The physical machine of NICEBD, an object oriented DBMS

C. SALGADO, N. LE THAN, E. VITTORI, J. T. DONG, G. MOPOLQ, S. MIRANDA

BAOU project, University of NICE-CNRS, LISAN, Rue A. Einstein, bat. 3, 06560 Sophia-Antipolis - Valbonne, FRANCE

ABSTRACT:
This paper presents the NICEBD internal level, called physical machine. NICEBD is an object oriented DBMS which has four main goals: persistent complex object management, deduction taking in account, multi-device storage and multi-user environment. This physical machine, called NICEMP, implements logical independance, in the sense it can be used for supporting any logical data model. On the first hand NICEMP is an object manager and on the other hand a set of tools for complex system development. Object oriented approach underlies its design and implementation.

KEY-WORDS:
Object-oriented DBMS, complex-object, deduction mechanism, multi-media database, object programming.

I - INTRODUCTION

The NICEBD project is a seven year research program (1985-1992) of Nice University focused on object oriented programming and data base systems. It meets several research topics:
- persistent object management system,
- object oriented data model,
- design of object oriented multi-media interface,
- data model implementation and object programming,

The use of DBMS in different application fields (CAD, office automation, ...) leads to the definition of new logical data models and consequently new DBMS, having a physical kernel able to support them. In recent literature, physical systems called object managers [BENZ88], [PUNCH86], object servers [HORN87], or simply object-oriented systems [BANE87] were proposed. The NICEBD physical machine is part of this research stream; its salient features are detailed in this article.

This paper summarizes some researches and developments made within the NICEBD prototype framework during the last two years. In section I, we give a general overview of the NICEBD system along with its main goals. Then we will concentrate on the description of its physical machine NICEMP, an object oriented DBMS kernel. Section III presents the NICEMP architecture and advantages. Section IV deals with persistent and temporary object storage performed by two system managers: the segment and virtual table manager and the device manager. Section V presents the tools used for object handling; these tools are gathered into four system managers: the virtual memory manager, the type manager, the path manager and the virtual variable manager.

II - PRESENTATION AND OBJECTIVES OF NICEBD

NICEBD is a multi-media DBMS based on an object oriented extension of the CODD's relational model [CODD70], the B-relational model [LETH86], [VITT87], [VITT88], [MIRA88] ... In order to answer the main objectives of the NICEBD system, it has been necessary to integrate some concepts which stem from various research domains:
(i) Software Engineering: basic concepts of abstract data types and object oriented languages, for complex object management and dynamic data types definition.
(ii) Artificial Intelligence: rule management and theorem prover for deduction.
(iii) Operating Systems: failure recovery mechanisms, serialization and concurrency control in a multi-user environment.
(iv) Data Base Systems: efficient paged memory management.

In addition to these goals we can also point out the multi-device aspect of our system. Since NICEBD is a multi-media system, we deal with the manipulation of various data types like texts, pictures, sounds which require specific device utilization (videodisc, WORM optical disc, CD-ROM, digitaliser,...) in an uniform manner.
Figure 1 shows the NICEMP general architecture. NICEMP encompasses four levels called machines which are object-oriented (both in their design and implementation): 

(i) the PHYSICAL machine represents the system internal level (DBMS kernel);
(ii) the LOGICAL machine able to support various logical data models and moreover different data processing models; these models may be seen as several independent software levels above the physical machine;
(iii) the TRANSACTIONAL machine which manages synchronization in a multi-task or network environment, implements data security mechanisms (acces control, integrity) and performs system optimization;
(iv) the INTERFACE machine that includes all the user interfaces provided by the system : B-SQL [LETH86], hypermedia interface [MIRA89], deductive interfaces, graphical interfaces, ...

III - NICEMP GENERAL ARCHITECTURE

Before beginning the NICEMP architecture description, let us point out the "persistent complex object" concept: A complex object [VITT88] is defined as an identified object that can be handled either as a whole thru specific operators (encapsulated object in programming-laiquage sense) or thru generic operators (structured object in the database sense). A persistent complex object is a complex object stored in persistent memory associated with a physical identifier and a physical structure.

The physical machine, called NICEMP, represents the NICEMP system internal level. It is an object oriented system which has three major functions:

(i) the storage of persistent and temporary objects that consist in managing the different device types as well as the system storage logical organization,
(ii) the complex object restructuration in volatile memory which includes the three following aspects:
   * virtual memory management allowing dynamic and efficient space allocation for the objects to be built,
   * types and type constructor management for the object interpretation,
   * data structures and constructors allowing navigation in the base to find persistent objects necessary to rebuild any "complex" object to be processed in the volatile memory,
(iii) the uniform interface provided to the NICEMP system user for the encapsulation and the manipulation of complex objects built inside the volatile memory.

The NICEMP architecture is an object oriented architecture. NICEMP is by the one hand an object server [HORN87], and on the other hand a tool box allowing complex systems development. It represents the internal level of the DRMS in charge of object storage and physical acces as well as concurrency and failure recovery. Moreover, it provides virtual memory management mechanism not only for data but also for working areas, called contexts, used for the development of higher system levels. As an object server, NICEMP perform the "system object" management. A system object is defined as a basic component of the system represented by a persistent complex object. We can point out six system objects: devices, segments (virtual tables), memory stocks, data types, access paths and virtual variables. It is thus composed by six distinct system managers that communicate among them.

Figure 2 presents NICEMP general architecture. Every NICEMP manager performs the management of only one system object type and its associated methods. We can classify them into two groups according to the task they deal with:

1. OBJECT STORAGE

   -The Device Manager which performs the management of the different device types connected to the system and so allows new device integration. It guarantees the physical storage of system objects.

   -The Manager of Segments and Virtual Tables which performs the management of:
     * segments allowing logical storage of data and system objects;
     * virtual tables allowing temporary working areas management.

   This manager includes conversion functions that allow logical/physical mappings of object addresses. Object storage or access in segment/virtual tables is done by their physical object identifier (surrogate) called POI. POI consist of the segment or virtual table identifier and the object identifier inside the segment or virtual table.

2. OBJECT PROCESSING

   -The Virtual Variable Manager which performs the management of the working areas called contexts. It provides a set of software tools allowing the manipulation of objects extracted from persistent memory and/or temporary objects created by the user inside a virtual buffer area. The external interface for complex objects manipulation is based upon the virtual variable concept (see V.4).

   Figure 2 : NICEMP ARCHITECTURE

   -The virtual memory manager which establishes the link between the storage level and the object processing level. It dynamically manages volatile memory in order to avoid saturation during simultaneous manipulation of several objects, data and contexts. Therefore, the virtual memory includes two parts:
     *persistent virtual memory associated with data management,
     *dynamical virtual memory associated with context management.
The two others managers that perform object processing are related to predefined constructors. A constructor is a tool that allows to define the system basic objects; it authorizes object recursive definition and therefore permits to define hierarchical objects.

- The Type Manager which performs management of predefined basic data types (integer, real, character, boolean, date, POI) and predefined constructors (tuple, set, stack, array, list, range, union). This manager ensures also the control of new data types definition.

- The Path Manager which performs management of the system different access paths. The access paths allow to navigate within the data structure and to express inheritance links between objects. These two aspects are fundamental for the execution of optimization and deduction mechanisms. Several constructors (access methods) can be associated to the path manager: B-tree, hashing, A-graph, mult-list ...

These functionalities allow an implementation of the complex object concept in NICEMP. The double dimension "structured object" and "encapsulated object" is treated in NICEMP as shown in figure 3. NICEDB is therefore a "fully object oriented" DBMS according to K. Dittrich's definition [DIT88].

Figure 3 : PERSISTENT COMPLEX OBJECTS IN NICEMP

IV - OBJECT STORAGE MANAGEMENT

Concerning data storage, two main aspects are pointed out: the data physical storage with multi-device management ensured by the device manager and the data logical storage based upon the segment and virtual table concepts.

IV.1 - Device Manager

NICEMP authorizes several storage and communication device type connections (until 256 in the present MS-DOS version). Our Goal is to manage different device types connected to the system and to allow dynamic device dynamic integration. We take into account not only the input/output devices but also the storage devices that can be classified in two categories:

(i) numerical devices on which informations are stored in the form of digital data.

(ii) non-numerical devices: all the other kinds of device.

The device manager is in charge of the physical storage of system objects. It allows the definition of a new device in the system and the suppression of a system device.

The definition of a new device type includes its characteristics and the input/output specific associated operations. Before introducing the device definition in detail, let us point out a concept which will be useful for the understanding of notions connected to physical and logical data storage. A physical object is defined by a set of coded informations addressable on a device.

A device is considered as a set of cells that permits persistent storage of physical objects. It represents the storage physical unit. As a general rule, a device is characterized by:

(i) an identifier,
(ii) the smallest area addressable on the device,
(iii) a transfer unit: input/output atomic part addressable by the system,
(iv) a set of specific input/output operators.

Our first idea was to standardize the management of any device connected to the system. But this approach included several constraints. Actually, according to the type of device, characteristics are very different, mainly those concerning input/output operations. This is why a device definition includes its specific characteristics along with associated input/output operations.

For example, the operations associated to a videodisc player may be:

- random access to a picture,
- variable scrolling thru the pictures (slow and fast / forward and reverse),
- audio handling,
- access to the current picture number,
...

We call "main device", the device the system is defined upon; it is considered as a predefined device.

IV.2 - Segment and Virtual-Table Manager

This manager is responsible for the object logical storage. This storage is permitted thru logical identification references - called POI (Physical Object Identifier) - to physical objects stored on the associated device. POI may be seen as a basic predefined type identifying the object during its manipulation in the whole NICEMP system. It is settled by two logical structures:

- segments for the storage of persistent data associated with the system different databases,
- virtual tables for the temporary data storage.

We are going to present them in turn.

IV.2.1 - Segment Concept and Storage Management

A segment represent the storage logical unit for persistent data. It is composed by a set of POI and characterized by:

(i) an identifier,
(ii) an associated physical device,
(iii) the smallest addressable area in the segment,
(iv) a transfer unit: segment logical division,
(v) processing unit: elementary information processing unit.

For segments associated with some devices (videodisc for instance) this characteristic is not applicable,
(vi) addressing mechanisms: they include a couple of bidirectional logical/physical mapping functions that ensure the interface between physical and logical data storage.

We distinguish two kinds of segment according to the associated device type: pageable segments and non-pageable segments.

**Pageable Segments**

Pageable segments are associated with numerical devices. In a pageable segment, informations are stored in digital form and may be broken down into elementary units in order to facilitate their transfer and manipulation in volatile memory. Due to this splitting, we can point out two other concepts associated with pageable segments:

(i) the **Block** concept (related to the device): physical unit of data transfer to volatile memory; its size is a system parameter;

(ii) the **Page** concept (connected to the segment): logical unit of data transfer composed with a set of contiguous blocks. The number of blocks in a page is variable.

A pageable segment consists of a set of (unnecessary) contiguous and variable-sized pages. Pages are connected together in a segment with a permanent logical order. We call logical address of a page in a segment its ordering number in the ordered set of associated segment pages. This address is fixed. The page physical address refers to the physical page cell on the associated segment device it belongs to. This logical address varies dynamically with the pages updates.

**Non-pageable Segments**

They are segments associated with non-numerical devices. In such a segment, informations are not stored in digital form. Consequently, they can not be split up and can just be processed as a whole. A non-pageable segment is stored on its associated device, outside the reserved space of the database it belongs to. Let us take for example the associated segments of a videodisc like device: an associated non-pageable segment represents a set of picture references. As for pageable segments, addressing concepts are also defined.

A non-pageable segment consists of a set of elements (picture references in the videodisc case). We call logical address in a non-pageable segment its ordering number in the ordered reference set the segment is composed of. The element physical address refers to the physical cell information on the associated device of the segment it belongs to. This logical address varies dynamically with the pages updates.

**Data Base Composition**

A NICEMP database is a storage area managed by the DBMS itself. It is composed with a segment set identified as follows:

(i) **System Segments** related to the database control informations (blocks and tables); these segments are necessarily pageable ones.

(ii) **Main Data Segments** corresponding to the database and defined inside the area reserved for the database by the DBMS. These segments are also necessarily pageable ones.

(iii) **Annex Data Segments** also related to data corresponding to the database but defined outside the reserved database area. They can be pageable or non-pageable ones.

Therefore, a database is inevitably composed with pageable segments (system segments and main data segments), along with data annex segments which can be non-pageable or pageable. The concept of data annex segment allows us to talk about an open DBMS. Actually, it permits the connection between a database and an external DBMS environment, and the integration of various device types. Annex segments can then be controlled by other access methods independent from those which are defined inside the DBMS. If they are related, for example, to a videodisc or a file defined by the file management system, they are then to be controlled by the corresponding specific system methods. Figure 4 presents the integration of multi-device aspect and the persistent memory organization.

Data logical storage is only related to pageable segments. As such, it permits:

(i) new segment creation in the database, 
(ii) segment suppression, 
(iii) addition, modification and suppression of an element in a segment.

An element in a segment is viewed as a variable-sized byte string; this size may be changed during update operations. Each object stored in a segment is identified by a POI.

Data logical storage is only related to pageable segments. As we mentioned previously, a pageable segment is composed with a set of pages which consist of a set of contiguous blocks.

**IV.2.2 - Virtual Table Concept**

NICEMP allows managing both persistent data storage memory and virtual working memory. In an object programming and management environment, management of virtual dynamic memory is crucial [WILL87]. Actually, the storage and working structures of complex objects are generally different and therefore require mappings during

---

**Figure 4**

**UNIFORM MANAGEMENT OF DIFFERENT DEVICE TYPES AND PERSISTENT MEMORY PARTITION**

---

**Figure 4**

**V.2.2 - Virtual Table Concept**

NICEMP allows managing both persistent data storage memory and virtual working memory. In an object programming and management environment, management of virtual dynamic memory is crucial [WILL87]. Actually, the storage and working structures of complex objects are generally different and therefore require mappings during
A virtual table represents the logical unit of the system working virtual memory overflow area. Contrary to the segment management mechanism, virtual-table management requires simplicity in order to meet the goal of efficiency. First, the shadow mechanism [LORI77] cannot be used for virtual tables. Furthermore, allocation units used for virtual tables must be related to the smallest allocation size of volatile memory.

A virtual table encompasses some fixed-sized elements themselves composed with a set of cells - the smallest volatile memory reservation unit for a working area (64 bytes in the present MS-DOS version). As for segments, each virtual-table element is related to a physical identifier (POI) which is permanent throughout table life.

This table management is performed by the following functions:

(i) creation of new virtual tables,
(ii) suppression of virtual tables,
(iii) granting an element in a table,
(iv) loading of an element in a table,
(v) revoking an element in a table.

Virtual tables take part in bringing into play the mechanisms of dynamic virtual memory (context stacks) allocation and management mechanisms. These aspects are detailed in section V.1.

V. COMPLEX OBJECT MANIPULATION

In this section, we present the four managers in charge of the following objects: virtual memory, types, paths and virtual variables.

V.1 - Virtual Memory Manager

In DBMS, the use of persistent memory is absolutely necessary to allow the management of large amount of data. During a database manipulation, data must be loaded in volatile memory and saved in persistent memory after modification. The virtual memory manager performs data continuous exchanges in order to hide the limited size of volatile memory. Everything behaves just as if the user had a very large volatile memory, able to contain the whole database: the virtual memory.

As a general rule, virtual memory management requires development of stack management mechanisms performing data transfers between persistent memory and volatile memory. Different kinds of mechanisms have been proposed in literature: LRU, LFU, FIFO, CLOCK, ... [EFFE64]. A thorough study of these methods points out LRU ("Last Recently Used") types algorithms as the simplest and, consequently the most used. Our approach implements a modified form of this method: the DLRU technique detailed below.

V.1.1 - The DLRU (Double LRU) Technique

One of the problems not solved by the LRU technique is the lack of preventing a page from "going out" of the stack if we know that it will be often swapped in memory during a given transaction processing. As the stack capacity in page number is limited, it is not really desirable to see some of them going out of the working area before the end of the processing. In order to solve this problem, some commands were defined. They enable:

(i) to lock a page in a stack;

In order not to alter the page replacement mechanism, locked pages are not considered as being in the stack. This operator is useful when loading pages which are to be modified.

(ii) to unlock a locked page;

It is the locking inverse mechanism; unlocked pages will be put back into the stack and handled by the replacement mechanism.

(iii) to remove the top page into the bottom of the stack;

This operator is useful when we know in advance if the given page will not be used anymore during the current processing of a transaction.

The control of these functions is in charge of the user. The system operates at the end of the working session in order to free the pages (locked or not) that have been loaded exclusively by the session.

The implementation mode of these mechanisms in NICEMP presents a particular characteristic. There is a control stack named the Stack of Stacks (justifying the term "Double LRU") whose purpose is to apply the LRU mechanism to the stacks themselves (cf V.1.2). Actually, the stacks are not part of volatile memory (except the stack of stacks); they are loaded and saved dynamically in persistent memory. This permits to optimize the occupied space in volatile memory and to improve virtual memory mechanism performance [ERNS87].

V.1.2 - The Two Types of Memory Stacks

In NICEMP system, volatile memory encompasses two parts (cf II).

(i) Persistent Data Memory: it represents the loading and saving areas for persistent object in volatile memory. It is directly related to data segments and used for data page exchanges coming from different databases. Such memory is managed by data stacks that constitute the data virtual memory.

(ii) Working Dynamic Memory: it provides the working areas allocated dynamically by a process. It is related to the virtual tables in order to constitute working virtual memory. This memory management is done thru the context stacks.

For example, let us assume there exists a persistent object called TUPLE. For the tuple (instance of TUPLE) object manipulation, two stacks may be defined:

- the TUPSTACK stack, a data stack in which the tuple values saved in data segments will be loaded;
- the TUPVTAB stack, a context stack in which will be loaded the working areas necessary to the tuple manipulation.

As a general rule, data stacks are universal, that means a data stack may be used for different database object types. In the opposite, context stacks are often either specialized according to the processed object type, or private for a given transaction. If data stacks come from a DBMS environment, context stacks allow the integration of NICEMP in a programming environment.

V.2 - Type Manager

The type manager performs management of data types at the system internal level, regardless of the logical model
implemented above. Rational for this manager lies in the ability of:
(i) defining new dat types,
(ii) specifying and implementing operators on this new types,
(iii) making easier new data model definition.

The type manager is fundamentally based upon abstract data type theory - ADT - [GUTT76], [CHAB82], ... and object oriented programming languages [COPE84], ... The following properties may be pointed out:
- simple or multiple inheritance of types and associated operator properties organized in class hierarchy;
- encapsulation making one type visible only thru operators defined upon;
- type parametrizing justified here by the fact that constructors are applicable on the POI and so independent from the datatypes;
- renaming, restriction and redefinition of type operators.

This manager provides the following operations:
(i) creation of new types inside the system,
(ii) suppression of types inside the system,
(iii) control of a value type,
(iv) addition of an operation associated with a type,
(v) modification of an operation associated with a type,
(vi) suppression of an operation associated with a type.

Dynamic definition of new data types is performed by using generic existing types and type constructors. System basic predefined types are: integer, real, character, boolean, POI (Physical Object Identifier), method and date. Basic type POI is the storage identifier of all the system objects. It represents the link between the segment and virtual table manager, and the other managers. The method type represents the unique identifier of the running code (stored in a specific segment), of the procedures that implement operators defined on these types. This type is associated with operators like load, run, ...

**Figure 5**

Constructors associated with the type manager are: tuple, set, list, stack, range, array and (type) union. Constructor elements are basic types, represented by POI. Since the POI are independent from types, basic operations defined upon constructors are applicable on any derived type or object. We can recursively define new types from basic types and constructors. Figure 5 describes every basic constructors and their associated methods.

**V.3 - Path Manager**

The purpose of this manager is threefold:
(i) it permits the representation of links between physical objects coming from one or several different segments,
(ii) it uses these links to have access to objects according to various criterias,
(iii) it provides a way to perform optimization and inference mechanisms inside the DBMS.

One of the path manager functionalities is to allow implementation of new access paths (logical or physical) necessary to object manipulation inside the DBMS. In a complex type real world, traditional access methods (hashing, B-tree, ...) may be inadequate. For example, for three-dimensional objects, R-trees [GUTT84], [SELL87] are more suitable while for a "text" type B*-trees [CARE86], will be a well adapted mechanism. In addition to new access path implementation, the path manager permits insertion and research of object occurrences thru available access paths.

In NICEMP, this manager is in charge of the management of two system-object types:
(i) index: it is a set of data structures stored on some segments and allowing access to physical objects belonging to another segment;
(ii) path: it is either an index or a logical composition of some other paths according to the composition rules of an access path algebra.

Index are called elementary paths and paths built from other paths are called generalized paths.

Paths may be built from several constructors provided by access methods like: B-tree, A-graph, Multi-list, ... Each access method has its own constructor and a set of functions for the management and use of its data.

An index may be built by composing several access methods. In the present version of NICEDB, three kinds of index have been defined:
This mechanism allows defining inference in a DBMS (organized by traversal) mechanism in the given index tables in order to have paths for rule graphs. The navigational mechanism acts like a virtual access to physical objects regarding some qualification criteria.

The path manager integrates a navigational (graph traversal) mechanism in the given index tables in order to have access to physical objects regarding some qualification criteria. This mechanism allows defining inference in a DBMS [LETH84]. For example, using artificial intelligence terminology, we can use index to describe graphs of facts and paths for rule graphs. The navigational mechanism acts like a theorem prover of the system. The full specification of the path manager is under current research.

### V.4 Virtual Variable Manager

The objective of the virtual variable manager is to provide a set of software tools allowing to handle with objects extracted from persistent memory and/or temporary objects created by the user in a working virtual memory area. First, it automatically manages the space of virtual memory reserved for every variable declared by the user according to the object size associated with this variable. Actually every declared variable is associated with one or several context memory areas allocated in the dynamic virtual memory area.

The DLRU management mechanism insures the availability and the security of data associated with declared virtual variables. Then it provides a way to dynamically handle objects through associated types. At any moment, it is actually possible to associate a value and a type to a virtual variable. Value may come from outside (through an interface) or from inside extracting a physical object from the persistent memory and the type has to be known by the type manager.

A virtual variable is said to be in the **associative mode** if its value corresponds to a persistent object extracted from the database. Every modification made on the object can be automatically saved in persistent memory upon user request. The virtual variable type said to be in the associative mode if it corresponds to the type associated with the concerned physical object. Type control is automatically made by functions provided by the type manager. Associated object may be a system object (managed by other managers) or an object stored as data in the database.

A virtual variable is said to be in the **free mode** if it is associated with temporary working variables defined by the user. The type of a virtual variable in the free mode must be a type known by the type manager; the virtual variable manager provides tools in order to define programming techniques in a multi-task environment.

A free virtual variable associated with a set of contexts is said to be in the **collective mode**. Each context can represent a task descriptor. The task is represented by a context activated when a virtual variable selects this context. The collective form is a generalization of the free one.

In the same way, the **selective mode** is a generalization of the associative one, in case the reserved contexts are associated with persistent objects in the database.

The fact that a variable may be associated with different contexts allows to define the concept of versions of an object during a long transaction (transient versions) as well as different techniques for selective object handling.

System objects managed by this manager deals with virtual variables. A virtual variable has the following main characteristics:

- **name**,
- **mode of use**, 
- address of the reserved virtual memory area,
- address of the corresponding physical object (in the associative mode),
- rank of the associated type,
- address of the associated context (in the selective or collective mode),
- ...

Some operators are provided by this manager:

- The "Designation" operator allows creation of an associative-type variable and the association of this variable to a predefined data type. A designation is a triplet:

  `<NAME, POI, C-ADR> where
  NAME : variable name,
  POI : associated physical object identifier,
  C-ADR : associated context address.`

- The "Assignment" operator allows the creation of a variable of free mode and the association of this variable to a predefined data type. An assignment is a triplet:

  `<NAME, T-ADR, C-ADR> where
  NAME : variable name,
  T-ADR : associated type address,
  C-ADR : associated context address.`

- The "Scheduling" operator allows the creation of a variable for selective or collective type and the association of this variable to a set of n contexts. A scheduling is a tuple:

  `<NAME, n, T-ADR1, C-ADR1, ..., T-ADRn, C-ADRn> where
  NAME : variable name,
  n : number of elements in the set of contexts,
  T-ADRi : type address (POI-selective mode-) associated with the ith context, (i=1..n),
  C-ADRi : address of the ith context, (i=1..n).`

The full specification of the virtual variable manager is beyond the scope of this paper.
IV - CONCLUSION

In this article, we concentrated our description of NICEBD prototype in its physical machine. The central idea in designing and implementing the physical machine is the decomposition of the architecture in several system objects. This way of thinking leads us to dissociate from the layered design of relational DBMS architecture we defined. In NICEBD, everything which is implemented is an object; everything the user sees and manipulates is also an object. The same philosophy was chosen for the interface machine. We can sum up how NICEMP can be a support for implementing persistent encapsulated and structured objects:

- encapsulated object inheritance is taken into account by the type manager;
- structured objects and structures operators (of the algebra type) are handled thru both the type manager at logical level and the path manager for physical structure implementation.

We have made some positive performance benchmark on a simplified form of NICEMP physical machine in a mono-user environment [ERN87] compared to a relational physical machine (both in a fail-safe and unsafe medium). These figures have to be repeated with NICEMP to validate our system as soon as we implement a logical machine.

A first MS-DOS version of NICEMP, written in C language, is operational since December 1987. This version was used for the set-up of a relational documentary DBMS on PC/AT. The realization of a version on UNIX operating system is in progress.

REFERENCES


