D1-T3.2.2-890401

ESPRIT Project n. 834.

COMANDOS

Construction and Management of Distributed Office Systems

ODMS Final Design

Istituto dell'Elaborazione dell'Informazione
v. S. Maria 56100 - Pisa
Italy

E. Bertino
R. Gagliardi
M. Gamboni
G. Mainetto
F. Marinaro
F. Rabitti
C. Thanos

April 1989

Document Structure

The present documentation upgrades what has been already presented in the last Comandos deliverable document on ODMS [ServDes88] . Many concepts have been enhanced or specified in more detail. In particular sect. 1.2 emends ODMS data model features. Sect. 1.3 revises the query language and presents relevant updates to it. Sect. 2 illustrates the abstract specifications of ODMS's internal components. Index Management is detailed in sect. 2.3.1 while the Query Processor abstract specifications is presented in sect. 2.4. The Query Processor specifications have been used by ARG for implementing the CIS prototype. Sect. 3 lists some of the ODMS dictionaries. Finally sect. 4 and apendix give implementation level specification.

1 ODMS concepts

1.1 Overview

In order to simplify the understanding of the contents of the following chapters, we would like to recall here the ODMS architectural framework.

The overall objective of the ODMS is to provide object management facilities for advanced applications in LAN based environment. An important component of the overall architecture is the **Object Store**, which represents the whole set of objects within the system. The object store is distributed across server and client workstations. We call one of such sites as **Object Store Nodes**.

In the present design the ODMS plays a passive role with respect to the overall Comandos computational model. In particular there is no notion of message passing or behavior at the ODMS level. Therefore Objects migrates (at least logically) from the Object Store to the sites where they are used. However ODMS Objects react to specific operations that are related to the ODMS three major services:

- Management of classes. To accomplish this task the **Class Manager** provides a uniform interface to access classes, even when there are different implementations of the same class at the various Object Store Nodes.
- Distribution of class instances. This service is provided by an internal component of the ODMS, called **Placement Manager**.
- Associative object access. A **query language** provides content based access to objects and it is supported by the **Query Processor**. This last component uses **Index Manager** facilities for performing fast object. The Index Manager has a bigger task in ODMS, that is it supports also object reconstruction.
- The **Resource Manager** manages resources in terms of usable space for creating, instantiating and destroying classes.

Executions of Comandos applications follows a generic client/server model: the client part of ODMS, on behalf of the application, requests services to the server part of ODMS. In terms of activities, one can think of the client/server model as an activity residing on one client node that at a given time processes the methods of the client_ODMS instance in order to request a service to the ODMS. When a service is requested to the client_ODMS for the first time, the activity diffuses to the server node [ServDes88].

3) For a component property, we can specify the denoted object(s) either as dependent or independent. An object is dependent when its existence depends on another object. This means that when one object is deleted, all its dependent component(s) are also removed from the database. Notice that the model does not allow any cycle in the path constituted by the dependent component links, at instance level; that is, each PART_OF object subgraph, defined by dependent component links, is a tree. The concept of dependent component can be defined both for single and multi_valued components.

4) We can give a default value to an attribute property. This is the value given to this attribute when the object is created, if the value is not specified at instance creation time.

5) A property (both an attribute and a component) can be defined as constant or variable. A constant property cannot be modified once a value is explicitly assigned. A multi_valued component cannot be declared as constant.

6) A property (both a component and an attribute) can be defined as optional or mandatory. The value of a mandatory property must always be specified when the object instance is created. The value of an optional property can be left null. Both a single and multi_valued component can be declared as mandatory.

7) For a component property it is possible to specify the inverse component property. Assuming that we define the property pc1 of a class C2 as the inverse component of a component property pc2 of a class C1, if an instance object O1 of class C1 has a value O2 for the property pc2, then the instance object O2 must have a value O1 in the property pc1. This capability is available for single and multi_valued components.

1.2.2 Class Definition

In the sequel, we will give a BNF specification of the syntax for defining class in the Data Model. This syntax will be used to implement an operation of the Class Manager that creates a class template and that has a single parameter which is a string belonging to the language generated by this grammar. It is worth noting that the actual syntax does not constrain the number of classes that can be defined on a type.

Syntactic conventions:

* ::= and | and angled brackets (<,>) are the usual metacharacters of the BNF representing respectively the expansion of a synctatic category, the alternatives of one expansion and, finally, **words** in boldface included between angled brackets are the syntactic categories;
* syntactic categories surrounded by square brackets ([,]) are optional i.e. the specified synctatic category may or may not be present;
* words in *italics* are "tokens" i.e. they represent synctatic categories that can be recognized by a simple lexical analysis phase;
* all the other words or characters are keywords of the class definition language i.e. they are made of sequences of terminal symbols.

```
<class> ::=        def_class <class_name>
                   of_type <type_name>
                         [<super_classes>]
                         { <property_list> } ;
```

```
<super_classes> ::=      subclass_of <class_list>

<class_list> ::=         <class_list> , <class_name>
                       | <class_name>

<property_list> ::=      <property_list> , <property>
                       | <property>

<property>      ::=      <attribute>
                       | <component>

<attribute>     ::=<attribute_name> :
                         [<multival>] <primitive_class> [<attr_spec>]

<component>     ::=<compon_name> :
                         [<link_type>] [<multival>]
                         <class_name> [<comp_spec>]

<comp_spec>     ::=      <inverse_spec>
                         | <optionality>
                         | <variability>

<attr_spec>     ::=      (default <value>)
                         | <optionality>
                         | <variability>

<inverse_spec>  ::=      ( inverse < class_name>.<compon_name>)

<multival>      ::=      set_of | sequence_of

<link_type>     ::=      dependent | independent

<optionality>   ::=      optional | mandatory

<variability>   ::=      constant | variable

<primitive_class> ::= string | integer | real | boolean | date | text

<class_name>      ::= IDENTIFIER
<type_name>       ::= IDENTIFIER
<attribute_name>  ::= IDENTIFIER
<compon_name>     ::= IDENTIFIER

<value>         ::=      STRING_VALUE
                       | INTEGER_VALUE
                       | REAL_VALUE
                       | BOOLEAN_VALUE
                       | DATE_VALUE
                       | TEXT_VALUE
```

1.2.3 Types vs Classes.

In this section we will describe in a more precise manner the relationship between types and classes in our Data Model. This relationship is extremely

important because the semantics of the type system of a generic object-oriented programming language must be compatible with the corresponding semantics in the Data Model; furthermore, some type constructors of the programming language must be mapped onto types provided by the Data Model.

The concept of type plays an important role in object-oriented programming language due to the inheritance idea: a method of a supertype can safely be applied to an object belonging to a subtype of that supertype. This property of objects in object-oriented programming language must be valid also for objects retrieved from the database.

Types in object-oriented programming languages are placed in an hierarchy and there are well defined rules that determine the place in the hierarchy occupied by a particular type. The hierarchy in the most complex case is a lattice and the relationship between a node T at a certain level and another node T' in the next level connected to the previous node T is called subtype relationship. The subtype relationship is reflexive (for every T: T<=T), transitive (if T<=T' and T'<=T'' then T<=T'') and partial (the relationship must not be defined for every generic T' and T''). Here are the rules that govern the type and the class system in the COMANDOS Data Model:

1) Every type T is subtype of a type *TOP*;
2) Every *primitive* type P (string, integer, real, boolean, date, text) is a type and primitive types are not in any relation each other;
3) the type of an aggregation is represented by the *labeled set of types with distinct labels*, i.e. an aggregation is a finite set of pair p_i=<label_i,type_i> i=1,n where each label_i is different;
4) the top of the aggregation hierarchy is represented by the *NULL* or *( )* aggregate type;
5) the aggregation constructor is the unique type constructor that introduces hierarchical information in the type lattice; given two types of aggregate AT1 and AT2, they are in a subtype relationship AT1<=AT2 if and only if the set of labels of AT1 contains that of AT2 and for every labeled type LT2 of AT2 the corresponding labeled type LT1 of AT1 is in the subtype relationship LT1<=LT2;
6) the type of a set of elements and of a sequence of elements is the *set* and *sequence* of the type of the elements; given two set or sequence types S1(T1) and S2(T2), they are in the subtype relationship S1(T1)<=S2(T2) if and only if T1<=T2;
7) given a set of aggregation types AT1 ... ATN, the multiple inheritance combines these types creating a new aggregation type AT with a union operation based on labels; every pair <L,T> of AT is equal to the unique one present in the union set or, if the same label L' occurs more than once in AT1 ... ATN, it is equal to <L',T'> with T'=min{T'': <L',T''> in AT1 .... ATN} ;
8) the type of a class of an aggregation type T is equal to *sequence*(T) ;
9) the type of a single_valued component is equal to the type of the component ;
10) the type of a multi_valued component is equal to *sequence* of the aggregation type of the component ;
11) the type of a specialized class is *sequence* of the aggregation type obtained using the multiple inheritance algorythm given in 7) where the ATi i=1,N are the aggregation type associated to the superclasses involved in the specialization.

1.2.4. Operations on Classes.

In this section we will describe the operations that are available at programming level or query processing interface, i.e. the operations that a language or query developer can use in order to implement its own algorythms. They must not be intended for the end-user but instead for a generic system programmer.

When a program begins its execution, there will be in the process address space an instance of a database object, which logically represents the visible part of the database for a particular user. This database object logically maintains the association between class names and class definitions and also preserves the extension of every class. In practice, it only handles a table of association between class names and class objects.

This database object is the object from which the program starts to manipulate classes and objects in classes. It will be indicated as DBobj of type DBtype.

Every class will be represented at run_time by a suitable object which responds to a set of messages similar to those available on sequence objects. The class object, identified by CLobj of type CLtype, maintains the sequence of objects belonging to the class. It stores on the secondary storage only those objects that have the direct type of the class, i.e. only those objects that belong to the aggregation type of the class and that are not also belonging to any of its subtypes. Every class object has a cursor which remembers the position of the last touched object.

We will use the following syntax to specify such operations, with the usual conventions:

<operation> ::= [<type>] <rec_oid>.**oper-name**([<type-parms>])
<type-params> ::= <type> | <type>,<type-parms>
<rec_oid> ::= *IDENTIFIER*
<type> ::= [<type>]

The phrase "returns an object with identifier ..." means that the object identifier, which is unique for every both persistent and volatile object, is returned to the caller of the operation. Usually the ODMS interface will return object identifier, except for primitive types.

Messages to which the DBobj answers:

1) **CreateClass**:*string X string* ⟶ *ClassType*
   Syntax: Cl_obj DB_obj.**CreateClass**(str1,str2)
   Semantics: This operation creates a new class object of name str1 and definition str2 and returns the new created object identifier Cl_obj. If a class with name str1 already exists, or the string str2 is not synctatically correct, the operation will not succeed. The class object is opened and the cursor is positioned before the first persistent object (it still has to be inserted one object in the class).

2) **OpenClass**: *string* ⟶ *ClassObj*
   Syntax: Cl_obj DB_obj.**OpenClass**(str)
   Semantics: This operation activates an existing class object of name str and returns the new opened object identifier Cl_Oid. The cursor is positioned before the first persistent object. If a class with name str is already opened, or the string does not represent a class name, the operation will not succeed.

3) **DeleteClass:** *string* ⟶ *( )*
Syntax: DB_obj.**DeleteClass**(str)
Semantics: This operation closes and deletes an empty class. It removes its definition, too. The class name is also removed from every superclass. If the class object contains some instances of persistent objects, the operation will not succeed. The CL_obj associated of the class object of name str becomes invalid.

4) **CloseClass:** *string* ⟶ *( )*
Syntax: DB_obj.**CloseClass**(str)
Semantics: This operation closes the class object having identifier associated to str. If the class object was already closed, or the string str is not associated to any class object, the operation will not succeed.

Messages to which the CLobj answers:

5) **Get_First:** *()* ⟶ *Dir_Type*
Syntax: Dir_Type CL_obj.**Get_First**()
Semantics: This operation returns an object of type Dir_Type, that is having the direct (aggregation) type of the class. If the class object was already closed, or if it is empty, the operation will not succeed. The cursor will be positioned between the first and the second object stored in the class.

6) **Get_Next:***()* ⟶ *Dir_Type*
Syntax: Dir_Type CL_obj.**Get_Next**()
Semantics: This operation returns the next object of type Dir_Type. If the class object was already closed, or if there is not any object after the current position of the cursor (the sequence is empty or the cursor is after the last element) the operation will not succeed. The cursor will be positioned after the position occupied by the returned object.

7) **Get_Last:***()* ⟶ *Dir_Type*
Syntax: Dir_Type CL_obj.**Get_Last**()
Semantics: This operation returns the last object of the class.. If the class object was already closed, or if it is empty, the operation will not succeed. The cursor will be positioned after the position occupied by the returned object.

8) **Get_Nth:***Integer* ⟶ *Dir_Type*
Syntax: Dir_Type CL_obj.**Get_Nth**(int)
Semantics: This operation returns the n-th object of the class (the sequence starts from one). If the class object was already closed, or if int exceeds the number of objects of the class, the operation will not succeed. The cursor will be positioned after the position occupied by the returned object.

9) **Cardinality:** *( )* ⟶ *Integer*
Syntax: Integer CL_obj.**Cardinality**()
Semantics: This operation returns size of the class i.e. the number of objects in the sequence. If the class object was already closed, the operation will not succeed. The cursor will not be moved.

10) **Subclasses:** *( )* ⟶ *Integer*

Syntax: Integer CL_obj.**Subclasses** ()
Semantics: This operation returns the number of subclasses of CL_obj class. If the class object was already closed, the operation will not succeed. The cursor will not be moved.

11) **Subclass:** *Integer* ⟶ *String*
Syntax: String CL_obj.**Cardinality**(int)
Semantics: This operation returns the name of the n-th subclass of the current class CL_obj. If the class object was already closed, or if CL_obj has less than int subclasses, the operation will not succeed. The cursor will not be moved.

12) **Insert:** *Dir_Type* ⟶ *( )*
Syntax: CL_obj.**Insert**(NewObj)
Semantics: This operation inserts the object passed as parameter to the sequence of objects of the class. It is inserted after the position indicated by the cursor and the cursor advances one position. If the class object was already closed, or the new object has not the direct type of the class, the operation will not succeed.

12) **Delete:** *Integer* ⟶ *( )*
Syntax: CL_obj.**Delete**(int)
Semantics: This operation deletes the object that occupies the int position in the sequence of objects of the class. The cursor will be positioned before the object following the one deleted one in the sequence. If the int parameter is equal to zero, then the object to be deleted is the one following the cursor and the cursor is not modified. If the class object was already closed, or the int parameter exceeds the size of the sequence, the operation will not succeed.

Operations available on classes that represent the usual operations on sequences are also provided for multivalued components. This means that it is possible to open for scanning a particular component of an object member of an opened class. The following operation provides this capability:

13) **OpenComp:** *string* ⟶ *CompObj*
Syntax: Comp_obj CL_obj.**OpenComp**(str)
Semantics: This operation activates an existing component object of name str and returns the new opened object identifier Comp_obj. The cursor is positioned before the first persistent object of the multivalued component. If that component is already opened, or the string does not represent a component name, the operation will not succeed.

14) **CloseComp:** *string* ⟶ *( )*
Syntax: DB_obj.**CloseClass**(str)
Semantics: This operation closes the component object having identifier associated to str. If the component object was already closed, or the string str is not associated to any class object, the operation will not succeed.

15) **Select:** *string* ⟶ *CL_obj*
Syntax: CL_obj1 CL_obj2.**Select**(str)
Semantics: This operation performs the select operation on CL_obj2 . The input parameter represents the selection criteria against a single class.

The objid of a temporary class is returned. This class contains the subset of objids satisfying the selection criteria.

17) **CreateObj**: *Cl_obj* ——▶ *NewObj*
Syntax: PropValObj Obj.**ReadProperty**(str)
Semantics: This operation reads the value of the property specified as input parameter.

17) **ReadProperty**: *string* ——▶ *PropValObj*
Syntax: PropValObj Obj.**ReadProperty**(str)
Semantics: This operation reads the value of the property specified as input parameter.

18) **WriteProperty**: *PropValObj,string* ——▶ *( )*
Syntax: Obj.**WriteProperty**(ProPValObj,string)
Semantics:this operation writes the value of the property specified as input parameters.

1.3. Query Language concepts and updates.

In this section we recall some concepts of the Comandos query language already described in [ServDes88], then we will discuss some updates to the language grammar.

1.3.1 Query language basic features

In defining the query language we have followed the approach of specifying first a logic based language, that we call **calculus query language.** In this way we can express in more clean way the language. However, other languages (such an SQL-like language) should be defined to be used by the application. In this case mappings will be defined to express the semantic of these language in terms of the calculus query language.

A number of requirements arising from the data model have influenced the design of the language.

A first requirement is the capability of navigating through the object structures. In fact the data model allows to define objects as aggregate of other objects; aggregations can be nested at several level. Queries must allow the restrictions on objects based on predicates on component objects at any level in the aggregation hierarchy. As an example let's suppose of having the class definitions in table 1.

Class A:	Class B:	Class C:
X: integer;	Z: string;	I: string;
Y: B;	K: C;	J: integer;
End.	End.	End.

Figure 1

Let's suppose that we want to retrieve the value of property X for all objects of class A having as value of property Y an object of class B, having in turn as value of property K an object of class C having as value of property J an integer equal to the constant 5. Supposing to have the relational calculus [MAIE83] extended with the equality predicate among complex objects (see the following subsection) this query would be expressed as follows:

{t(X)] (EXISTS u) (EXISTS w) (A(t) AND B(u) AND C(w) AND
t(Y)=u AND u(K)=w AND w(I)=5}

(EXISTS indicates the existential quantifier)

From the previous example it can be seen that this query requires to introduce two variables *u* and *w* and two predicates $t(Y)=u$ and $u(K)=w$ to navigate through the structure of complex objects. To simplify queries it is therefore useful to introduce the **dot function**: given a variable x denoting an object and a property name p , p.x returns the object wich is value of property p of x. Dot functions can be nested.

A second requirement arises from the fact that components can be multivalued. From the query language point of view this implies that queries must allow to restrict objects by specifying for instance that all objects values of a multivalued component satisfy a given predicate or that there exists an object among the

values of a multivalued component that satisfies a given predicate. Therefore a predicate must be provided to test the membership of objects to such collection.

A third requirement concerns properties having alternative domains. This is a common situation when modelling complex objects [BATO85]. The query language should provide the possibility of restricting objects by specifying **composite predicates**, for properties with alternative domains, of the form:

if domain(prop) is C1 then P1 ;

if domain(prop) is C2 then P2 ;
.....
if domain(prop) is Cn then Pn ;

where:

prop is a property name of the class to which the query is addressed;

C1 ,....., Cn are classes that are alternative domains for the property prop ; and

P1 ,....., Pn are predicates expressed in terms of properties/and components of classes C1 , ... , Cn respectively.

In the following we recall the BNF grammar of the language:

query ::= <aggregate formation> : <list_of_targets> : <condition>

<aggregate formation> ::= <X>; <R>; <F>

<X> ::= <list_of_objects>

<R> ::= <list_of_objects>

<list_of_objects> ::= <object> | <object>, <list_of_objects>

<F> ::= <list_of_f_spec>

<list_of_f_spec> ::= <f_name> (<property>) |
<f_name> (<property>), <list_of_f_spec>

<list_of_targets> ::= <bind_spec> | <bind_spec>, <list_of_targets>
<bind_spec> (list_prop_names)|
<bind_spec> (list_prop_names), <list_of_targets>

<bind_spec> ::= <variable_name> / <class_name>

<condition> ::= <list_of_quantifiers> (<condition>) |
(<condition>) | NOT <condition> |
<condition> <bool_op> <condition>
<simple_cond>

<list_of_quantifiers> ::= <quantification> |
<quantification> <list_of_quantifiers>

<quantification> ::= (<quantifier> <bind_spec>) |

(<quantifier> <bind_spec>: <quant_restr>)

<bind_spec> ::= <variable_name> /<class>

<class> := <class_name> | (<class> <bop> <class>)

<quant_restr> ::= ELEMENT_OF (<object>, <object>.<prop_name>)

<quantifier> ::= EXISTS | FOR EACH

<simpl_cond> ::= <property> <num_pred> |
<property> <str_pred> |
<property> <join_pred> |
<property> <text_pred> |
ELEMENT_OF(<object>,<property>) |
CLASS_OF (<object>) <str_pred> |
<class_name> (<variable_name>) |
<composite_pred>

<object> ::= <variable_name> | <property>

<property> ::= <variable_name>.<property_spec>

<property_spec> ::= <property_name> |
<property_name>.<property_spec>

<list_prop_names> := <property_spec> |
<property_spec>, <list_prop_names>

<num_pred> ::= <rel_operator> <num_value> |
BW (<num_value>, <num_value>)
IS IN { <list_of_num_values> }

<str_pred> ::= <rel_operator> <str_value> |
LIKE <str_value>|
IS IN { <list_of_str_values> }

<comp_pred> ::= <rel_operator> <object>

<composite_pred> ::= (CLASS_OF(<object>) = list_of_alternatives)

<list_of_alternatives> ::= [<class_name> : <condition>] |
[<class_name> : <condition>]
<list_of_alternatives>

<text_pred> ::= CONTAINS {<list_of_strings>}

<f_name> ::= AVG | SUM | MIN | MAX | COUNT

<rel_op > ::= = | > | < | <= | >=

<bool_op> ::= AND | OR

<list_of_string> ::= <str_value> | <str_value>, <list_of_strings>

<class_name> ::= STRING

<variable_name> ::= IDENTIFIER

<property_name> ::= IDENTIFIER

<num_value> ::= INTEGER | REAL

<str_value> ::= STRING

Two query examples expressed against the set of classes below :

```
Class Projects
     p_name :  string;
     target :  string;
     participants :collection of TEAMS;
End.

Class Teams
     t_name :   string;
     t_addr :   ADDRESS;
     staff :    collection of EMPLOYEE;
End.

Class Employees
     e_addr :   ADDRESS;
End.

Class Address
     nation  :  string;
     city:      string;
     num  :     string;
     street:    string;
     code:      string;
     phone:     string;
End.
```

are the followings:

• *Retrieve the name and the status of all employees of IEI andARG.*

⇒ e/Employee (e_name, status):

(EXISTS t/Teams) (t.t_name IS IN { "IEI", "ARG"} AND

ELEMENT_OF(e, t.staff)).

• *Retrieve the Project name in which employees of Pisa or Milano work.*

⇒ p/Project(p_name) :

(EXISTS t/Teams: ELEMENT_OF(t,p.participants)

(EXISTS e/Employee

```
( (e.e_addr.city = 'PISA' OR  e.e_addr.city = 'MILANO' ) AND

ELEMENT_OF(e, t.staff))).
```

1.3.2 Query Language Updates.

The Query Language has been shown to satisfy the Comandos major requirements. This has been proved by mean of the prototype implemented for CIS. From this experience we have identified the following enhancements to the language grammar:

i) the target list specification should contain the SELF operand (denoted as "."). It is useful whenever the OID of the current object satisfying the query is explicitly required in the target list.

ii) the "always true" restriction clause must be added in order to allow queries like "get all the XProperty values of class C" or "get all object of class C".

iii) The class domain of a query should be extended to subclasses. Since this is not always agreeable, domain extension should be explicitly specified .

iv) It has been outlined that structural equality is a desirable feature in the context of object oriented data base. In addition equality to NIL object should be directly supported by the language.

v) Null values for attributes should be explicitly managed. This improves the system performance at null field retrieval time.

The points above suggest the following extensions to the query language grammar (see [SERVDES88] page 21-23):

i) the followings should be added to the list_of_targets syntax:

<list_of_targets> ::= <bind_spec > (<self>, <list_prop_names>)
| <bind_spec> (<self>,<list_prop_names>) , <list of targets>

being

<self>::= '.'

For example, let us to refer to the following class schema:

```
class  Project
        {       int  DepartmentCode  ;
                Employee  Manager;  }

class  Employee
        {       int  EmpCode;
                string  EmpName;    }
```

The query "get Projects of department '123' and their Manager Names " can be formulated as:

p/ Project(. , p.Manager.EmpName) :
(p.DepartmentCode=123)

ii) the followings should be added to <condition> syntax:

<simple_cond> ::= TRUE

iii) the followings should be added to the <bind_spec> syntax:

<bind_spec>:: = <variable_name>'$'<class>

Example: the query

p$Project(.) :
(TRUE);

gets all the Projects examining possible Project subclasses.

iv) <comp_pred> should be replaced by:

<comp_pred>: <IdnEquality_op> <object>
| <StrEquality_op> <object>

object: property | OID | NIL

v) <rel_op> should be added with the following operator:

| IS_NULL.

2 ODMS components abstract specifications.

2.1 Resource Manager

In order to create classes, the user must have the right to use ODMS resources. At present the only type of resource managed inside the ODMS is the **ObjectSpace**. The ObjectSpace organization is flat. The ObjectSpace definition holds the name of the ObjectSpace, the owner and the physical specifications in term of memory amount managed by the Comandos Storage Subsystem.

2.2 ObjectSpaces Acquisition and Administration

Not everybody can obtain and use ObjectSpaces. The privilege of obtaining the ObjectSpaces is controlled by the authorization mechanism. However the user is allowed to create classes even without having the privilege of obtaining ObjectSpaces . This is possible because a special user, let us call him ODMS Administrator (ODMSA), can allocate ObjectSpaces for other users. The space administration is accomplished by mean of two commands:

- GetObSpace *[owner] ObjectSpaceName [PhysicalSpecs];*
- DropObSpace *ObjectSpaceName.*

Owner is mandatory when ODMSA allocate ObjectSpace for other users.

2.3 Index Manager .

2.3.1 Index manager Concepts

In this section the indexing method adopted in Comandos is described. This topic has been already discussed in the previous ODMS documentation [ServDes88]. However a number of concepts have been either enhanced or refined , while others aspects have been simplified.

Indices are auxiliary structures for increase the efficiency of constrained retrievals. Object oriented database offers some peculiarities with respect usage of indices. The most relevant are the followings. One might want to find an object containing other given objects. In this case indices should consider the PART_OF hierarchy and the search condition can be possibly defined on the **object identity.** In addition the domain on which the search is performed possibly includes the instances of subclasses.

The ODMS supports directly the concept of strong identity (that is object identity is not implemented by using concepts like primary keys or tuple identifier or other tricks). Every object in the ODMS has a persistent identifier called **surrogate** (the opportunity of using surrogates are deeply discussed in [Kho86]) .

The surrogate is a system generated entity, and it is always referred to one ODMS object (the object must exist). The Low Level Identifier defined in Comandos Kernel Design could partially match the structure of the surrogate. In fact the surrogate is independent from the physical locations, and holds inside the class to which the object belongs.

The purpose of index in the ODMS is twofold:

- to support the fast retrieval required by the query processor;
- to support the object reconstruction. Indices implement efficiently the link between the current object and the father object in the PART_OF hierarchy. In this way the usage of index can speed the reconstruction of one object by navigating either in Top-Down or in Bottom-Up mode the object PART_OF structure.

Indices are specified via **path expressions** (briefly path). The path is a structural notation, such as:

cl.p1.p2...pn

where **cl** is the (abstract) *class name* and **pi** are *property names*. The first element of the path is called prefix . The subpath p1...pn is called suffix. We will use the dot notation instead of message notation for simplicity. A path prefix is always a class name . Let A, B and C classes defined as follows:

```
class A        class B        class C
      :              :              :
      p1:B           p2:C           p3: integer;
      :              :              :
 end A;         end B;         end C;
```

then ***A.p1.p2.p3*** is a legal path. In this case p3 is a numeric basic type (in Comandos those properties have been called **attributes**) therefore the index can be used for evaluating numeric predicates on queries. We will call this kind of index **value index.**

More generally however the last element of the index path can be a **component** (i.e. a structured object). This case is specific for O-O environment. For example the path ***A.p1.p2*** is still a legal index declaration path. Such indices are used for evaluating expressions containing predicates on "object identity" (we will call this kind of index **object identity index**). For example the expression "Project.manager EQ a_given_employee" can appear in queries testing in which projects a given employee has management position. Let us suppose Project and Employee classes being defined as :

```
class Project                  class Employee
      Manager: employee;             FirstName: string;
        :                          LastName:string;
        :                          JobCode:integer
end;                           end;
```

Therefore the declaration of index **Project.manager** speeds the query execution up.

The mechanism for index creation used in Comandos derives from the mechanism adopted in Gemstone [mai86] . However the original technique has been modified and multivalued properties and the explicit use of **join indices** [val87] have been included.

The previous ODMS has discussed the idea of join indices. Here we recall the main usage of join indices.

Join indices implements father- son object link. This link is generally a couple (su_2,su_1) where:
- su_1 is the surrogate of one object (the *father*);
- su_2 is the surrogate of the object that is a direct component of *father*.

Therefore, for a set of objects , join indices are a set of binary tables that materialize the PART_OF relationships. The usage of join indices is in general more spread ,since the link can be specified on predefined operations and not only values. This feature is not considered here.

Indices definitions originated a lattice in the following way. The index defined as **A.p1.p2.p3** implicitly defines all the subpath associated indices, namely the object identity indices **A.p1.p2**, **A.p1** and **ClassOf(p1).p2**. The last one defines in turn **ClassOf(p1).p2.p3** and **ClassOf(p2).p3** (ClassOf(pi) denotes the class name of property p*i*) .

Indices sharing subpath's suffixes share the same data structures. That means that the index **B.p2.p3** shares the data structure of **A.p1.p2.p3**, or, in other words, that the B_tree used for indexing the string of p3 is used possibly by the indices **C.p3**, **B.p2.p3** and **A.p1.p2.p3** and so on. The sharing of the data structure is made easier by using surrogates.

The data structures used for implementing indices are of two kinds: ***index dictionary*** and ***B-trees*** . Their usage is discussed below.

2.3.2 B_Tree usage

The usage of B_tree is twofold:

- to index attributes: in this case the B-tree links the value of the attribute to the surrogate of the object to which it belongs.

- to improve the access to join index table's columns for object reconstruction. Two (binary) and three entries join indices are considered. Three entries join indices are used for handling multivalued properties.

For example the implementation of the index

i) department.department_location.building_numb

defined on the class schema:

department
:
department_location: location;
:
end ;

location
building_name: string
building_numb: integer
:
end;

uses two B_trees :

- *integer->location_sur* B_tree performs fast access to building numbers returning the surrogate of *Location* objects (location_sur);
- *location_sur->department_sur* B_tree performs a fast access on the binary table (location_surrogate, departments_surrogate). This table implements the join index holding the link between *Location* and *Department* objects. B_Tree can be defined as indexing either the second surrogate column (we call it **TopDown** join index) or the first surrogate column (**BottomUp** join index) .

In the proposed mechanism the values of a property are always indexed in the same B_tree and there is not any memory about which index has originated the index entry for that value. Therefore additional run-time computation is required for chaining back the join indices. In the previous schema let be:

ii) *location.building_number*

an additional index defined on **Location** class. Occurrences of *Location* can be inserted without being PART_OF Department objects. However every location is indexed through the same B_tree. The evaluation of the query "retrieve the departments in building 123" is performed by using the index *i)* . Therefore the B-tree location_sur -> department_sur must be accessed in order to check whether a location_sur surrogate found in the B_tree integer -> location_sur refers to an object PART_OF of an instance of *Department* .

This run time computation can be expensive and depends on the complexity of the object hierarchy. On the other hand this mechanism allows to easily support query on all the class instances whatever is the object hierarchy that has caused the entry in the index. Other methods might keep memory of that hierarchy, but in this case the maintenance of indices becomes very complex.

A specific case, due to the Comandos data model, occurs if *pi* is a multivalued property. The semantic of index defined on multivalued properties is discussed below. Let us consider the following class schema:

```
class project
        p_name : string;
      :
         partners: collection of Team;
      :
end;


class team
        t_name: string;
        t_address: Address;
      :
         staff: collection of Employee;
      :
end;

class employee
        e_name: string;
        e_address: Address;
      :
      :
end;

class address
        street: string;
        city : string;
      :
end;
```

Table. 2

Let *idx_1* be the index defined on class **project** having the following path: ***project.partners.t_name.*** Since partners is a multivalued property of project, this index may be used for retrieving the projects in which a given team appears in the set of participants. In particular such indices can be used for evaluating the Element_of predicate of queries .

2.3.3 Index dictionary.

The index dictionary describes the index structure. As already mentioned in [SisServ88] the ODMS adopts the data dictionary approach. Therefore Index dictionary is spread over several system classes. The Class Manager and the Query Processor obtain informations on indices by querying directly the index dictionary.

Logically the index dictionary describes a lattice, since path expressions defining indices can partially overlap. In sect. 3.3 the design of ODMS classes structuring the index dictionary as well as the operations for maintaining the dictionary are discussed.

2.3.4 Revised Index Definition Statement Grammar.

The index definition statements have been revised since the last version [ServDes88]. The new BNF grammar is the following.

<index_operations>::= DEFINE INDEX <path_exp> <option_list> AS <index_name> |
DROP INDEX <index_name>
<path_exp> ::= <class_name> . <suffix_path>
<suffix_path> ::= <property>.<suffix_path> | <property>
<option_list> ::= <option>, <option_list> | nil
<option> ::= TopDown | BotUp | Inherited | ASC | DSC
<index_name> ::= string

ASC and DSC mean that the collating order of indexed values is from low to high and from high to low respectively. Of course ASC and DSC apply only to the last suffix_path and only if it is a value index. In this case if the suffix is shared (i.e. already existing) then ASC and DSC apply only when the first index definition is made.

2.3.5 Notes on Placement Manager.

Since the first design of ODMS is centralized, no further activities have been developped on Placement Manager specifications . Therefore the specification level for this component is the one which has been described in [Servdes88].

2.4 Query Processor.

In this chapter the overall description of the Query Processor (QP) logical architecture is discussed . Although the architecture of the QP has been already described in [SERVDES88], the basic concepts will be briefly revisited first.

Most of the considerations that will be outlined in this section have been conceived during the collaboration of CNR researchers and ARG staff involved in the implementation of CIS. The collaboration was aimed at the implementation of the first QP prototype in Comandos.

The present design of QP is based on a Client - Server architectural frame. Client and Server do not identify two different physical levels (for example two machines having special hardware). In fact this bipolar architecture has been used for splitting logical functionalities. Therefore QP_Client and QP_Server can be thought as processes or Comandos Activities running on the same or on different physical sites.

Among different possible Client Server configurations (see [SERVDES88], QP section), the following one has been chosen. The QP_Client performs the syntax checks of queries and constructs a linearized parse tree. The parse tree is then sent to the QP_Server that performs the query type checking and executes the query.

The QP accepts a string containing the query. Queries are specified by means of Query Language grammar . The Query Language Grammar is described in the previous Comandos documentation [SERVDES88] and in this documentation on chap. 2.4 where some updates have been discussed. However the implemented prototype do not include these improvements. The major purpose of the languages are:

- To allow multiclass target list and restrictions, i.e. the query can involve objects belonging to several classes.

- Aggregate functions (Average, Mean , Group By) can be specified in the target list.

- A number of predicates can be specified on simple values . Equality of object identity can be possibly specified for testing (nonstructural) object equality (i.e. two objects are equal if they have the same identity).

- Restrictions on multivalued components can be specified and the Element_Of operator allows to handle object having multivalued components.

- Usual Join predicates can be applied to objects belonging to different classes.

2.4.1 Embedding query language into Comandos language.

This part has not been covered in the past. Here we do not intend to give the implementation solution of the embedding of the Comandos Query Language into the Comandos Language. In fact this section suggests some alternatives to the language implementors in order to attain this difficult task. Pro and cons of two different approaches are considered.

One possibility is to fully merge the Query Language within the programming language. This solution is the most agreeable for the performance point of view but it is the most complex to realize and inflexible for the following reasons:

1) the language syntax must be upgraded in order to host query language grammar.

2) Query must be type checked and optimized. In order to perform type checking the compiler must access classes's informations. Query optimization strategies should be considered among the actions performed by the compiler.

3) After program compilation the strategy chosen by the optimizer can be invalidated because of physical schema changes (for example after adding or deleting indexes). Therefore the program must be recompiled and the plan reevaluated. That constrains the program to be tightly bound to the physical data storing, while the ***physical data independence*** of user applications is an agreeable property.

One alternative is to charge the QP of query type checking and query optimization. More precisely the control flow may be the following:

1) The Query Language statements are syntactically checked and parsed by the language compiler.

2) The Query Language statements are extracted from the source program. They are replaced with special QP method invocation (for example ExecuteThisQuery) . The string containing the query can be passed as argument to the QP.

3) The QP create a new ***optimized plan***. This action can be issued either by interpreting or compiling the query (see below).

4) The plan is called at runtime. If the plan is invalidated the QP should be able to regenerate a new plan and to bind it to the program. In this way the application is unaware of the plan implementation, and physical data indipendence is warranted.

It is clear that the previous mechanism consider a preprocessing phase of the source code.

One additional problem is the management of objects returned from the query execution. Query results can not be associated to a defined type. On the contrary very often the query define implicitly a new type to which the resulting objects match. One solution to this problem is to let a preprocessor to analyse the target list of the query and to create a new type having a generic ***scan*** operation (this is similar to the concept of cursor in database [DATE86]) allowing to browse the resulting set of objects.

2.4.2 Interpretation vs. Precompilation strategy.

The section above has sketched one possible approach to the problem of embedding the query language into Comandos language. By insulating the query statements outside the source program and by charging the QP of compiling or interpreting them , the language is not affected by schema changes and do not need to perform semantic actions on query statements.

Two alternatives can be adopted for the interaction between the language and the QP. First , interactions happen only at runtime. In this case the query interpretation seems to be the most natural way. Query interpretation is not very efficient, in particular if optimizer is called any time the query is evaluated.

Queries can be compiled at compilation time. Query compilation requires in the average more time than the interpretation in order to be issued. However it is a better approach because no overhead is required at run time and because error the query type checking is performed at compilation time. In the query language is not syntactically integrated with the programming language, than a precompilation phase can be use for:

i) performing query syntax checks;
ii) parsing the query;
iii) replacing the query statements into the source program (see below point 2).

After the precompilation , the source program compilation and the query compilation can be performed in parallel by the compiler and by the QP respectively. The output of the QP is a executable optimized plan. It must be catalogued in order to be loaded for later executions.

2.4.3 QP present design.

Although the first prototype of the QP in CIS implements an interpreted version, the design of QP can be adapted to both of the two strategies mentioned above. Let this feature be shown by recalling the functional steps of QP:

- Syntax checks and query parsing phase.

  The QP_Client performs a syntactical analysis of query statement and produces a parse tree.

- Query type checking phase.

  The QP_Server gets as input the parse tree and performs the checks on classes, properties and predicates on them. The type checker needs to access the class catalogs. Class catalogs are secondary storage data structure holding metainformations on classes, properties, indexes etc..

- Query decomposition phase.

  The original query is decomposed, when possible, into a number of subqueries, according the algorithm described in [SERVDES88]. During this phase the *query connectivity* is also tested. Query connectivity states whether the query has been correctly formulated by the user and can be reduced to disjointed queries without operating any transformation.

- (sub)query scheduling and optimization phases

  This is the crucial phase. The data structures describing the steps performed by the executive are generated. These data structures are called evaluation tree. It is generated according to optimization rules. Therefore if the compilation approach is taken, the evaluation tree must represents the compiled version of the query. They can be stored in secondary memory and reload at any later evaluation of the query.

- (sub)query execution phase.
  The evaluation of query is performed on object at the time. The evaluation tree is visited and the predicates are evaluated. A result is returned according to the target list specifications.

Since the parsing, type checking and decomposition phases have been already detailed in the previous Comandos documentation, major emphasis will be given in the next sections to the behaviour of QP during the scheduling, optimization and execution phases.

2.4.3.1 Parsing phase

The query processing is then organized by performing first the query parsing at the Client QP. During this phase the program variables occurring in the query are detected and their values fetched and substituted within the query. The result of this phase is the query parse tree. The parse tree, in linear form. The structure of parse tree is relocatable by mean of relative references within the parse tree data structure.

2.4.3.2 Type Checking phase

This module performs the checking of the query, i.e. it verifies the consistency of each entity appearing within the query statement (classes, attributes, components, variables) with respect to the operand to which it is applied. In the following we discuss the actions performed by the type checker in order to verify the type consistency of the query.

We make the assumption that information on classes are obtained from the Abstract Class Definition Manager (a component of the Server Subsystem). However for the moment we do not make any hypothesis on how these information are organized in main memory, since this is not relevant for the following discussion.

As stated in the previous sections the query is constituted by the target part and the qualification part.

The type checker must perform the following checks on the target part:

- The classes appearing within the <list_of_binding> (see the Calculus grammar) must be existing classes.

- Variables defined in the <list_of_binding> must be declared only once within the scope of the query (uniqueness check). In addition variables declared in the target part can appear in the qualification part as not bounded variables. The consistency of the use of those variables must be verified. In the the example 1. of the "Query language" section, the type of x and y must match with the type of the property of the left hand term.

- If aggregate functions are present then the domains of the aggregate functions must be legal variables for that functions (for example AVG and SUM accepts only numeric domains).

The type checker performs the following checks on the qualification part:

Classes that are domains of \f2existential\f1 and \f2universal\f1 quantifier must be existing classes. Furthermore only variables defined with EXISTS and FOR_EACH can appear within the qualification part (in addition to the variables which are declared in the target part).

Properties that are specified as parameters of the "dot" function must be properties of the class on which the "dot" function is applied. Checking that the "dot" function is applied correctly is more complex when several "dot" function are nested as in the following example:

```
Class A {....;  integer i  ; ....}
Class B {....;    A    x  ; ....}
Class C {....;    B    y  ; ....}

z / C

z.y.x.i
```

In this case the checker must verify that y is a component of C, as well as that x is a component of the class domain of y (i. e.
B), and so on recursively, until the attribute i is reached.

<property> BETWEEN (<num_value1>, <nume_value2>).

The following conditions must be verified: the type of <property> must be numeric (i.e. the type of the object resulting from a READ operation). Num_value1 must be less than num_value2.

<property> IS IN <list_of_num>.

the type of <property> must be numeric. Objects in <list_of_num> must be numeric.

<property> IS IN <list_of_string>

the type of <property> must be string. Objects in <list_of_string> must be of string type.

<property> LIKE <str_value>

<property> must be of type string as well as <str_value>.

<property> CONTAINS <list_of_string>
<property> must be of type text.

ELEMENT_OF (<object1>,<object2>.<property>)

<property> must be a multivalued property containing values elements having the same type than <object1>; <property> must be a property of <object2>.

Relational operators can be applied only to objects belonging to basic classes as usually. The comparison operator "=" can be applied to objects of any type.

2.4.3.3 Query decomposition

The following rules have been already discussed in [SERDES88] . However they are represented for completeness.

The heuristics discussed in the previous section do not imply query transformations. However the query execution may well become unbearably complex, if the query involves many join and not join predicates. In addition one should keep in mind that, since objects can be very complex and large, the performances are bound to the memory capability of the machine more than in the traditional database environments.

Therefore the approach that has been followed is to reduce (if possible) an arbitrary multiclass query to a sequence of subqueries. This process, called decomposition, allows to simplify the query optimization by performing simpler queries in turn. At this aim a technique similar to that used for QUEL [WONG76] is used. This mechanism allows to split the query into pieces (so called irreducible components) which are joined to the remainders by a single joining class. However, once irreducible components are detected, the methods discussed in the previous section are used for performing the subqueries instead of the Tuple Substitution algorithms adopted in QUEL.

In addition an adjustment to the decomposition procedure has been introduced in order to consider the fact that not relational tuples but objects are managed. This extension involves the way join predicates are detected. If the cost estimate of a predicate like "root.comp.attr operator value" approximates the cost of the join (again, the class implementor must decide it), then this predicate is considered as a join during the decomposition process.

The following example will help the reader in understanding the decomposition process (the class schema is that in Chapter 2.4).

Retrieve the team name and the title of all documents containing the word 'computer' in the abstract published by P.Rossi.

```
Q = d/Documents(d_name), t/Teams(t_name):

      (EXISTS e/Employee)
1     (d.abstract    LIKE "computer"      AND
2      e.ename.fname LIKE "P*"            AND
3      e.ename.lname LIKE "Rossi"         AND
4      ELEMENT_OF(e,tm.staff)          AND
5     ELEMENT_OF(e,d.authors)           AND
```

Let C=(C1,C2,...CN) denote the classes of query Q and let T(C) and B(C) denote the target list and qualification respectively. B(C) is the set of the clauses in Q that, according the query specifications, are in conjunctive normal form. Consider the matrix with n columns corresponding to classes C1...CN, and k+1 rows corresponding to T(C) and the k clauses. An entry is set to 1 if the class (column) appears in the given clause (row). This matrix, is called incidence matrix. For the query Q the matrix above is given below. One should notice that clauses 2 and 3 are considered two class rows, i.e. we suppose the class implementor's estimate costs for them are compared to that of join.

	Team	Employee	Document	Name
T	1	0	1	0
1	0	0	1	0

2	0	1	0	1
3	0	1	0	1
4	1	1	0	0
5	0	1	1	0

First of all the query connectivity is tested. The connectivity algorithm reduces step by step the initial matrix. It the process results in a matrix with a single row which is not all 1's, the the class corresponding to the zero entries can be eliminated. If the final matrix is not one row, then the sets of variables corresponding to different rows must be disjoint. The procedure for reducing the matrix is the following :

- find the rows having 1 on the first column;

- perform the logical OR of all those rows;

- apply the previous two steps on the reduced matrix, finding the rows with 1 on the second column and so on.

The matrix above will be reduced as follows:

	Team		Employee	Document	Name
T,4	1	1	1		0
1	0	0	1		0
2	0	1	0		1
3	0	1	0		1
5	0	1	1		0

	Team	Employee	Document	Name
T,4,2,3,5	1	1	1	1
1	0	0	1	0

	Team	Employee	Document	Name
T,4,2,3,5,1	1	1	1	1

The query is therefore connected. The second step is the reduction into irreducible components. Let Q be a connected multiclass query, then it is reducible if the elimination of any one variable results in Q being disconnected. The variable having this property is called joining variable. Thus if such a variable does not exist then the query is irreducible. The interesting aspect of the technique is that in order to obtain the irreducible components, the connectivity algorithm can be used.

The procedure is made by eliminating each column of the matrix in turn and by testing for the connectedness. This procedure is applied to the submatrix of the incidence matrix, obtained by discarding every one class row (i.e. the rows having just one 1). For the example above the steps are the followings:

Eliminate class Team.

	Employee	Document	Name
T	0	1	0
2	1	0	1

3	1	0	1
4	1	0	0
5	1	1	0

	Employee	Document	Name
T	0	1	0
2,3,4,5	1	1	1
T,2,3,4,5	1	1	0

Class Team does not disconnect.

Eliminate Employee.

	Team	Document	Name
T	1	1	0
2	0	0	1
3	0	0	1
4	1	0	0
5	0	1	0

	Team	Document	Name
T,4	1	1	0
2	0	0	1
3	0	0	1
5	0	1	0

	Team	Document	Name
T,4 ,5	1	1	0
2	0	0	1
3	0	0	1

	Team	Document	Name
T,4 ,5	1	1	0
2,3	0	0	1

Class Employee disconnects the matrix into two components..

Since Class Name can be discarded from the component (T,4,5) and (2,3) components result in being a zero's matrix, those components con not be further decomposed. Therefore the irreducible components correspond to the following subqueries:

```
Q1: z/Employee:
 2    (e.ename.fname LIKE "P*"          AND
 3       e.ename.lname LIKE "Rossi" )
```

Let Z be the set of Employee's instances resulting from Q1.

```
Q2:  d/Documents(d_name),  t/Teams(t_name):
     (EXISTS e/Z)
 1    (d.abstract  LIKE "computer"        AND
 4     ELEMENT_OF(e,tm.staff)                  AND
 5      ELEMENT_OF(e,d.authors));
```

As stated before, those subqueries can be executed separately. For obtaining the right sequence of evaluation the reduced incidence matrix (i.e. the one that contains components) might be rearranged (it is convenient for example that the component containing the target list is the last row and so on). Q1 and Q2 can be then executed according the heuristics given in the previous section.

2.4.3.4 Detailed description of scheduling/optimizing phases.

2.4.3.4.1 Overview

In the present design of the query optimizer, sophisticated optimization techniques are not provided, rather few heuristics will be addressed. This choice is motivated by the fact that query optimization in the framework we are addressing requires the optimizer being flexible enough to manage all the possible access methods provided by the underlying store nodes. Since the optimizer must adapt to a number of subsystems, this requires the set of alternative strategies for executing queries be represented as input parameters rather than being embedded in the optimizer code. The class implementor should then provide those data as part of the integration. Although some experiences on the field of extendible optimizer are under development, the problem is still a open research issue [LOHM87]. This is outside the scope of the first prototype development.

In particular we make the following assumptions:

- Queries are in Prenex form, that is in the qualification part of the query there are first all the quantifiers, followed by the query predicates. Furthermore we assume that the query predicates are in conjunctive normal form. These assumptions do not lead the generality of the discussion since queries in general form can be transformed in this form.
- No information about the class cardinality (or other statistical information) are available.
- Information about the indexes available at the underlying system interface include the class properties on which the indexes are defined and the types of relational operators supported by the indexes.
Instead no information about the access costs of the various indexes are available.

2.4.3.4.2 Optimization Concepts.

During those phases an **Optimized Plan** is produced. In the present design the Optimization Plan is composed of two data structures called *Variable Evaluation Tree* and *ElementOf Evaluation Tree* (shortly VarTree and ElofTree).

The first structure is used for evaluating predicates not involving multivalued components while the second is used for solving predicates involving only predicates on multivalued components.

In other words the ElofTree keeps the left-hand variable of ElementOf. Giving the expression *<QU v/ Class ElementOf (v,mc)>* where *QU* is either the quantifier EXISTS or FOREACH , *v* is a variable name and *mc* is the path expression denoting a multivalued component, then *v* is kept inside the ElOf tree (we will call those variables *ElofVar*). Other variables are Join variable or

Simple_variable (shortly *JoinVar* and *SimpVar* respectively) . The JoinVar are the variables that appear in join predicates while the SimpVar do not. Both of them are handled into VarTree.

In order to illustrate steps performed in the scheduling/optimizing phase for the construction of ElofTree and VarTree, let us refer to the following class schema:

```
Class A:  {   int a1;
              int a2 ;  /* a2 is key , i.e. has and index */
              collection a3 of C};

Class B:  {   int b1;
              int b2; }

Class C   {   int c1
              int c2  } ;

Class D:  {   int d1;
              int d2; }
```

and the query:

```
(q1)   a/A (a1) , d/D(d1):
       ( EXISTS b/B     AND
       a.a2 = d.d1      AND
       b.b2 =345        AND
       ( EXISTS c/C : ElementOf(c,a.a3) AND
       c.c1=b.b1  AND c.c2= 123) );
```

In this case a and d are JoinVar, b is the SimpVar and c is the ElofVar.

The construction of the VarTree is issued in the following way (see fig 1.):

1) JoinVar are detected and positioned randomly into VarTree.

2) The VarTree is then revisited and the original tree is changed depending on the variable selective level . The most selective variable is moved down to the tree (see below).

3) a conjunct list is associated to each node. The list contains all the predicates that are defined on the variable representing the node. If the predicate is a join then it is associated to the variable that has been placed in a lower node (i.e. higher tree level).

4) the conjunct list is then analysed for selecting the better way for accessing the class.

5) SimpVar are positioned into the VarTree. They are set at the first level nodes of the tree (the tree root is a dummy var) and they are fathers of the JoinVar beginning the join chain down to the tree.

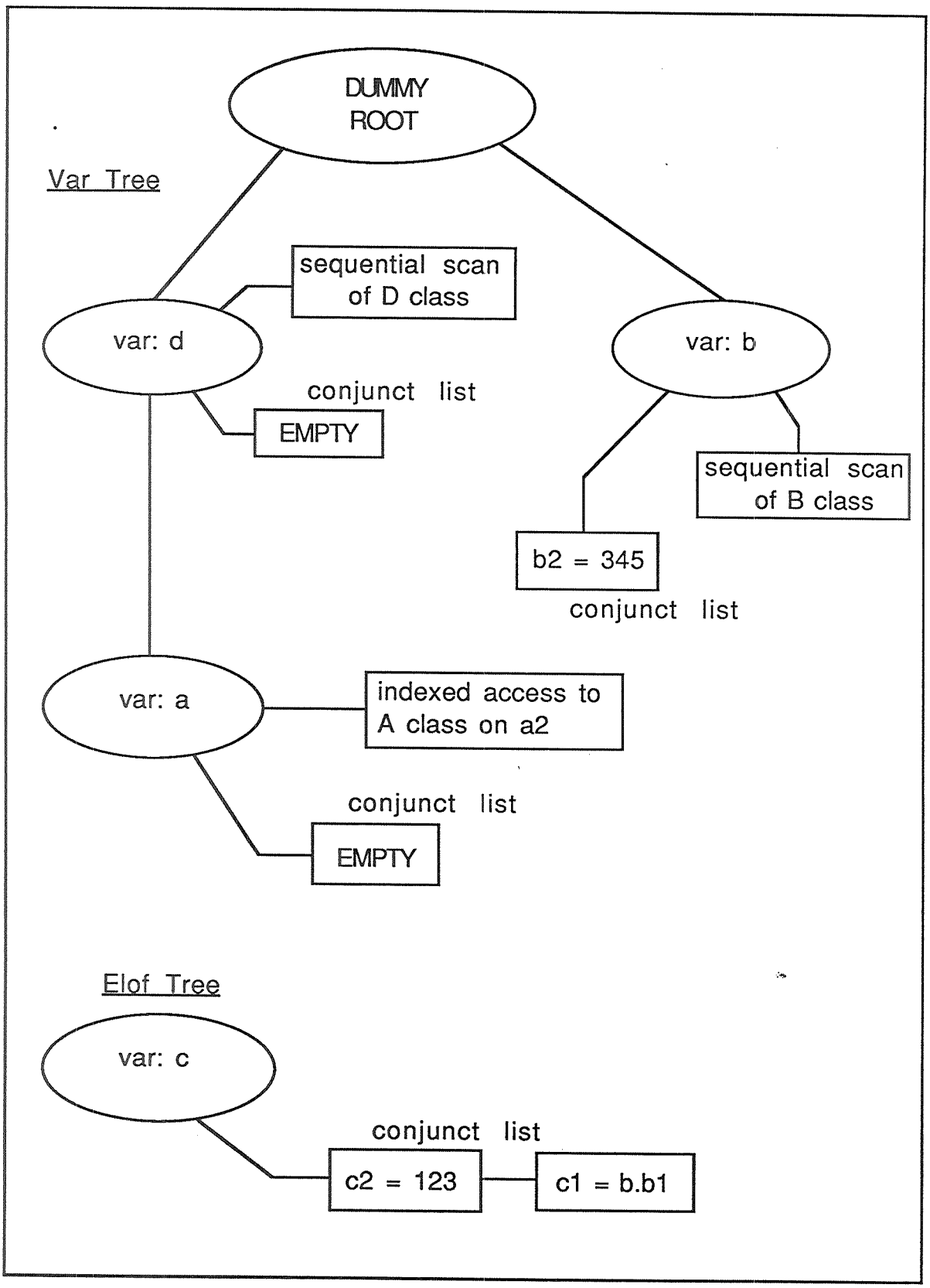
DUMMY
ROOT
Var Tree
sequential scan
of D class
var: d
var: b
conjunct list
EMPTY
sequential scan
of B class
b2 = 345
conjunct list
var: a
indexed access to
A class on a2
conjunct list
EMPTY
Elof Tree
var: c
conjunct list
c2 = 123
c1 = b.b1

fig. 1

Therefore each node of VarTree specifies: the class which has to be accessed ; and how to access it.

The class access method must be as much selective as possible. It generally depends on several factors: the size of its class, the existence of indexes and how selective those index are, how data are physically clustered in secondary memory. The present design of QP takes into account mainly the index existence; that is issued in the following way.

Let us consider again query q1. The positioning of a and d variables within the VarTree cause which of the A and B class is accessed first. The following cases are considered:

- an index exists only on property a2. Then b variable is positioned in the upper node. This node becomes father of node of variable a.
- an index exists only on property b1. Then a variable is positioned in the upper node. This node becomes father of node of variable b.
- indexes exist on both properties. Then the conjunct lists related to both variables are analysed. The variable with the higher number of conjuncts is moved to the upper node.
- no indexes exist at all. As in the previous case, the conjunct lists related to both variables are analysed. The variable with the higher number of conjuncts is moved to the upper node.

The construction of the ElofTree takes into account some restrictions of the languages. In particular the join operation between ElofVar is not allowed. A more interesting operation is the chain of ElofVar (that originates subtrees of ElofTree) defined as:

ElementOf(x,y.y1) AND ElementOf(z,x.x1) AND ElementOf(w,z.z1) etc....

This case has been considered in the present design. In fact while in the VarTree the hierarchy is originated by chain on join predicates, in the ElofTree the hierarchy is originated by chain of ElementOf cascades.

Therefore the ElofTree construction is similar to that of VarTree :

6) detect ElementOf cascades and create subtree of ElofTree;

7) if the same ElofVar is used inside two ElementOf predicates then the Optimizer applies rules that are similar to the ones listed above. In particular the access priority is evaluated in the opposite way, i.e.: the multivalued having keyed access available is accessed *before* the one which has not keyed access and so on. The difference here is that once a multivalued is accessed, the other ElementOf takes the meaning of testing whether a given object is member of a multivalued component. Unfortunately the present design consider the OID never being key.

The last two steps need to be exemplified. Let us consider the following class schema:

```
Class A:  {     int a1;
                int a2 ;
                collection a3 of C};

Class B:  {     int b1;
                collection b2 of C ; }

Class C   {     int c1
                collection c2 od D  } ;

Class D:  {     int d1;
                int d2; }
```

and the query:

```
(q2)   a/A (a1) , b/B(b1):
       (        ElementOf (c,a.a3)      AND
                ElementOf (c,b.b2)      AND
                c.c1<100                AND
                (EXISTS d/D : ElementOf(d,c.c2)  AND
                d.d1>1000));
```

The query q2 represents the case in which an ElementOf cascade occurs (of variables c and d) and the case in which c appears twice in the ElementOf predicate.

The Optimizer must select which of the two ElementOf has to be evaluated first (i.e. which multivalued component should be scanned first). Since the CIS data model supports key operations on multivalued components (namely KeyFirstComponent and KeyNextComponent) , the selection depends strictly from the existence of keys.

For example, if property c1 is key and can be used for accessing multivalued component a3 but not b2, then a3 is accessed first by using c1<100 as key search argument. Once the object belonging to the multivalued a3 has been found, let us say OID1, then multivalued b2 can be lately scanned in order to test whether OID1 belongs to it as well.

The authors are aware of the weakness of some of the previous heuristic. The reader can easily imagine other strategies once statistics on class or multivalued components are accessible or index selective power is known. Further developments of the QP may address those improvements.

2.4.3.5 Detailed description of Execution phase.

The Execution phase is performed by visiting the VarTree and the ElofTree. The tree nodes contain all the informations needed for accessing one class or a multivalued component and for evaluating predicates.

The Executive translates those informations in terms of class generic operations described in the Comandos Data Model, then it issues class access requests to the Object Server (either the ClassManager or CIS). The Object Server

is represented by a component called ObjectServerInterface. In particular for each VarTree node the Executive performs the following steps :

- if the class is keyed accessible, then it requests a KeyFirst/KeyNext access to the ObjectServerInterface passing the search argument (i.e. the predicate simple or composite that must be verified);

- if only sequential scan is available on class then it requests a First/Next access to the ObjectServerInterface;

- once the object has been accessed and the OID is returned it evaluates other conjunct predicates by issuing ReadAttribute requests to the ObjectServerInterface.

Then the Executive visits the ElofTree. For each ElofTree node the Executive performs the following step :

- if the multivalued component is key accessible then it requests a KeyFirstComponent/KeyNextComponent access to the ObjectServer Interface passing the search argument.

- if only sequential scan is available on class then it requests a FirstComponent/NextComponent access to the ObjectServerInterface.

- once the object has been accessed and the OID is returned it evaluates other conjunct predicates by issuing ReadAttribute requests to the ObjectServerInterface.

Finally the Executive builds the query result (one object at the time) by examining the target list. Since the query has been checked to be connected (i.e. each target list variables are referenced at least once within the restriction part, see [SERVDES88]), the result is producted by issuing additional ReadAttr or ReadComp operations on the already accessed objects.

3. ODMS dictionary

The following classes describe metadata for handling : resources, classes, properties and indices. As already mentioned in [SysServ88] the approach adopted in the ODMS is the data dictionary approach, in which system owned classes hold informations about themselves plus the informations of any other class handled by ODMS. In this way system classes are kind of metadata which are cached at the ODMS start time .

3.1 ObjectSpace Dictionary.

Class **ObjectSpace**

```
    ObjSpaceName  string;
    Owner  surrogate;
    NClass  int;
    Containers: collection of ContainerInfo*;
end;
```

- *ObjSpaceName* name of the object space;
- *Owner* surrogate of the owner of the ObjectSpace;
- *NClass* number of classes created in the ObjectSpace.

*: ContainerInfo class is not specified. It hold memory amount information matching the Comandos Storage Subsystem specifications.

3.2 Class and Property Dictionary.

Class **ClassDictionary**:

```
    ClassGroup  string;
    ClassName: string;
    Owner: surrogate;
    ClassType: int
    SuperClass: surrogate;
    Properties: collection of PropertyDictionary;
    ClassSpace: surrogate
    OccurrenceCount: int
end;
```

- *ClassGroup* environment in which the class is created;
- *ClassName* : holds the class name.
- *Owner* : is the surrogate of the object's owners.
- *ClassType*: is the type of the class. At present classes can be real or virtual. Virtual classes are similar to Views in Relational Databases.
- *SuperClass:* So far we have implicitly supposed to deal only with single inheritance. Surrogate 0 corresponds to the TOP class.
- *Properties:* is a multi-valued property whose components store the description of properties of the class.

- *ClassSpace:* is the reference to the descriptor of the physical space in which the class has been created.

- *OccurrenceCount*: is the number of the objects of the class. Is used by query optimiser.

Class **PropertyDictionary.**

PropertyName: string;
Offset: integer;
PropertyType: string;
Length: integer;
IsNull: int
OccurrenceCount: int
AverageLength:int
IdxMainElement: collection of IndexMainDescriptor;
IdxElement: surrogate;
end:

- *PropertyName* : stores the name of the property.

- *Offset* : contains the offset (the sequential number) of the property in the class.

- *PropertyType* : stores the type of the property; legal values are basic types and class names. For multivalued object (attribute or component) PropertyType is the type of the objects belonging to that collection.

- *Length*: stores the length associated to the type declaration.

- *IsNull*: specifies whether NULL values or objects are allowed.

- *OccurrenceCount*: is the number of unique values in the property. It is used as statistic by the optimizer.

- *AverageLength*: average length of property values.

- *IdxMainElement* : references to the index main element instances describing the indices defined on this property (see sect. 4.2.)

- *IdxElement* : this value can be either *nil* or a surrogate of an IndexIntermDescriptor or an IndexFinalDescriptor instance. This property has been added for performance reasons (see index maintenance in section 5.1) .

3.3 Index Dictionary.

The dictionary is a logical entity constituted of several classes. It is structured in three levels:

- The Main Descriptor level describes the hole index in terms of offsets of properties inside their classes.

- The Intermediate Descriptor level describes not final suffix elements, it contains references to Join Indices structures and it allows the sharing of subpaths between different index definitions.

- The Final Descriptor level implements the link between attribute's values and its class object and it allows the sharing of final subpaths between different index definitions.

Logically the index dictionary describes a **lattice** since elements of the suffix can be shared by many indices. Some information are stored also in a system maintained classes holding the metadata of classes, called **ClassDictionary** and **PropertyDictionary.** For indexing purpose the relevant properties of this class are the followings:

class **IndexMainDescriptor**

```
    IdxName:  string;
    IdxType:  integer
    PredicateType:  integer;
    PathLength:  integer;
    IdxStructure:  collection of PathDescriptor;
end;
```

- *IdxName :* the name of the index.

- *IdxType:* says whether the index is used for evaluating predicates on object identity or object equality (i.e. on attributes).

- *PredicateType*: is significant only for the object equality index. It stores the type of last component of the path.

- *PathLength*: holds the length of the suffix part of the index definition.

- *IdxStructure*: the elements of this multivalued describe the structures of the index path defined on the properties in terms of offsets.

class **IndexIntermDescriptor**

```
    BottUpJoin:  surrogate;
    TopDownJoin:  surrogate;
    JoinTable:  surrogate;
    OperationType:  integer;
    TotalSharingDegree:  integer;
    InheritanceDescriptors:  collection of Surrogate;
    IdxStructure:  collection of PathDescriptor;
end;
```

This class describes intermediate links in the path suffix.

- *BottUpJoin*: objid of the B_tree implementing the join index in which surrogates are clustered on in the bottom_up way (that is the access is made on the component object surrogates), or nil value.

- *TopDownJoin*: objid of the B_tree implementing the join index in which surrogates are clustered on in the top-down way (that is the access is made on father the object surrogates), or nil value.

- *JoinTable:* objid of the JoinIndex class instance describing the physical join index table.

- *OperationType:* defines the type of operators supported by this level join index. For all components but last only identity operator is legal. For last component of an identity index, the legal operator is also the identity operator. For last component of an equality index, legal operators are relational or string operators.

- *TotalSharingDegree:* stores the overall number of index paths sharing this component.

- *InheritanceDescriptor:* the members of this multivalued are, if any, the surrogates of all the Index Interm Descriptors corresponding to the current property in the direct sub-classes. If the IS-A hierarchy doesn't exist or the index is not inherited then the content of this property is nil.

- *IdxStructure*: is a multi_valued property whose elements store the offset and the surrogate of the next links of each index path sharing this component.

class **IndexFinalDescriptor**

BTreeRoot : surrogate;
ClassIdentity: surrogate;
PredicateType: integer;
InheritanceDescriptors: collection of surrogate;
ObjectSize: integer;
TotalSharingDegree: integer;
end;

This class describes the last link in the path suffix.

- *BTreeRoot:* contains the objid of the B_tree's root that provides for fast retrieving from attribute values to surrogate values.

- *ClassIdentity:* stores the surrogate of the class to which the current attribute belongs. If the attribute is a multivalued one it contains the surrogate of *IntList, RealList, CharList* or *StringList* depending on the attribute type. The three classes above are employed for storing the object instances of the collections of basic types.

- *PredicateType*: stores the basic type of the last component of the path; legal values are the following : INT, REAL, STRING and CHAR.

- *InheritanceDescriptor:* as in the class above but with regard to Index Final Descriptors.

- *ObjectSize:* length in bytes of the records containing the instances of the class to which this attribute belongs. This Property has been added for performance reasons .

- *TotalSharingDegree*: as in the class above;

class **PathDescriptor**

Offset: integer;
IndexElement: surrogate;
SharingDegree : integer;
end:

- *Offset*: offset of the path element inside its class;

- *IndexElement*: contains the surrogate of either an IndexFinalDescriptor or of an IndexIntermDescriptor.

- *SharingDegree*: stores the number of index paths sharing this single PathDescriptor. Inside IndexMainDescriptor the only possible value of this property is one.

class **JoinIndex**

TwoEntryJI: collection of SurCouple ;
ThreeEntryJI: collection of SurTriplet;
end;

- *TwoEntryJI*: ; two entries join index table
- *ThreeEntryJI*: ; three entries join index table

class **SurCouple**
SonSur :Surrogate
FatherSur: surrogate
end;

class **SurTriplet**
SonSur :Surrogate
FatherSur: surrogate
GranFatherSur: surrogate

end;

- *GranFatherSur*: surrogate of the object containing the multivalued
- *FatherSur:* surrogate of the father or of the multivalued
- *SonSur* :Surrogate of members of the multivalued.

4. ODMS internal component specifications

4.1 Index Manager Specifications

The Index Manager (IDXM) interface provides with functions for creating, deleting indices and updating index entries. The internal structure of IDXM is layered in two levels: the IDXM High Level Component (IDXHC) and the IDXM Low Level Component (IDXLC). While the IDXHC performs operations on the hole index, the IDXLC reacts to operations on simple links and interfaces the Comandos memory management layer. The following subsections depict their behaviour.

The emphasis of the presented indexing mechanism is on index definitions organization. Lower features like B_Tree implementations are not considered. Therefore the pseudocode described in Appendix 1 and 2 makes the assumption that a B_Tree based lower component exists (in this case C_ISAM features have been considered) .

Note: the uppercase parameter type specifications are specified in Appendix 1 and 2.

4.2 High Level Index Manager Component

This component provides for the management of indices definitions lattice and index look-up.

• create_idx (*runner, idx name, idx path , options*)

runner : surrogate
needed for authorization checking .

idx_name : string
name of the index to be built.

idx _path : string.
is a legal path expression (see section 2.3.1)

options: T_OPTION_SPECS
it specifies option list (see 2.3.4).

This function updates the index dictionary entry. The new index definition is inserted and already existing subpath of index path are shared. BTrees are initialized.

• delete_idx (*idx name*)

idx_name: string;
name of the index to be deleted.

This function remove the definition of the index to be deleted from the index dictionary. Sharing degrees of shared subpaths are decreased. This function deletes BTree entries implementing Join Indices and possibly BTree entries for attribute values.

• lookup_idx (sarg, scanObj,mode)

sarg: SEARCH_ARGUMENT.
parsed form of the predicate to be used as index search argument.

scanObj: is NIL the first time the function is called. Otherwise it holds the scan identifier of the multipath index . This identifier is set by look_up the first time it is called.

mode: int
has to be passed to the lower component in order to specify the index scanning modality (namely FIRST, NEXT, CLOSE).

The call to this function is generated by the optimizer. The optimiser choices the index to be used for evaluating a query restriction. Hence it compacts a search argument that can be used with the index. The search argument is coded in parsed form . The expression to be evaluated is in prenex canonical form. The sequential scan (NEXT) allows to retrieve sequentially object according the index collating order while the random access is used when one single object is needed to be accessed (FIRST). In last case all the links involved in the index path are opened . On the other way CLOSE closes all opened links.

4.3 Low Level Index Manager Component.

In this section we will call **link** either the couple describing: i) the connection between an attribute value and the surrogate of the object to which it belongs ; ii) the connection between an object component surrogate and the surrogate of the object to which it belongs (the JoinIndex entry). Whenever differentiation is needed , the link of type i) will be called **value_link** while link of type ii) will be called **sur_link**.

The operations on index at this level is restricted to single links. Therefore this interface provides with the insertion (**insert_link**), the deletion (**delete_link**) and the modification (**modify_link**) of links. In addition the access to information about indices (**idx_cmp_information**), the opening (**open_link**) and the closing (**close_link**) of scannings as well as sequential (**value_sequential_scan, join_sequential_scan**) and random (**value_index_scan, join_index_scan**) access to Indices will be described.

• close_link (scanObj)

scanObj: T_LOW_SCAN_OBJECT
returning data from the opening operations below.

This function closes the specified link.

• insert_link (*idxElement, isColl, fSur, cmpValue, mcSur*)

idxElement : surrogate;
surrogate of the IndexDescriptor (Interm or Final) relative to B_tree(s) to be updated.

isColl : boolean;
FALSE indicates that the link to be inserted is related to either an attribute or to a single_valued component, whereas TRUE indicates that the link is relative to a multi_valued component.

fSur : surrogate;
surrogate of father object.

cmpValue : T_PROP_VALUE;
this parameter stores either the basic type value (integer, real or string) if the link is a value_link or a surrogate if the link is a sur_link. This depends whether idxElement is a final or intermediate IndexDescriptor respectively.

mcSur : surrogate;
this parameter is significant only if isColl value is TRUE. It stores the surrogate of the multivalued component of a (three entries) Join Indices.

This function updates for insertion either BTree holding value_link or BTree holding sur_link. Appendix A2 provides for the detailed description of this function.

• delete_link (*idxElement, delSpecs, delObj*)

idxElement: surrogate;
as in the function above.

delSpecs integer;
if idxElement object is an FinalIntermDescriptor then this parameter specifies that the current link is a value_link (BTREE). In the other case it is a sur_link . In addition the link can be related to a BottomUp link (see section 3) or to a TopDown link. This parameter details also if the current Join Indices table has TWO or THree entries. Furthermore, in the bottomUp delete case relative to a three entries Join Indices table it is also necessary to specify whether ***delObj*** object is a single_valued Component or a Multi_valued component. Therefore this parameters takes one of the following values: BUTWO, BUTHC, BUTHM, TDTWO, TDTHR.

delObj : surrogate;
surrogate of the object to be deleted.

This function updates for deletion either BTree holding value_link or BTree holding sur_link. Appendix A2 gives the detailed description of this function.

• modify_link (*idxElement, isColl, fSur, cmpValue, mcSur*)

idxElement: surrogate;
as in the function above.

isColl : boolean;
analogous as in the insert_link function .

fSur : surrogate;
surrogate of the father object .

cmpValue : T_PROP_VALUE;
as in insert_link function; it is the new attribute/component value.

mcSur : surrogate;
as in insert_link function; it is the possible new multivalued component value.

This function updates for value modification either BTree holding value_link or BTree holding sur_link.

• idx_cmp_information *(clsSur, propName, idxCmpInf)*

clsSur: surrogate;
surrogate of the class .

propName: string;
name of the property on which index existence is queried.

idxCmpInf: T_IDX_CMP_INFORMATION;
this is an output parameter; it is the object that the procedure has to fill (see appendix A2.).

This function return information about index existence on a given property.

• open_link *(idxElement, openSpecs, mode, lockSpecs, lowScanObj)*

idxElement: surrogate;
IndexDescriptor related to the index component on which the scan must be opened .

openSpecs: integer;
The values allowed for this parameters are those already specified for delInf in delete_link. Therefore it specifies whether the scan must be open on a final component (BTREE value), or on an intermediate component. In the latter case it must specify whether the scan is opened on either bottom_up or top_down Join Indices and component is single or multivalued.

mode: integer;

specifies whether the index component is to be opened for reading, writing or both; legal values are the following constants: INPUT, OUTPUT and INOUT respectively.

lockSpecs: integer;
specifies locking informations Three locking modes are allowed: exclusive lock (EXCLLOCK constant value), manual lock (MANULOCK) and automatic lock (AUTHLOCK). The first mode locks all the links of the current index component from the opening to the closing time. The second mode locks a link object only if at the reading time the user submits the explicit request using the LOCK flag (see value_sequential_scan algorithm). Finally the automatic lock is set by system before the read of the link is performed.

lowScanObj: T_LOW_SCAN_OBJECT;
this is an output parameter; it is the data structure the procedure has to fill (see appendix A2).

This function causes the opening of the index either for sequential or random access.

This function causes the closing of an already opened index.

• value_sequential_scan *(lowScanObj, mode, lspecs, btreeObj)*

lowScanObj: T_LOW_SCAN_OBJECT;
the state of this object identifies the required sequential scan.

mode: integer;
it is used to indicate that the current (CURR constant value), first (FIRST) or next (NEXT) link in the specified value index is to be read.

lspecs: integer;
only values for this parameter are the constants LOCK and NLOCK. LOCK value is legal only if manual lock has been specified at scan opening time and in this case the link is locked before being read (that is lspecs set to LOCK is the explicit request submits by the user that wishes an object to be locked).
NLOCK value indicates that either no manual locking mode has been specified at scan opening time (other possible locking modes are: exclusive and automatic) or it has been specified but the user doesn't wish this object to be locked for reading.

btreeObj: T_FINAL_LINK;
this is an output parameter; it is the object that the procedure has to fill (see below).

This function reads class instances according to the order (ascending or descending) specified for the index at index definition time.

• join_sequential_scan (lowScanObj, mode, lspecs, joinObj)

lowScanObj: T_LOW_SCAN_OBJECT;
as in the function above.

mode: integer;
as in the function above.

lspecs: integer;
as in the function above.

joinObj: T_INTERMEDIATE_LINK;
this is an output parameter; it is the object t the procedure has to fill (see Appendix A2).

The join index specified by lowScanObj is scanned sequentially.

• value_index_scan *(lowScanObj, operator, basSchVal, lspecs, idResult)*

lowScanObj: T_LOW_SCAN_OBJECT;
the state of this object identifies the required scan.

operator: integer;
legal specifications are: EQ, GT, GE, LT, LE, LIKE.

basSchVal: T_FINAL_LINK_VALUE;
this object contains the value to be matched. (See appendix A2 for this type specifications).

lspecs: integer;
as in the function above.

idResult: surrogate;
this is an output parameter; it is the object that the procedure has to fill .

• join_index_scan *(lowScanObj, idSchVal, scanSpecs, lspecs, idResult)*

lowScanObj: T_LOW_SCAN_OBJECT;
as in the function above.

idSchVal: surrogate;
it is the searched key value.

scanSpecs: integer;
legal values of this parameter are the following constants: SHCTWO (search for idSchVal in object component entries of a two entries Join Indices), SHCTHR

(search for idSchVal in object component entries of a three entries Join Indices) and SHMTHR (search for idSchVal in multivalued object component entries) for a bottom_up join scan and RTCTWO (return object component surrogate of the located (two entries) Join Indices link), RTCTHR (return object component surrogate of the located (three entries) Join Indices link) and RTMTHR (return multivalued object component surrogate of the located Join Indices link) for a top_down join scan.

lspecs: integer;
as in the function above.

idResult: surrogate;
this is an output parameter; it is the object that the procedure has to fill (see appendix A2).

4.3 Example

The Example in fig. 2 shows the data structures used for implementing the index ***project.p_name***. In the figure ***su_i*** are surrogates. Circles bound multivalued components. A multivalued component has a surrogate itself. Dictionaries entries and B_trees (triangles) are labeled with surrogate as well. Each B_tree is shown whether it indices attributes or surrogates (join indices). In the first example (fig. 2) no join indices appear, since the p_name value (a string) is associated directly to the surrogate of the occurrences of the class Project.

The sequence of steps following the `create_idx(User1, "index1", "project.p_name")` request (which create the data structures of Example 1) are the followings:

- Initially only ClassDictionary and PropertyDictionary entries exist. The properties IdxMainElement and IdxElement of PropertyDictionary are set to *nil*.

- The occurrence of IdxMainElement which has surrogate su_4 is created.

- the IndexMainDescriptor entry su_5 is created as element of the multivalued su_4. EQ means that the index is an "equality index", i.e. it is used for evaluating relational predicates (<,>,= etc..) or specific predicates on strings (LIKE, etc..). String_code is the numeric code specifying that legal operator are string operators. 1 is the path length. index1 is the index name.

- The occurrence of IdxStructure is created with surrogate su_6.

- An element of su_6 (a PathDescriptor's occurrence) is created having surrogate su_7. The offset of p_name inside Project class is 1.

- The IndexFinalDescriptor entry is created with surrogate su_8. String_code gives the type of the B_tree associated. A B_tree is initialized having surrogate su_9. The property TotalSharingDegree is set to 1 since this final component is shared by one index. The surrogate of this element is recorded in IdxElement of the PropertyDictionary entry describing p_name.

Let now make some considerations on object insertion. The insertion of one instance of the class *Project*, is done "**one property at a time**". Therefore first an occurrence of *Project* must be created (obtaining back the surrogate, for example su_100) then the Write operation is performed on the property project_name passing the property value, for example "Comandos".

At insertion time it is asked to the IDXM to check whether indices exist defined on that property and, if positive, it finds the B_tree to be updated by querying the index dictionary (IdxElement property value of the current PropertyDictionary instance).
In our case such query returns su_8, in fact the IndexFinalDescriptor having su_8 surrogate gives the reference of the B_tree to be updated.

In the second example (fig.3) an index is added defined as ***Project.partners.t_address.city***. This case considers the multivalued property partners as suffix's element. Let us give some comments for this example.

- the object su_12 is a multivalued object containing the description of the index structure. Surrogates stored in the *IndexElement* property references two intermediate elements su_16 and su_21 implementing the link *(partners,team)* <-> *project* and *address* <-> *team*. In the first case the join index has three entries because partners is a multivalued property.

- The object su_15 references a final descriptor implementing the index on the attribute **city**.

- In order to increase the efficiency of the **create_idx** algorithm, the surrogate su_16 , su_21 and su_26 are stored in the IdxElement property of the corresponding PropertyDictionary instances: **partners** (su_3), **t_address** (su_53) and **city** (su_94) respectively (fig. 2.4 shows the overall situation).

The example in fig.4 and 5 is more complex. It shows how the proposed mechanism shares dictionary data structures. Fig.4 shows the status of PropertyDictionary after the index : ***Projects.partners.staff.e_address.city*** is added to the situation depicted in fig.3. Here object su_10 contains the MainDescriptors of the two indices defined on **partners**. Su_29 contains the description of the new index. The object su_16 is the descriptor of the **partners** property that is now shared by two indices. In fact su_29 and su_12 (fig. 3) contains the surrogate su_16. The TotalSharingDegree of su_16 is thus set to 2. Likewise the final descriptor su_26 (related to Address.city) is shared by the two indices therefore the TotalSharingDegree is 2. One should notice that a new PathDescriptor instance (su_34) is added in su_19 because the indices suffix matches on the first suffix element **partners** but no on the second suffix element, **t_address** in the former case and **staff** in the latter case.

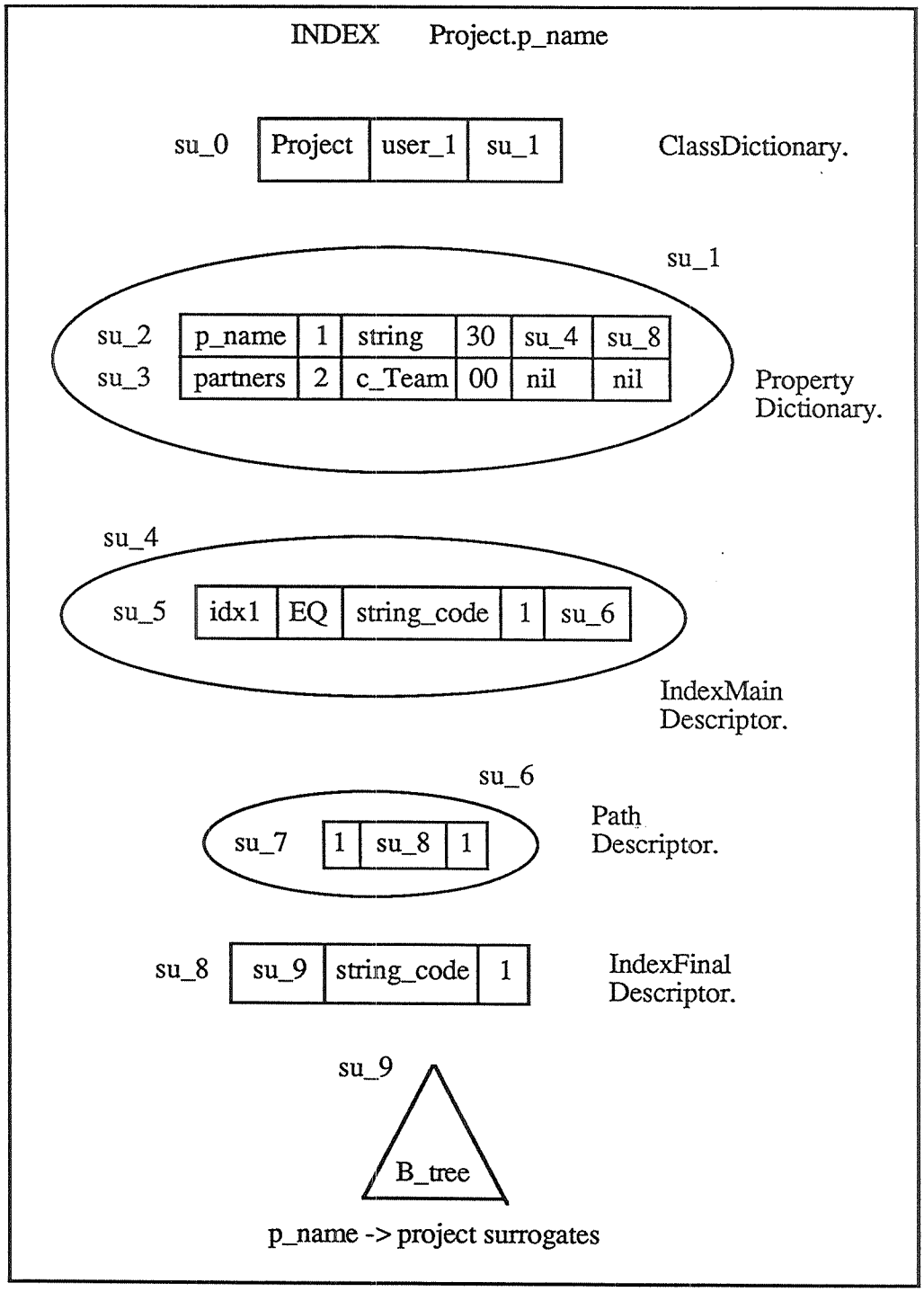
INDEX Project.p_name
su_0
Project
user_1
su_1
ClassDictionary.
su_1
su_2
p_name
1
string
30
su_4
su_8
su_3
partners
2
c_Team
00
nil
nil
Property Dictionary.
su_4
su_5
idx1
EQ
string_code
1
su_6
IndexMain Descriptor.
su_6
su_7
1
su_8
1
Path Descriptor.
su_8
su_9
string_code
1
IndexFinal Descriptor.
su_9
B_tree
p_name -> project surrogates

Fig. 2

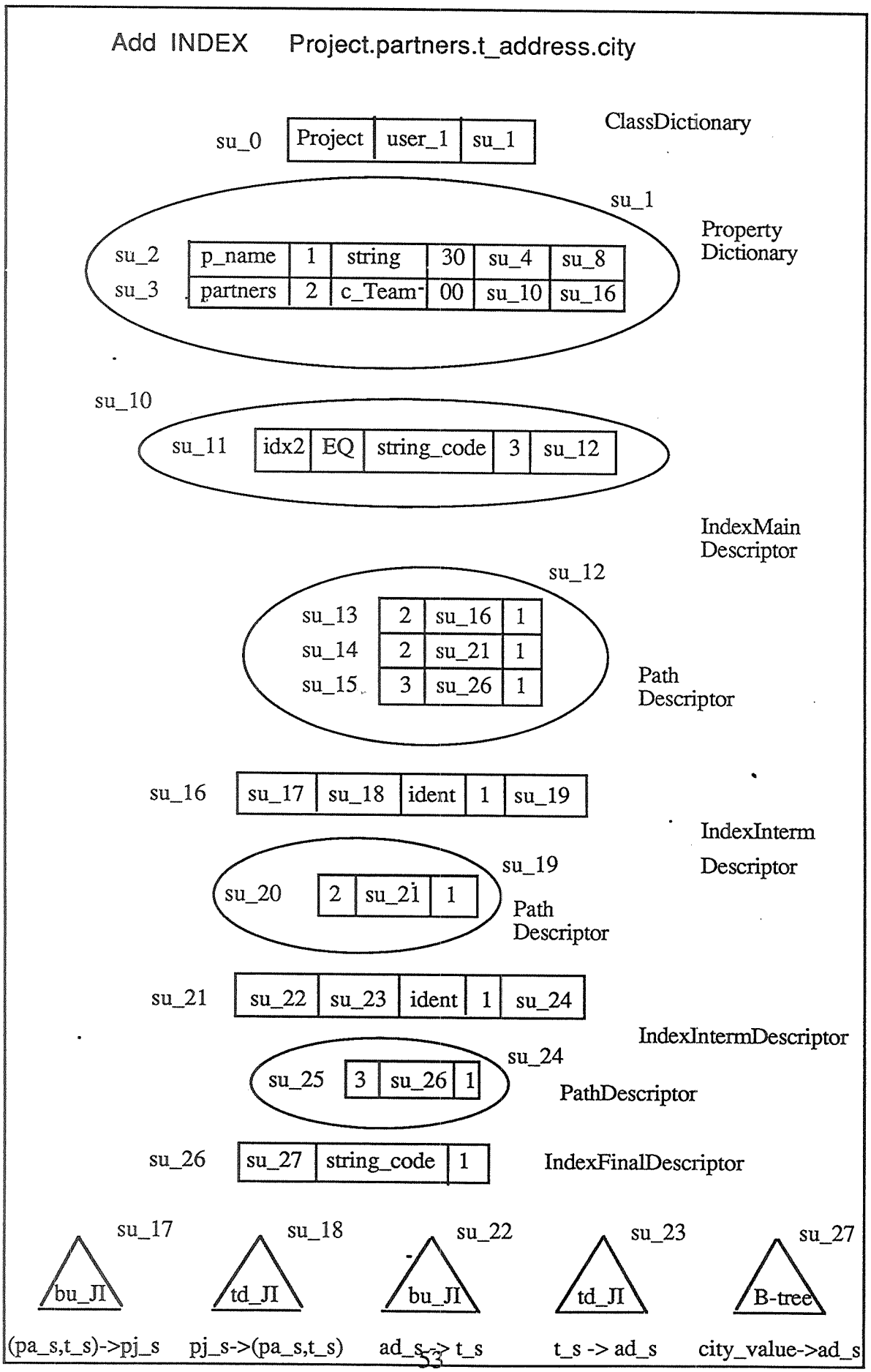
Add INDEX Project.partners.t_address.city
su_0
Project
user_1
su_1
ClassDictionary
su_1
Property Dictionary
su_2
su_3
p_name
1
string
30
su_4
su_8
partners
2
c_Team
00
su_10
su_16
su_10
su_11
idx2
EQ
string_code
3
su_12
IndexMain Descriptor
su_12
su_13
su_14
su_15
2
su_16
1
2
su_21
1
3
su_26
1
Path Descriptor
su_16
su_17
su_18
ident
1
su_19
IndexInterm Descriptor
su_19
su_20
2
su_21
1
Path Descriptor
su_21
su_22
su_23
ident
1
su_24
IndexIntermDescriptor
su_24
su_25
3
su_26
1
PathDescriptor
su_26
su_27
string_code
1
IndexFinalDescriptor
su_17
su_18
su_22
su_23
su_27
bu_JI
td_JI
bu_JI
td_JI
B-tree
(pa_s,t_s)->pj_s
pj_s->(pa_s,t_s)
ad_s -> t_s
t_s -> ad_s
city_value->ad_s

Fig 2.3

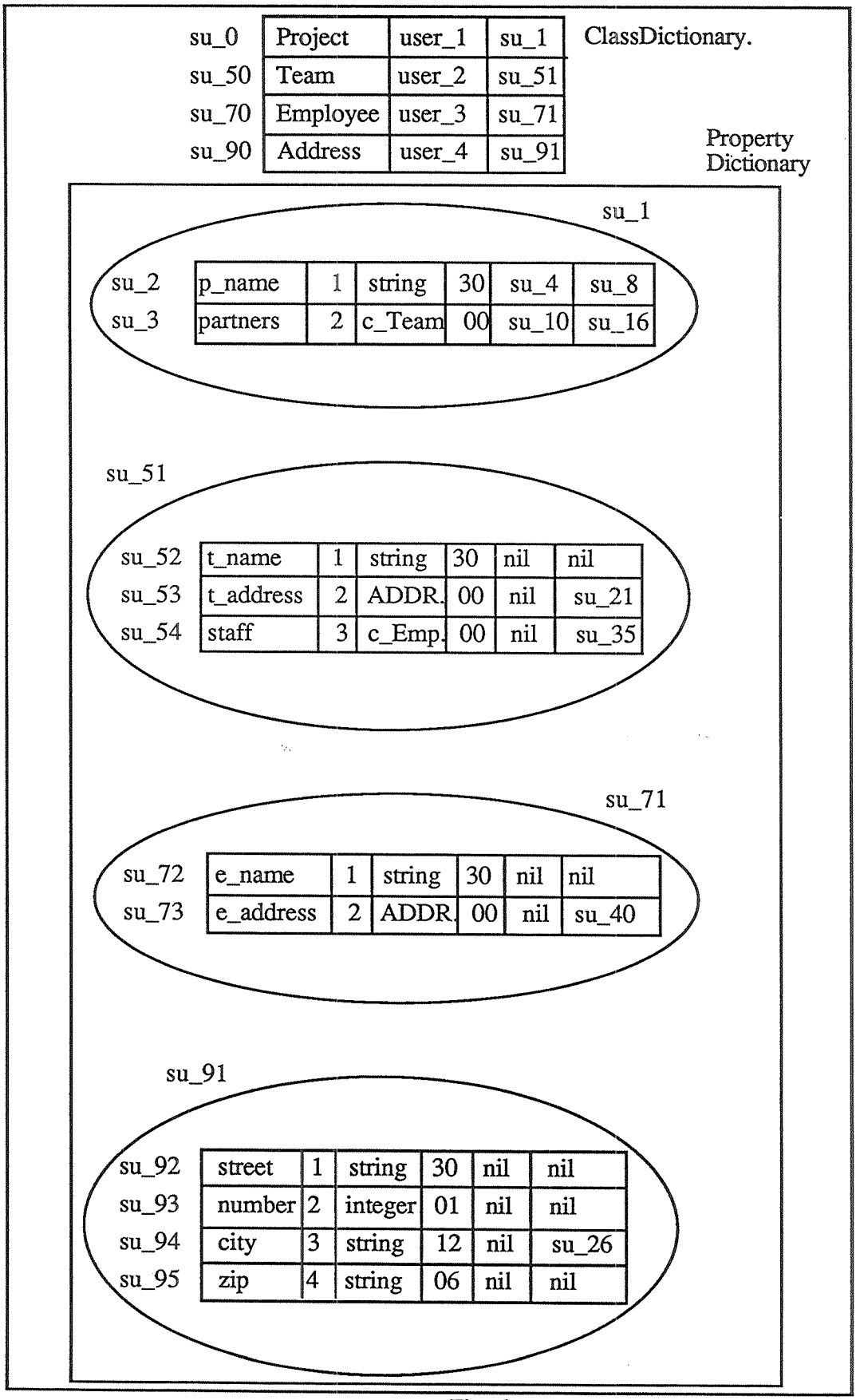
su_0 Project user_1 su_1
su_50 Team user_2 su_51
su_70 Employee user_3 su_71
su_90 Address user_4 su_91
ClassDictionary.
Property Dictionary
su_1
su_2 p_name 1 string 30 su_4 su_8
su_3 partners 2 c_Team 00 su_10 su_16
su_51
su_52 t_name 1 string 30 nil nil
su_53 t_address 2 ADDR. 00 nil su_21
su_54 staff 3 c_Emp. 00 nil su_35
su_71
su_72 e_name 1 string 30 nil nil
su_73 e_address 2 ADDR. 00 nil su_40
su_91
su_92 street 1 string 30 nil nil
su_93 number 2 integer 01 nil nil
su_94 city 3 string 12 nil su_26
su_95 zip 4 string 06 nil nil

Fig. 3

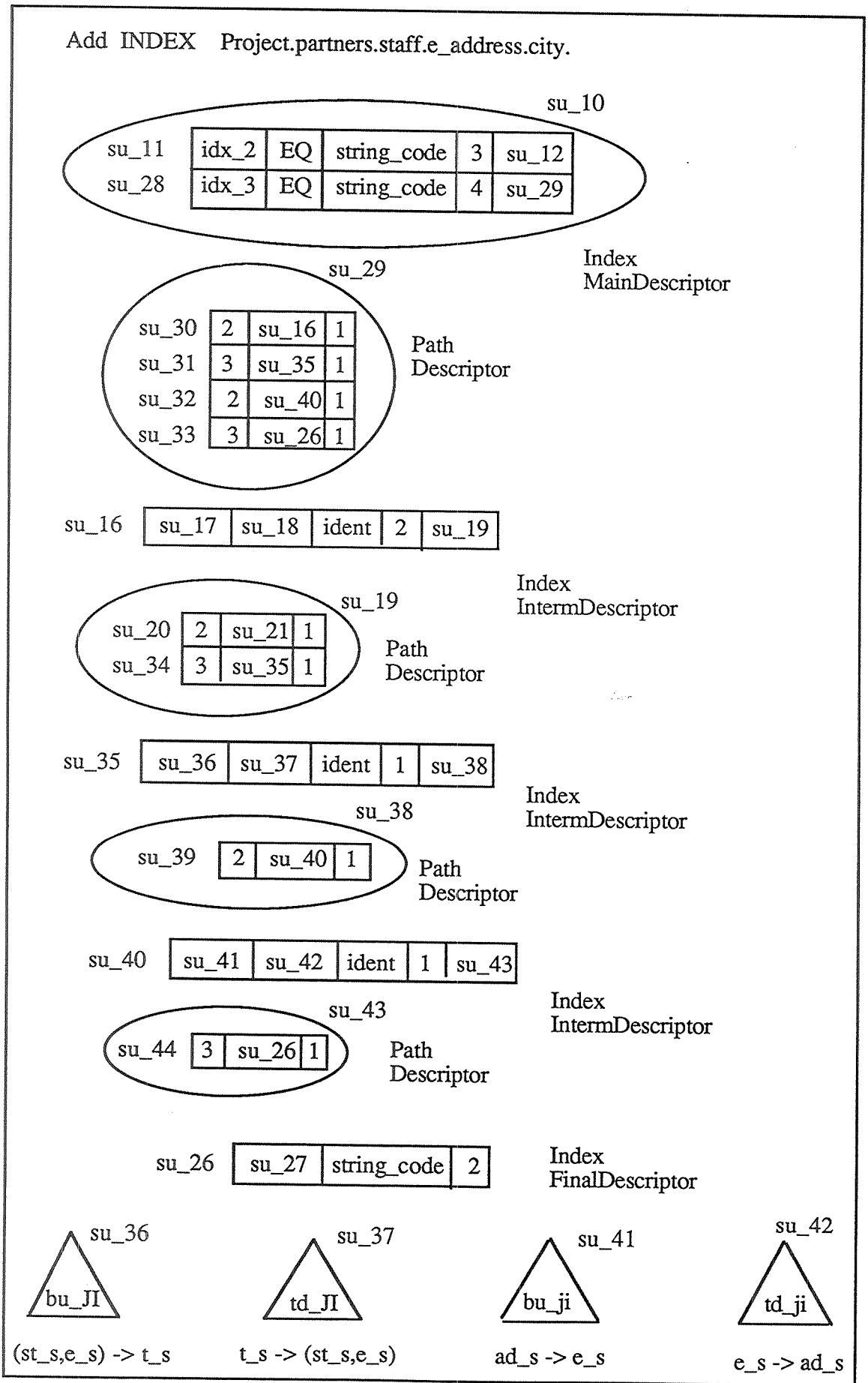
Add INDEX Project.partners.staff.e_address.city.
su_10
su_11
su_28
idx_2 EQ string_code 3 su_12
idx_3 EQ string_code 4 su_29
Index MainDescriptor
su_29
su_30 2 su_16 1
su_31 3 su_35 1
su_32 2 su_40 1
su_33 3 su_26 1
Path Descriptor
su_16 su_17 su_18 ident 2 su_19
Index IntermDescriptor
su_19
su_20 2 su_21 1
su_34 3 su_35 1
Path Descriptor
su_35 su_36 su_37 ident 1 su_38
Index IntermDescriptor
su_38
su_39 2 su_40 1
Path Descriptor
su_40 su_41 su_42 ident 1 su_43
Index IntermDescriptor
su_43
su_44 3 su_26 1
Path Descriptor
su_26 su_27 string_code 2
Index FinalDescriptor
su_36
bu_JI
(st_s,e_s) -> t_s
su_37
td_JI
t_s -> (st_s,e_s)
su_41
bu_ji
ad_s -> e_s
su_42
td_ji
e_s -> ad_s

Fig 4

4.4 Query Processor Specifications

The following functions describe the major steps of query processor. Their behaviour has been detailed in sect. 2.4

Client side.

- qp_parser(externalQuery)

externalQuery: : string.
It is a string containing the query formulated according the query language grammar. This function create a flat byte stream that can be sent over the network and reinterpreted by the server using the parser data structures described in Appendix A3.

Server side.

- qp_checker(query,cArea)

query : struct Query;
header for query data. (See Appendix A4).

cArea: struct ControlArea;
Output formatted data describing query data in easier way. See Appendix A3.

- qp_decomposer(subqueryHeader, cArea)

subqueryHeader: struct SubqHead;
header structure pointing to the evaluation trees.

cArea: struct ControArea;
See above and Appendix A3.

- qp_executive(cArea,subqueryHeader,subqNum)

subqueryHeader: struct SubqHead
See above and Appendix A3.

cArea: struct ControArea
See above

subqNum: int
subQuery to be evaluated.

APPENDIX

In the following appendices the flows of the most relevant functions of the Index Manager are presented (App. 1 and 2). The functions that will be detailed concern the management of the index definition lattice. Revised Query processor data structures are also given in App. 3.

A.1 High Level Index Manager Functions descriptions

• create_idx (*runner, idx name, idx path ,options*)

- 1 ThisClass = GetPrefix(***idx_path***).
- 2 retrieve in ClassDictionary where (ClassName == ThisClass). Let class_des the surrogate of this entry.
- 3 properties_des=SurrogateOf(class_des.Properties).
- 4 check authorization for "creating index " being performed by this user.
- 5 suffix_pos = 1.
- 6 suffix_element=GetSuffixElement(***idx_path*** , suffix_pos).
- 7 retrieve element of properties_des in PropertyDictionary where (PropertyName==suffix_element). Let curr_property = SurrogateOf(this_property).

  /* this step is performed by scanning sequentially properties_des multivalued object. In order to increase efficiency of the algorithm an index can be defined on the PropertyDictionary's property PropertyName */
- 8 if curr_property.IdxMainElement == nil
  then create IdxMainElement object and store the surrogate in curr-_property.IdxMainElement.
- 9 else if curr_property.IdxMainElement exists then if IndexMainDescript addresses an index_path == ***idx_path*** then signal "index previously defined" and goto step 17.
- 10 create new member in curr_property.IdxMainElement and set curr_main=sur_of_this_member.
- 11 create IdxStructure object and set curr_main.IdxStructure=surr_of_ this_object.
- 12 set First = TRUE.
- 13 create a member in the multivalued curr_main.IdxStructure and set curr_path_desc=sur_of_this_member.

```
• 14  set curr_path_desc.Offset ;
      set  curr_path_desc.SharingDegree=1.

• 15  if  IsAttribute(curr_property) == TRUE then
      15.1 if  curr_property.IdxElement==nil  then
              15.1.1 set new_descriptor= create_descriptor (FINAL,
              class_des,  suffix_element).

              /* create_descriptor    is described below.  */

              15.1.2  curr_path_desc.IndexElement  =  new_descriptor.
              15.1.3  curr_property.IdxElement  =  new_descriptor.
              15.1.4  if not First then
                      15.1.4.1   create a  member in the multivalued
                                 curr_interm.IdxStructure and set
                      new_path_desc  =  sur_of_this_member.
                      15.1.4.2   set   new_path_desc.Offset.
                      15.1.4.3   new_path_desc.IndexElement  =
                      new_desriptor.
                      15.1.4.4   set new_path_desc.SharingDegree  =  1.
      15.2    else

              /* the present path index component descriptor already exists in
              the index dictionary so it will be shared (by simply increasing
              the corresponding TotalSharingDegree property, see below)   */

              15.2.1  new_descriptor=current_property.IdxElement.
              15.2.2  increase  new_descriptor.TotalSharingDegree.
              15.2.3  curr_path_desc.IndexElement=new_descriptor.
              15.2.4  if not First then
                      15.2.4.1   retrive object of curr_interm.Idx
                      Structure  where (object.Offset==
                                 Offset_of_suffix_element) and set
                                 new_path_desc=sur_of_this_object.
                      15.2.4.2   if new_path_desc <> nil
                                 then    increase  new_path_desc.Sharing
                                        Degree.
                                 else {
                                        create a member in the multi_ valued
                                        curr_interm.IdxStructure  and  set
                                        new_path_desc
                                        =  sur_of_this_member.
                                        set Offset of new_path_desc.
                                        new_path_desc.IndexElement  =
                                        new_descriptor.
                                        set  new_path_desc.Sharing
                                        Degree = 1.
                                       }
      15.3    set properties ( IdxName, IdxType, PredicateType and
              PathLength) of curr_main object.

• 16  if curr_property is not an attribute then
      16.1 if  curr_property.IdxElement==nil  then
              16.1.1 set new_descriptor=create_descriptor (INTERM,
              class_des,  suffix_element).
```

```
        16.1.2  curr_path_desc.IndexElement  =  new_descriptor.
        16.1.3  curr_property.IdxElement  =  new_descriptor.
        16.1.4  if not First then
                16.1.4.1   create a  member in the multivalued
                           curr_interm.IdxStructure and set
                new_path_desc=  sur_of_this_member..
                16.1.4.2   set Offset of new_path_desc.
                16.1.4.3   new_path_desc.IndexElement =
                new_desriptor.
                16.1.4.4   set new_path_desc.SharingDegree = 1.
16.2    else

        /* the present path index component descriptor already exists in
        the index dictionary so it will be shared (by simply increasing
        the corresponding TotalSharingDegree property, see below)   */

        16.2.1  new_descriptor=current_property.IdxElement.
        16.2.2  increase  new_descriptor.TotalSharingDegree.
        16.2.3  curr_path_desc.IndexElement=new_descriptor.
        16.2.4  if not First then
                16.2.4.1   retrive object of curr_interm.Idx
                Structure  where (object.Offset==
                           Offset_of_suffix_element) and set
                           new_path_desc=sur_of_this_object.
                16.2.4.2   if new_path_desc <> nil
                           then    increase new_path_desc.Sharing
                                   Degree.
                           else {
                                   create a member in the multi_valued
                                   curr_interm.Idx  Structure  and  set
                                   new_path_desc
                                   =  sur_of_this_member.
                                   set Offset of new_path_desc.
                                   new_path_desc.IndexElement  =
                                   new_descriptor.
                                   set new_path_desc.SharingDegree = 1.
                                  }

16.3    increase suffix_pos.
16.4    suffix_element=GetSuffixElement(idx_path ,suffix_pos)
16.5    if suffix_element==nil then
        /* it is an object identity index */
        16.5.1  set properties (IdxName, IdxType, PredicateType
                and PathLength) of curr_main object.
        16.5.2  goto step 17.
16.6    prefix  =  curr_property.PropertyType.

        /* if curr_property is a multivalued then consider the
             type of the objects belonging to the collection        */

16.7    retrive descriptor in ClassDictionary where (ClassName ==
prefix) and set class_des = sur_of_this_descriptor.
16.8    set properties_des = sur in class_des.Properties.
16.9    retrive element of properties_des in PropertyDictionary
        where  (PropertyName==suffix_element)
        and  set  curr_property=sur_of_this_property.
```

16.10 set curr_interm=new_descriptor.
16.11 First=FALSE.
16.12 goto step 13.

• 17 END.

• SURROGATE create_descriptor (type, class_des, property_name)

type : integer
holds the type (INTERM or FINAL) of index descriptor to be instantiated.

class_des :surrogate
surrogate of the class to which the property relative to the index descriptor to be created belongs.

property_name: string
name of the property.

This function instantiates a new Index Intermediate or Final Descriptor class entry and returns its surrogate. This is achieved either by instantiating the appropriate Index (Intermediate or Final) Descriptor or by simply increasing the TotalSharingDegree. Furthermore it recoursively expands the instantiation within the subclass hierarchy .

{

SURROGATE: new_descriptor, inh_des, sub_class_des, properties_des, curr_property, other_descriptor;

• if (type==FINAL) new_descriptor=new.IndexFinalDescriptor
• else new_descriptor= new.IndexIntermDescriptor ;

• <initialize properties of new_descriptor>

• if ((sub_class_des = SubClassesOf(class_des)) == nil) {
new_descriptor.InheritanceDescriptors=nil;
return (new_descriptor); }

• else {
<create InheritanceDescriptors object and set
inh_des = sur_of_ this_object>;

• new_descriptor.InheritanceDescriptors=inh_des;

• while (sub_class_des != nil) {

• properties_des=sub_class_des.Properties;

• <retrieve element of properties_des in Property Dictionary where (PropertyName == property_name) and set curr_property = surr_of_this_property>;

• if (curr_property.IdxElement==nil) {

```
    • other_descriptor = create_descriptor (type, sub_class_des,
property_name);
    • inh_ des object=other_descriptor
    • curr_property.IdxElement = other_descriptor; }

    else {
       ++ curr_property.IdxElement.TotalSharingDegree;
       inh_des=curr_property.IdxElement }
    • sub_class_des = GetNextSubClass (class_desc);
  }
       return (new_descriptor);
  }
}
```

• delete_idx (*class, idx_name*)

class: string;
prefix class name of the path expression describing idx_name index.

idx_name: string;
name of the index to be deleted.

Description.

An informal description of the delete_idx procedure is given below. The procedure delete the entries in the index dictionary and the B_trees related to the index specified by *index_name*. .

Steps:

• 1 retrive descriptor in ClassDictionary where (ClassName == ***class***) and set class_desc = sur_of_this_descriptor.

• 2 set prop_desc = class_desc.Properties.

• 3 retrieve in prop_des.IdxMainElement where Index MainDescriptor.IdxName == ***idx_name***) and set curr_main = sur_of_this_Descriptor.

• 4 set path = curr_main.IdxStructure.

• 5 delete curr_main .
If this was the only instance of the corresponding IdxMainElement multivalued object then delete also it and set the prop_desc.IdxMainElement = nil.

• 6 set curr_path_desc = FirstComp (path,nil).

/* **FirstComp** function returns the surrogate of a PathDescriptor object */

- 7 set curr_descriptor = curr_path_desc.IndexElement.

- 8 if curr_descriptor object is an IndexIntermDescriptor instance then

8.1 if curr_descriptor.TotalSharingDegree == 1 then

8.1.1 set curr_idx_structure = curr_descriptor.Idx Structure.

8.1.2 set next_path_desc = NextComp (path, curr_path_desc).

8.1.3 if (next_path_desc != nil) then

8.1.3.1 set nx_offset = next_path_desc.Offset.

8.1.3.2 retrieve in curr_idx_structure collection where (instance.Offset == nx_offset) and set path_desc = sur_ofthis_instance.

8.1.3.3 remove path_desc instance from curr_idx_structure collection.

8.1.4 delete (no transitively) curr_idx_structure object.

8.1.5 < retrive element of class_desc.Properties in PropertyDictionary where (Offset == curr_path_ _desc.Offset) and set curr_property = sur_of_this_ _property >.

/* curr_property stores the surrogate of the property related to the index descriptor to be deleted with regard to the current path index (see below). */

8.1.6 set curr_inh_desc = sur in curr_ _descriptor.InheritanceDescriptors.

8.1.7 delete (no transitively) curr_descriptor object.

8.1.8 set IdxElement definitional property of the Property Dictionary instance (it has surrogate value equal to curr_property) related to curr_descriptor object to nil value.

8.1.9 if (curr_inh_desc != nil) then **delete_inh_descriptor** (class_desc, curr_ _inh_desc, curr_property.PropertyName).

/* **delete_inh_descriptor** function provides for the updating of the related Is_a hierarchy; */

8.2 else

/* curr_descriptor.TotalSharingDegree > 1, that is this path index component descriptor is shared by several path indexes. This descriptor has not to be deleted. It is only necessary to reduce the corresponding TotalSharing Degree property . Updates to the relative Is_a hierarchy are not necessary. */

8.2.1 decremente curr_descriptor.TotalSharingDegree.

8.2.2 set curr_idx_structure = sur in curr_descriptor.Idx Structure.

8.2.3 set next_path_desc = NextComp (path, curr_path_desc).

8.2.4 if (next_path_desc != nil) then

8.2.4.1 set nx_offset = next_path_desc.Offset.

8.2.4.2 retrive instance in curr_idx_structure multivalued component where (instance.Offset == nx_offset) and set path_desc = sur_of_this_instance.

8.2.4.3 if path_desc.SharingDegree == 1 then remove path_desc instance from curr_idx_structure collection. else decremente path_desc.Sharing Degree.

- 9 if curr_descriptor object is an IndexFinalDescriptor instance then

9.1 if curr_descriptor.TotalSharingDegree == 1 then

9.1.1 < retrieve element of class_desc.Properties in PropertyDictionary where (Offset == curr_path_ _desc.Offset) and set curr_property = sur_of_this_ _property >.

/* curr_property stores the surrogate of the property related to the descriptor to be deleted with regard to the current path index . */

9.1.2 set curr_inh_desc = sur in curr_ _descriptor.Inheritance Descriptors

9.1.3 delete (in no transitive way) curr_descriptor object.

9.1.4 set IdxElement definitional property of the Property Dictionary instance (it has surrogate value equal to curr_property) related to curr_descriptor object to nil value.

9.1.5 if (curr_inh_desc != nil) then **delete_inh_descriptor** (class_desc, curr_ _inh_desc, curr_property.PropertyName).

/* **delete_inh_descriptor** utility provide for updating of the pertinent Is_a hierarchy; this procedure is described below. */

9.2 if (curr_descriptor.TotalSharingDegree > 1) then

9.2.1 decrement curr_descriptor.TotalSharingDegree.

- 10 remove curr_path_desc instance from path multivalued object.

- 11 retrieve descriptor in ClassDictionary where (ClassName == curr_ _property.PropertyType) and set class_desc = sur_of_this_ _descriptor.

- 12 set curr_path_desc = NextComp (path).

- 13 if (curr_path_desc != nil) then goto step 7.

- 14 delete (in no transitive way) path object.

• 15 END.

• delete_inh_descriptor (class desc, inh desc, property name)

class_desc: surrogate
surrogate of the class to which the property of the index to be deleted belongs.

inh_desc: surrogate
surrogate of the multivalued storing the references of the index descriptors corrisponding to *property_name* and which reflects the direct sub_classes of *class_desc* in the Subclass hierarchy.

property_name: string
name of the property mentioned above .

This function recoursivelly expand index deletion to the index inherited by class_desc subclasses. This aim is achieved either by removing the appropriate index descriptor or by simply decreasing their associated TotalSharingDegree properties.

```
SURROGATE      sub_class_desc, properties_desc, curr_property,    other_desc;

• sub_class_desc = nil;

• while ( < inh_desc isn't an empty collection > )      {
    other_desc = FirstComp (inh_desc);
    sub_class_desc = NextComp (class_desc.SubClasses,
                        sub_class_desc);

/* Notice: if the second parameter of NextComp is equal to nil then it has the
same behaviour as FirstComp function.  */

    if (other_desc.TotalSharingDegree == 1)     {

        delete_inh_descriptor (sub_class_desc, other_
        _desc.InheritanceDescriptors,  property_name);

        < delete (in no transitive way) other_desc object >;

        properties_desc = sub_class_desc.Properties;

        <retrieve element of  properties_des in Property Dictionary where
        (PropertyName == property_name) and set curr_property =
        surr_of_this_property>;

        curr_property.IdxElement = nil; }

  • else -- other_desc.TotalSharingDegree;
```

```
< remove other_desc surrogate from inh_desc collection >;
}
```

A2. Low Level Index Manager Component Specifications

The functions of the Low Level are supposed to interact with an additional lower layer for the handling of access data structures like B_trees or Hash tables. In our case C_ISAM-like functionalities are supposed to be available on this lower layer. The informal description of functions partecipating to this level is given below.

==========

- insert_link (*idxElement, isColl, fSur, cmpValue, mcSur*)

Description.
Let us define some useful types using C-like syntax.

T_PROP_VALUE type has the following definition:

```
typedef   struct {
                T_PROP_VALUE_TYPE type;
                T_VALUE           value;
} T_PROP_VALUE;
```

furthermore:

```
typedef   enum {
                INT,
                REAL,
                STRING,
                SURROGATE,
} T_PROP_VALUE_TYPE;

typedef   union {
                int          intvalue;
                double       realvalue;
                STRING       strvalue;
                SURROGATE    survalue;
} T_VALUE;

typedef   char STRING [MAXNAME];
```

So far we have defined the surrogate of an object as being composed of: 1) the unique identifier of the class, to which the object belongs, within the Class Dictionary ; 2) the unique identifier of the object within its own class.

```
typedef   struct {
                long     ClsId;
                long     ObjId;
} SURROGATE;
```

Here and in the next algorithms the **GetProp** has been used for accessing properties of classes. Two parameters must be passed to it: the class's surrogate and the name of the property to be read. It returns its value. An informal description of the insert_link procedure is given below. The procedure creates the new entries in the specified B_tree(s) and, then it inserts it into the respective Join Indices Table if the given link is an intermediate link,.

Steps:

- 1 if IdxElement object is an IndexIntermDescriptor instance then

1.1 JoinTableInsert (GetProp(idxElement, JoinTable), isColl, fSur, cmpValue.value.survalue, mcSur).

/* **JoinTableInsert** function provides for the insertion of the specified link into the table storing the Join Indices of this level. * /

1.2 set cmpUse = Use (idxElement).

/* Use function returns one of the following possible values: BOTHJOIN, for that intermediate component both Join Indices exist in which surrogates are clustered either in the bottom_up way and in top_down way, BOTTUPJOIN or TOPDOWNJOIN in the other cases. */

1.3 switch (cmpUse) {

```
case "BOTTUPJOIN":
    set root=GetProp (idxElement, BottUpJoin).
    if isColl==0  then

    /* it is a two entries Join Indices  */

    BTreeInsert (root, cmpValue.value.survalue,
                fSur).
    /* BTreeInsert function has the task of updating the
    B_tree whose root surrogate is root. This operation is to
    be made in accordance with root.ClsId value since it
    specifies whether you have to update a bottom_up Join
    Indices, a top_down Join Indices, or a final B_tree.  */

    else      /* it is a three entries Join Indices  */
        MultBTreeInsert (root, mcSur, cmpValue.
        value.survalue, fSur).

        /* MultBTreeInsert function has the same
        behaviour as BTreeInsert function but with
        regard to a three entries B_tree.  */

    break.
case "TOPDOWNJOIN":
    set root=GetProp (idxElement, TopDownJoin).
    if isColl==0  then
        BTreeInsert (root, cmpValue.value.survalue,
                fSur).
    else
        MultBTreeInsert (root, mcSur, cmpValue.
        value.survalue, fSur).
    break.
case "BOTHJOIN":
    if isColl==0  then
        set root=GetProp (idxElement, BottUpJoin).
```

```
                    BTreeInsert (root, cmpValue.value.survalue,
                            fSur).
                    set root=GetProp (idxElement, TopDownJoin).
                    BTreeInsert (root, cmpValue.value.survalue,
                            fSur).
               else
                    set root=GetProp (idxElement, BottUpJoin).
                    MultBTreeInsert (root, mcSur, cmpValue.
                    value.survalue, fSur).
                    set root=GetProp (idxElement, TopDownJoin).
                    MultBTreeInsert (root, mcSur, cmpValue.
                    value.survalue, fSur).
               break.
          }
```

- 2 if idxElement object is an IndexFinalDescriptor instance then
  2.1 set root=GetProp (idxElement, BTreeRoot).
  2.2 switch (GetProp(idxElement, PredicateType)) {

```
               case "INT":
                    BTreeInsert (root, compValue.value.intvalue,
                            fSur).
                    break.
               case "REAL":
                    BTreeInsert (root, compValue.value.realvalue,
                            fSur).
                    break.
               case "STRING":
                    BTreeInsert (root, compValue.value.strvalue,
                            fSur).
                    break.
          }
```

- 3 END.

- delete_link (***idxElement, delInf, delObj***)

Description.

Th considerations made in insert_link procedure are also significant for this procedure. An informal description of the delete_link procedure is given below. The procedure removes the specified entry(entries) and/or leaf(leaves) relative to delObj parameter from the appropriate B_tree(s) and, if the current descriptor is an intermediate descriptor, then it deletes every link holding delObj parameter from the respective Join Indices Table .

Steps:

- 1 if IdxElement object is an IndexIntermDescriptor instance then
  1.1 set cmpUse = Use (idxElement).

```
          /* Use function is described in insert _link algorithm */
```

  1.2 switch (delInf) {

```
               case "BUTWO":
```

```
        case "BUTHC":
        case "BUTHM":
             if (cmpUse==BOTHJOIN :: cmpUse==BOTTUPJOIN)
             then   JoinTableRandDelete ( GetProp (idxElement,
                          JoinTable), delInf, delObj).

             /* JoinTableRandDelete function uses in this
             specific case the BottomUp Join Indices for direct
             search in the specified Join Indices table (key value is
             delObj) .Then it delete all the links holding delObj
             parameter.  */

             else   JoinTableSeqDelete ( GetProp (idxElement,
                          JoinTable), delInf, delObj).

             /* JoinTableSeqDelete  function provides for the
             same behaviour as JoinTableRandDelete function but
             the table is now entered in sequential way. */

             break.
        case "TDTWO":
        case "TDTHR":
             if (cmpUse==BOTHJOIN :: cmpUse==TOPDOWJOIN)
             then   JoinTableRandDelete ( GetProp (idxElement,
                          JoinTable), delInf, delObj).
             else   JoinTableSeqDelete ( GetProp (idxElement,
                          JoinTable), delInf, delObj).

             /* the deletion in top down way removes more than one
             link only in TDTHR case. */

             break.
   }

1.3  switch (cmpUse) {
        case "BOTTUPJOIN":
             set root=GetProp (idxElement, BottUpJoin).
             if (delInf==BUTWO :: delInf==TDTWO)
             then      BTreeDelete (root, delInf, delObj).

             /* BTreeDelete  function performs the task of
             removing either the entries related to delObj parameter
             from the two-entries B_tree whose root surrogate is
             root. This operation has to be made according to the
             delInf value which, in the current case (: BOTTUPJOIN,
             specified by root.ClsId value) is equal to TDTWO (delete
             leaf) or BUTWO (delete entry(entries) ).  */

             if (delInf==BUTHC :: delInf==BUTHM ::
                                        delInf==TDTHR)
             then      MultBTreeDelete (root, delInf, delObj).

             /* MultBTreeDelete function has the same behaviour
             of BTreeDelete function applied to three entries B_tree.
             * /
```

```
                    break.
               case "TOPDOWNJOIN":
                    set root=GetProp (idxElement, TopDownJoin).
                    if (delInf==BUTWO :: delInf==TDTWO)
                    then      BTreeDelete (root, delInf, delObj).
                    if (delInf==BUTHC :: delInf==BUTHM ::
                                                        delInf==TDTHR)
                    then      MultBTreeDelete (root, delInf, delObj).
               break.
               case "BOTHJOIN":
                    if (delInf==BUTWO :: delInf==TDTWO)
                    then  set root=GetProp (idxElement, BottUpJoin).
                          BTreeDelete (root, delInf, delObj).
                        set root=GetProp (idxElement, TopDownJoin).
                          BTreeDelete (root, delInf, delObj).
                    if (delInf==BUTHC :: delInf==BUTHM ::
                                                        delInf==TDTHR)
                    then  set root=GetProp (idxElement, BottUpJoin).
                               MultBTreeDelete (root, delInf, delObj).
                        set root=GetProp (idxElement, TopDownJoin).
                               MultBTreeDelete (root, delInf, delObj).
                    break.
          }
```

- 2 if idxElement object is an IndexFinalDescriptor instance then
  2.1 set root=GetProp (idxElement, BTreeRoot).
  2.2 BTreeDelete (root, BTREE, delObj).

- 3 END.

- modify_link (*idxElement, isColl, fSur, cmpValue, mcSur*)

Description.

Considerations made in insert_link and delete_link procedures are also relevant to this procedure. An informal description of the modify_link procedure is given below. The procedure updates B_tree(s) and, if the current descriptor is an intermediate descriptor, deletes every link holding fSur parameter from the Join Indices table; then it inserts the specified new link.

The algorithm does not provide for the deletion of the bottom_up links of the old component value since either that object continues to exist within the database as independent object or the application will delete it by invoking again the delete_link function.

Steps:

- 1 if idxElement object is an IndexIntermDescriptor instance then
  1.1 set cmpUse = Use (idxElement).

  /* Use function is described in insert_link algorithm */

  1.2 if (cmpUse==BOTHJOIN :: cmpUse==TOPDOWNJOIN)
  then JTRandModify (GetProp(idxElement, JoinTable), isColl,

```
                    fSur, cmpValue.value.survalue, mcSur ).

/* JTRandModify function performs the direct search in the
specified Join Indices table (key value is fSur). Hence it either
1) updates the retrieved link (old object component surrogate is
substituted with cmpValue.value.survalue) or 2) deletes all links
holding fSur parameter, then it inserts the specified link
corresponding to first component instance of the updated
collection, depending on whether the isColl value is 0 or 1
respectively. In the latter case the links corresponding to
possible other component instances must be inserted by invoking
again insert_link . */

else    JTSeqModify ( GetProp(idxElement, JoinTable), isColl,
                fSur, cmpValue.value.survalue, mcSur ).

/* JTSeqModify function has the same behaviour as JTRand
Modify but the table is now entered sequentially . */
```

1.3

```
switch (cmpUse) {
    case "BOTTUPJOIN":
        set root=GetProp (idxElement, BottUpJoin).
        if isColl==0
    then        BTreeDelete (root, TDTWO, fSur).

                BTreeInsert (root, cmpValue.value.sur
                value, fSur).
        else    MultBTreeDelete (root, TDTHR, fSur).
                MultBTreeInsert (root, mcSur, cmpValue.
                value.survalue, fSur).
        break.
    case "TOPDOWNJOIN":
        set root=GetProp (idxElement, TopDownJoin).
        if isColl==0
    then        BTreeDelete (root, TDTWO, fSur).
                BTreeInsert (root, cmpValue.value.sur
                value, fSur).
        else    MultBTreeDelete (root, TDTHR, fSur).
                MultBTreeInsert (root, mcSur, cmpValue.
                value.survalue, fSur).
        break.
    case "BOTHJOIN":
        if isColl==0  then
            set root=GetProp (idxElement, BottUpJoin).
            BTreeDelete (root, TDWO, fSur).
            BTreeInsert (root, cmpValue.value.survalue,
                    fSur).
            set root=GetProp (idxElement, TopDownJoin).
            BTreeDelete (root, TDWO, fSur).
            BTreeInsert (root, cmpValue.value.survalue,
                    fSur).
        else
            set root=GetProp (idxElement, BottUpJoin).
            MultBTreeDelete (root, TDTHR, fSur).
            MultBTreeInsert (root, mcSur, cmpValue.
            value.survalue, fSur).
```

```
                    set root=GetProp (idxElement, TopDownJoin).
                    MultBTreeDelete (root, TDTHR, fSur).
                    MultBTreeInsert (root, mcSur, cmpValue.
                    value.survalue, fSur).
                break.
        }
```

- 2 if idxElement object is an IndexFinalDescriptor instance then
  - 2.1 set root=GetProp (idxElement, BTreeRoot).
  - 2.2 BTreeDelete (root, BTREE, fSur).
  - 2.3 switch (GetProp(idxElement, PredicateType)) {

```
            case "INT":
                BTreeInsert (root, compValue.value.intvalue,
                             fSur).
                break.
            case "REAL":
                BTreeInsert (root, compValue.value.realvalue,
                             fSur).
                break.
            case "STRING":
                BTreeInsert (root, compValue.value.strvalue,
                             fSur).
                break.
        }
```

- 3 END.

- ***idx_cmp_information (clsSur, propName, idxCmpInf)***

Description.

This procedure checks whether indices exist that are defined on *propName*. Hence it finds the reference (surrogate) of the IndexDescriptor (Final or Intermediate) structure by accessing the index dictionary.

The type of the returned object has the following C-like definition:

```
typedef  struct {
                SURROGATE   idxElement;
                int         idxCmpUse;
                char        statInf [MAXINF];
}   T_IDX_CMP_INFORMATION;
```

idxElement : legal values are either nil (on *propName* property no index has been defined) or the surrogate of an IndexDescriptor (intermediate or final) instance if index has been defined on *propName* .

idxCmpUse : is significant only if idxElement is not equal nil. If idxElement value is the surrogate of an IndexIntermDesriptor instance then it holds the access type (bottom_up, top_down, or both) provided by the Join Index. Therefore legal values are the following constants: BOTTUPJOIN, TOPDOWNJOIN and BOTHJOIN. The tegal value of this field for the case of IndexFinalDescriptor is BTREE.

statInf : it is significant only if any index is defined on *propName* property, in this case it holds index statistics (among other data the key cardinality, key size, index component entries cardinality, etc..).

An informal description of the algorithm follow s.

Steps:

- 1 access the instance having *clsSur* surrogate into ClassDictionary and set props_des=GetProp (clsSur, Properties).

- 2 retrieve element of props_des object where (PropertyName==*propName*) and set curr_prop=sur_of_this_ property.

  /* this step is performed through the sequential scan of the multivalued props_des . In order to increase efficiency of the algorithm an index should be defined on the property PropertyName of PropertyDictionary class */

- 3 set idxCmpInf.idxElement=GetProp (curr_prop, IdxElement).

- 4 if idxCmpInf.idxElement==nil
  then /* no index is defined on propName, the other fields of idxCmpInf structure are so unneeded. */
  goto step 7.

- 5 /* at least one index is defined on PropName */

  if idxCmpInf.idxElement object is an IndexIntermDescriptor instance
  then set idxCmpInf.idxCmpUse = Use (GetProp(curr_prop, IdxElement)).

  /* Use function is described in insert_link algorithm */

  else set idxCmpInf.idxCmpUse = BTREE.

- 6 get statistic informations (by querying ClassDictionary class) and store it into idxCmpInf.statInf string.

- 7 END.

- **open_link** (***idxElement, openSpecs, mode, lockSpecs, lowScanObj)***

Description.

This procedure opens either the value-index or the join index specified in idxElement , in order to process the index scan.
lowScanObj identifies the active scan for subsequent accesses to this index (therefore it allows the IDXM to manage several active scannings).

The type of the returned object has the following description

```
typedef    struct  {
```

```
                    int             fdIdxCmp;
                    SURROGATE       idxElement;
                    int             openSpecs;
}   T_LOW_SCAN_OBJECT;
```

fdIdxCmp: in the first design it has been chosen to use files as containers for persistent objects. More precisely it is the descriptor of the file storing either the Join Indices table (if the scanning is demanded on an intermediate component) or the data file (if the scanning is demanded a final component).

idxElement: holds the reference of the Index Descriptor of the opened scan

openSpecs: describes the scan type (see above) .

Steps:

- 1 if (openSpecs == BTREE)
  then openClass = GetProp (idxElement, ClassIdentity).
  else openClass = GetProp (idxElement, JoinTable).

  /* **openClass** holds the surrogate of an instance of ClassDictionary, more precisely it holds the surrogate of the class to be opened for processing the current (value/join) index scan. */

- 2 openFile = GetProp (openClass, FileName).

- 3 lowScanObj.fdIdxCmp = Open (openFile, mode, lockSpecs).

  /* **Open** function returns the file descriptor that should be used for processing the scan */

- 4 according to openSpecs initialize the current settings for the file (Join Indices table or data file) .

- 5 lowScanObj.idxElement = idxElement.

- 6 lowScanObj.openSpecs = openSpecs.

- 7 END.

- **close_link** (***lowScanObj***)

Description.

This function closes the scan of the specified value or join index. The file containing the entries of such index is closed and all the active locks are released.

- **value_sequential_scan** (***lowScanObj, mode, lspecs, btreeObj***)

Description.

According to the specified **mode**, **lspecs** and **lowScanObj** the sequentially located B_tree link is read from secondary memory into **btreeObj** object.
The **btreeObj** type definition follows :

```
typedef    struct {
                T_FINAL_LINK_VALUE      atrvalue;
                SURROGATE               fhidentity;
}  T_FINAL_LINK;

typedef    struct {
                T_BASIC_TYPE type;
                T_BASIC_VALUE      value;
}  T_FINAL_LINK_VALUE;

typedef    enum {
                INT,
                REAL,
                STRING,
}  T_BASIC_TYPE;

typedef    union {
                int          intvalue;
                double       realvalue;
                STRING       strvalue;
}  T_BASIC_VALUE;
```

where:

atvalue: basic type value in a final link.
fhidentity: surrogate of the father object in a final link.

value_sequential_scan algorithm prepares and forwards the parameters required by underlying retrieval system (recall the assumption made at the beginning of this section) .

- **join_sequential_scan** *(lowScanObj, mode, isMult, lspecs, joinObj)*

Description.

According to the specified **mode**, **lspecs** and **lowScanObj** the sequentially located Join Index 1 link is read from secondary memory into **joinObj** object.
The **joinObj** type definition follows :

```
typedef    union {
                T_THREE_ENTRIES     thrjoinindices;
                T_TWO_ENTRIES       twojoinindices;
}  T_INTERMEDIATE_LINK;

typedef    struct {
                SURROGATE           mulcmpidentity;
                SURROGATE           cmpidentity;
                SURROGATE           fhidentity;
}  T_THREE_ENTRIES;
```

```
typedef    struct {
                    SURROGATE          cmpidentity;
                    SURROGATE          fhidentity;
} T_TWO_ENTRIES;
```

where:

mulcmpidentity: surrogate of the multi_valued object component in the intermediate link.
cmpidentity: surrogate of the object component in the intermediate link.
fhidentity: surrogate of the father object in the intrmediate link.

join_sequential_scan algorithm prepares and forwards the parameters required by underlying retrieval system (recall the assumption made at the beginning of this section) .

- **value_index_scan** (*lowScanObj, operator, basSchVal, lspecs, idResult*)

Description.

According to the specified **operator**, **basSchVal** and **lspecs** parameters the value index identified by **lowScanObj** parameter is randomly accessed, then the father object surrogate of the located B_tree link is read into **idResult** output parameter. If no entry (having key value specified in **basSchVal**) is found then a specific error code is returned.

value_index_scan algorithm prepares and forwards the parameters required by underlying retrieval system (recall the assumption made at the beginning of this section) .

- **join_index_scan** (*lowScanObj, idSchVal, scanSpecs, lspecs, idResult*)

Description.

In accordance with the specified **idSchVal** and **lspecs** parameters the join index identified by **lowScanObj** is randomly accessed (the operator parameter is not needed since only equality predicate are allowed). Hence depending on **scanSpecs** parameter either the father object surrogate (bottom_up join scan) or the (single/multi_valued) object component surrogate (top_down join scan) of the located Join Indices link is read into **idResult.**

join_index_scan algorithm prepares and forwards the parameters required by underlying retrieval system (recall the assumption made at the beginning of this section) .

A.3 Query processor Data Structures

A.3.1 Parser Data Structure

Parse tree structures.

The structures listed in the following form the parse tree.

Structures	Signification
Query	- Root node of the parse tree.
Aggregate	- Root of the aggregate part.
Object_list	- Element of an object list.
F	- Element of an aggregate function list.
Target	- Root of the target part.
Class	- Node containing a class specification.
Bind_spec	- Element of a list of bind specification.
Condition	- Root of the condition part.
Quantifier	- Element of a list of quantifier specification.
Object	- Object specification node.
Property	- Initial property specification node.
Prop_Spec	- Intermediate property specification node.
Prop_List	- List of property specification nodes.
Num_Predicate	- Node containing a numerical predicate specification
Common	- Node containing information in common between a string predicate and a class predicate.
Str_Predicate	- Node containing a string predicate specification.
Join_Predicate	- Node containing a join predicate specification.
Text_Predicate	- Node containing a text predicate specification.
Memb_Predicate	- Node containing a membership predicate specification.
Class_Predicate	- Node containing a class predicate specification.
Comp_Predicate	- Node containing a composite predicate specification.
Alt_List	- Element of a list of alternatives.
In_Predicate	- Node containing a composite predicate specification.
Namelist	- Element of a list of names.
Num_Value	- Element of a list of numerical values.
Str_Value	- Element of a list of string values.

```
struct Query {
        int aggr_formation;
        int target_list;
        int first_cond;
};

struct Aggregate {
        int x_spec;
        int r_spec;
        int f_spec;
```

```
};

struct Object_list {
        int object;
        int next;
};

struct F {
        int  Aggr_Function;     /* values AVG, SUM, MIN, MAX, COUNT */
        int object;
        int next;
};

struct Target {
        int bind;
        int prop_names;
        int next;
};

struct Class {
             int class_spec; /* values: SIMPLE or COMPLEX */
             int class_name;
             int class1;
             int class2;
             int bool_op;
};

struct Bind_spec {
        int var_name;
        int class;
};

struct Condition {
        int cond_type;  /* values : PT_QCOND, PT_ANDCOND,
                                PT_ORCOND, PT_NOTCOND, PT_TERMCOND */
        int termtype;  /* values (only in case of PT_TERMCOND */
        int l_quant; /* only for PT_QCOND */
        int cond1;   /* NULL in case of
                                        PT_TERMCOND */
        int cond2;   /* NULL in case of PT_QCOND,
                                        PT_TERMCOND, PT_NOTCOND */
        union {
                int num_pred;
                int str_pred;
                int join_pred;
                int text_pred;
                int memb_pred;
                int class_pred;
                int comp_pred;
                int in_pred;
               } pred;
};

struct Quantifier {
        int quantifier;         /* PV_EXISTS, PV_FOR_EACH */
        int bind;
```

```
        int memb_pred;
        int next;
};

struct Object {
        int object_type;
        union {
               int var_name;
               int property;
              } otype;
};

struct Property {
        int var_name;
        int prop;
};

struct Prop_Spec {
        int prop_name;
        int next;
};

struct Prop_List {
        int prop;
        int next;
};

struct Num_Predicate {
        int property;
        int operator;
        int num_val1;
        int num_val2;   /* only for PV_OPER_BE*/
};

struct Common {
        int operator;
        int str_value;
};

struct Str_Predicate {
        int property;
        int common;
};

struct Join_Predicate {
        int object1;
        int operator;
        int object2;
};

struct Text_Predicate {
        int property;
        int operator;      /* PV_OPER_CON */
        int str_list;
};
```

```
struct Memb_Predicate {
        int obj;
        int m_property;
};

struct Class_Predicate {
        int obj;
        int common;
};

struct Comp_Predicate {
        int obj;
        int alternative;
};

struct Alt_List {
        int class_name;
        int condition;
        int next;
};

struct In_Predicate {
        int property;
        int set_type;
        union {
                int str_list;
                int num_list;
                } set;
};

struct Namelist {
        char name[STRING_LENGTH];
        int  next;
};

struct Num_Value {
        int    val_type; /* values: PT_INTVAL, PT_REALVAL */
        int    intval;
        double realval;
        int    next;
};

struct Str_Value {
        char str [STRING_LENGTH];
        int  next;
};


/*****************************************************************/
/*                                                               */
/*                    Other data structures                      */
/*                                                               */
/*****************************************************************/

struct Lin_Ptree {
        int p_length;
```

```
        char *ptree;
       };

struct Stack        {     /* Query parse tree stack */
       int st_hi;         /* Height of the stack; 0=empty */
       struct Condition *co_ptr[MAXDIM];
       int co_ty[MAXDIM]; /*type of condition: OR or AND*/
                          /* array of pointers to the Conditions */
                                 /* Whenever there is no left brach in
                                  * the current condition, the pointer
                                  * at the top of the stack is used;
                                  * empty stack indicates end of parse
                                  * tree
                                  */
       };

struct Path { /* This structure contains informations
                 about the left or right hand of a
                 predicate. */
       int    num_el;
       char   prop[MAX_PATH][NAME_LENGTH];
       char   class[MAX_PATH][NAME_LENGTH];
       int    type[MAX_PATH];
};
```

A.3.2 Type Checher Data Structures

```
/*****************************************************/
/*                                                   */
/*  TABLES  CONTAINING INFORMATION ABOUT THE QUERY   */
/*                                                   */
/*  The structure Control_Area is the master         */
/*  control structure and points to all the other    */
/*  structures needed                                */
/*                                                   */
/*****************************************************/


struct Control_Area {
        struct B_Table *b_table;
        struct C_Table *c_table;
        struct D_Table *d_table;
        struct F_Table *f_table;
        struct X_Table *x_table;
};

/* This structure contains the list of bindings */

struct B_Table {
int             num_element;
struct BElement *b_element[BTABLESIZ];
};


/* This structure contains the list of conjuncts */

struct C_Table {
int             num_element;
struct CElement *c_element[CTABLESIZ];
};


/* This structure contains the list of disjuncts */

struct D_Table {
int             num_element;
struct DElement *d_element[DTABLESIZ];
};

/* This structure contains the list of aggr. functions */

struct F_Table {
int             num_element;
struct FElement *f_element[FTABLESIZ];
};

/* This structure contains the list of grouping objects */
```

```
struct X_Table {
int             num_element;
struct XElement *x_element[XTABLESIZ];
};


struct BElement {
        char var_name  [NAME_LENGTH];
        char class_name[NAME_LENGTH];
        int  quantifier;   /* it can assume the following
                              values: PV_EXISTS, PV_FOR_EACH,
                              TARGET */
        int  restricted;   /* if 0 the variable is not
                              restricted to a range of a
                              multivalued component */
        int  me_pred;      /* offset inside the parse tree
                              of the membership predicate */
};


struct CElement  {
       int c_loc;          /* offset inside of the parse tree
                              of this disjunct */
       int type;
       int disj;           /* if TRUE the conjunct contains
                              several disjunct; otherwise is
                              simple predicate */
};


struct DElement {
        char var_name[NAME_LENGTH];     /* name of the variable
                                        representing the object
                                        on which the predicate is
                                        applied */
        char var_name1[NAME_LENGTH];    /* name of the variable
                                        representing the second
                                        object in the predicate,
                                        only for predicates
                                        that involve two objects */
        int d_loc;          /* offset inside of the parse tree
                               of the predicate */
        int type;           /* type of the predicate */
                            /* the possible types are listed before */
        int op;             /* operator used in the predicate
                               it is significant only for the
                               following predicates:
                               num_pred, str_pred, join_pred, class_pred */
        int mv;             /* if TRUE the predicate concerns
                               a multi-valued component */
        int inv;            /* if TRUE the predicate has a NOT
                                  before */
        int c_entry;        /* entry in the C_Table of the
                               conjunct of this disjunct */
```

```
        struct Path path_name;  /* path name for the object in the pred. */
        struct Path path_name1;
};

struct FElement {
        int    f_loc;
        int    function;
};


struct XElement {
        int         type;
        int         x_loc;
        char        varname[NAME_LENGTH];
        char        class[NAME_LENGTH];
        int         in_R;
        struct Path path;
        char        *res_ptr;
};
```

A.3.3 Scheduler and Executive Data Structures

```
/*************************************************************************
*
*
*             constants, types and variables for scheduler*
*
*************************************************************************/


#define VOIDNAME (nullstrng)
#define VARNULL  -1

#define is_component(pth,prn) ((pth).type[(prn)] == SCOMP || \
                (pth).type[(prn)] == MCOMP || (pth).type[(prn)] == VAR)
#define is_sing_comp(pth,prn) ((pth).type[(prn)] == SCOMP)
#define is_eof(pnum) (dtabpoin->d_element[(pnum)]->type == MEMB_PRED)
#define is_join(pnum) (dtabpoin->d_element[(pnum)]->type == JOIN_PRED || \
                dtabpoin->d_element[(pnum)]->type == OBJECT_JOIN_PRED)
#define is_sjoin(pnum) (dtabpoin->d_element[(pnum)]->type == JOIN_PRED)
#define is_simple(pnum) (! is_join(pnum) && ! is_eof(pnum) )
#define is_restricted(bnum) (btabpoin->b_element[(bnum)]->restricted)
#define is_void(poin) ((poin) == QPNULL)
#define disjelem(pnum) (dtabpoin->d_element[(pnum)])
#define pathsize(pnum) (dtabpoin->d_element[(pnum)]->path_name.num_el)
#define pathsiz1(pnum) (dtabpoin->d_element[(pnum)]->path_name1.num_el)
#define predconj(pnum) (dtabpoin->d_element[(pnum)]->c_entry)
#define predvarb(pnum) (dtabpoin->d_element[(pnum)]->var_name)
#define predvar1(pnum) (dtabpoin->d_element[(pnum)]->path_name1.prop[0])
#define predpath(pnum) (dtabpoin->d_element[(pnum)]->path_name)
#define predpat1(pnum) (dtabpoin->d_element[(pnum)]->path_name1)
#define btabvarb(pnum) (btabpoin->b_element[(pnum)]->var_name)
```

```
#define xtabvarb(pnum) (xtabpoin->x_element[(pnum)]->varname)
#define xtabpath(pnum) (xtabpoin->x_element[(pnum)]->path)
#define is_foreach(bnum) (btabpoin->b_element[(bnum)]->quantifier == PV_FOR_EACH)



/*************************************************************************/
/*                                                                       */
/*                 INCLUDE FILE FOR QUERY EXECUTIVE                      */
/*                                                                       */
/*                 Version : 1.0                                         */
/*                                                                       */
/*************************************************************************/



/* Type of operation for the selected access method:  */

typedef enum {
      KEYEDACC,       /* A simple or composite keyed access is       */
                      /* detected.                                   */
      JOINVAR,        /* The access is made by reading the joinva-   */
                      /* riable OIDs from temporary file.            */
      SEQSCAN,        /* the access is made by sequential scanning.  */
      MULTIKEY,       /* multivalued accessed by key                 */
      MULTSCAN        /* multivalued accessed by scanning            */
             } QP_SKIND;

/* Type of variable into VarList  */


typedef enum {
     NEWTARGET,       /* the variable is the new target after decomp. */
     NEWJOINV,        /* the variable is the join variable after dec. */
     UNCHANGED        /* the variable is unchanged after decomposit.  */
            } QP_MODIF ;



/* SubqHead: for each subquery points to its Bindings and Conjuncts in
   Btable and Ctable  */

struct SubqHead {
                  int subqnumb ;          /*               num of subqueries */
                                          /* Btab elements  of this subquery */
                  struct VarList  *vlist[MAXSUBQ];
                                          /* Ctab elements  of this subquery */
                  struct ConjList *clist[MAXSUBQ];
                                          /*                evaluation  tree */
                  struct VarTree  *vtree[MAXSUBQ];
                                          /*            element of part tree */
                  struct VarTree  *eloftree[MAXSUBQ];
                };
```

```
/* VarTree: variable for this subquery : initialized by decomposer */

struct VarList {
                int btabindx ;          /* entry in Btable                  */
                QP_MODIF newbvar ;      /* newtarget or newbind or unchang */
                struct VarList *next;   /* ptr to next element              */
               };


/* ConjList: points to the Celement of this subquery (initial. by decomp */

struct ConjList {
                int ctabindx ;              /* entry in Ctable        */
                struct ConjList *next ;     /* ptr to next     elem   */
               };


/* VarTree: implements the variable tree and the element of tree        */

struct VarTree {
                int btabindx ;              /* entry in Btable        */
                int dtabindx ;              /* ind in Dtab case ELOF  */
                QP_MODIF newbvar ;          /* newtarget or newbind   */
                QP_SKIND acckind;           /* access method type     */
                T_OID  curobj   ;           /* cur. obj in multival.  */
                struct Schedule *selem;     /* ptr to the sched.list  */
                struct Schedule *acclist;   /* ptr to the accmeth.list*/
                struct PathDesc *pathd;     /* ptr to the path descr. */
                char *predicate;            /* ptr to predic. if key  */
                char predname[NAME_LENGTH] ;     /* name of this predicate */
                struct VarTree *prev;       /* ptr to previous elem   */
                struct VarTree *next ;      /* ptr to next     elem   */
                struct VarTree *side ;      /* ptr to side     elem   */
               };



/* Schedule : schedule list element   */

struct Schedule {
                int cost;                   /* cost for evaluate this */
                int ctabindx ;              /* index in ctable        */
                struct Schedule *next ;     /* ptr next schedule elem */
               };

/* structure holding the path descriptions and the OBJids */

struct PathDesc {
                char prop[NAME_LENGTH] ;        /* name of var. or prop.  */
                T_OID thisobj;              /* current obj for this   */
                struct PathDesc *next;      /* ptr to son             */
                struct PathDesc *side;      /* ptr to brother         */
               };
```

[KLUG82] Klug A., "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions", Journal of ACM, Vol.29, N.3 (July 1982), pp.699-717.

[MAIE83 Maier D., "The Theory of Relational Databases", Computer Science Press, 1983.

[MAI86] Maier D., Stein J, "Indexing in an Object_Oriented DBMS", IEEE Asilomar Conference , 1986.

[SELI79] Selinger P.G., et Al., "Access Path Selection in a Relational Database Management System", Proc. of ACM SIGMOD, May 1979.

[SERVDES88] ,Esprit project n. 834 Comandos, "Services Design", d1.t3.2.2.2-880331, March 1988.

[STON77] Stonebraker M., and Neuhold E., "A Distributed Version of INGRES", Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks, May 1977.

[STON83]Stonebraker M., and Al., "Performance Enhancement to a Relational Database System", ACM Trans. Database Systems, pp.167-185, 1983.

[WONG76] Wong E., Youssefi K., "Decomposition - a Strategy for Query Processing", ACM Trans. on Database Systems, Vol.1, N.3 (1976), pp.223-241.

[VALD86] Valduriez P., et Al., "Implementation Techniques of Complex Objects", Proc. VLDB, Kyoto, Aug.1986.

[VALD87] Valduriez P., "Join Indices", ACM Trans. on Database Systems, Vol.12, N.2, (June 1987), pp.218-246.

REFERENCES

[BAYE72] Bayer R., and McCreight E., "Organization and Maintenance of Large Ordered Indexes", ACTA INFORMATICA, Vol.1, N.3, 1972.

[BATO85]
Batory D.S., and Kim W., "Modeling Concepts for VLSI CAD Objects" in Proc. ACM-SIGMOD Conference, Austin (Texas), May 1985.

[BERT83] Bertino E., C.Meghini, G.Pelagatti, C.Thanos, "The Update Problem in the Distributed Database System HERMES/1", Proc. Second International Conference on Databases, Cambridge (U.K.), 1983.

[CERI84] Ceri S., and Pelagatti G., "Distributed Databases - Principles and Systems", MacGraw-Hill Book Company, 1984.

[CHAM81] Chamberlin D., et Al., "Support for Repetitive Transactions and ad hoc Queries in System R", ACM Trans. Database Systems, pp.70-94, 1981.

[CIS88]ARG, "Comandos Integration System - Detailed Design", Version 1, February 1988.

[COM87] Comandos (ESPRIT Project 834), "Object Oriented Architecture", Project Report, D1-t2.1-870904, September 1987.

[COME79] Comer D. "The Ubiquitous B-tree", Computer Surveys, Vol.11, N.2, (June 1979), pp.121-137.

[DEPP86] Deppish U., Paul H.B., Schek H.J., "A Storage System for Complex Objects", Proc. IEEE Workshop on Object Oriented DBMS, Asilomar (calif.), Sept.1986.

[EGE87] Ege A., and Ellis C., "Design and Implementation of GORDION, an Object Base Management System", Third Int. Conf. On Data Engineering (IEEE), Los Angeles (U.S.A.),

[LIND81] Lindasy B.G., "Object Naming and Catalog Management for a Distributed Database Manager", Proc. 2nd International Conference on Distributed Computing Systems, Paris, France, April 1981.

[LOHM85] Lohman G., et Al., "Query Processing in R*", in Query Processing in Database Systems, W.Kim, D.Reiner, D.Batory (eds.), Springer-Verlag, Heidelberg, 1985.

[LOHM87] Lohman G., "Grammar-like Functional Rules for Representing Query Optimization Alternatives", RJ 5992,IBM Almaden Research Center, San Jose, CA 95120,
Dec. 1987.

[KIM87] W. Kim et al., "Enhancing the Object-Oriented concepts for Database support",Proc. Workshop on Object Oriented DBMS, Asilomar (Calif.), September 1986.

[KHO86] Khoshafian S. N., Copeland G. P., "Object Identity", OOPSLA '86, Portland, Oregon

```
/* undelying  interface */

struct CisInterf {
int             op;                             /* called operation        */
char            schema[NAME_LENGTH];            /* schema name             */
char            name[NAME_LENGTH];              /* class/multiv name       */
char            prname[NAME_LENGTH];            /* predicate name          */
char            predicate[PRED_STRING_LENGTH];  /* predicate string        */
T_OID           obj1;                           /* current class object    */
T_OID           obj2;                           /* current component obj.  */
T_OID           retobj;                         /* returned object         */
T_ATTR_VALUE    retval;                         /* returned value          */
char            *retptr;                        /* returned pointer        */
};
```