versions are standard requirements in the target application domains. Corresponding concepts in the data model makes easier the semantic modellization; and therefore, the resulting schema is more understandable.

- One can only achieve efficient implementation of these two concepts by using specific techniques (objects clustering and delta mechanisms for instance).

4.2 Other requirements

4.2.1 Data consistency

Consistency concept indicates the fact that there is a one to one correspondence between the content of the DB and the part of the world it models. **Consistency constraints** cover the real world laws that must be enforced to data, and that cannot be described only by means of the data model. An operation violating a consistency constraint is usually rejected, where in the engineering field it must be tolerated because the operation is complex and costly.

Engineering applications comprise numerous consistency rules of high complexity that apply to database states as well as transitions between states. A large number of constraints are embodied algorithmically in existing consistency control programs that have to be exploited.

What's more, design is a lengthy stepwise and iterative creative (sometimes trial-error) process involving a large amount of data. Consistency cannot be achieved all at once; in fact inconsistencies must be tolerated over varying periods of time. Consequently, consistency checking cannot be done automatically after each DB update. The DBMS must provide consistency checking facilities that

can be requested by the designer when needed. To allow flexible check times and a variety of reactions when constraints are violated, the DBMS can only provide basic mechanisms for consistency control that the user/tools have to apply appropriately.

4.2.2 Transaction management

Transactions are user-defined units of work including sequences of DB read and update accesses that transform a consistent DB state into another consistent state. A transaction is **atomic**, that is all updates within a transaction success or none is completed. A transaction can **commit**, that is all operations within the transaction have succeeded and deliver the DB in a consistent state. Or it can **abort**, that is at least one operation of the transaction failed leaving the DB in a inconsistent state. In that case, the transaction must be **rolled back**, that is all operations completed since the beginning of the transaction must be undone. And finally, an external incident (system crash) can occur while a transaction is going on. When the DBMS is restarted, all actions done since the beginning of the transaction must be ignored (**crash recovery**). But the classical notion of transaction (atomical, short duration, few data involved) does not hold here and consequently transaction management must be adapted to special requirements.

On the contrary to traditional applications with short transactions (duration of some seconds at most), engineering applications are characterized by very long transactions that may last hours or days and even span across several sessions. One can even say that a transaction begins with the initialisation of the design, and ends when the object is completely designed. These transactions access large volumes of data that are however often confined to one design object or selected parts of it.

4.2.3 Data integration

The DBMS integrates all the tools of a given design environment by integrating its data and enforcing certain standards for their representation. But in contrast to traditional applications with frequent interactive data access, engineering applications are made up of tools (programs) that access the database. Consequently, the programming interface of the DBMS is much more important than its interactive interface.

4.2.4 Performance

In spite of the eminent advantages discussed so far, a DBMS will not be acceptable for use in engineering applications if it does not offer adequate performance. Tools accessing large amounts of data have to be supported without causing intolerable delays. At best, slightly longer times are tolerable for updates. Among others, efficient storage structures and access mechanisms must be used in the DBMS in order to achieve the needed performance. Some painful compromises seem unavoidable; for instance, long fields might be used for complex objects even though this would relegate the consistency control to the application programs.

5. Shortcomings of existing DBMS

Experiences reported in the literature have shown that today's commercially available database systems do not meet most of the requirements listed above. The main reasons for these shortcomings are as follows:

5.1 Data modelling

Existing systems offer traditional data models that are well suited for simple, 'flat' data structures with a small number of attributes per entity. These data models, especially the relationnal model, are too poor to allow the total expression of all the semantics

of the data. Attribute domains are simple numerical or text types. Objects of complex structure with complex attributes are not supported. The DBMS leaves the contruction and management of such objects totally to the application program, thus lacking user comfort, implicit consistency, and above all efficiency. Current DBMSs offer sophisticated mechanisms to access atomical data, where an engineering object is often composed of sub-objects thus giving many data when accessing one object. Often also, nothing is provided for versions management.

5.2 Consistency control

As far as today's DBMSs offer mechanisms for explicit consistency control, it is accomplished in a rather fixed manner. The only way to define consistency constraints is by logic predicates. Thus procedural definition and integration of existing control algorithms are not provided for, and one cannot deal with the large number of highly complex consistency constraints that have to be expressed in terms of test procedures.

All constrains have to be met at the end of each DB-transaction which does not allow for user-defined check times, discrimination between local and global constraints, and so on. If an inconsistency is detected, the whole transaction will be rolled back, which is not tolerable in case of long transactions.

5.3 Transaction management

The classical transaction management is geared to short transactions only. These transactions are used as the units of consistency, recovery, and synchronization. At least the first two of these issues need to be reconsidered in an engineering environment. In design automation, neither the long duration of transactions, nor their high data volume, nor the temporary consistency violations fit into the traditional concept. All this raises serious doubts as to

whether transaction management as known in current DBMSs makes any sense to engineering DBMSs.

In the recovery area, differences between commercial and engineering applications are less drastic. Recovery from system crash or media failure should still stand. The main difference is found in handling transaction abort, where rollback only extends to the most recent save point. The notion of atomicity must also be revised because each operation is costly, and the designer wants to restore after a crash, as much as possible of his work. Often, temporary data inconsistency must be tolerated, because one cannot reject a set of lengthy and costly operations that lead to an inconsistent object. This object must be temporarily admitted till it is transformed into a consistent designed object (which is the final goal).

5.4 Performance

Bad performance is perhaps the most important reason for the low acceptance of DBMSs in engineering. The storage structures and access mechanisms offered by database systems so far are suited to simple structures with few inter relationships. They are tailored to a database with many data records belonging to a relatively small number of different types. On the other hand, CAD/CAM applications for instance, are characterized by a great number of different types, each of which has rather few instances. Furthermore, in existing DBMSs the user has to do extensive navigation to collect all the information describing one complex object.

6. Principal solutions to DBMS support

As it has been pointed out, DBMS support exhibits a number of advantages for engineering applications, though current solutions are far from satisfactory. There are at least four principal ways to overcome this situation [Ditt86-b] :

6.1 Restricted DBMS support

A solution chosen in some existing systems is to keep the proper design data on files as before and to store only some structural design management information in a (common) DBMS. This approach, however, is only a first step towards comprehensive DBMS support as all major advantages are not available for the engineering data themselves.

6.2 Database Design

A conventional DBMS is used to store all the data. The complex mapping of the application's semantic is completely left to the user (database design) and is thus hidden in programs and their documentation. With this approach, all the shortcomings of conventional DBMS are retained.

6.3 DBMS extention

This approach tries to overcome at least the data model shortcomings by extending a conventional DBMS or putting an additional layer on top, the interface is augmented to comprise application specific modelling concepts. These concepts are automatically mapped onto relations, sets etc. There is, however, no way to adapt the system performance to the new 'on top' concepts, and all the other drawbacks regarding transaction management, consistency control and the like remain or need special treatment.

6.4 New DBMS

A completely new DBMS is developed providing all necessary features. This solution is the only one that can offer an application-specific interface combined with efficient storage structures and adequate capabilities for synchronization, etc.

6.5 Conclusion

While the first solution is at least suitable as an intermediate approach until better support is available, the second solution clearly is a 'non-solution'. Though the third solution seems to be rather attractive, it incorporates surprisingly high system development effort (if all issues are considered) and still does not render the desired efficiency. It is however a good rapid prototype solution to try out novel data models. In the long run, only the development of specific design database systems can provide the necessary functionality and performance.

Section C :

Specification of the

PYRAMIDE Project

1. The models of PYRAMIDE

The term "database" is generally used to denote a collection of data which represents a subset of the real world; the so-called application domain. To describe what a PYRAMIDE database is, we shall speak in terms of Entity types, Attributes and Relationship types, as its data model is a subset of the Entity-Relationship model. We will give a short definition for each of these terms, the reader looking for more explanation can consult for instance [Boda83] or [Chen76]. The specific E/R model supported by the upper layer will be more formally described in the dissertation of Feraille patrick and Tomasi mathias realizing the E/R layer.

Let's take the following (somewhat simplified) example defining a database for a SEE as described in the requirements analysis. A software product is described by documents. Each document can have many versions. And each version has been realized by one or more authors under a contract number. This will be used as a basis for our next programming examples.

1.1 The Entity-Relationship model

An **Entity** is an abstraction of an object being important for the application domain (e.g. a chip, a circuit in VLSI design, a function or a program in Software Engineering). Entities are classified into **Entity types** (e.g. PART is the collection of all the parts used in a CAD system).

When entities are in relation one with another, we speak of **Relationships**. The collection of similar relationships is called a **Relationship type**. Each relationship type is described by the number, the type and the role of the entities that participate to the association. A relationship type can connect two (**binary**) or more entity types (**n-ary**) together. Each entity member of the relationship type can also be characterized by its **Connectivity** (in how many relationships of this type must we in minimum and can we in maximum

find the same entity). For example, in the 'DESCRIPTION' binary relationship type, each entity of the 'SOFTWARE' entity type can be associated with any number (say N) of entities of the 'DOCUMENT' entity type, and each 'Document' entity can be associated with only one 'Software' entity. Such a relationship type is therefore called 1-N (one to many), from 'SOFTWARE' to 'DOCUMENT', and N-1 (many to one) from 'DOCUMENT' to 'SOFTWARE'. In the same way, a 'REALIZATION' relationship type for instance, associating any number of 'Version' (of a document) entities to any number of 'Author' entities, and vice-versa is said to be many to many (N-N). We can also speak of a 1-1 relationship type in case of a one to one association.

Properties of the application domain's objects are represented by **Attributes** attached to entity types or relationship types. A particular entity or relationship has got **attribute values** for each of the attributes attached to its type. An attribute can either be **simple** or **repetitive**, **elementary** or **decomposable**, **mandatory** or **optional**. For instance, one can say that the 'DOCUMENT' entity type has a 'NAME' and a 'SUBJECT' as attributes (see figure C.1).

An **identifier** of an entity type is a group of attributes and/or roles, for which two entities of this type cannot have the same combination of values. The 'NAME' attribute is an identifier for the 'SOFTWARE' entity type, as there can't be two different softwares that have the same name. A relationship of a given type can also be identified by a group of attributes and/or the roles assumed by the entities on which it is defined. (e.g. A BORROW relationship is identified by the BOOK that is borrowed and the beginning date of the borrowing).

1.2 The logical model of the DBMS

The current data model supported by the PYRAMIDE-DBMS is a subset of the E/R model. It comprises the concepts explained above with the following restrictions. The relationship types are only binary and of the 1-N kind (and N-1 in the reverse order). There are attributes only for entity types, and they are always mandatory. And only a

SOFTWARE

Name
Release

(1-N) | Is_Described_By

DESCRIPTION

(1-1) | Describes

DOCUMENT

Name
Subject

(1-N) | Has_Version

VERSIONING

(1-1)   Is_Version_Of

AUTHOR

Name
Team
Address[2]
Street
Number
Locality

(1-N) | Realizes

REALIZATION

Contract_Nbr

(1-N)   Realized_By

VERSION

Num
Date
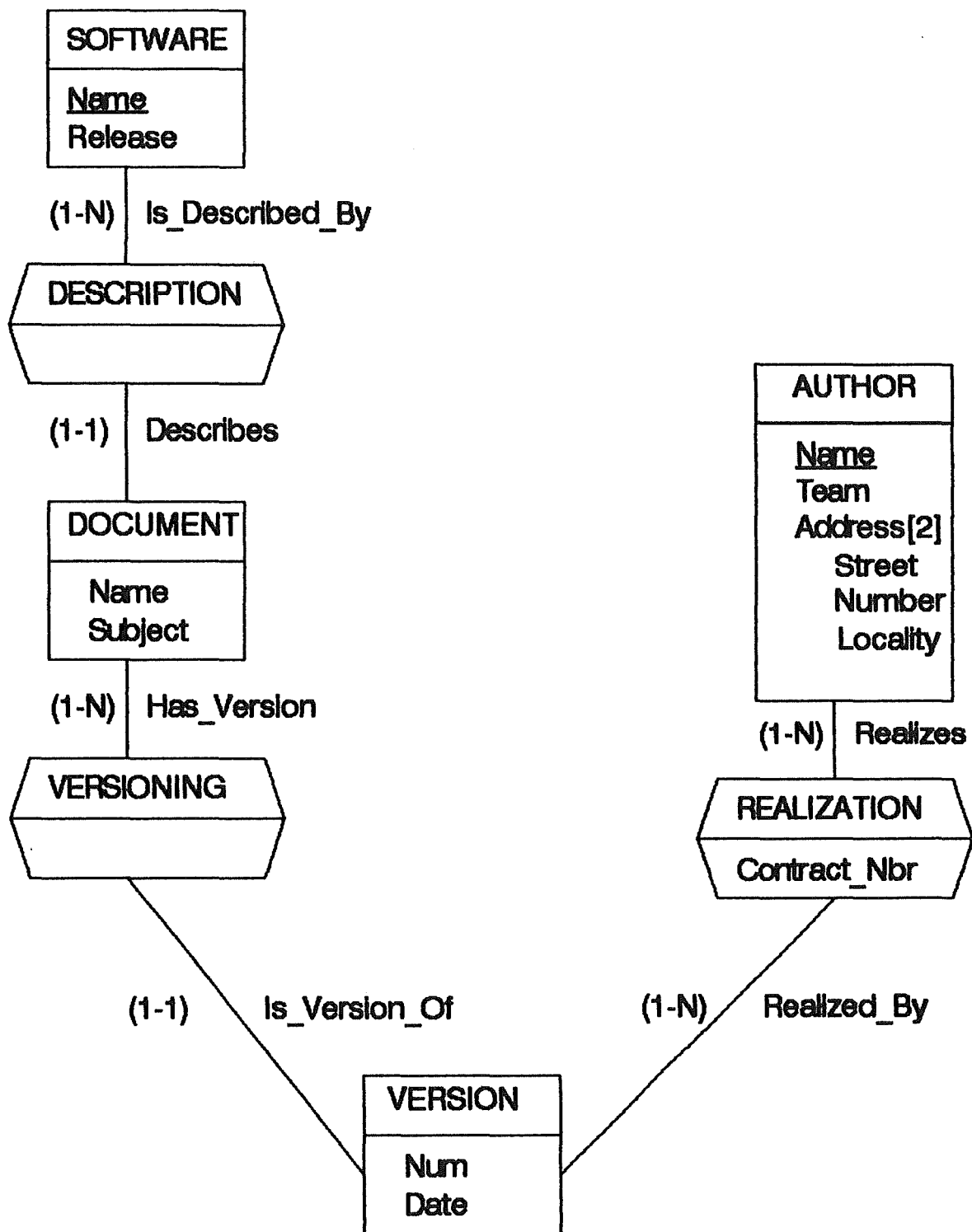
Figure C.1 : The E-R schema of the Soft_Env database

simple, elementary, (mandatory) entity attribute can be an identifier for an entity type.

Entity attributes are represented by PASCAL data types. Each entity of the application domain is represented together with its attribute values by some sort of data stored in a file, that we shall call an **PYRAMIDE Entity**.

One must also notice that relationships are used to obtain entities that are logically interconnected. For instance an application program can ask for all the 'Document' entities that are connected to a given 'Software' entity through 'Description' relationships. Or also, to get the 'Software' entity related to a given 'Document' entity. That is, the program navigates through the database following the **paths** corresponding to the relationship types. A given relationship type is mapped onto two **path types**; a 1-N path type and its inverse N-1 path type. A path is always directed from its **origin** entity towards its **target** entities.

1.3 The physical model of the DBMS

We should give a short description of the physical components of a database. However, these concepts are not necessary to be known by the novice programmer as all that is needed to write a program is the database schema and the programming interface of the DBMS. A more detailed explanation of these components will be given later on.

The prime component of a database is the internal representation of entities. An entity is physically represented by a string of bytes that we will call an **entity record** or simply **record** for short. A record contains the attribute values of the entity together with some technical data needed by the DBMS. The complete description of a record will be given in the next section.

These records are stored in a database file divided into **pages** of 1024 bytes. Records of any entity type can be stored in any page provided a record do not span across several pages. A page is

identified by its number which range from 1 TO 65000. Therefore the DBMS can actually manage databases up to about 65 MegaBytes, which seems to be sufficent in the current state of the art in personal computers storage devices.

There are two storage schemes for records; they can either be stored in a **clustered** manner or **randomly**. With the cluster storage scheme, a new record is stored in the same page of the last record that has been accessed. This storage scheme is very efficient for sequential access, as records that are created in burst tend to be stored in contiguous pages. Another possibility is to store the records in a random manner. In that case a page range must be specified as it will be explained in the programming interface. With that storage scheme the page where to store a new record is chosen randomly within the page range, so that records are distributed as uniformly as possible, leaving space between them. By default, the records are stored in a cluster manner.

In both storage schemes, if the page where to store a new record is full, other pages are searched for space according to the particular storage scheme. A **free space index** is used in order to accelerate the search. If no sufficient space is found, new pages can be added to the database file. The free space index garanties that the closest page with at least the needed amount of free bytes will be loaded in the buffer in no more than two physical accesses.

The DBMS uses a **Buffer** to store the pages it reads from the database file. The buffer is an area in main memory that can contain several pages from which the records are read and written to. When a record is asked by an application program, the DBMS first searches the buffer in order to spare physical accesses. If it is not present, it is loaded from the file into the buffer. The bigger the buffer, the higher the probability to find a requested record in main memory and therefore, the lower the number of physical accesses. But a larger size requires more memory and more time to manage the buffer. The user can always fine tune the buffer size at runtime depending on his

needs. A enhanced LRU strategy is used to manage the buffer, and will be discussed later on.

PYRAMIDE offers also a build-in **index** structure to quickly access entities according to their identifier value. The index structure is implemented by B+trees, and allows much better performance than sequential access. It is contained in the database file so that it can't be lost. The existence of an index for a given entity type depends only on whether or not an identifier has been declared in the schema for this entity type, and is transparent to the user. There is no connection between the presence of an index and the particular storage scheme that is chosen for the entity type.

1.4 Schemata

The specification of the entity types, entity attributes and relationship types of a database is called its **schema.**

As in the PYRAMIDE data model, the relationship types are restricted to 1-N binary types, a PYRAMIDE database schema accepts a simple and intuitive graphical representation that is derived from the usual graphical representation used for E/R concepts [Hain86-a]. Each entity type is represented by a box containing its name and the names of its attributes (an identifier being underlined). A relationship type is represented by an arc labelled with its name and joining the boxes of the entity types. The 1-N direction is indicated by a small triangle sticked on the arc which mimics the way the entities are connected (the base of the triangle being on the N side).

2. Schemata mapping

The schemata are represented accordingly to a given data model. Therefore the system must insure the correct mapping between a schema expressed in the E/R model towards the PYRAMIDE logical model. And also from a schema expressed in the PYRAMIDE model towards a physical PYRAMIDE schema.

2.1 From E/R to the logical model

This mapping is accomplished by the upper layer and thus, will not be discussed herein in details. We shall instead refer the interested reader to the dissertation of Feraille patrick and Tomasi mathias realizing the E/R layer.

In short we can say that there are direct translation rules between the two models. For instance :

- a (N-N) binary relationship type would be decomposed into an intermediate entity type and two (1-N) binary relationship types;

- a N-ary relationship type will be represented by an entity type and N binary (N-1) relationship types from the new entity type to each of the member entity types;

- a relationship type having attributes can also be transformed as in the preceding example, each attribute being mapped onto an attribute of the new entity type.

Because the PYRAMIDE Data Manipulation Language (DML), that is the programming interface, is embedded in the programming language 'PASCAL', the data structures of the schema description have also to be mapped to corresponding 'PASCAL' data structures. These 'PASCAL' data structures are part of the programming interface and are generated by the MetaCompiler in the form of a 'PASCAL' include file.

Here follows an example of such a mapping between the two models (figure C.2) and of the resulting 'PASCAL' include file (figure C.3).

2.2 From the logical to the physical model

Each entity together with its attribute values will be represented at the user level by 'PASCAL' data types, and stored at the physical level in a language-independent structure on a mass storage device such as a magnetic disk. Relationships will be represented by a connection between entity representations. The DBMS needs therefore information on the physical structures on which the

SOFTWARE

Name
Release

Description

DOCUMENT

Name
Subject

Versioning

VERSION

Num
Date

AUTHOR

Name
Team
Address[2]
Street
Number
Locality

_Realized_By_

Realizes

Realization

Contract_Nbr
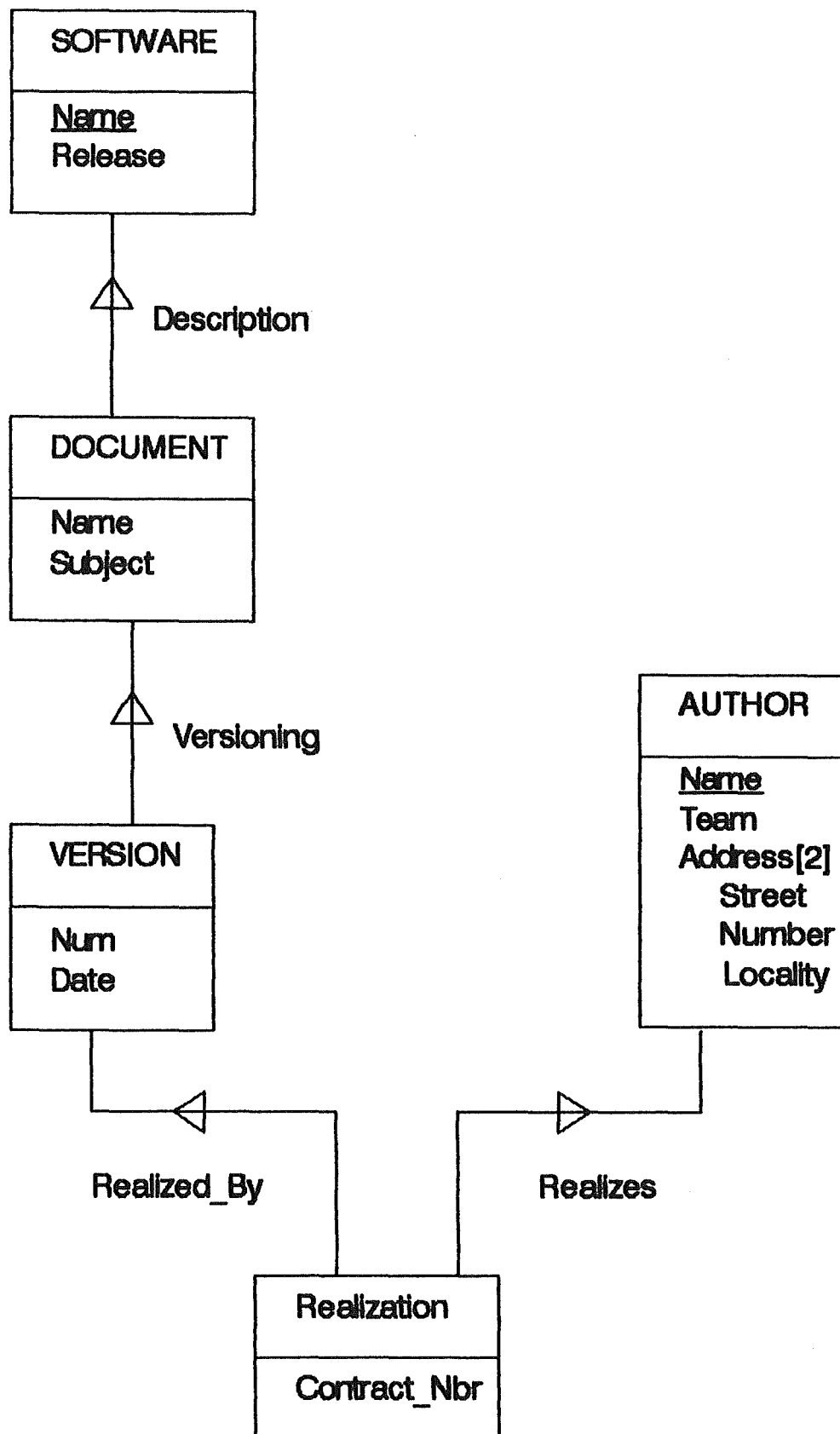
Figure C.2 : The logical schema of the Soft_Env database

```
CONST
     SOFTWARE    = ... ;
     DOCUMENT    = ... ;
     VERSION     = ... ;
     REALIZATION = ... ;
     AUTHOR      = ... ;

     DESCRIPTION = ... ;
     VERSIONING  = ... ;
     REALIZED_BY = ... ;
     REALIZES    = ... ;


TYPE
     TSOFTWARE = record
                     ...
                     NAME    : String[35];
                     RELEASE : String[4];
                 end;

     TDOCUMENT = record
                     ...
                     NAME    : String[35];
                     SUBJECT : String[160];
                 end;

     TVERSION  = record
                     ...
                     NUM  : integer;
                     DATE : Sting[6];
                 end;

     TREALIZATION = record
                      ...
                      CONTRACT_NBR : integer;
                    end;

     TAUTHOR   = record
                     ...
                     NAME : String[35];
                     TEAM : byte;
                     ADDRESS : array [1..2] of
                                    record
                                       street : string[30];
                                       number : integer;
                                       locality : string[15];
                                    end;
                 end;
```

Figure C.3 : The resulting PASCAL include File.

logical schema components are mapped. It needs for instance to known what strorage scheme has been chosen for each entity type and the corresponding page range. This mapping occurs when a PYRAMIDE database schema is compiled into what we shall call the schema internal tables of the DBMS.

Here follows an example of such a mapping (figure C.4).

3.   The Data Dictionary of PYRAMIDE

3.1 Functions

The first function of the PYRAMIDE Data Dictionary is to store E/R schemata. Thus, it allows the user to model the data in a fashion as close to reality as possible, so that the data and relationships do not need to be contrived to fit a given structure. The second function is to store the mappings between schemata of a database. The third function is to be an information resource for various processors and application programs.

3.2 Principles

As outlined before, one of the characteristic of this DBMS is that each database file incorporates its own description. In short, one can say that each database file includes a data dictionary (DD). The DD contains data describing the user schema. These data specifies for instance that there are entity types 'SOFTWARE' and 'DOCUMENT', that relationship type 'DESCRIPTION' is defined between them, and that 'NAME' is an attribute of 'DOCUMENT'. Since these data are about user data, they are called Meta Data. They are organized according to a specific schema called the Meta Schema. The Meta Schema includes such entity types as 'ENTITY_TYPE', 'REL_TYPE' and 'ATTRIBUTE'. It is defined according to the PYRAMIDE logical model (see figure C.5).

The Meta Schema is part of the schema of any database. A database schema includes the objects of the Meta Schema **plus** the

ENTITY TYPE

    Software
        Storage scheme : random
        Page range   : 1-50
        Index on   : Name

    Document
        Storage scheme : cluster

    Version
        Storage scheme : cluster

    Realization
        Storage scheme : cluster

    Author
        Storage scheme : random
        Page range   : 51-100
        Index on   : Name

RELATIONSHIP

    Description
        From : Software
        To   : Document

    Versioning
        From : Document
        To   : Version

    Realized By
        From : Version
        To   : Realization

    Realizes
        From : Author
        To   : Realization

Figure C.4 : An example of a possible physical mapping
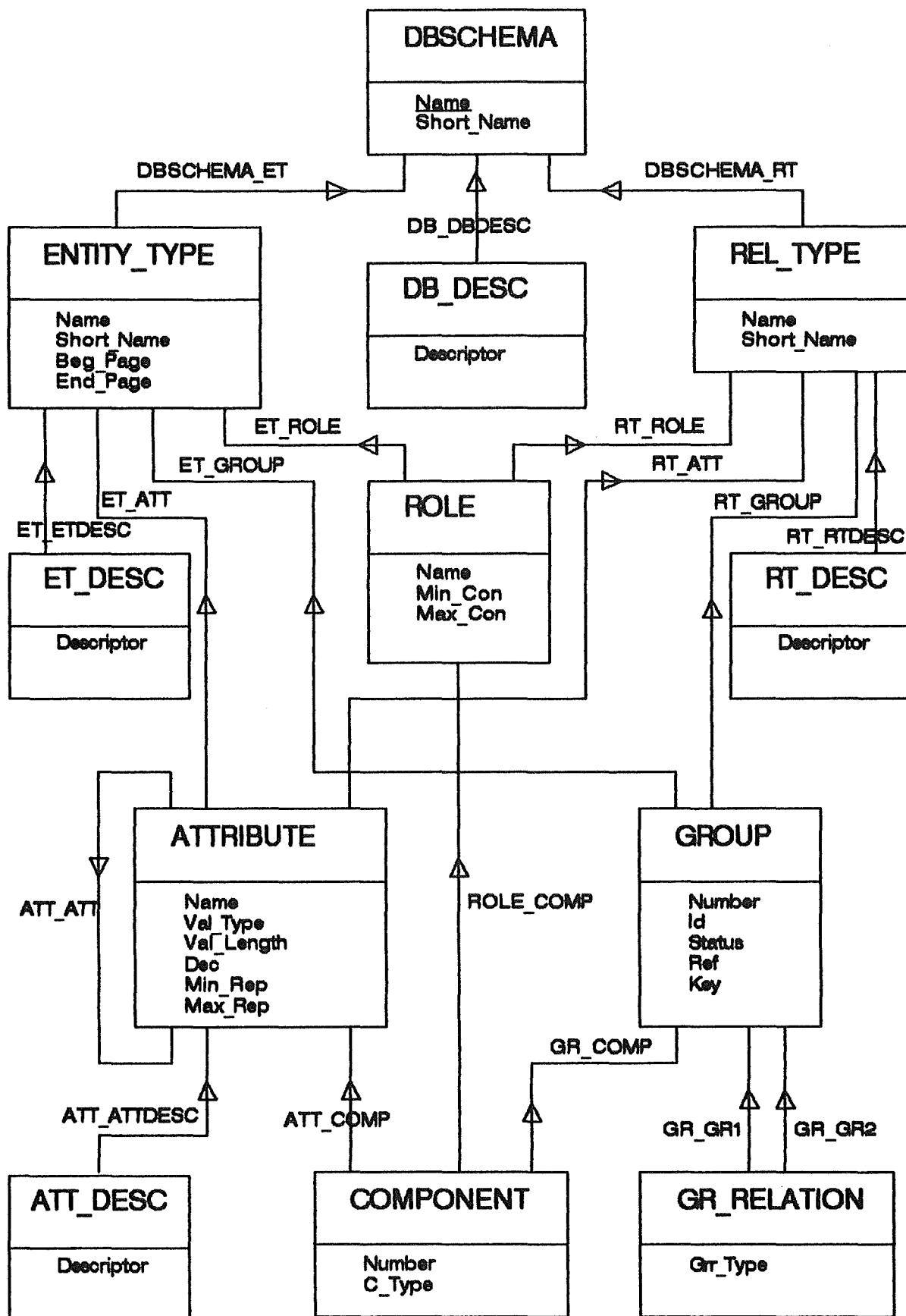
Figure C.5 : The Meta Schema of the data dictionary.

objects of the user schema. A database includes Meta Data **plus** user data.

They are no technical difference between Meta Data and user data except that they are of different entity types. Meta Data can be retrieved by an application program using the standard programming interface. However, updating these data means updating the user schema. That operation must be carried out with much care, and will be allowed, in the current version of the DBMS, when defining a new user schema only (i.e. no dynamic schema restructuring for now).

The Meta Data are essential for the operations of the programming interface. Checking the correctness of an operation and translating a logical operation into a physical operation requires the knowledge of the logical and the physical schemata and of the mapping between both. However, the way the meta data are stored is not efficient enough (accessing the data dictionary for each elementary operation would slow down operations considerably) for driving user data manipulation. Therefore, an internal version of the Meta Data is also stored in the database in a highly packed format called the schema internal tables. These internal tables are loaded in main memory when opening a database, so that no physical access will be needed at runtime. In principle, this internal version cannot be seen by application programs.

An interesting (but somewhat puzzling) feature of the Meta Schema is that it is self describing. A Meta Schema is a schema in its own right. It must therefore be described by Meta Data, that are in turn described by the Meta Schema ! Moreover, these Meta Data have a schema internal table version as well. Therefore, consulting the data dictionary gives, among others, information about its own schema.

To try to make things a bit clearer, we shall describe the various states of a database starting from an empty database to a database containing user data (figure C.6).

Meta data

Tables

Meta
schema
tables

Data describing
the meta
schema

'Standard.dtb'

Meta data

Tables

Meta
schema
tables

Data describing
the meta
schema

Data describing

the Soft_Env

E/R schema

'Soft_Env.dtb' with an E/R schema

Meta Data

Tables

Data

Meta
schema
tables

Data
describing
the meta
schema

Data
describing

the Soft_Env

E/R schema

Data
describing

the Soft_Env

logical

schema

'Soft_Env.dtb' with a logical schema

Figure C.6 : the various states of a database

3.2.1 The "empty" database

An empty database contains no user schema, and therefore no user data. Its only contents are :
- a minimal data dictionary;
- the schema internal tables of the data dictionary;

The schema of the empty database is the Meta Schema and its only data are the Meta Data describing the meta schema. Therefore, the only significant operation is consulting the Meta Data of the Meta Schema.

The 'standard.dtb' database file is the origin prototype of any empty database. Building the 'Soft_Env.dtb' database starts with copying the 'standard.dtb' into 'Soft_Env.dtb'.

3.2.2 The database containing a E/R schema

As the "empty" database already contains the schema internal tables of the data dictionary, the progammer can call the DBMS primitives to work on the Meta Schema. He can for instance, describe and load the 'Soft_Env' E/R user schema into the 'Soft_Env.dtb' database. For this, he can use the Meta Data PASCAL types contained in the file 'Standard.typ'. A compiler for an E/R Data Definition Language (DDL) should be realized in the upper layer by Feraille Patrick and Tomasi Matthias that would make this work easier.

Now the database contains a E/R user schema in addition to the Meta Schema. Its data are the Meta Data describing the Meta Schema plus the Meta Data describing the E/R user schema. Still, it does not contain any internal tables for the E/R user schema. Therefore, the only significant operation added is consulting and/or modifying the Meta Data of the E/R user schema.

3.2.3 The database containing the logical schema

This step is achieved via a schema processor that is to be realized within the upper layer. It will realize the mapping between the E/R and the logical user schemata. This pre-processing solution has the advantage of freeing the DBMS of some problems such as consistency and conformity control of the logical user schema, enabling us to keep a very performing and compact physical engine.

A logical user schema can also be described using a DDL and its compiler, or if it is not available yet, it can even be loaded in the database by a application program working on the Meta Data types, as for the loading of the E/R user schema.

Now the database contains a logical user schema in addition to the E/R user schema and the Meta Schema. Its data are the Meta Data describing the Meta Schema plus the Meta Data describing the E/R user schema and the Meta Data describing the logical user schema. Still, it does not contain any internal tables for the logical user schema. Therefore, the only significant operation added is consulting and/or modifying the Meta Data of the logical user schema.

3.2.4 The initialized database

To make the schema description known to the PYRAMIDE-DBMS, it has to be compiled by the MetaCompiler, which analyzes the logical user schema and stores an internal representation in the Data Dictionary. The internal tables are updated, so that the DBMS will be able to work not only on the Meta Data, but also on the new database schema. Besides, the MetaCompiler will also create a 'PASCAL' include file named 'Soft_Env.typ' containing the data types needed to program on the database schema.

Now, the database contains, besides the Meta Data, the internal tables for the database schema. Therefore, the user can program on the logical user schema by using the PASCAL data types generated by the MetaCompiler in the include file.
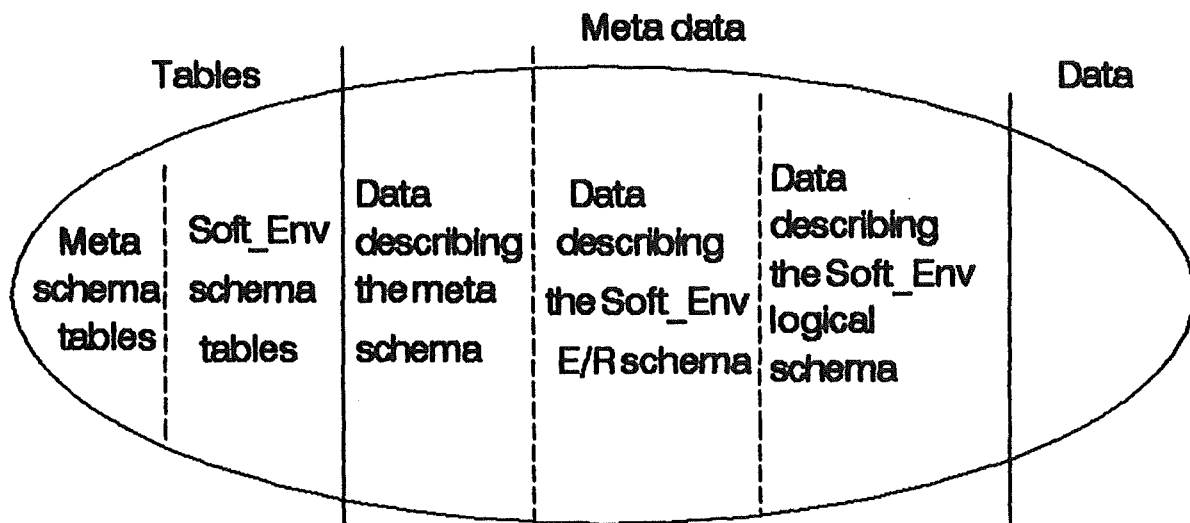
The initialized 'Soft_Env.dtb'
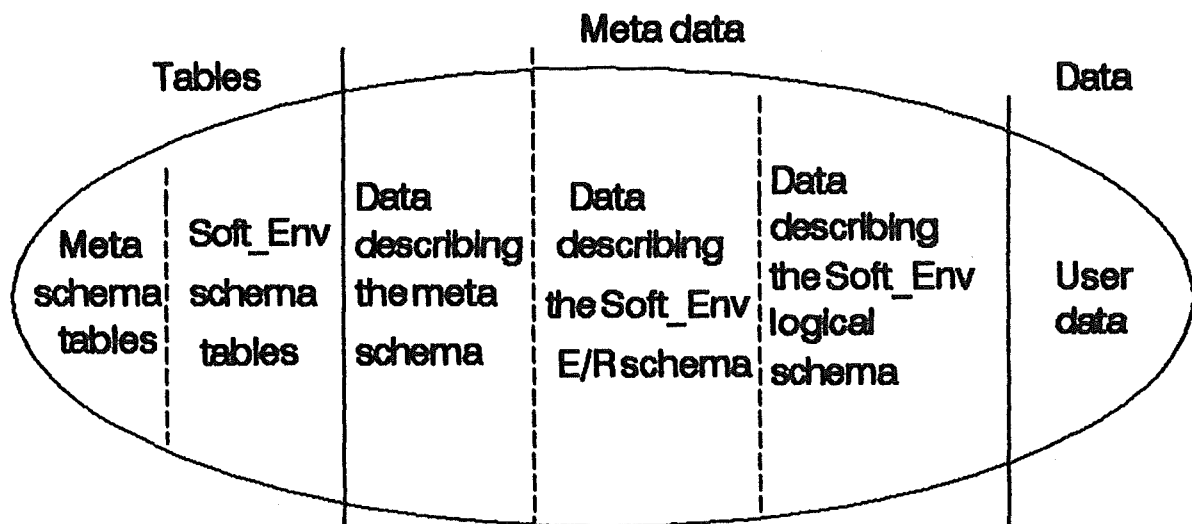
'Soft_Env.dtb' with user's data

Figure C.7 : the various states of a database (continuation)

3.2.5 The database with user's data

From now on, the user can create, update and retrieve data on basis of the database schema he has described. The database contains user data in addition to the Meta Data describing the various schemata.

Please note that the Meta Data describing the Meta Schema is present for completeness and documentation purposes. It should only be accessed in read mode. Updating this Meta Schema would have no harmful consequences as neither the DBMS, nor the MetaCompiler do take these modifications into account, but the documentation purpose won't be accurate anymore.

Besides documentation purposes, the data dictionary functionality is needed by the schemata compiling solution we have chosen to implement, and later, for the dynamic extension of a database schema. This aspect will be developped in an other section of this document.

3.3 The Meta Schema of the DD

A schema is described in the data dictionary in terms of entity types, entity attributes and relationship types. As shown in the Meta Schema (figure C.5), a **db schema** is given a **name** (optionally a **short name**), which must be a valid file name (*without extension*) for the Operating-Sytem (OS) used with the computer. Generally a 1 to 8 'normal' characters string will do.

Each **entity type** has a **name** (optionally a **short name**) which is a valid 'PASCAL' name, and which is *unique* within a given schema. The entities can also be stored in a 'cluster' manner or randomly within a range delimited by its **beginning** and **ending page.**

An **attribute** has a **name** which is a valid 'PASCAL' name. The attributes of a given entity have different names, but attributes of different entity types may have the same name. An attribute has a

value type which is a valid 'Pascal' data type such as Integer, Char, Byte, bOolean, Real, String and 'Group' as an attribute can be a composed as a Pascal record. In the case of the string type, the value length gives the maximum length of the string. In the case of composed attribute, the decomposition flag is set to true. The minimum and maximum repetitiveness must also be given. An entity type need not have attributes, but if it has at least one attribute, one of them can be the entity identifier. In that case it is the only component of a group of which the identifier flag is set to true.

A relationship type has a name (optionally a short name) which is a valid 'PASCAL' name, and which is *unique* within a given schema. It is defined between two entity types who need not be distinct. That is one can easily describe so-called 'recursive' entity types (e.g. the classical 'HAS_SON' relationship type on the recursive 'MAN' entity type). Each entity type being part of a relationship type play a specific role, with a name, and its minimum and maximum connectivity.

The other components of the Meta Schema are not used by the current PYRAMIDE version, but are uselful to other processors using for instance the data dictionary as an information resource.

We can see that the Meta Schema of the data dictionary has been conceived to allow the storage of schemata belonging to the different data models.

In our example, the E/R user schema could be represented by creating an entity of type DBSCHEMA with the identifier NAME 'Soft_E/R', and connecting it to entities of type ENTITY_TYPE with the NAME 'Version' and 'Author'. A ATTRIBUTE entity of NAME 'Name', VAL_TYPE 'String', VAL_LENGTH '80', DEC 'False', MIN_REP and MAX_REP equal to 'one' would be connected to the 'Author' ENTITY_TYPE entity. A REL_TYPE entity of NAME 'Realization' would be connected to the 'Soft_E/R' DBSCHEMA entity. It would also be connected to ROLE entities of NAME 'Realizes' and 'Realized_By'. The 'Realizes' ROLE entity with a MIN_CON of one and a MAX_CON of N, would be connected to the 'Author' entity. The 'Realized_By' ROLE entity with a MIN_CON of

one and a MAX_CON of N, would be connected to the 'Version' entity, and so on ...

In the logical user schema, the N-N 'Realization' REL_TYPE entity would be transformed into a ENTITY_TYPE entity of the same NAME. Two REL_TYPE entities of NAME 'Realizes' and 'Realized_By' would be created. The 'Realizes' REL_TYPE entity would be connected to ROLE entities of NAME 'Target' and 'Origin'. The 'Origin' ROLE entity with a MIN_CON and MAX_CON of one would be connected to the 'Author' ENTITY_TYPE entity. The 'Target' ROLE entity with a MIN_CON of one and a MAX_CON of N would be connected to the 'Realization' ENTITY_TYPE entity, and so on ...

And finally, at the physical level, we would have information such as the page range and the storage scheme for the ENTITY_TYPE entities 'Version' and 'Author'.

4. Architectural principles

As a first step towards understanding engineering-DBMSs, we thought a good prime objective was the development of a DBMS based on the Entity-Relationship model. This goal was reached by developing the system into two layers : a physical layer which implements a "low level" DBMS, and an upper layer which offers functionalities based on the E/R model. The subject of this thesis is the design and realization of the DBMS supporting the E/R layer : the PYRAMIDE - DBMS.

It seemed to us that a DBMS offering a restriction of the E/R model as a data model, was a good basis to realize this physical engine, as there are direct translation rules between the two models.

That is, the main architecture is made up of two layers. At the lower level, we have Entities and (1 to N) binary Relationships without attributes. The upper level being in charge of translating true E/R primitives into primitives of the physical motor. For

instance, a pre-processor could translate an access loop on the
Versions written by an Author, into two access loops : on the
Realizations of the Author, and then on the Version of the current
Realization.

4.1 Basic functions and processors

4.1.1 Loading of an E/R schema

The first function that the system must offer is the possibility
for a programmer to load and store a true E/R schema. This can be
done either by a specific application program working on the Meta Data
types of the data dictionary, or by the DDL-processor of the upper
layer (which is in fact also an application program running on top of
PYRAMIDE). That is, the programming interface can work on ordinary
data, as well as on Meta Data (depending on the (Meta) Data types that
are used).

4.1.2 Loading or production of a logical schema

The second function is to load and store a logical schema of the
database in the data dictionary. This can also be done by an
application program working on a subset of the Meta Data types. But
the best solution is to traduce the E/R schema to a logical schema, by
using the schemata-processor to be offered by the upper layer. Thus
the schema can be checked and the result is guaranteed to be a conform
PYRAMIDE logical schema.

4.1.3 Initialization of the database

When a logical schema has been stored in the data dictionary, it
must be compiled into a physical schema that will be stored in the
internal tables, before the DBMS can work on this schema. This is
accomplished by the PYRAMIDE MetaCompiler that generates the new
database file and the data types needed to program on the new schema.

4.1.4 Accessing the database data

   In order to fulfill all the preceding functions and the basic one
that is to access user data contained in the database, the user is
provided with a programming interface. The interface comprises
'PASCAL' data types (generated by the MetaCompiler on basis of a
logical schema) and a set of 'PASCAL' procedures and functions to
operate on the data of the database.

4.2 Secondary processors

   A second objective, was that the programming interface should be
kept as close as possible to the **N.D.B.S.** (a Network Data Base System)
interface [Ross87] and [Hain87]. That is, the numerous existing
NDBS-programs and tools should be able to run on top of PYRAMIDE at
the lowest possible adaptation cost. The database compatibility at
the physical level between the two DBMSs was however not possible to
keep, as PYRAMIDE offers much more powerful features than NDBS does
(i.e. the data dictionary).

   Such tools already existing for NDBS and that could be easily
adapted to PYRAMIDE are a schema loader, a data input processor and a
report writer. A fourth generation language (4GL) is also currently
under development. That is, someone could work with PYRAMIDE without
having to program anything. The user would load his schema with the
schema loader, and compile it. Then he would use the data input
processor to enter data in the database (this processor asks the user
to input data on basis of tree-like structures of the schema). He
would then manipulate data with the 4GL. And finally, he would make a
report of the contents of the database with the help of the report
writer. One can notice that all these processors are in fact
application programs built on top of the PYRAMIDE interface, some of
them accessing the data dictionary contained in a database.

   Therefore, some concepts implemented in PYRAMIDE are inspired by
NDBS (which I programmed some years ago), where they have proven to be
effective. Nevertheless, PYRAMIDE is not an extension of NDBS. The

architecture has been designed to allow easy maintenance and further extension. Features like Indexed Sequential Access Method (**ISAM**) are supplied. New organization techniques at the file level have also been developed to support the present and forthcoming features like dynamic extension of the DB schema at runtime.

In summary, we can say that we have tried to build a new DBMS that could efficiently support the E-R layer, while keeping as much compatibility with NDBS as possible. That is, PYRAMIDE can be seen at one and the same time as a stand alone DBMS, and a part of a much larger Entity-Relationship database system.

Section D :

The Programming interface

1.   General Presentation

1.1 The basic choices and their motives

PYRAMIDE is, as explained before, a "two-goals" project. On one hand, it must play the role of the lower layer of a much larger Entity-Relationships system. But on the other hand, it must also be the true successor of NDBS, in order not to loose the compatibility with programs already existing or still under development.

Therefore, simplicity, low-cost and efficiency in the context of microcomputers are the qualities that the PYRAMIDE-DBMS has kept from its NDBS predecessor. But besides these, the new DBMS offers features necessary to the upper layer which can also benefit the PYRAMIDE programmer.

Simplicity, as the programming interface is embedded in the PASCAL language. Therefore, it is very easy for a pascal programmer to quickly write programs to handle databases of any complexity. This aspect is also very important in engineering applications where the DBMS must play the role of the common integrating interface for the different tools.

Low-cost, particularily in terms of disks accesses, processing time and main memory requirement. As shown in the requirements analysis, this is of prime importance in order not to slow down the already fairly time consuming CAD/CAM tools. Special care has also been given in order to favour the read accesses, where larger delays seem to be tolerated by designers while writing to secondary storage.

Efficiency, as all the needs of the upper layer have to be fullfilled by PYRAMIDE, while it must be able to run within the limitations of Personnal Computers currently available on the market.

1.2 Description of the PYRAMIDE environment

PYRAMIDE is a so-called Network-DBMS and its data model is a subset of the Entity-Relationship (E-R) model. Thus it can easily be understood by non technical users, as a clear distinction between the semantic structure of the data and its physical representation is established.

Its programming interface deriving directly from NDBS, has been kept simple. Thus complex programs can easily be written using only a subset of the database functions PYRAMIDE offers. The absence of implicit arguments (such as currency indicators in Codasyl) or side effects of the database functions ensure a total visibility of the programming objects concerned by database operations. What is more, the programming interface is directly traced from the PASCAL syntax, totally complying with its data types and programming rules. Thus making easy to solve even recursive problems generally considered as difficult to program in existing DBMSs.

Therefore, even if PYRAMIDE is an educational DBMS, the features it offers are close to commercial DBMSs. A multi-layers architecture implemented as a "unit" in TURBO-PASCAL (Version 5) allows easy maintenance and further extension. "All-in-one" self-describing data file, compact variable-length data storage, bi-directional record chaining, Indexed Sequential Access Method (ISAM) implemented by B+Tree-like indexes, enhanced Least Recently Used (LRU) buffer management, parametrised storage schemes (clustered or random), transaction and recovery management, embedded data dictionary, embryon of dynamic schema management, possibility to work with several databases at the same time, are some of the characteristics that make PYRAMIDE a very competitive DBMS for handling complex and large databases on microcomputers. Moreover, features like multi-user support, a true dynamic schema management, and many other enhancements could be developed in the future that would make together with the compatible NDBS tools currently available or under development, a very complete stand alone database system environment. Besides the necessary and existing schema compiler, such tools coming soon are for

instance a Fourth Generation Language (4GL), a data input processor, a report writer and a Data Definition Language (DDL) with its compiler that will make easier the work of the PASCAL-PYRAMIDE programmer.

The current "prereleased" version (0.00) developed within the framework of this dissertation includes the two essential components; that is, a database handler and a schema compiler. The database handler comprises the set of database functions offered to the programmer to handle efficiently its data. And the schema compiler is a program using the PYRAMIDE-DBMS to read the description of a database (its schema which is included in the database itself), and produce as an output the database file (ready to be processed) and the PASCAL types needed to program. The logical schema description of the database can be loaded using a program (in PYRAMIDE-PASCAL) or even better, the DDL if it is available.

2. The schema compiler

The MetaCompiler is an application program build on top of the PYRAMIDE-DBMS. It automatically generates from the Database schema description, a database file processable by the DBMS and the 'PASCAL' data types needed to program.

2.1 The general mechanism of creating a database

It is not our aim to give any guidelines for the design of a database. The reader needing information on how to build his database schema and the programs to manipulate data, will find a thorough discussion of this subject in [Boda83] and [Hain86-a]. We shall therefore stick only to functional, architectural and technical problems involved by the PYRAMIDE-DBMS itself and describe the basic mechanism to create a database.

Before the user can program on a database, he must create the database file itself. He first has to copy the standard database file named 'standard.dtb' to the file that will contain his database (e.g.

'Soft_Env.dtb'). Then he has to load the database schema description in this file. This can be done by using a program that manipulates the 'Soft_Env' database file. As a matter of fact, the initial 'Soft_Env' database is not empty; its **data dictionary** contains the Meta Schema description which has already been compiled into the internal tables needed to program on this schema. A **'standard.typ'** file containing the 'PASCAL' data types corresponding to this Meta Schema is also available. An example of such a program is given in the case studies.

Then the user can run the **MetaCompiler** program which will read the schema description of the Soft_Env database contained in its data dictionary. That is, a database contains its own description (self describing file). Then it will modifie the 'Soft_Env.dtb' database file and create a 'Soft_Env.typ' file containing the 'PASCAL' data types needed to program. From now on, the database will be ready to be processed.

2.2   A Data Definition Language

Another much easier way for loading the schema description would be to use a Data Definition Language (**DDL**) and its compiler. Such a helpful tool should be provided for PYRAMIDE shortly. As it is not part of this dissertation we shall refer the interested reader to Professor Hainaut for further information. Nevertheless, the principles of such a tool will be outlined herein.

The task of the user is made easier as he only has to describe the database schema in the DDL. That is, he must observe strictly the rules defining the language, but doesn't have to program anything. Let's assume that a text file containing the DDL description of the schema exists. The DDL-compiler will read this description and load consequently the data dictionary of the database file. When the compilation is done, the description of the database schema is inserted in the data dictionary of the database. The user only has then to run the Metacompiler to get all what he needs.

One of the advantages of this way of working is that the analyzer first checks whether the schema description is syntactically and semantically correct according to the language rules, before loading it. If the schema contains an error, the loading process is aborted and the error is signaled. The DDL-compiler has then to be called again after correction of the schema description.

2.3 The compilation of a database schema

When a schema description is loaded in the data dictionary of the database, the Metacompiler must be called to process it. For the time being, the Metacompiler interface is quite simple. The user only has to give the name of the database file and the name of the schema within this database to be processed. The validity of these names is checked and in case the Metacompiler does not find either the database or the DB schema, it writes an error message and asks for a new name.

Then the database schema is read and if it is syntactically correct, the internal tables and other information needed by the DBMS are generated in a new '.DTB' database file and a new '.TYP' file containing the data structures corresponding to the schema is also created. The old '.DTB' and '.TYP' files are respectively renamed '.ODB' and '.OTP'.

In case of a schema error, a message is displayed and the compilation process is aborted. The database schema description will have to be corrected before calling the Metacompiler again.

2.4 Dynamic updating of a database schema

This problem has also been taken into account during the realization of this DBMS. We think the dynamic updating of a database schema is an interesting functionnality for an engineering DBMS. But in order to be able to offer this possibility at the upper level, this lower layer has got to support this concept also.

The problem is to enable the user to create, delete or modify data structures at any moment during his work session. As this is not known during the compilation of the database schema, it has to be performed dynamically. This is a very difficult problem to solve as all preceding data have to be modified in some cases.

As a first step towards a true dynamic schema management, we have included the definitions of the Meta schema and the database schema into the database file itself. That is, each and every database file has got a kernel which is also a true Data Dictionary containing its own database schema. In other words, each database contains a Meta Database which is a data dictionary. Therefore, the Metacompiler accesses the database during the database schema compilation (as the DB contains the data dictionary). And an application program can also access it, during execution, in order to read and/or modify the schema describing the data structures.

A modification of the data structures of a database is performed by updating its database schema. These changes can be performed by the use of the programming interface operating on the special data structures that are described in the Meta Schema. The programming is kept very simple as there are no special functions but the basic procedures of the DBMS. A more general example of updating a database schema is to create a new one; and a program doing this is given in the case studies. The reader will also find there, a program that lists the schema description of a database.

Special care must be given when working on the Meta Schema as the user can completely corrupt the database. No protection is provided as the main objective was to keep the programming interface as straight as possible. It is only the objects the procedures are working on that change. Therefore, preventing the user to do some things depending on the object he his working on, would have been a blow to the simplicity of the programming interface. We believe the advanced programmer dealing with dynamic updating of the schema or other very special purpose functionalities should be aware of what he is doing. The special types needed to program on the Meta schema are

contained in the 'STANDARD.TYP' file. Therefore, this file must be included in the program that is intended to update the schema. If it is not present, there is no danger to corrupt the schema in any way.

As this is an advanced functionality, a thorough explanation of the mechanism used in the current version of PYRAMIDE as en embryon of dynamic schema management will be given in the physical description part of the DBMS. We can say for now that after a database schema has been updated, it must be recompiled before the user can work on the new data structures. This implies for the time being, calling again the MetaCompiler which will generate the new files. The preexisting data are still contained in the old version of the database file, and thus can be retrieved from this file, processed according to the updating of the schema, and finally re-inserted in the new database file.

We are aware this is not a true dynamic process. But it is a good first step towards it, as all what is necessary to implement it, is physically present in the file structure.

3.   The programming interface

Besides the descriptive objective of a database that has been discussed, another purpose of a database is also to be an efficient and reliable data server for a large class of needs. This objective is fullfilled by the Data Manipulation Language which will be described herein in details.

The PYRAMIDE-DML comprises a set of generic operations that may be used to create, retrieve and update data stored in PYRAMIDE Databases. To enable tools to utilize the DML operations, the DML has to be embedded in the programming language in which those application programs are implemented. Since the PYRAMIDE-DBMS is intended to run primarily on microcomputers, and bearing in mind the main objectives exposed previously, it naturally provides for an embedding in the PASCAL programming language. The PASCAL programming interface is made up of generic PASCAL procedures contained in a TURBO-PASCAL 'UNIT',

and of a '.TYP' include file containing the result of mapping the
PYRAMIDE data structures to corresponding PASCAL data structures.
This mapping is schema-dependent and is performed by the PYRAMIDE
MetaCompiler.

N.B. : A Turbo-pascal "unit" is in fact a way of implementing a
module. It provides a set of capabilities through procedures and
functions, with supporting constants, data types and variables; but it
hides how those capabilities are actually implemented by separating
the unit into a (public) interface and a (private) implementation
section. When a program uses a unit, all the unit's declarations
become available, as if they had been defined within the program
itself. That is, the procedures and functions visible to any program
using the unit are declared in the interface, while their actual
bodies are found in the implementation part. The unit interface can
also contain constants, data types and variables that are to be known
by the using programs. The implementation part can have additional
declarations of its own, although these are not visible to any program
using the unit.

3.1 Data types and variables

After the application programmer has defined its database schema
and has compiled it, he can begin working on the data itself. For
this purpose he has at his disposal the set of operators of the
PYRAMIDE Data Manipulation Langage (DML). These operators are
themselves mapped to PASCAL procedures that constitute the PYRAMIDE
programming interface. These procedures allow him to open many
databases at the same time and close them, to access entities of a
given type sequentially or directly on basis of an identifier value,
to access sequentially entities linked to another via a path, to
update entities (create, modify and delete) and paths (insert and
remove), and to manipulate variables.

These procedures are available by inserting a 'USES PYRAMIDE'
statement at the beginning of the PASCAL application program. In

order to communicate with the DBMS, the application program needs also additional data types and variables. The application-specific parameters are mapped onto PASCAL data types and variables which are automatically generated in a '.TYP' file (e.g. 'Soft_Env.typ') by the MetaCompiler on basis of the database schema. This file must also be included in the application program by inserting for instance a '(*$I Soft_Env.typ *)' statement. From now on the programmer is provided with new data types and variables that allow him to work on entities, paths and to transmit attribute values.

As a necessary background, we must introduce the general concepts of the programming interface. These are that of reference, database reference, reference variable, entity variable and the schema component designators.

- **Database reference** : before using some procedures of the DBMS, the user must specify the database he wants to work with. This can be done at the opening of the DB or with a selection procedure if he is working with several opened DB. Each database that is opened is referenced by some kind of pointer that is called the database reference.

- **Reference** : while navigating through the database, the application program must be able to fix particular entities to proceed to other entities. For this, PYRAMIDE provides the concept of Reference. A Reference is a data type (**DbRef**) of which a value can designate an entity. A special **NULL** value says that no entity is referenced.

- A **Reference Variable** is a PASCAL variable of DbRef type. Such a variable can either designate an entity of any type, or be NULL or undefined (before it is assigned a value). In fact, references can be seen as so-called **surrogate keys** as each entity stored in a PYRAMIDE database is identified by a system-wide unique reference generated by the system itself. The reference remains unchanged during the lifetime of the referred entity, and a reference of an entity that has been deleted will never again be assigned to another entity. An

application program has the possibility to store a reference into a reference variable in its own address space, and to use it to refer the entity. But special care must be given when handling these variables, and most of the time they should be manipulated only by the DBMS's procedures. In particular, the programmer should never modify the value of such a variable or pass an undefined variable as an input argument to the DBMS, or unpredictable results could occur.

- An **entity variable** is a PASCAL variable of one of the entity variable types automatically generated by the MetaCompiler in the '.TYP' file. These types are prefixed with the letter 'T', as for instance **'TSOFTWARE'** for 'Software' entity variables. Such an entity variable can contain the attribute values for entities of the concerned type and no other. Besides this, it also has some components needed by the DBMS (such as the reference of the current entity), that should only be manipulated by the DBMS's procedures.

Designators for the schema components.

First of all, one must designate the **database**. This is done by using its name, possibly prefixed by a path, but without any extension. Each **entity type** is designated by a given numeric code automatically generated by the MetaCompiler in the '.TYP' file. The code is a PASCAL integer constant with the name of the entity type as for instance, 'SOFTWARE = 13'. Each **relationship type** is designated by a given numeric code automatically generated by the MetaCompiler in the '.TYP' file. The code is a PASCAL integer constant with the name of the relationship type as for instance, 'DESCRIPTION' = 19'. This code also designates the 1-N paths related to that relationship type, and the inverse N-1 paths can be designated by negating that code. For instance, '- DESCRIPTION' designates the paths from Documents to Software entities.

One must note that the application programmer is not aware of the actual values of these constants that are automatically generated by the MetaCompiler. Instead, the programmer will use the names of these constants that he knows from the schema definition.

3.2 Procedure arguments

Most of the arguments are common to several procedures and will therefore be described herein.

DataBase : is a string or constant expression containing from 1 to 64 characters giving the name of a database, possibly including a path, but without any extension.
   Examples : 'SOFT_ENV' or 'C:\data\pyramide\PART' are valid.
              But 'SOFT_ENV.DB' is NOT valid.

DbDesc : is a variable of the DDB type. That is, a pointer towards a Database Descriptor Bloc. The DDB type is defined within the '.TYP' file generated by the MetaCompiler.
   Example : Var DB1, DB2 : DDB;

Entity Type : is any positive integer expression giving the numeric code of a valid entity type of the database. These codes are generated in the '.TYP' file in the form of a predefined integer constant named after the entity type it designates.
   Examples : SOFTWARE, DOCUMENT, VERSION.

Path Type : is any integer expression giving (in its absolute value) the numeric code of a valid relationship type of the database. These codes are generated in the '.TYP' file in the form of a predefined integer constant named after the relationship type it designates. If the value is positive, it designates the 1-N path type. If it is negative, it designates the N-1 inverse path type.
   Example : 'DESCRIPTION' designates the 1-N path type from SOFTWARE to DOCUMENT, and '- DESCRIPTION' designates the N-1 inverse path type from DOCUMENT to SOFTWARE.

Ent Var : is any entity variable of the type Tentity_type_name.
   Examples : Var Pyramide, Ndbs : TSOFTWARE;
              Var ReqAnalysis, SourceCode : TDOCUMENT;

Origin : is any entity variable of the type Tentity_type_name designating an origin entity of a path.

Target : is any entity variable of the type Tentity_type_name designating a target entity of a path.

Ref_Var : is any reference variable of type DbRef.

R/E_Var : is either a Ref_Var or an Ent_Var.

3.3 The DbStatus return codes

After a procedure of the DBMS has been called by an application program, the integer global variable named **DBSTATUS** indicates how the operation has been carried out. Detailed explanation will be given for each and every DBMS's procedures, but as the returned code has generally a common meaning we list here the possible values and the corresponding meaning.

0 : the operation has been correctly carried out.
1 : the requested object (entity, database) has not been found.
2 : identifier uniqueness violation during an update operation.
10 : incorrect entity type code.
11 : incorrect relationship type code.
30 : incorrect reference value given as an input.
70 : out of main memory space.
80 : out of disk memory space.
90 : incorrect reference found in the database; the database is corrupted.
99 : I/O or system error.

As one can see, the seriousness of the incident increases with the return code value. One digit codes define normal conditions. Return codes 10 and 11 define a wrong designation of a schema component, probably from a syntactic error in the program. Return code 30 is more severe, and greater return codes are due to grave external accidents.

3.4 Describing the Kernel procedures

The PYRAMIDE Data Manipulation Language may be characterized as a "one entity/relationship at a time" interface. This means that in contrast to set-oriented interfaces such as SQL, every PYRAMIDE operation always return at most one entity to the calling program. Furthermore, PYRAMIDE is a database system of the **Network-DBMS** family. That is, the application program can access and update the data stored in a PYRAMIDE database by **navigating** from entity to entity via the relationships defined between entities.

We will describe each procedure separately, giving summary examples in the case studies. The programmer need not use all these procedures. Less than a dozen of them are sufficient to write easily simple database management programs.

PYRAMIDE also allows to work with several databases. Each database that has been opened at some point in time and not closed is called an **opened** database. Among the opened databases, there is a particular one that is the **active** database. Except the open and close procedures, all procedures of the interface implicitly operate on the active database. The last opened database is by default the active one. If the user wants to work with another database he can change the active database with a special selection procedure.

3.4.1 Diagnostic functions

As seen previously, by the end of a function or a procedure of the programming interface that has been called, a code is returned giving information on how the operation has been carried out. Some boolean diagnostic functions corresponding to these return codes are also offered to the programmer.

NAME

    <u>dbfound - found, DbStatus = 0</u>

INTERFACE

    dbfound : Boolean

DESCRIPTION

    Returns true if the last DBMS's procedure called returned dbstatus = 0; false otherwise.

NAME

    <u>dbnotfound - not found, DbStatus = 1</u>

INTERFACE

    dbnotfound : Boolean

DESCRIPTION

    Returns true if the last DBMS's procedure called returned dbstatus = 1; false otherwise.

NAME

    <u>dbnonunique - not unique, DbStatus = 2</u>

INTERFACE

    dbnonunique : Boolean

DESCRIPTION

    Returns true if the last DBMS's procedure called returned dbstatus = 2; false otherwise.

NAME

    <u>dbsevere - severe problem, DbStatus > 2</u>

INTERFACE

    dbsevere : Boolean

DESCRIPTION

Returns true if the last DBMS's procedure called returned dbstatus > 2; false otherwise.

NAME

dbpanic - very serious problem, DbStatus > 30

INTERFACE

dbpanic : Boolean

DESCRIPTION

Returns true if the last DBMS's procedure called returned dbstatus > 30; false otherwise.

3.4.2 Database Management

NAME

dbopen - database opening

INTERFACE

dbopen (DataBase, var DbDesc)

DESCRIPTION

This is the first procedure to be called before the user can work on the database. If the database named DataBase exists, opens it, makes it available for the program (it becomes the active database of the program), initialises the DDB (DbDesc points to the Database Descriptor Bloc), 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, DbDesc is undefined and dbstatus is different from 0.

RETURN CODES

    dbstatus = 0 : the database has been opened;

    dbstatus = 1 : the database has not been found;

    dbstatus = 70 : no sufficient main memory to create a new DDB;

    dbstatus = 99 : I/O or system error;

NAME

    dbclose - database closing

INTERFACE

    dbclose (var DbDesc)

DESCRIPTION

    The application program must call this procedure at the end of
its work in order to save all the update operations done till the
opening of the database. If several databases had been opened, this
procedure must be called for each of them. Closes the database
designated by DbDesc, if any; from now on there is no active database
for the program, and DbDesc is set to nil. Please, pay attention that
this procedure must be called before using DbDesc again; otherwise
unpredictible results could occur.

RETURN CODES

    dbstatus = 0 : the database has been closed;

    dbstatus = 1 : no database pointed by dbdesc has been found;

    dbstatus = 99 : I/O or system error;

EXAMPLE

    Var db1 : DDB;

        dbopen ('Soft_Env',db1);
            if not dbfound then goto err_open;

        . . .

        dbclose (db1);
            if dbnotfound then goto err_close;

3.4.3 Sequential and Direct access to entities

NAME

dbfirst - get first entity

INTERFACE

dbfirst (Entity_Type, var Ent_Var)

DESCRIPTION

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. If that entity doesn't exist, the content of Ent_Var is unchanged.

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;

NAME

dblast - get last entity

INTERFACE

dblast (Entity_Type, var Ent_Var)

DESCRIPTION

Finds the last entity of the type Entity_Type in the active database and stores its reference and its attribute values into the variable Ent_Var. If that entity doesn't exist, the content of Ent_Var is unchanged.

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;

NAME

    dbnext - get next entity

INTERFACE

    dbnext (Entity_Type, var Ent_Var)

DESCRIPTION

    If Ent_Var designates an entity of type Entity_Type of the active
database which is not the last one, the procedure finds the entity of
the type Entity_Type that follows the entity designated by Ent_Var in
the database. It then stores the reference and the attribute values
of the next entity into the variable Ent_Var. If Ent_Var was the last
entity, the content of Ent_Var is unchanged. If Ent_Var has a null
reference, dbnext acts as dbfirst (the successor of none is the first
one). The dbfirst procedure is therefore redundant.

RETURN CODES

    dbstatus = 0 : an entity has been found;

    dbstatus = 1 : entity not found;

    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;

NAME

dbprior - get prior entity

INTERFACE

dbprior (Entity_Type, var Ent_Var)

DESCRIPTION

If Ent_Var designates an entity of type Entity_Type of the active database which is not the first one, the procedure finds the entity of the type Entity_Type that precedes the entity designated by Ent_Var in the database. It then stores the reference and the attribute values of the next entity into the variable Ent_Var. If Ent_Var was the first entity, the content of Ent_Var is unchanged. If Ent_Var has a null reference, dbprior acts as dblast (the predecessor of none is the last one). The dblast procedure is therefore redundant.

RETURN CODES

dbstatus = 0 : an entity has been found;

dbstatus = 1 : entity not found;

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

Let's give an example of a program that opens the database named 'Soft_Env', writes to the screen the name of all software entities stored in that database, and then closes it.

```
Label exit, err_open, err_trt;

Type TSOFTWARE = record              (* generated by MetaComp *)
                    ...
                 name : string[35];
                 release : string[4];
              end;
```

```
    Var db1  : DDB;
        soft : TSOFTWARE;

    BEGIN
            dbopen ('Soft_Env',db1);            (* opening of the DB *)
                if not dbfound then goto err_open;

            dbfirst (Software, soft);    (* gets first entity *)
            while dbfound do
            begin
                writeln ('Software name = ',soft.name);
                dbnext (Software, soft)   (* gets next entity *)
            end;

                if dbsevere or dbpanic goto err_trt;

        goto exit;
        err_open : ... ;
        err_trt : ... ;

        exit : dbclose (db1)              (* closes the working DB *)
    END.
```

NAME

<u>dbid - get entity based on identifier value</u>

INTERFACE

dbid (Entity_Type, var Ent_Var)

DESCRIPTION

Finds in the active database the entity of the type Entity_Type that is identified by the identifier value previously stored in Ent_Var; stores its reference and its attribute value into the variable Ent_Var. If the entity has not been found, the content of Ent_Var is unchanged.

RETURN CODES

    dbstatus = 0 : the entity has been found;

    dbstatus = 1 : no entity has been found;

    dbstatus = 10 : Entity_Type has an incorrect value; maybe this
                      entity type hasn't got an identifier;

    dbstatus = 90 : corrupted database;

    dbstatus = 99 : I/O or system error;

EXAMPLE

Here follows a part of a program that gives the number of the team to which belongs an author identified by his name.

```
Type TAUTHOR = record
                      ...
                 name : string[35];
                 team : integer;
                      ...
               end;

Var designer : TAUTHOR;

           ...

       readln (designer.name);
       dbid (Author, designer);
       if dbfound
           then writeln ('Team number :', designer.team)
           else if dbnotfound
                   then writeln ('Designer unknown')
                   else goto err_trt;
```

NAME

    dbdirect - direct access to an entity

INTERFACE

    dbdirect (Entity_Type, var Ent_Var, var R/E_Var)

DESCRIPTION

Finds in the active database the entity of the type Entity_Type referenced by the entity variable or the reference variable Var; stores its reference and its attribute values into the variable Ent_Var. Ent_Var and Var don't need to be distinct. If the entity has not been found, the content of Ent_Var is unchanged.

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 : Var has an incorrect value;
dbstatus = 90 : corrupted database;
dbstatus = 99 : I/O or system error;
```

EXAMPLE

The following example accesses softwares of which the references have previously been stored in the array soft_lst, and writes their name onto the screen. NS gives the actual number of references stored in the array.

```
Const max = 100;

Var soft : TSOFTWARE;
    soft_lst : array[1..Max] of DbRef;
    NS, i : integer;

       ...

    for i : = 1 to NS do
    begin
      dbdirect (Software, soft, soft_lst[i]);
          if dbstatus <> 0 then goto err_trt;
      writeln (soft.name)
    end;
```

3.4.4 Sequential access in a path

NAME

dbfpath - get first entity in path

INTERFACE

dbfpath (var Target, var Origin, Path_Type)

DESCRIPTION

Finds the first entity connected to the entity Origin by the path Path_Type in the active database, and stores its reference and its attribute values into the variable Target. If the entity has not been found, the content of Target is unchanged. This procedure can be used for both 1-N and N-1 paths. If Path_Type is positive, the 1-N way is used, else it is the N-1 way.

RETURN CODES

dbstatus = 0 : the entity has been found;

dbstatus = 1 : no entity has been found; no entity is connected
                to Origin entity;

dbstatus = 11 : Path_Type has an incorrect value;

dbstatus = 30 : Origin has an incorrect value;

dbstatus = 90 : corrupted database;

dbstatus = 99 : I/O or system error;

EXAMPLE

Here follows an example of getting the software to which belongs a document (using the N-1 path).

```
Var soft : TSOFTWARE;
    doc  : TDOCUMENT;

    ...

    dbfpath (soft, doc, - Description);
    if dbfound
        then writeln ('software name = ',soft.name);
```

NAME

dblpath - get last entity in path

INTERFACE

dblpath (var Target, var Origin, Path_Type)

DESCRIPTION

Finds the last entity connected to the entity Origin by the path Path_Type in the active database, and stores its reference and its attribute values into the variable Target. If the entity has not been found, the content of Target is unchanged. This procedure can be used for both 1-N and N-1 paths. If Path_Type is positive, the 1-N way is used, else it is the N-1 way.

RETURN CODES

dbstatus = 0 : the entity has been found;

dbstatus = 1 : no entity has been found; no entity is connected to Origin entity;

dbstatus = 11 : Path_Type has an incorrect value;

dbstatus = 30 : Origin has an incorrect value;

dbstatus = 90 : corrupted database;

dbstatus = 99 : I/O or system error;

EXAMPLE

```
Var soft : TSOFTWARE;
    doc  : TDOCUMENT;

    ...

    dblpath (doc, soft, Description);
    if dbfound
        then writeln ('Last document is ',doc.name);
```

NAME

dbnpath - get next entity in path

INTERFACE

dbnpath (var Target, var Origin, Path_Type)

DESCRIPTION

If Target designates an entity of type Entity_Type of the active database which is not the last one in the path, the procedure finds the entity that follows the entity Target among those connected to the entity Origin by the path Path_Type in the database. Then, it stores its reference and its attribute values into the variable Target. If the entity has not been found, the content of Target is unchanged. If Target has an initial null reference, dbnpath acts as dbfpath (the successor of none is the first one). The dbfpath procedure is therefore redundant. If used for N-1 paths, it consistently returns DbStatus = 1.

RETURN CODES

dbstatus = 0 : the entity has been found;

dbstatus = 1 : no entity has been found; the Target entity was the last one;

dbstatus = 11 : Path_Type has an incorrect value;

dbstatus = 30 : Target or Origin have an incorrect value;

dbstatus = 90 : corrupted database;

dbstatus = 99 : I/O or system error;

EXAMPLE

Here is a part of a program that lists all documents connected to a software.

```
Var soft : TSOFTWARE;
    doc  : TDOCUMENT;

    ...
```

```
        dbfpath (doc, soft, Description);
        while dbfound do
        begin
          writeln ('document name = ', doc.name);
          dbnpath (doc, soft, Description)
        end;
```

NAME

dbppath - get prior entity in path

INTERFACE

dbppath (var Target, var Origin, Path_Type)

DESCRIPTION

If Target designates an entity of type Entity_Type of the active database which is not the first one in the path, the procedure finds the entity that precedes the entity Target among those connected to the entity Origin by the path Path_Type in the database. Then, it stores its reference and its attribute values into the variable Target. If the entity has not been found, the content of Target is unchanged. If Target has an initial null reference, dbppath acts as dblpath (the predecessor of none is the last one). The dblpath procedure is therefore redundant. If used for N-1 paths, it consistently returns DbStatus = 1.

RETURN CODES

```
    dbstatus = 0   : the entity has been found;
    dbstatus = 1   : no entity has been found; the Target entity was
                     the last one;
    dbstatus = 11  : Path_Type has an incorrect value;
    dbstatus = 30  : Target or Origin have an incorrect value;
    dbstatus = 90  : corrupted database;
    dbstatus = 99  : I/O or system error;
```

EXAMPLE

```
Var soft : TSOFTWARE;
    doc  : TDOCUMENT;


    ...

    writeln ('Here follows a list of documents beginning from
             the last to the first one');
    dblpath (doc, soft, Description);
    while dbfound do
    begin
      writeln ('document name = ', doc.name);
      dbppath (doc, soft, Description)
    end;
```


3.4.5 Updating entities

NAME

dbcreate - create entity

INTERFACE

dbcreate (Entity_Type, var Ent_Var)

DESCRIPTION

Creates and inserts in the database an entity of type Entity_Type. The attribute values are obtained from Ent_Var and the reference of the new entity is stored 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 is created;
dbstatus = 10 : Entity_Type has an incorrect value;
dbstatus = 90 : corrupted database
dbstatus = 99 : I/O or system error;
```

EXAMPLE

```
VAR soft : TSOFTWARE;

        . . .

    soft.name := 'TURBO_PASCAL';
    soft.release := '5.00';
    dbcreate (Software, soft);
    case dbstatus of
        0 : ;
        2 : goto err_id;
        else goto err_serious;
    end;
```

NAME

dbdelete - delete entity

INTERFACE

dbdelete (Entity_Type, var Ent_Var)

DESCRIPTION

Erases from the active database the entity designated by Ent_Var of type Entity_Type. The reference part of the 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. These entities can still be accessed sequentially or by other paths to which they participate.

We must also warn the user against the problem of so-called 'dangling references' when using several entity variables to designate the same entity. When that entity is deleted, the entity variable passed in the procedure as an argument is set to NULL. But other entity variables may still reference the deleted entity, and can

therefore lead to some problems if they are used before being updated. This problem is well known for PASCAL pointers as well.

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 soft : TSOFTWARE;

    . . .

    soft.name := 'TURBO_PASCAL";
    dbid (Software, soft);
        if dbstatus > 0 then goto err_trt;
    dbdelete (Software, soft);
        if dbstatus > 0 then goto err_trt;

    . . .
```

NAME

dbmodify - modify entity

INTERFACE

dbmodify (Entity_Type, var Ent_Var)

DESCRIPTION

Modifies the attribute values of 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 particular attribute.

RETURN CODES

        dbstatus = 0   : the entity has been modified;

        dbstatus = 2   : an entity with the same identifier value already

                         exists; no modification occured;

        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 soft : TSOFTWARE;

            . . .

        soft.name := 'TURBO-C';

        dbid (Software, soft);

            if dbstatus > 0 then goto err_trt;

        soft.name := 'TURBO_C';

        soft.release := '2.00';

        dbmodify (Software, soft);

            if dbstatus > 0 then goto err_trt;

            . . .

3.4.6 Updating paths

NAME

    dbinsert -   insert entity into path

INTERFACE

        dbinsert (var Target, var Origin, Path_Type)

DESCRIPTION

        Connects the entity Target to the entity Origin; more precisely,
inserts the entity designated by Target as a target of the 1-N path of
type Path_Type with origin designated by Origin. If that entity was
already a target in a path of that type, it is first removed from that

path. That procedure should only be used for 1-N paths; if used for N-1 paths, it always returns DbStatus = 11.

RETURN CODES

    dbstatus = 0 : the entity has been inserted;

    dbstatus = 11 : Path_Type has an incorrect value;

    dbstatus = 30 : Target or Origin have incorrect value;

    dbstatus = 90 : corrupted database

    dbstatus = 99 : I/O or system error;

EXAMPLE

The following example creates a document entity and connects it to a software.

```
VAR soft : TSOFTWARE;
    doc  : TDOCUMENT;

    . . .

    doc.name := 'Requirements';
    doc.subject := 'Analysis_of_Needs';
    dbcreate (Document, doc);
        if dbstatus > 0 then goto err_trt;

    soft.name := 'Pyramide';
    dbid (Software, soft);
        if dbstatus > 0 then goto err_trt;

    dbinsert (doc, soft, Description);
        if dbstatus > 0 then goto err_trt;
```

NAME

    dbremove - remove entity from path

INTERFACE

    dbremove (var Target, var Origin, Path_Type)

DESCRIPTION

Disconnects the entity Target from the entity Origin; more precisely, removes the entity designated by Target from the 1-N path of type Path_Type with the origin designated by Origin. That procedure should only be used for 1-N paths; if used for N-1 paths, it always returns DbStatus = 11.

RETURN CODES

dbstatus = 0 : the entity has been removed;

dbstatus = 11 : Path_Type has an incorrect value;

dbstatus = 30 : Target or Origin have incorrect values;

dbstatus = 90 : corrupted database

dbstatus = 99 : I/O or system error;

EXAMPLE

```
VAR doc : TDOCUMENT;
    ver : TVERSION;

        . . .


    dbremove (ver, doc, Versioning);
        if dbstatus > 0 then goto err_trt;
```

3.4.7 Variables manipulation

These procedures handle reference and entity variables in order to avoid user mismanipulation.

NAME

dbclear - clear reference variable

INTERFACE

dbclear (var Ent_Var)

DESCRIPTION

The Ent_Var variable reference is set to null. Therefore it references no entity any more. The attribute values are left

unchanged. No type checking is performed, therefore this operation always succeds and no return code is provided.

NAME

    dbcopyatt - copy attribute values

INTERFACE

    dbcopyatt (Entity_Type, var Ent_Var1, var Ent_Var2)

DESCRIPTION

    Copies the attribute values of Ent_Var1 of type Entity_Type to Ent_Var2. Other information (e.g. reference,type) are left unchanged.

    One must note that as the Entity_type parameter is present, no type checking is needed nor performed on the entity variables. It is therefore possible to copy the attribute values between entity variables of different entity types (provided they have the same attributes structure).

RETURN CODES

    dbstatus = 0 : the attribute values have been copied;

    dbstatus = 10 : Ent_Var1 is not of type Entity_Type;

NAME

    dbcopyall - copy attribute values and reference

INTERFACE

    dbcopyall (Entity_Type, var Ent_Var1, var Ent_Var2)

DESCRIPTION

    Copies entirely (e.g. reference, type and attribute values) the variable Ent_Var1 of type Entity_Type to Ent_Var2. Therefore the two entity variables designate the same entity and contain the same attribute values.

RETURN CODES

      dbstatus = 0 : the attribute values have been copied;

      dbstatus = 10 : Ent_Var1 is not of type Entity_Type;

NAME

      dbcopyref - copy reference

INTERFACE

      dbcopyref (var Ent_Var1, var Ent_Var2)

DESCRIPTION

      Copies the reference of Ent_Var1 to Ent_Var2. Therefore the two entity variables designate the same entity. No type checking is performed, therefore this operation always succeds and no return code is provided. Other information (e.g. reference,type) are left unchanged.

NAME

      dbequal - verify reference equality

INTERFACE

      dbequal (var Ent_Var1, var Ent_Var2) : Boolean

DESCRIPTION

      Returns TRUE if the references of Ent_Var1 and Ent_Var2 are equal; that is, the two variables designate the same entity in the database or are both null. No type checking is performed, therefore this operation always succeds and no return code is provided.

NAME

      dbnull - verify null reference

INTERFACE

      dbnull (var Ent_Var) : Boolean

DESCRIPTION

Returns TRUE if the references of Ent_Var1 is null; that is, the variable designates no entity in the database. No type checking is performed, therefore this operation always succeds and no return code is provided. Note that a variable that has not received any value yet is undefined and not Null. There is therefore no way to see whether a variable has been initialized or not.

3.4.8 Setting the buffer size

The buffer size can dynamically be modified at runtime with the following procedure.

NAME

dbbuffer - change buffer size

INTERFACE

dbbuffer (length : integer)

DESCRIPTION

Sets a new buffer length given as a number of 'pages' (of 1024 bytes) stored in the main memory buffer. The default value is set to 8; the minimum is 1 and the maximum is 100 pages. If length is not within this range, no changes are performed. When this procedure is called every pages contained in the buffer are saved to secondary memory. Thus, all preceding changes that occured in the database are secured on disk. The buffer and the variables needed to manage it are reset, so that the global situation is the same as after the opening of the database.

RETURN CODES

dbstatus = 0 : the buffer size has been changed;

dbstatus = 1 : the length parameter has an incorrect value;

dbstatus = 70 : no sufficient main memory space to create new
                buffers;

dbstatus = 99 : I/O or system error;

3.4.9 Handling several databases

As already said, PYRAMIDE allows to work on several databases at the same time. The databases must have been opened thus initializing the database descriptor bloc pointers. Then, the last database that has been opened is active and all primitives that are being called operate on the active DB. If the user wants to perform some operations on another opened database, he must select it, so that it becomes the active database.

NAME

dbselect - select a database

INTERFACE

dbselect (DbDesc)

DESCRIPTION

Selects a database descriptor bloc. From now on, the active database is the one pointed by this descriptor. Pay attention that the pointed database must have been opened previously. The previous active database is disactivated, but not closed !

RETURN CODES

dbstatus = 0 : the database pointed by DbDesc is active;

dbstatus = 1 : no opened database is pointed by DbDesc;

EXAMPLE

In the following example, we suppose that the 'Employee' database contains among others, the Analyst entity type which has the same attribute 'name' as Author in the 'Soft_Env' database. Therefore the following program creates in the Employee database, the Analysts that are Authors in the Soft_Env database.

Label err_open;

```
    Type TAUTHOR = record

                        . . .

                    name : string[35];
                    team : integer;

                        . . .

                end;


        TANALYST = record

                        . . .

                    name    : string[35];
                    address : record
                                street : string[30];
                                number : integer;
                                locality : string[15];
                            end;

                end;

Var designer : TAUTHOR;
    analyst  : TANALYST;
    db1, db2 : DDB;

    . . .

    dbopen (Employee, db1);
        IF DbStatus <> 0 then goto err_open;
    dbopen (Soft_Env, db2);
        IF DbStatus <> 0 then goto err_open;

    Dbfirst (Author, designer);
    While DbFound do
    Begin
      Analyst.name := Author.name;
      dbselect (db1);
      dbcreate (Analyst, designer);
      dbselect (db2);
      dbnext (Author, designer)
    End;
```

NOTE

An entity variable is divided in three main parts. The attribute part can be handled by the application program, while the two other parts (reference and type code part) should normally never be accessed by anyone but the DBMS.

There are two kinds of primitives in the programming interface. The first kind allow to initialize an entity variable (Dbfirst, Dblast, Dbcopyall, Dbclear, Dbcreate). That is, these primitives fill in the various parts of the resulting entity variable.

The primitives of the other kind need that the rigth entity type code be present in the entity variables passed as parameters. Therefore, these entity variables must have been previously filled in by some initializing procedure. These primitives that have to access the type code are dbfpath and dblpath for the origin entity variable; Dbnpath, dbppath, dbremove and dbinsert for the origin and the target entity variables; and Dbnext, Dbprior, Dbmodify and dbdelete for the entity variable.

3.5 Integrity management

PYRAMIDE will allow embedded transactions. This is possible by building a hierarchy of transactions. Each transaction is composed of a set of child transactions that must begin after and end before their parent transaction. Embedded transactions offer a dynamic control structure allowing to distribute the work between the child transactions and to do roll back on a limited number of child transactions. This particular transactions management is a first answer to the specific needs already discussed in the requirements analysis.

D : The Programming Interface

NAME

dbbgtr - begin a new transaction

INTERFACE

dbbgtr (var ta_id : integer)

DESCRIPTION

Within the active database, a transaction is started as unit of
consistency, synchronisation and recovery. If the operation is
successful the transaction identifier is given as a result in ta_id.
If the transaction is started inside an already on-going transaction,
it is considered as being its child.

RETURN CODES

dbstatus = 0 : a new transaction has begun;

dbstatus = 30 : problem creating a new transaction;

NAME

dbendtr - end transaction

INTERFACE

dbendtr (ta_id : integer)

DESCRIPTION

Within the active database, the transaction identified by ta_id
and all its child transactions will be finished. After this operation
has terminated correctly, all changes that have been performed since
transaction begin are made permanent in the database.

RETURN CODES

dbstatus = 0 : the transaction has ended;

dbstatus = 90 : problem ending the transaction;

NAME

    <u>dbabtr - abort transaction</u>

INTERFACE

    dbabtr (ta_id : integer)

DESCRIPTION

    Within the active database, the transaction identified by ta_id and all its child transactions will be aborted by undoing all updates that have been performed since the begin of that transaction.

RETURN CODES

    dbstatus = 0 : the transaction has aborted;

    dbstatus = 90 : problem aborting the transaction;

3.6 · The DbSys procedures

    A set of DbSys procedures allow the user to work on the physical structures of the DBMS which are hidden in the Unit. Using these procedures can be dangereous. They should only be used by advanced programmers being acquainted with the physical level of PYRAMIDE. Therefore these procedures are only outlined in this document.

    These additional primitives are interesting for performance measuring, and also to understand how the Kernel primitives work. Such procedures allow for instance the programmer to do debugging of the Kernel primitives at a very low level. Functionalities such as to enable/disable tracing along the internal procedures of the DBMS are supported. Procedures to read internal schema information or internal pointers are also provided.

    As conditional compiling is performed, these procedures are not always availaible depending on the compiled version in use. Therefore, parameters setting at source code level is sometimes necessary before re-compiling to have these procedures available. If you need such enhancements or simply more information, please contact

either the author or Professor Hainaut. Here follows a short description of some of these primitives.

NAME

    dbsys traceon - enables tracing

INTERFACE

    dbsys_traceon (level : integer, output : string[64])

DESCRIPTION

    Enables tracing among internal primitives of the DBMS at a given level, and directs the result towards the indicated output "file" ("debug.trc", screen, printer, ...).

RETURN CODES

    dbstatus = 0 : tracing is enabled;

    dbstatus = 90 : not possible to open the output;

NAME

    dbsys traceoff - disables tracing

INTERFACE

    dbsys_traceoff

DESCRIPTION

    Disables any previously active tracing.

RETURN CODES

    dbstatus = 0 : tracing is disabled;

    dbstatus = 90 : not possible to close the output;

NAME

dbsys ref - decodes a reference

INTERFACE

dbsys_ref (var Ref_Var, var page : integer, var posit : integer)

DESCRIPTION

Decodes an internal reference of an entity into its page number and its position inside that page.

NAME

dbsys space - gives the free space

INTERFACE

dbsys_space (page : integer) : integer

DESCRIPTION

Gives the remaining free space in a given page designated by its physical number.

NAME

dbsys first - get first record

INTERFACE

dbsys_first (page : integer, var Ent_Var)

DESCRIPTION

Finds the first entity record **whatever its type** contained in the given page of the active database and stores its reference and its attribute values into the variable Ent_Var. If that entity doesn't exist, the content of Ent_Var is unchanged.

RETURN CODES

    dbstatus = 0 : an entity has been found;

    dbstatus = 1 : entity not found; there aren't any entity in that
                    page of the active database;

    dbstatus = 99 : I/O or system error;

NAME

    dbsys next - get next record

INTERFACE

    dbsys_next (page : integer, var Ent_Var)

DESCRIPTION

    If Ent_Var designates an entity of the given page in the active
database which is not the last one, the procedure finds the next
entity whatever its type that follows the entity designated by Ent_Var
in the database. It then stores the reference and the attribute
values of the next entity into the variable Ent_Var. If Ent_Var was
the last entity in that page, the content of Ent_Var is unchanged. If
Ent_Var has a null reference, dbnext acts as dbfirst (the successor of
none is the first one). The dbfirst procedure is therefore redundant.

RETURN CODES

    dbstatus = 0 : an entity has been found;

    dbstatus = 1 : entity not found; no more entities in that page;

    dbstatus = 99 : I/O or system error;

N.B. : These two last procedures enables the programmer to dump the
entity records contained in a given page. Therefore, it is very easy
to completely dump the content of a whole database, by scanning all
the pages with those procedures.

NAME

   dbsys info ent - get info about entity type

INTERFACE

   dbsys_info_ent (type : integer, var Latt, Lptr, posid, Lid,
                   typid, Pbeg, Pend : integer)

DESCRIPTION

   Gives information about the entity records of a given type. It
gives the length of the attributes and of the pointers, the position,
the length and the type of the identifier, and the page range for the
entity type.

RETURN CODES

   dbstatus = 0 : the entity descriptor has been found;
   dbstatus = 10 : the entity type doesn't exist;

NAME

   dbsys info path - get info about a path type

INTERFACE

   dbsys_info_path (path : integer, var typorig, typtarg, posorig,
                    postarg : integer)

DESCRIPTION

   Gives information about the paths of a given path type. It gives
the entity types of the origin and the target, and the position of the
path pointers in the respective entities.

RETURN CODES

   dbstatus = 0 : the path descriptor has been found;
   dbstatus = 10 : the path type doesn't exist;

3.7 Fine tuning of parameters

Some physical parameters can be set to user defined values in order to get optimized performance from a given set of applications. These parameters are the storage scheme and the page range of each entity type. A detailed explanation of the resulting effects will be given in the physical description of the DBMS.

The storage scheme of each entity type can be set either to Clustered or Random. The default setting is clustered. If the storage scheme is Random, an explicit page range must be given. The page range of each entity type can be set as the beginning and ending page numbers such that $0 < Beg <= End <= 65000$. This is done at schema description time, and cannot further be modified after schema compilation.

4   Case studies

4.1 The Soft Env database

4.1.1 Description of the application domain

A software product is described by documents. Each document can have many versions. And each version has been realized by one or more authors under a contract number. The schema of the database is given in figure D.1.

4.1.2 The programming environment

The following definitions are available in the Soft_Env.TYP file after compilation of the schema with the MetaCompiler. This file must be included in any program working with the Soft_Env database.
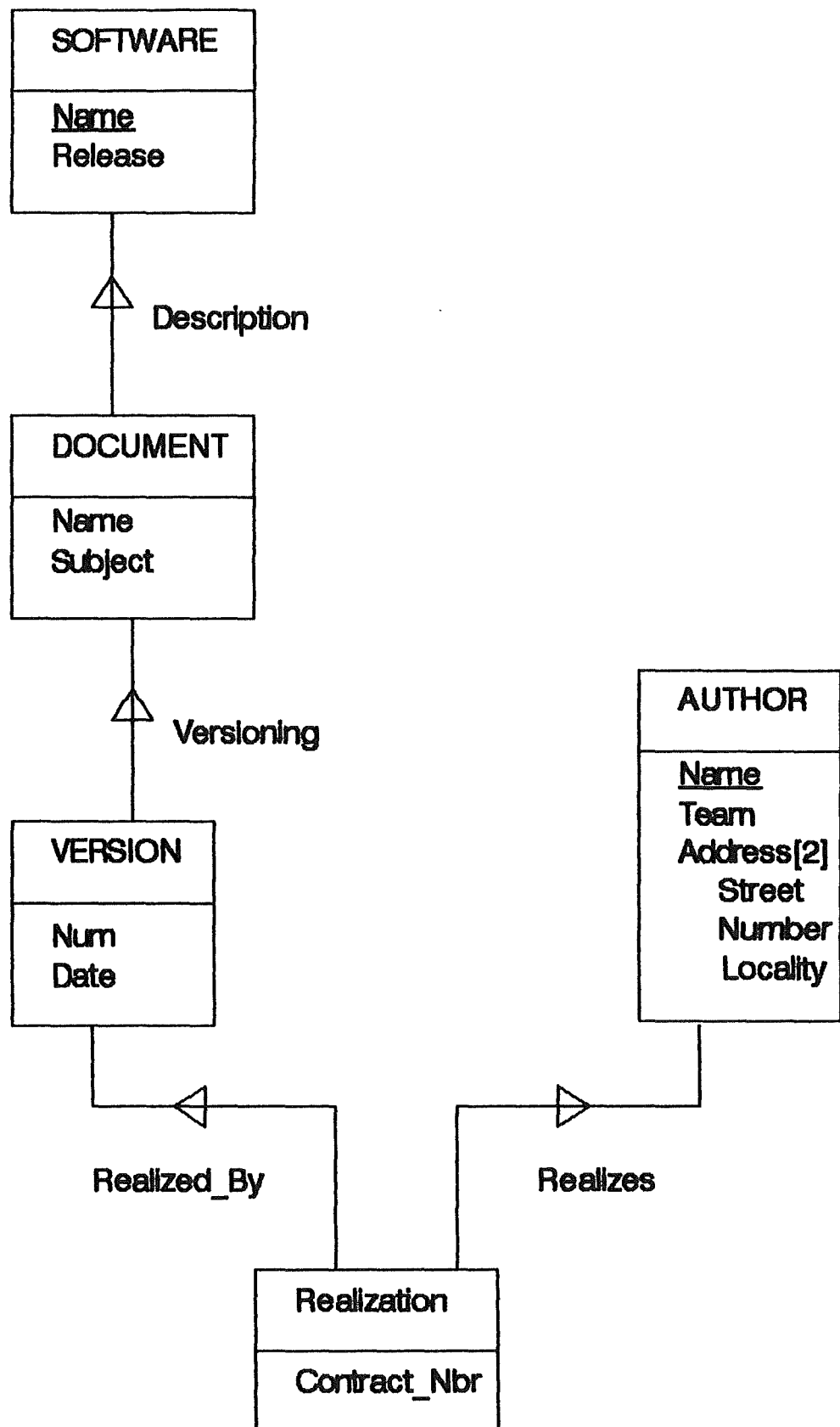
Figure D.1 : The logical schema of the Soft_Env database

```
CONST

        SOFTWARE    = ... ;
        DOCUMENT    = ... ;
        VERSION     = ... ;
        REALIZATION = ... ;
        AUTHOR      = ... ;

        DESCRIPTION = ... ;
        VERSIONING  = ... ;
        REALIZED_BY = ... ;
        REALIZES    = ... ;


TYPE

        TSOFTWARE = record
                        ...
                        NAME    : String[35];
                        RELEASE : String[4];
                    end;

        TDOCUMENT = record
                        ...
                        NAME    : String[35];
                        SUBJECT : String[160];
                    end;

        TVERSION  = record
                        ...
                        NUM  : integer;
                        DATE : Sting[6];
                    end;

        TREALIZATION = record
                           ...
                           CONTRACT_NBR : integer;
                       end;
```

```
          TAUTHOR    = record

                          . . .

                          NAME : String[35];
                          TEAM : byte;
                          ADDRESS : array [1..2] of
                                         record
                                             street : string[30];
                                             number : integer;
                                             locality : string[15];
                                         end;

                      end;
```

4.1.3 Simple sequential scanning

This first program lists the characteristics of all the SOFTWARE
entities. It simply scans the SOFTWARE entities in the database and
displays the values of NAME and RELEASE for each of them.

```
program seq1;
uses pyramide;
{$I soft_env.typ }

Var SOFT : TSOFTWARE;
    dtb  : DDB;

Begin
     dbopen ('soft_env',dtb);
     dbfirst (SOFTWARE,SOFT);
     While DbFound DO
     Begin
          writeln (SOFT.NAME, SOFT.RELEASE);
          dbnext (SOFTWARE,SOFT)
     End;
     dbclose (dtb)
End.
```

4.1.4 Selective sequential scanning

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

```
program seq2;
uses pyramide;
{$I soft_env.typ }

Var DOC  : TDOCUMENT;
    Name : String[35];
    dtb  : DDB;

Begin
      dbopen ('soft_env',dtb);
      write ('Enter document name : '); readln(Name);
      dbfirst (DOCUMENT,DOC);
      While DbFound DO
      Begin
           if DOC.NAME = Name
           then
                writeln (DOC.RELEASE);
           dbnext (DOCUMENT,DOC)
      End;
      dbclose (dtb)
End.
```

4.1.5 Immediate access based on identifier

The program displays the characteristics of the SOFTWARE entity (if any) whose NAME matches the value given by the user at the terminal. A NAME value identifies at most one SOFTWARE entity.

```
program ident;
uses pyramide;
{$I soft_env.typ }

Var SOFT : TSOFTWARE;
    dtb  : DDB;

Begin
     dbopen ('soft_env',dtb);
     Write ('Enter software name : '); readln (SOFT.NAME);
     dbid (SOFTWARE,SOFT);
     if DbFound DO
          then writeln (SOFT.NAME, SOFT.RELEASE);
     dbclose (dtb)
End.
```

4.1.6 A 2-level embedded access program

The program prints a report giving the NAME of all DOCUMENT entities for each SOFTWARE entity of the database. For each SOFTWARE entity, the program examines all the DOCUMENT entities that are connected to it via DESCRIPTION and gets the value of their NAME.

```
program path;
uses pyramide;
{$I soft_env.typ }

Var SOFT : TSOFTWARE;
    DOC  : TDOCUMENT;
    dtb  : DDB;

Begin
     dbopen ('soft_env',dtb);

     dbfirst (SOFTWARE,SOFT);
```

```
            While DbFound DO
            Begin
                    writeln (SOFT.NAME);

                    dbfpath (DOC, SOFT, DESCRIPTION);
                    While Dbfound DO
                    Begin
                            writeln ('    ', DOC.NAME, DOC.SUBJECT);
                            dbnpath (DOC, SOFT, DESCRIPTION)
                    End;

                    dbnext (SOFTWARE,SOFT)
            End;

            dbclose (dtb)
        End.
```

4.1.7 A 5-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 NAME of all the AUTHOR that have been working on a given SOFTWARE.

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

```
get the specified SOFTWARE
        for each of DOCUMENT of the SOFTWARE
                for each VERSION of the current DOCUMENT
                        for each REALIZATION of the current VERSION
                                get the corresponding AUTHOR
                                print his NAME
```

One can note that there is also a protection against the multiple printing of the same AUTHOR name. The Putset procedure puts a new name into a set of names. The boolean InSet procedure is true if the name is already in the set. So with the additional test, we are sure not to print trwice the same name.

```
program Aut;
uses pyramide;
{$I soft_env.typ }

Var SOFT : TSOFTWARE;
    DOC  : TDOCUMENT;
    VER  : TVERSION;
    REAL : TREALIZATION;
    AUT  : TAUTHOR;
    dtb  : DDB;

Begin
    InitSet;
    dbopen ('soft_env',dtb);

    write ('Enter software name : '); readln(SOFT.NAME);
    dbid (SOFTWARE,SOFT);
    if DbFound then
    Begin
        dbfpath (DOC, SOFT, DESCRIPTION);
        While Dbfound DO
        Begin
            dbfpath (VER, DOC, VERSIONING);
            While DbFound DO
            Begin
                dbfapth (REAL, VER, REALIZED_BY);
                While DbFound DO
                Begin
                    dbfpath (AUT, REAL, - REALIZES);
                    if DbFound and Not InSet(AUT.NAME)
                    then Begin
                            PutSet (AUT.NAME);
                            writeln (AUT.NAME)
                        End;
                    dbnpath (REAL,VER,REALIZED_BY)
                END;
                dbnpath (VER,DOC,VERSIONING)
```

```
                    End;
                    dbnpath (DOC, SOFT, DESCRIPTION)
            End
        End;

        dbclose (dtb)
    End.
```

4.2 The Bill-Of-Material (BOM) database

4.2.1 Description of the application domain

This example is taken from [Hain87]. It 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 directly nor indirectly.

The database schema (figure D.2) includes only one entity type, namely PART. A PART entity is characterized by its NAME, which is an identifier, the QUANTITY of such parts that enter into the super part and the WEIGHT of one such part. The super/sub-part relationship is represented by the PSP relationship type.

4.2.2 The programming environment

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

```
CONST
        PART = ... ;

        PSP = ... ;
```
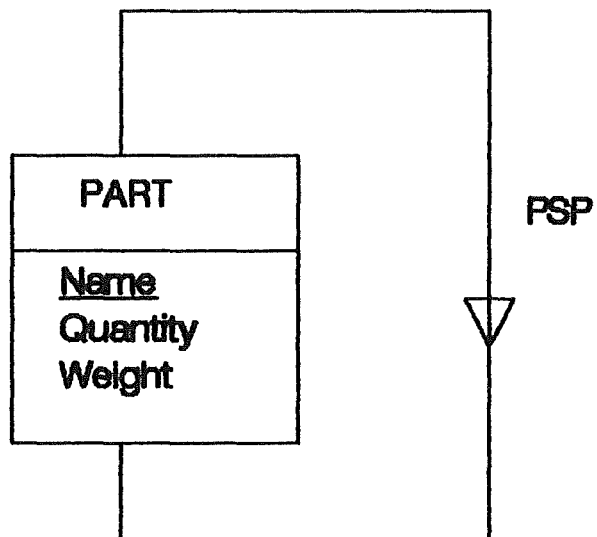
Figure D.2 : The logical schema of the BOM database

```
TYPE
     TPART = record
                  . . .
                  NAME      : String[35];
                  QUANTITY  : Integer;
                  WEIGHT    : Real;
            end;
```

4.2.3 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.

```
program BOM1;
uses pyramide;
{$I bom.typ }

Var P : TPART;
    dtb  : DDB;

Procedure EXPLODE (var P : TPART);
Var SP : TPART;
Begin
     writeln (P.NAME);
     dbfpath (SP,P,PSP);
     While DbFound DO
     Begin
           EXPLODE (SP);
           dbnpath (SP,P,PSP)
     End
End;
```

```
    Begin
         dbopen ('soft_env',dtb);
         write ('Enter id-name of the part to explode : ');
         readln (P.NAME);
         dbid (PART,P);
         if DbFound
             then EXPLODE (P);

         dbclose (dtb)
    End.
```

4.2.4 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 BOM2;
    uses pyramide;
    {$I bom.typ }

    Var P : TPART;              {the root of the part decomposition}
        dtb  : DDB;

    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;
         dbfpath (SP,P,PSP);
         While DbFound DO
         Begin
             PW := PW + SP.QUANTITY * PWEIGHT (SP);
             dbnpath (SP,P,PSP)
         End;
```

```
        if PW > 0 {update part P if it has at least one component}
        then Begin
                P.WEIGHT := PW;
                dbmodify (PART,P)
            END;
        PWEIGHT := PW        {return the weight of part P }
End;

Begin
        dbopen ('soft_env',dtb);
        write ('Enter id-name of the part to update : ');
        readln (P.NAME);
        dbid (PART,P);
        if DbFound
            then writeln('Weight of part ',P.NAME,' = ',PWEIGHT(P));

        dbclose (dtb)
End.
```

4.3 Listing of a schema

We give herein a program that lists the ENTITY_TYPE entities of a schema together with the ATTRIBUTE entities connected to them, and the REL_TYPE entities.   The Process_Att procedure processes recursively the component attributes of a super-attribute. And from each REL_TYPE entity, the program scans the ENTITY_TYPE entities connected to it via a ROLE entity. This example shows clearly that working on the Meta Data is the same as working on normal data. There is only the types (contained in standard.typ) the primitives are working on which change.

```
program listdb;
{ -- listing of a schema description -- }

uses pyramide;
{$I standard.typ }

label EXIT;
var SCH       : TDBSCHEMA;
    ET1, ET2 : TENTITY_TYPE;
    RT        : TREL_TYPE;
    ROL       : TROLE;
    ATT       : TATTRIBUTE;

    DTB : DDB;
    I : integer;

Procedure Process_Att (var ATT : TATT);
Var SATT : TATT;            {one of the component attribute of Att}
Begin
     writeln (ATT.NAME);
     dbfpath (SATT, ATT, ATT_ATT);
     While DbFound Do
     Begin
          Process_Att (SATT);
          dbnpath (SATT, ATT, ATT_ATT)
     End
End;

begin
    dbopen('standard', DTB);
        if DbStatus <> 0 then begin
                              writeln ('problem opening');
                              goto exit
                          end;



    write ('Enter a schema name : '); readln (SCH.NAME);
```

- 56 -

```
        dbid (DBSCHEMA,SCH);
            if DbNotFound then begin
                                writeln ('Schema not found');
                                goto exit
                            end;

        dbfpath (ET1,SCH,DBSCHEMA_ET);
        WHILE DBFOUND DO           { list the ENTITY_TYPE entities }
        BEGIN
          writeln ('ET : ',ET1.NAME);
          dbfpath (ATT,ET1,ET_ATT);
          WHILE DBFOUND DO         { list the ATTRIBUTE entities }
          BEGIN
            Process_Att (ATT);   {process recursively the attributes}
            dbnpath (ATT,ET1,ET_ATT);
          END;
          dbnpath (ET1,SCH,DBSCHEMA_ET);
        END;

        dbfpath (RT,SCH,DBSCHEMA_RT);
        WHILE DBFOUND DO           { list the REL_TYPE entities }
        BEGIN
          writeln ('RT : ',RT.NAME);
          dbfpath (ROL, RT, RT_ROLE);
          While DbFound DO
          Begin {list the ENTITY_TYPE entities connected by a ROLE}
              dbfpath (ET2, ROL, - ET_ROLE);
              if DbFound then
                  writeln ('ET: ',ET2.NAME,' with role ',ROL.NAME);
              dbnpath (ROL, RT, RT_ROLE)
          End;
          dbnpath (RT,SCH,DBSCHEMA_RT);
        END;

  dbclose(DTB);
  EXIT :
  end.
```

4.4 Loading of a schema description

Here follows a program that loads the description of the BOM schema in the form of meta data.

```
program loaddb;
uses pyramide;
{$I standard.typ }
label EXIT, fin;
var SCH : TDBSCHEMA;
    DTB : DDB;
    ET  : TENTITY_TYPE;
    RT  : TREL_TYPE;
    Rol : TROLE;
    Att : TATTRIBUTE;
    Com : TCOMPONENT;
    GR  : TGROUP;
    I : integer;

begin
    dbopen('BOM',DTB);

    SCH.NAME := 'BOM_Schema'; {creating a new BOM schema}
    SCH.Short_Name := '';
    dbcreate(DBSCHEMA,SCH);

    ET.Name := 'PART';          {creating the entity type PART}
    ET.Short_Name := '';
    ET.Beg_Page := 0;
    ET.End_Page := 0;
    dbcreate(ENTITY_TYPE,ET);
    dbinsert (ET,SCH,DBSCHEMA_ET);

    ATT.NAME := 'NAME';         {creating the attribute NAME}
    ATT.VAL_TYPE := 'S';
    ATT.VAL_LENGTH := 35;
    ATT.DEC := 0;
    ATT.MIN_REP := 1;
```

```
        ATT.MAX_REP := 1;

        dbcreate (ATTRIBUTE,ATT);

        dbinsert (ATT,ET,ET_ATT);

        COM.NUMBER := 1;

        COM.C_TYPE := 'S';

        dbcreate (COMPONENT,COM);

        dbinsert (COM,ATT,ATT_COMP);

        GR.NUMBER := 1;

        GR.ID := 'Y';

        GR.STATUS := 'P';

        GR.REF := 'N';

        GR.KEY := 'Y';

        dbcreate (GROUP,GR);

        dbinsert (COM,GR,GR_COMP);


        ATT.NAME := 'QUANTITY';   {creating the attribute QUANTITY}

        ATT.VAL_TYPE := 'I';

        ATT.DEC := 0;

        ATT.MIN_REP := 1;

        ATT.MAX_REP := 1;

        dbcreate (ATTRIBUTE,ATT);

        dbinsert (ATT,ET,ET_ATT);


        ATT.NAME := 'WEIGHT';     {creating the attribute WEIGHT}

        ATT.VAL_TYPE := 'R';

        ATT.DEC := 0;

        ATT.MIN_REP := 1;

        ATT.MAX_REP := 1;

        dbcreate (ATTRIBUTE,ATT);

        dbinsert (ATT,ET,ET_ATT);


        RT.NAME := 'PSP';         {creating the relationship type}

        RT.SHORT_NAME := '';

        dbcreate (REL_TYPE,RT);

        dbinsert (RT,SCH,DBSCHEMA_RT);


        Rol.NAME := 'ORIGIN';
```

```
        Rol.MIN_CON := 1;
        Rol.MAX_CON := 1;
        dbcreate (ROLE,Rol);
        dbinsert (Rol,RT,RT_ROLE);
        dbfirst (ENTITY_TYPE,ET);
        dbinsert (Rol,ET,ET_ROLE);

        Rol.NAME := 'TARGET';
        Rol.MIN_CON := 1;
        Rol.MAX_CON := 9999;
        dbcreate (ROLE,Rol);
        dbinsert (Rol,RT,RT_ROLE);
        dbnext (ENTITY_TYPE,ET);
        dbinsert (Rol,ET,ET_ROLE);

fin: dbclose(DTB);
EXIT :
end.
```

Section E :

The PYRAMIDE-DBMS

1. The architecture of the DBMS

PYRAMIDE has been programmed into three main layers corresponding to different levels of abstraction. The well-known advantages of such a hierarchical structure are to reduce the complexity of the system (as it is decomposed into smaller components), while insuring its independance with respect to maintenance (as interrelations between the different layers are restricted). The view offered at each level hides the concepts belonging to the other layers, thus simplifiyng the perception of each part making up the global system. And the independency principle is that the modification of given layer doesn't induce any change in the other layers.

The higher layer deals with the concepts of entity type, relation type, attribute and database. It comprises primitives to manage databases, to access and update entities and relations and to work on attribute values. The intermediate layer deals with the concepts of reference, logical accesses and schema information tables. It comprises primitives to retrieve information from the tables, to access and update records, to navigate among paths between records and to manage references. And finally, the lower layer deals with physical concepts such as page, buffer, string of bytes and pointers. We find here primitives to work at the file and byte level.

Between the logical and the physical layers, we can also find the ISAM (Indexed Sequential Access Method) that allows to quicken access to entities according to their identifier value.

2. Physical data structure of PYRAMIDE-DBMS

2.1 Inter records chaining (bi-directional ring)

The sequential scanning of records of the same type is made possible by chaining them one to another with pointers. A pointers table gives the reference of the first record for each type (if it doesn't exist, the reference is Null). Then each record is chained bi-directionally in a ring fashion. That is, each record has a first pointer towards its preceding record in the chain and a second pointer towards the next one (see figure E.1). Thus, enabling very easy access to the first, last, previous and next record, in no more than one physical access (if it is not already in the buffer).

Here the ring solution was chosen because it doesn't cost anything to find the first record in the ring thanks to the presence of an indirection table pointing towards it. The same solution was not chosen for path pointers as we will see in the next point.

2.2 Intra path chaining (bi-directional)

Here, the sequential scanning of the records in a path is realized as follow. The origin entity record has a pointer towards its first target record in the path and another towards its last target. In turn, each target entity record has a pointer towards the preceding target in the path, another towards the next one and a third one towards its origin entity record (see figure E.2). Thus, enabling easy access to the first, last, prior and next target, and back to the origin record from a target, in no more than one physical access (if not present in the buffer).

The ring method was not implemented here, because it would cost one more physical access (to the origin record) to find the first target in the ring. Instead, the first target in our system is the
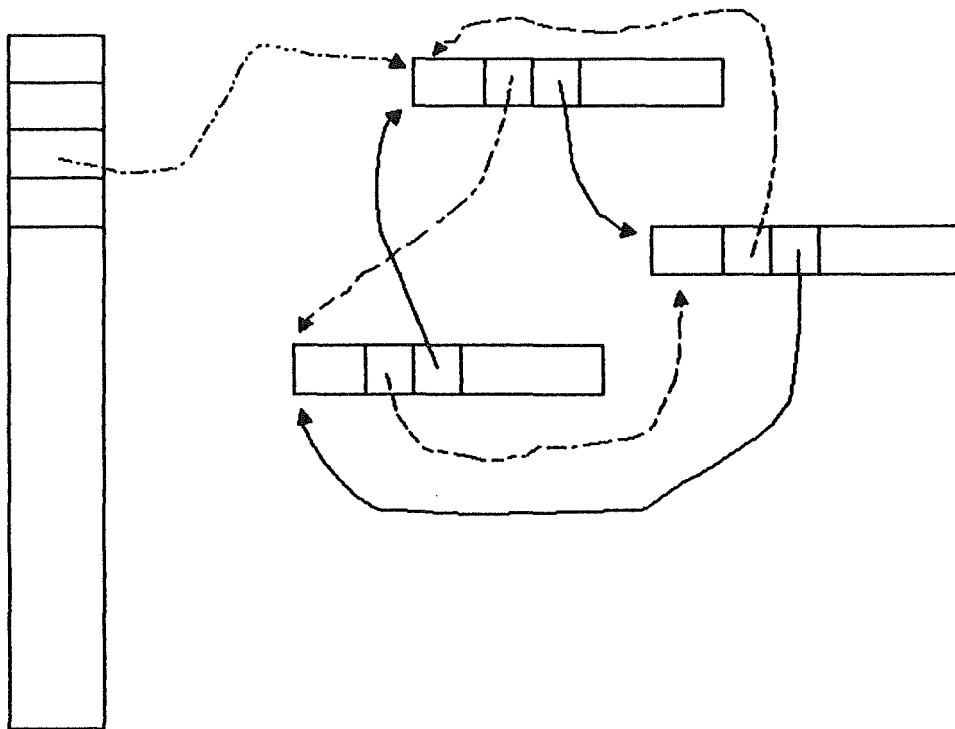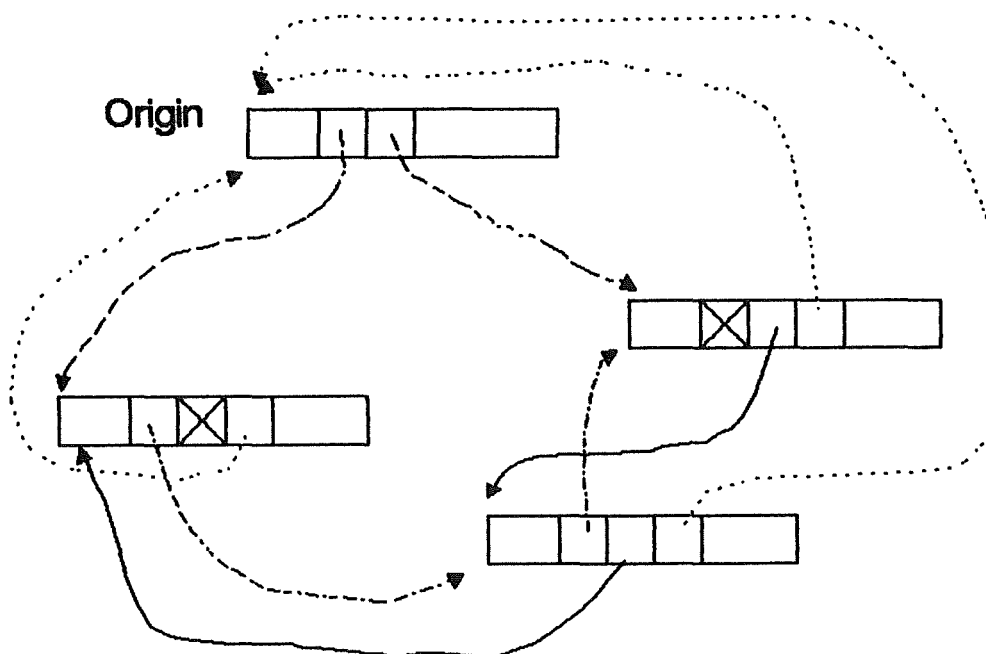
Figure E.1 : Inter entities chaining

Origin

Figure E.2 : intra path chaining

one without any preceding target; the last one having not any following target (pointers set to the Null reference).

2.3 Physical record composition

We must distinguish the record composition between records stored into reference variables (in main memory) and records stored in the database (in the file or the buffer).

In the database, a record is composed of its entity type coded into one byte, a string of pointers (three bytes) towards other records to implement sequential access among records of a given type and inside paths between records, and the attribute values of the entity record (figure E.3).

In reference variables, a record is preceded by its reference (three bytes) and the pointers are not present (figure E.4).

So, this explains the formula to calculate the length of a record. That is, besides its attribute values, a record in the DB is composed of a type (one byte), two inter records pointers (2 * 3 bytes), and (3 * the number of paths in which its type is the target type) + (2 * the number of paths in which its type is the origin type). In a reference variable, the actual length occupied by the record can be calculated as the total attributes length + 1 (for the type byte).

2.4 Physical structure of a page

As already said, the database is completely contained in one file that is divided into pages of 1024 bytes.

There are various types of pages. For the most part, pages contain user data and/or meta data, they are called data pages. But some others contain schema information tables, or records pointers, or free space indexes, or B-trees indexes. Each page has its four first bytes reserved for special purpose. In particular, the third byte

Type	String of Pointers	Attribute values

Figure E.3 : record composition in the database

Reference	Type	Attribute values

Figure E.4 : record composition in a variable

always give the type of the page. "1" is for pages containing schema information tables. "2" is for the entity record pointers page. "3" is for level-one free space index and "4" for the second-level pages. "5" is for the B-Trees indirection table page. "6" is for the B-Trees indexes and "7" for the B-Trees leaf pages. And finally, "10" denotes a page where data is strored in the form of entity records. This is important for recovery purposes and also to control access to some pages.

The different kinds of page have also different structures. We will described here the pages containing data stored in the form of records. The other kinds of page will be studied later.

A data page is composed of a data zone and an indirection pointers table. The first two bytes of the page are pointers to these two zones that avoid collision. The first byte points towards the last pointer of the indirection table. Where the second byte points to the last bloc of bytes occupied by data. One must note also that the first byte gives the right number of the pointer, while the second gives a value in terms of data blocs (that is four bytes). The pointers (one byte) in the indirection table also give a value in terms of data blocs. That is, we lose in fact an average of two bytes per record which seems negligible (see figure E.5).

The address of a record is decomposed into a page number coded onto two bytes, and a rank number of the record into the given page coded onto one byte. An address is said to be NULL if the page number and the rank are set to zero. It is an invalid address since the first page (numbered zero) always contains DB management information and ranks begin with number one.

The address part that gives the rank number of the record stored in the current page, is in fact the number of the pointer towards this record in the indirection table. That is, we can theoretically have a maximum of 255 records stored in a page. A pointer in the indirection table having a value equal to zero means that the corresponding record has been deleted. This system insures the stability of the addresses
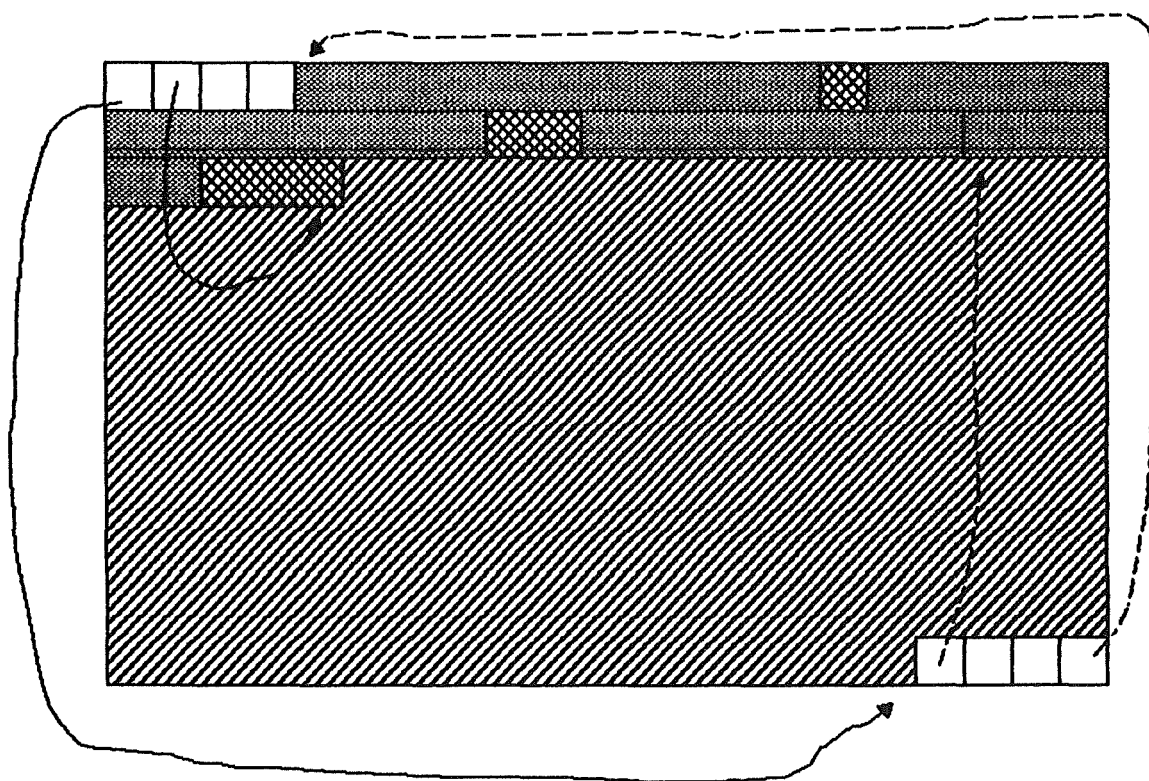
Figure E.5 : the structure of a data page

Attribute length	pointer length	position of id	id's length	id's type	Page range

Entity type descriptor

origin's type	target's type	pointers in origin	pointers in target

Path type descriptor

Figure E.6 : descriptors for entity and path types

of the records in case a page is recompacted. That is, after deletion of a record, the page is recompacted and the values of the indirection table pointers are updated. And as records are never moved from one page to another, our system insures the global stability of the addresses.

2.5 The schema information tables

We can easily understand that the DBMS needs to know some information about the entity records and the paths between them. These information are given by the schema internal tables.

These tables contain for each entity type, the length of its attributes, the length of its paths pointers, the position, the length and the type of the identifier attribute (if it there is one), and the storage page range. That is nine bytes for each entity record descriptor. In addition the tables contain for each path type, the origin and target entity types, and the positions of the path pointers in the origin and targets records; that is, four bytes per path type (figure E.6).

These tables are contained in specially structured pages. The first two bytes of the page are used as a pointer towards the next "table" page (if it exists, else zero). The third byte gives the page type (i.e. '1'), and the fourth byte gives an offset value for the beginning of the data pages. The next byte gives the number of entity record descriptors, and then come the descriptors one after another in increasing order of the type code. The byte following the last entity record descriptor gives the number of path descriptors which then come one after another in increasing order of the path code. The DBMS has therefore, all the information necessary to access these tables, and then to use them to navigate from one record to another and retrieve or update data. As the information contained in those tables is accessed very often, they are stored at opening time and stay always in main memory.

One can point out the redundancy between the database schema and its compiled form stored in the internal tables. The information needed by the DBMS could therefore be derived from the schema at the opening of the database. But this would involve a loss of time and could lead to internal errors if the schema is updated. In our solution, the changes are taken into account only if the schema is recompiled.

2.6 General physical structure of the file

The first page always contains the schema information tables that have previously been compiled by the MetaCompiler. If they span across more than one page, the pages containing them are chained one to another via pointers included in the pages themselves. This chaining mechanism allows further extension of these tables as they can be expanded anywhere in the database file provided that some protection is awarded to these particular pages. This is very important for dynamic updating facilities of the database schemata and is already supported by the current version of the DBMS.

Then we have a page containing pointers to access the first entity record of each entity type. The other records belonging to the same type being chained one to another, thus enabling sequential access. The next page contains an indirection table to B-Trees indexes corresponding to each entity type. And finally there is a page containing the "level-one" free space index, and another containing the first "level-two" free space index.

The schema information tables pages, the record pointers page, the B-Trees indirection table page and the level-one free space index page are loaded into tables in main memory at opening time, in order to speed up frequent access to them.

From this short description we can already note that the first page available from the user point of view is the page logically numbered one, which in fact begins with a physical number equal to 3 +

the number of pages for the internal tables located at the beginning of the file.

3. Implementation aspects

3.1 The enhanced LRU Buffer management

The pages that are read from the database file are loaded into a special area in main memory called a buffer. This buffer is decomposed into frames. A frame can contain exactly one page of the file and the default number of frames (8) can be changed at runtime. Each frame has a descriptor giving the number of the page that is currently loaded in the frame, a flag that is set to true when the page has been updated while being in main memory, and a date giving information on the age of the page (see figure E.7).

That is, each time a page is accessed in the buffer, a global date variable is increased and the date field of the frame's descriptor is updated. When there is no more empty frames in the buffer, one of the frames must be freed before another page can be loaded. The frame must be carefully chosen, as if it contains a page that has been updated, an access to the disk is necessary to unload it back into the database file so that the changes are made permanent. On the other hand, if the frame contains a page that is often accessed, even read only, it should stay in main memory. In short we can say that in our system, the oldest page is chosen (that is, the page that has been the least recently accessed). But pages that have been updated are protected has their age is divided by two.

Every time a page is needed by the DBMS, it first searches the buffer. In case the requested page is present, no physical access occurs. Otherwise, the page is loaded into the buffer, after having freed a frame containing the least recently used page, if needed. This strategy allows the DBMS to keep the working set of pages in main memory, thus reducing drastically physical accesses.
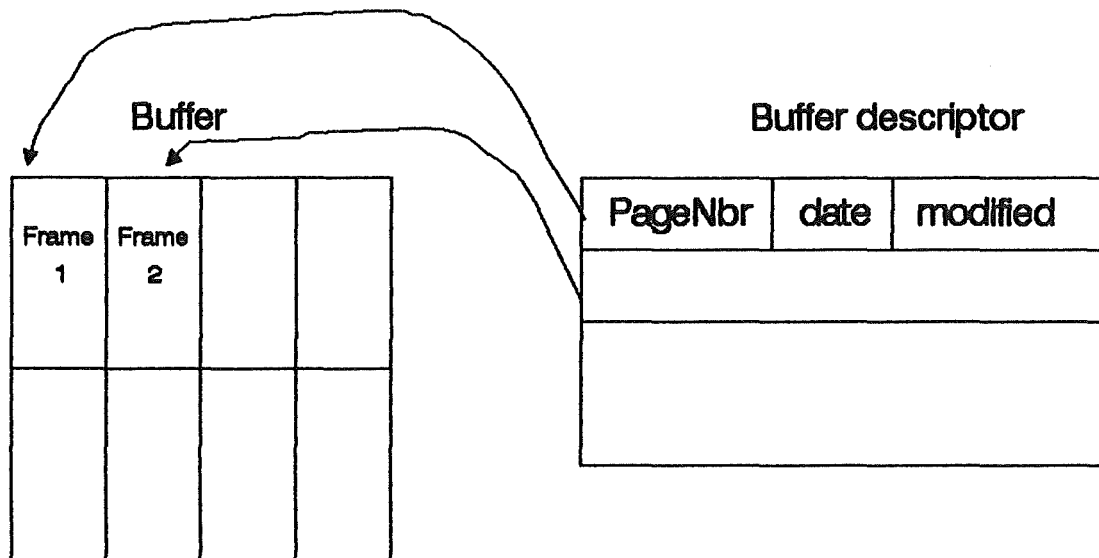
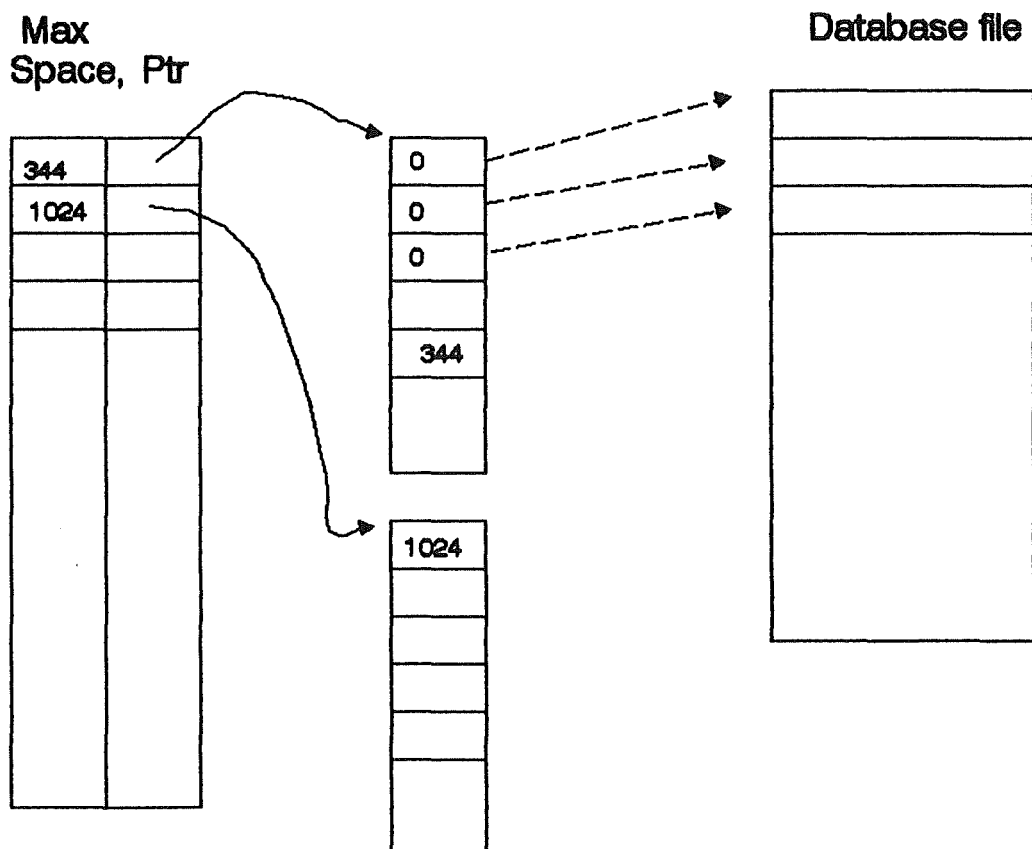**Figure E.7 : the Buffer**

Figure E.8 : the free space index

When the global date reaches its maximum value, it is re-initialized together with the frames descriptors. The system is so, that the relative age of the pages is kept the same. Thus, this re-initialization (which happens rarely) introduces no disfunction.

3.2 The free space index

When searching where to store data in the database (i.e. in which data page), a free space index is used. As we have seen previously, if the page where a record should be stored (according to its storage scheme) is full, some other pages are looked for. But if those pages were to be accessed to see how much space is left, it would increase dramatically physical accesses and thus, lower the DBMS performances.

Instead, a two level index is being searched. Each entry of the first level (contained in one database's page stored in main memory) gives the maximum space available that can be found in a given set of 510 database pages and a pointer towards a second level page indexing those data pages. Then, each entry of a page of the second level gives the remaining space in a given page (figure E.8). Therefore, our system insures the DBMS to find the right page where to store some data in no more than one physical access. Still, at most one physical access is needed to store the data (if the page was not in the buffer). That is, any data record can always be stored in the database in no more than two physical accesses (including the search for free space and the storing of the record).

The special purpose pages in the database are protected against user data insertion as they are declared full (the corresponding free space index's entry is zero) as soon as they are created.

3.3 The ISAM

In order to retrieve random information very quickly from a large database, an Indexed Sequential Access Method is provided to access data on basis of an identifier value. That is, for each entity type having an identifier, an index is build in the database.

- 8 -

3.3.1 General ISAM principles

An ISAM is composed of individual pieces of information called "keys". A key is an ASCII string representing some value in a data record. The index is arranged in such a way that keys can be retrieved randomly and sequentially. While there are many ways to implement an ISAM index, the B-Tree is generally accepted as being the current state of the art.

A tree structure is called such because if all the search paths are drawn out, they look like an inverted tree. The search starts at the root and progresses towards the bottom (the leaf level). In a simple binary tree, each key is stored in an individual node together with two pointers that make up the search paths through the tree. A search through a binary tree is very simple. The searched key is compared to the key in the root and if it matches, the search is successful. Otherwise, the search must go on. That is, if the key is smaller than the key in the root, the path designated by the left pointer is taken; else, the path to the right is taken. Then the key is compared to the key in the current node and appropriate action is taken. The search goes on until the key is found in some node or a leaf node is unsuccessfully evaluated.

As long as the nodes are kept in main memory this is an efficient method. But once the nodes must be stored on secondary storage devices (because of the large volume), the performance quickly degrades because of the large number of physical accesses required. Other problems can also arise when keys are inserted, as the tree can become unbalanced. That is, some search paths are made longer than others. This requires the insertion algorithms be aware of this possiblity, and subsequently these procedures become much more complicated. The B-tree (from R. Bayer) system tries to overcome these problems. The most obvious difference is that from any node there can be more than two paths to the next node. Thus, it allows many more keys to be stored in each node, and reduces drastically the physical accesses and therefore, the search time.

The first step of searching a B-Tree is to look at the root node and scan sequentially each key in the node (sorted in ASCII sequence). The process stops when either a matching key is found (the search is successful) or a higher key is found. In case no match is found, the path to follow to the next level is given by the pointer which sits where the searched key should be if it existed. The next node is read and the same procedure is followed until either the key is found or a leaf node is unsuccessfully scanned.

3.3.2 Inserting and deleting in a B-Tree

Insertion into a B-Tree uses the search procedure as it gives the place where the key should sit if it existed. Then the key is simply inserted into the node where it should be. Note that keys are always inserted into leaf nodes as an insertion is only allowed if the search failed. If the node into which the key is to be inserted is full, a split occurs. The keys are divided into two nodes and since there is a new node, a pointer to it must be inserted into the level above. Usually the middle key of the two nodes is brought up to the previous level to be used as a separator. If the node above is also full, then it too might be split. This can continue till the root, and if it also has to be split, a new root node is created so that the tree becomes one level higher. Since all node expansions are done on the leaf level, a B-Tree is always balanced. That is, an insertion will never increase the search path to one leaf node and not the others.

Deletion of a key is simply finding the key and taking it out of the node. If a key doesn't reside in a leaf, then a new key must take its place to provide the same paths as the deleted key. This new key is found by getting the next key in sequence from the deleted key.

3.3.3 The ISAM of PYRAMIDE

One of the most important variants of the B-Tree is the B+Tree, where all the keys are stored in leaf nodes. The upper levels simply provide pointers to the next lower level, and so on until the leaf level is reached. In addition, all the leaves are linked together, so

that we have a B-Tree type of path to the proper position in a sequential list of the keys. The general idea of the PYRAMIDE ISAM is driven from the B-Trees strategy to manage indexes [Baye72] and its B+Tree variant, but adapted in our particular case according to what we can call a "secondary index realized by inverted file" [Hain86-b] (figure E.9).

One of the peculiarity of our system is that the indexes are contained in the database file itself, thus suppressing the risks of loosing an index file or not updating data in such a file. A table stored in main memory gives for each entity type a pointer towards the first level's page of the index (Zero if there is no identifier, thus no index exists). Then we have an index composed of several levels. Each entry of an index page at a given level gives a pointer towards a next level's page and the maximum value of the identifier (key) contained in that page. The last index level points towards "B-Tree leaf pages". Each entry of a B-Tree leaf page points towards an entity record in the database and gives its identifier value (see figure). And each level of an index (and the leaf level too) is sorted on the increasing order of the identifier values.

In this system, the search uses the upper levels as a roadmap to the next level, and it is only until a leaf is reached that the key is actually looked for. Therefore, all searches use the same number of nodes, and every search will be a worst case of a normal B-Tree (which is still very good under most circumstances). Since all searches take approximately the same amount of time, a high degree of consistency is also achieved, which can have its benefits in a real-world situation.

The reader will also note that only one access key is allowed per entity type. In the future we could enhance the system so that it would allow any number of access keys per entity type or even access keys in paths, but these concepts were not needed by the current upper layer.

In order to be as much performing as possible, the loadrate of the index pages is supervised by the system. In particular, different
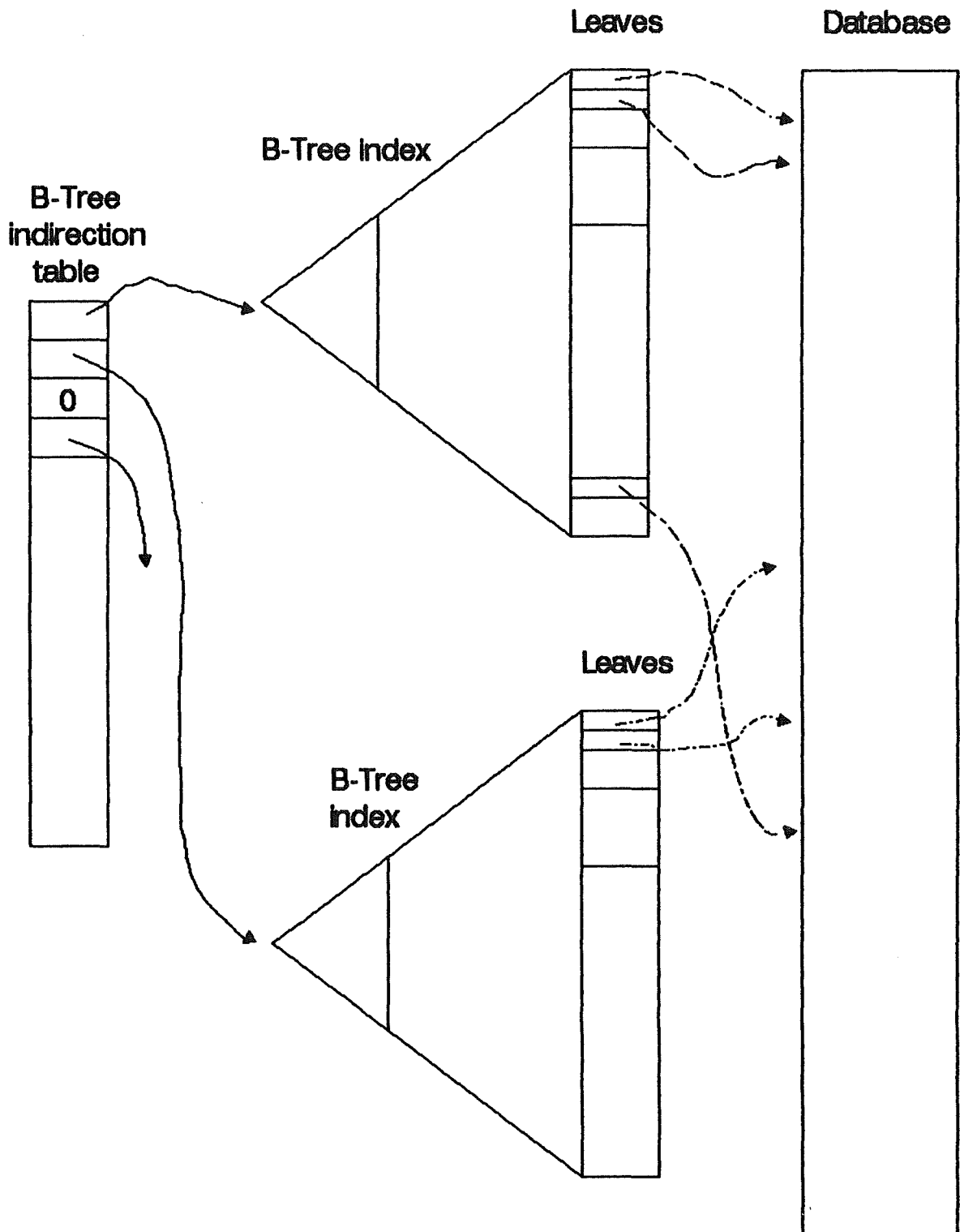
Leaves   Database

B-Tree index

B-Tree
indirection
table

0

Leaves

B-Tree
index

Figure E.9 : the ISAM of PYRAMIDE

insertion procedures (insert before and insert last) are applied depending on the loading strategy. Otherwise, in case of a sorted loading process, the index pages could be splitted each time a new record is inserted. That is, the system acts differently depending on whether the application program does an append or a loading process. There is a also a maximum loadrate for the pages, which is a parameter of the insertion procedures. And even when entries are deleted, the system keeps a satisfying loadrate by trying to catenate neighbouring pages. With those cares, the loadrate of an index page in our system is comprised between 50% and 80%.

When an entry is inserted before another, the normal append process in a B+Tree is performed (including splipping of pages if needed). But when an entry must be inserted as the last one (sorted loading process), the pages are not filled in completely. If the loadrate of the page would exceed 80%, then a new page is created where the entry is stored. Therefore, we can be sure not to have too many splitting when new entries will be stored in the index pages. Also, if the loadrate of a page becomes smaller than 50% after an entry has been deleted, the system tries to reorganize the neighbouring pages so that the entries are equally distributed. If possible, two pages can even be catenated. That is, the entries of the two pages are stored in one of them, and the other page is made free again.

3.4 The database definition block (DDB)

The database definition block is an area in main memory where all the information needed by the DBMS to manage a given database is stored. This area is referenced by a database pointer located in the application program memory area and that designates the database to work with (figure E.10).

The block contains :
- a pointer towards the database file;
- the database name as a string of characters;
- an offset value for the beginning of data pages;
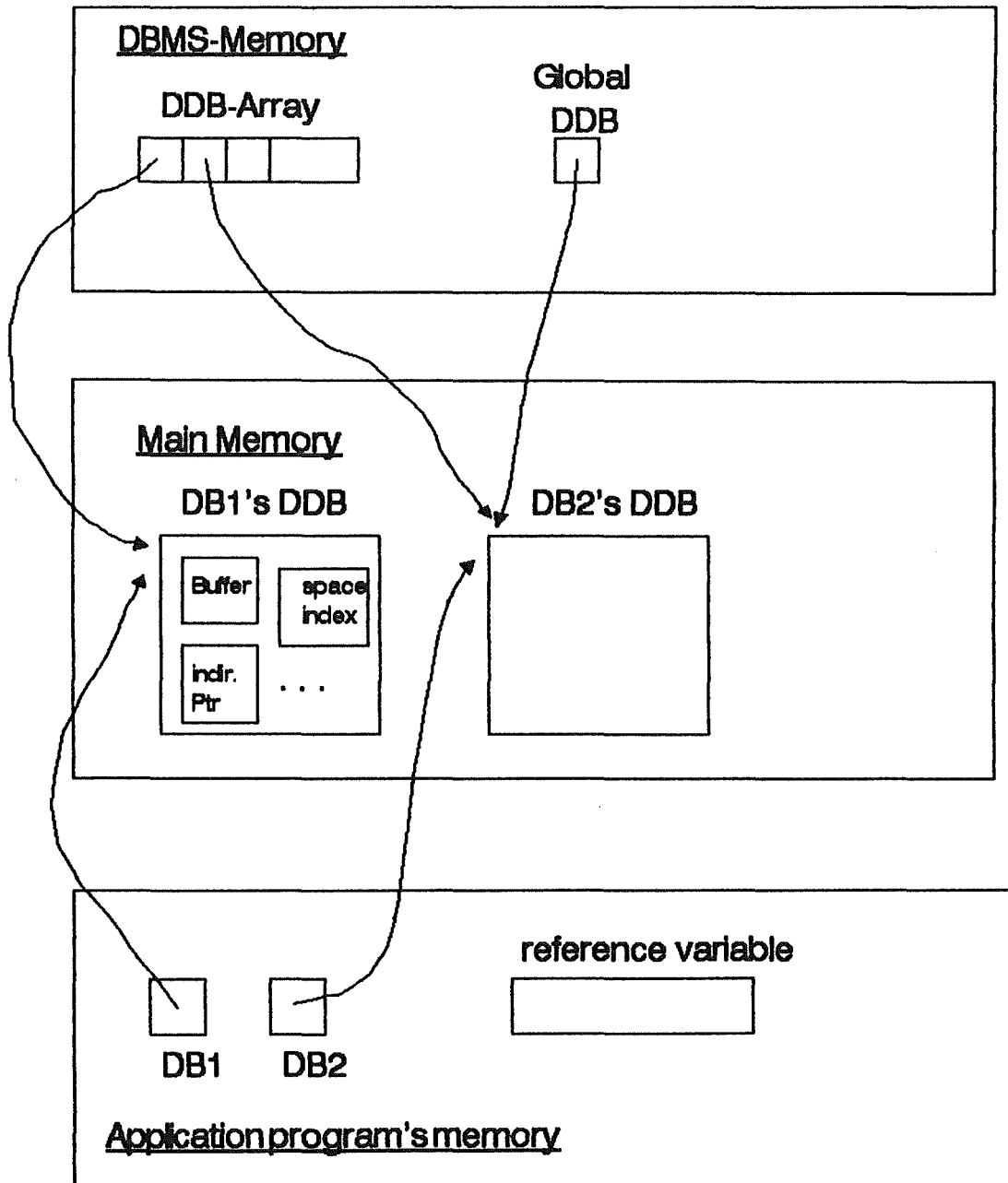
DBMS-Memory

DDB-Array

Global
DDB

Main Memory

DB1's DDB

| Buffer | space index |
| indir. Ptr | . . . |

DB2's DDB

reference variable

DB1   DB2

Application program's memory

Figure E.10 : The Database Definition Blocks

- the number of entity types and of relationship types;
- the buffer area;
- a timer giving the global date;
- the entity type descriptors table;
- the relationship type descriptors table;
- the indirection record pointers table;
- the level-one free space index table;
- the B-Tree indirection table;
- the reference of the last record that has been accessed.

When a new database is opened, its DDB pointer is stored in an array and will only be deleted when the database is closed. A global DDB pointer is set to the new value referencing the active database. If another database is selected, the array is searched for the value of its DDB pointer and if it is found, the global DDB pointer is updated.

3.5 Recovery management in PYRAMIDE

The reconstruction of a consistent DB state after an abnormal termination of a short transaction (abort) or after a system crash (crashrecovery) is called Recovery.

To recover the state of a transaction after its abnormal termination or after a system error, one must have a log-component that had stored this state. That component should then be read to re-build the current state.

The worst conditions occur with a crashrecovery, for here the re-build of a consistent DB state is also hampered by the loss of the main memory. So this implies that all data necessary for a crashrecovery must be stored in secondary memory, to be safe in case of a system crash.

The recovery function of PYRAMIDE is not implemented yet. However, some problems that have to be considered before implementing it will be discussed in the next pages that summarize the work Michael

Ranft, Andreas Geppert and I have made at FZI for the DAMOKLES-DBMS [Ranf88].

3.5.1 The log-granule

The quantity of information that is stored in a log-entry (log-record) is called the log-granule. The question here is what information on the DB state one must have at the beginning of a transaction to be able later to recover it quickly. First of all, there are many possibilities concerning the contents of a log-record.
- the content of a page that has been changed by the transaction;
- the content, the address and the length of a bytestring that has been changed by the transaction;
- an operation that does the opposite change (reverse operation).

The advantage if the first two solutions is the simple search of the log-record and a primitive recovery strategy : replace the new values by the old ones. The disadvantage is that with this strategy of logging, data changes that have not directly been done by the user, must also be recorded. These secondary data are for instance, access paths or other internal lists that are necessary for the manipulation of primary data by the user.

This problem can be solved by recording reverse operations, so that all secondary data will be changed automatically. What's more, the log-entry of a reverse operation is shorter than a complete page, or the bytestrings that would be recorded in the two first strategies. The disadvantage is that the reverse operations must be defined, and for some, be calculated during the on-going of the current operation. This problem is emphasized with operations on structured entities, as one only knows which entities are effectively concerned during the on-going of the operation.

One must also notice that there is a dependency between the log-granule and the locked-granule (= quantity of information that is used for the synchronisation of short transactions) that is expressed as :
log-granule <= locked-granule.

This dependency can be explained in the following example. Let's take as log-granule pages, and as locked-granule only records. Let's take also two transactions T1 and T2 that change two different records R1 and R2 located in the same page P1. During the execution of T1, page P1 will be stored in the log and then the record R1 will be changed; giving page P1'. Then the transaction P2 will record page P1' and then change record R2; giving page P1''. Then T2 ends, so the changes become permanent. But transaction T1 is then aborted, and its changes must be done reversely; that is, page P1'' is replaced by page P1. But in that case, the changes of T2 (that should have been left unchanged) are also reversed. This happens only because the log-granule was too big.

If the synchronisation component in PYRAMIDE is to be smaller than pages (e.g. entities or part of entities), the first proposal (pages as log-entries) should not be taken. The second proposal should also not be accepted if we are looking for as much concurrency as possible, as the fact that secondary data must also be stored does not correspond to that goal.

In conclusion, the log-granule in PYRAMIDE should be reverse-operations of the most upper level (interface primitives) as to ensure as much concurrency as possible.

3.5.2 Types of recovery

During a recovery operation, the results of transactions that did not terminate properly must be reversed (UNDO) and sometimes the results of transactions that did terminate successfully must be redone (REDO). This depends completely upon the buffering strategy that is implemented :

- if only UNDO-operations are implemeted, then by the successful end of a transaction, all changed pages must have been written in the DB, so that the changes are secure.
- if only REDO-operations are implemented, then this implies that the writing of changed pages is allowed only when the transaction has successfully ended.

- and if UNDO and REDO operations are implemented, then the buffering strategy doesn't matter; that is, the writing of the pages in the DB can occur at any moment.

The choice of one of the three possibilities only depends on the buffer strategy and the expense (in terms of DB operations) for the recovery. But one must be aware that in all three possibilities, the writting of the changed pages is bound to the writing of the corresponding log-entry.

Particularly, the expense in I/O operations should be smaller in the third strategy, while it should be almost the same in the two others.

It seems a bit foolish to implement only REDO-operations, for it would be very critic when working on very large entities. So this strategy will no longer be discussed.

The choice between UNDO and UNDO/REDO must therefore be done on basis of the expense in DB operations during the recovery. It is quite clear that the expense for the recovery logging itself would be smaller when only UNDO operations are to be written in the log. But on the other hand, the expense for the recovery, during an on-going operation should be less in the third strategy, for here only the log-entries must be stored in secondary memory at commit point (no matter the changed pages are saved or not).

But this last remark doesn't stand here for a reverse operation can only be carried out if all data to be reversed are consistent. That is, all changed pages must have been written in secondary memory before the log-entry is saved. And this is exactly the UNDO strategy! So as the third possibility does not lead here to any advantage, the strategy to be chosen is to implement only UNDO operations.

3.5.3 The recovery of a partly processed transaction

Reverse operations cannot deal with only partly processed transaction for it would lead to an inconsistent state of the DB. But the recovery must also support these cases, as for instance, after a system fault or a system crash during a DB-operation, to re-build a consistent DB state.

But here a characteristic of PYRAMIDE must be invoked : a DB-operation can be seen as an atomic action of a transaction. That is, a DB-operation must be completly ended before the next begins.

With the help of this concept, it will be possible during a DB-operation to use page-logging, for there can't be any locked-/log-granule conflict inside an atomic action.

If the operation could not be entirely done (i.e. there was a system crash during the operation), then with the help of the traced pages, the system can re-build a consistent DB state that enables to carry out reverse operations.

That is, there are two log concepts. During an operation, we'll have logging of the modified pages; and at the end of the operation, logging of the reverse operations which will replace the pages previously stored in the log.

3.5.4 The recovery

Principle

The re-build procedure of a consistent DB state after an abort transaction or a system crash is always the same. The log-entries will be read in a lifo strategy. When the entry is a before-page, the current page will be replaced by its original. Otherwise, the entry is a reverse-operation that will be processed. This is done until the read log-entry says all necessary recovery actions have been done. Then the system can get back to its normal work.

Recovery after abnormal ending of a transaction (abort) :

   To reverse a single transaction, one only has to read backwards the log-records of this transaction and of its child transactions, and process them, till arrival to the record that marks the parent transaction's begin.

Recovery after a system crash :

   The crash recovery will process all existing log-records, till it reaches the log-entry that marks the starting point of the DB system. Normally, it is the only existing record. Otherwise, all existing transactions will be reversed. Here, the reverse-operations of the various transactions can be processed in any order. This is possible because (thanks to the locking mecanism) the different transactions can only concern disjuncted entities and relationships. Still of course, inside a transaction, the log-records must be processed from the youngest towards the oldest (lifo).

Idempotency of the recovery :

   One has of course no garanty that another DBS crash won't occur during the recovery operations. This means that all recovery operations must be idempotent (that is, the results must be the same, no matter how many times it has been carried out). But reverse operations are only partly idempotent, and not at all when the system crash occurs inside a DB operation. The logical consequence to this, is that one has to do logging also during the recovery procedure. But it's a nonsense first to carry out the reverse operation of a traced reverse operation, and then that operation itself. So the crash recovery procedure must know which log-records have been correctly handled during the preceding recovery procedure.

   For instance, the crash recovery procedure could process only the traced pages, and then compare the log of the previous recovery actions with the original log, till it reaches the last correctly

handled recovery action. The original log can then be normally handled.

A much simpler strategy (still efficient) is to copy the current Database before doing crash recovery and to mark somewhere that from now on we are doing crash recovery. Then if another crash occurs during this recovery operation, we only have to copy back the saved Database and to redo all the recovery procedure as if nothing happened.

3.5.5 Reverse operations

The DB operations for which the recovery is necessary can be divided in four categories.

1. DB operations that can be reversed by only one reverse operation of which all parameters are known at least, at the end of the operation.

Ex : Dbbgtr, Dbendtr, Dbabtr, Dbcreate, Dbinsert, Dbremove.

2. DB operations that can be reversed by only one reverse operation of which parameters must be partly calculated.

Ex : Dbmodify.

3. DB operations that must be reversed using more than one reverse operation of which parameters must be partly calculated.

Ex : Dbdelete.

Remark :

Begin and abort transaction operations need reverse operations which constitues a proof of consistency at the end of the recovery action (as for a "normal" transaction).

Section F :

Further developments
And Conclusion

1   Multi-user support and server functionality

The first enhancement for the DBMS we could think of is to expand it from a single-user to a multi-user system. This is made easier by the fact the Kernel primitives are not much aware of the objects they manipulate. That is, all the information peculiar to a given application is stored in the application memory, and the DBMS only has a pointer towards the DDB containing the information it needs to operate on the proper data.

Therefore, we can easily think of a server-like system, that would be called by several application programs using interrupts to request some operation. The calling process would only have to pass the pointer towards its own DDB and information for the operation to be carried out by the DBMS. This is a major improvement, that would require among others, implementing a lock manager to regulate concurrency between the various calling processes.

2   Dynamic schema management

CAD/CAM data is characterized by its dynamic schema. As seen previously, a schema defines the allowable structures for data instances and is generally viewed as a static collection of data types. The data types represent attributes, entity types and relationship types of the application being modeled. CAD/CAM data differ from business-oriented data because the structure of CAD/CAM data actually grows with the design of the artifact and therefore cannot be completely defined in a static schema. At each manufacturing phase, schema specification is interleaved very closely with the construction of an object [Camm87].

Most of the DBMS adhere to a static schema definition. Generally, schema definition and generation are expensive off-line tasks. The enormous overhead for database reconfiguration due to schema modifications prohibits the practical use of an interactive

dynamic schema in most existing systems. Thus, the desired structure of the entities to be represented is completely defined at schema definition time and cannot be subsequently modified. An increasing number of DBMS are allowing schema revision to the extent that revisions are upwardly compatible with the existing schema, and previously loaded data. However, this limitation still requires a user to make a strict distinction between schema definition and data specification, and the operations for performing each.

The dynamic quality of the schema and data requires some highly flexible and interactive user facilities. Users need to view or navigate the database dictionary, and dynamically modify the schema. They need to retrieve schema information such as the attributes which are defined for a given entity type or the relationship types that have been described. Corresponding operations for modifying the schema and the data should also be available. These facilities include initial generation of a new schema when designing a new part, and interleaving the schema definition with the storing of specific data values.

The construction of such facilities entails the definition of the structure of the schema and data (from a user's viewpoint) and user operations for viewing and manipulating the schema and the data. These have already been realized within this dissertation. We would have now, to enable the system to do dynamic compilation and modification of a schema stored in the data dictionary. But this would be for another dissertation yet to come.

3   Conclusion

We have designed a very complete DBMS on basis of specifications driven from a thorough requirements analysis. When developing each part of the system, we have always kept in mind three major objectives. First of all, to comply with the needs of the upper E/R layer. Secondly, to keep as much compatibility with the NDBS

tools as possible. And finally, to always look for more performance, as it is of major importance for the target application domains.

In conclusion we can say that PYRAMIDE is a first step towards developing a much more powerful system. It already offers interesting features, such as its embedded data dictionary, that could be enhanced in the next future.

Literature

[Baye72]   Bayer, R.; McCreight, E. : Organization and
maintenance of large ordered indexes. Acta
Infomatica 1, 173-189, Springer Verlag 1972.

[Boda83]   Bodart, F.; Pigneur Y. : Conception assistée des
applications informatiques, 1- Etude d'opportunité
et analyse conceptuelle. Masson éditions &
Presses Universitaires de Namur, 1983.

[Camm87]   Cammarata, S.J.; Melkanoff, M.A. : An interactive
data dictionary facility for CAD/CAM databases.
Proceedings from the first international workshop
on expert database systems. Benjamin/Cummings
publishing company inc., 1987.

[Chen76]   Chen, P.P.-S. : The Entity/Relationship Model -
Towards a unified view of data. ACM Transactions
On Database Systems, Vol 1, N°1, March 1976.

[Date83]   Date, C.J. : Introduction to database systems,
Volume II, Addison-Wesley 1983.

[Depp80]   Deppe C.; Bartholomew, A. : B-Tree ISAM concepts.
Dr Dobb's Journal, Number 80, P. 289 - 292,
June 1980.

[Ditt86-a]   Dittrich, K.R. : Object-oriented database systems:
The notions and the issues. Proc. ACM/IEEE Int.
Workshop on object-oriented database systems,
1986.

[Ditt86-b]   Dittrich, K.R.; Kotz, A.M.; Mülle, J.A. : Database
support for VLSI design : The Damascus system. In
CAD-Schnittstellen und Datentransferformate im
Elektronikbereich, ZGDV-Buchreihe "Beiträge zur
Graphischen Datenverarbeitung", Springer Verlag
1986.

[Damo86]   Gotthard, W. and al. : Damokles : das Datenmodell
               des Unibase - Entwicklungsdatenbanksystems,
               Verbundprojekt Unibase. Projektbericht, FZI,
               Karlsruhe, März 1986.

[Damo87]   Damokles Projekt, Datenbankunterstützung für
               Software-Produktionsumgebungen.   In Technischer
               Bericht des FZI - 1987.

[Damo88]   Damokles, Reference Manual of Release 2.0.
               Forschungzentrum Informatik an der Universität
               Karlsruhe, March 1988.

[Doug79]   Douglas, C. : The ubiquitous B-Tree.   Computing
               surveys, Vol.1, N°2, June 1979.

[Fauv88]   Fauvet, M.C. : ETIC : Un SGBD pour la CAO dans un
               environnement partagé.   Thèse de Doctorat USTMG de
               l'Université Joseph Fourier - Grenoble 1,
               Spécialité Informatique, 1988.

[Gott86]   Gotthard, W.; Dittrich, K.R.; Lockemann, P.C. :
               Datenbanken In Software - Produktionsumgebungen :
               das Projekt Damokles und sein Entwurfsobjekt -
               Datenmodell. Proc. GI-Fachtagung "Die Zukunft der
               Informationssyteme", LINZ, Springer, 1986.

[Hask88]   Haskin, R. and al. : Recovery management in
               QuickSilver. ACM Transactions on Computer
               Systems, Vol. 6, N° 1, February 1988, pages
               82-108.

[Hain86-a]   Hainaut, J-L. : Conception assistée des
               applications informatiques, 2- conception de la
               base de données. Masson éditions & Presses
               Universitaires de Namur, 1986.

[Hain86-b]   Hainaut, J-L. : Technologie des fichiers. Notes
               de cours, Institut d'Informatique des Facultés
               Universitaires Notre Dame de la Paix à Namur,
               1986.

[Hain87]   Hainaut, J-L. : NDBS - a simple database system
               for small computers. Document interne, Institut
               d'Informatique des Facultés Universitaires Notre
               Dame de la Paix à Namur, 1987.

[Härt87]   Härtig, M. : Feinplanung und realisierung eines
               "Internal Object Managers" für das Datenbanksystem
               Damokles. Diplomarbeit - Institut II der Fakultät
               für Informatik - Universität Karlsruhe, 1987.

[Kemp87]   Kemper, A.; Wallrath M. : An analysis of Geometric
               Modeling in Database Systems. ACM Computing
               Surveys, Vol. 19, N° 1, March 1987.

[Knut73]   Knuth, D.E. : The art of computer programming,
               Vol3, Sorting and searching. Addison-Wesley
               Publishing Company, 1973.

[Lams87]   Van Lamsweerde, A. : Méthodologie de développement
               de logiciels. Notes de cours, Institut
               d'Informatique des Facultés Universitaires Notre
               Dame de la Paix à Namur, 1987.

[Lock83]   Lockemann, P.C. : Analysis of version and
               configuration control in a software engineering
               environment. In Entity-Relationship approach to
               software engineering. Elsevier Science Publishers
               B.V. (North.-Holland).

[Lock85]   Lockemann, P.C. and al. : Anforderungen
               technischer Anwendungen an Datenbanksysteme. In
               Blaser, A.; Pistor, P. (eds) : Datenbank - Systeme
               für Büro, Tehnik und Wissenschaft, GI-Fachtagung,
               Karlsruhe, Informatik - Fachberichte 94, Springer,
               1985, page 1-26.

[Obri88]   O'Brien, S.K. : Turbo Pascal - Advanced
               programmer's guide. Borland-Osborne/McGraw-Hill,
               Programming series, 1988.

[Ranf88]   Ranft, M. : Recovery für das Datenbanksystem
               Damokles. Technischer paper - Forschungzentrum
               Informatik an der Universität Karlsruhe, 1988.

[Ross87]   Rossi, D. : NDBS - Primitives de base du SGBD,
               Travail Personnel de première Licence et Maîtrise.
               Institut d'Informatique des Facultés
               Universitaires Notre-Dame de la Paix à Namur,
               1987.

[Turb88]   Turbo Pascal, User's guide and programmer's
               reference. Borland, 1988.

[VERH78]   Verhofstad, J.S.M. : Recovery Techniques For
               Database Systems. Computing Surveys, Vol.10, N°2,
               June 1978.