Contents

List of Tables

List of Figures

Chapter 1

Introduction

In typical software development, the software developer usually cannot avoid designing a data structure which supports access and update to data residing in main memory. The design of this data structure determines the main-memory data retrieval time and thus plays an important role in the overall software performance. (This is particularly true for software systems which interact mainly with main memory.) The logical component of the software which is responsible for managing this data structure can be viewed as a dedicated database management system (DDBMS) which manages data residing in main memory (*a memory resident database*). Developing such a data management component usually contributes a major fraction of the software development time. Moreover, any change to the design of the data structure implies a partial or total amendment to the existing data management component, which could have major impact to the whole software development process.

The *Resident Database Manager* (RDM) [21, 23] is a software tool which helps with the development of main memory data management components for software

systems. RDM designs and generates code for dedicated database management systems with good performance. What RDM requires from the software developer is the information about the target data and how this data is going to be used by the software being developed. The software developer can then concentrate on the development of the remaining parts of the software by using an extended C language, which has additional constructs to allow interaction with the generated dedicated database management systems. When there are changes to the usage of data, or even the nature of the data itself, RDM can generate a new dedicated database management system form an updated version of the data information.

RDM maintains a set of generic *index types*. A particular dedicated database management system is generated with chosen appropriate index types and instantiates them as required to create indices. For example, if a general binary tree index type is chosen for storing a particular record type, an actual binary tree index instance will be generated for that record type in the dedicated database management system. (The terms *index* and *index type* have analogy with the terms *data* and *data type*.) The implementation method, characteristics, and performance of any index instance is determined by its corresponding index type. Since RDM selects indices from the defined index types and generates code to manage these indices, it implies that RDM has to have built-in information on the defined index types. However, there are certain disadvantages for hard-wiring index type information inside RDM:

1. the set of built-in index types may not suit the needs, and

2. it requires an internal update to RDM in order to introduce new index types.

Consequently, the ability for users to introduce new index types to RDM externally is a essential.

RDM generates two kinds of output source code: The source code comprising a dedicated database management system, and the source code generated by replacing all the extended C constructs from an user application written in the extended C language. Both kinds of output source code are generated in ANSI C [12].

In this thesis we consider the issues of *index type extensibility* in RDM. We also describe how *code generation* is done in RDM.

1.1 Related Research

There has been considerable interest in the database research community in developing methods and mechanisms to support database *extensions*. Projects involving database extension research include the EXODUS project at the University of Wisconsin [3], POSTGRES at the University of California at Berkerley [18], ENCOMPASS at Tandem Computers [19], PROBE at the Computer Corporation of America [11], GENESIS at the University of Texas at Austin [2], STARBURST at the IBM Almaden Research Center [15], and GRAL at Universität Dortmund [9].

Database extensions on these projects can be (grossly) classified as *user data extensions* or *data management extensions*. User data extensions concentrate on providing support for user-defined abstract data types and functions for fields of database records. Data management extensions provide alternative implementations of database storage and access paths.

User data type extensions provide support for domains [7] and allow application developers to specify data type representations, conventions, and functions on user data types, which can be performed by the database management system during the execution of database queries and updates. User data types not only enhance

application development productivity and integrity, but also improve system performance because predicates involving user-defined types can be evaluated by the DBMS to eliminate items which would otherwise be returned to the application.

Data management extensions provide alternative data storage methods, data access paths, and integrity constraints. Data management extensions allow a database system to evolve to support new hardware and application opportunities and requirements. As database management systems are applied to an ever broader range of applications, it is often desirable to provide database facilities which are specialized to the application requirements, in order to enhance application performance or simplify the application implementation.

One common feature for all the projects mentioned above is that they are intended for databases stored in secondary memory. Data management extensions on these projects therefore focus on storage and access methods appropriate to secondary memory. Index type extensions for handling main memory data are not considered. This thesis takes an initial step and provides a method for index type extensions to memory-resident data.

Code generation for most of RDM is straightforward. The only difficulty is on the code generation for query access plans, which involve complex control-flow structures. An internal abstraction called control-flow boxes is used to assist the code generation process. There are two related work (although with different objectives) to this: the first one is the LDL project at Microelectronic and Computer Technology Corporation [5]. It uses a *graph* to assist its code generation process and the graph has nodes annotated to specify relevant details of the execution. The second one is a Ph.D. thesis at Harvard University [8], which describes the process of translating relation queries into iterative programs.

1.2 Thesis Overview

The structure of the thesis is as follows. The next chapter will give a general overview on RDM. The internal structure together with the input and output languages of RDM will be described. An internal language used for expressing design information of dedicated database management systems will also be described. The third chapter will describe the internal design of the database management system. The fourth chapter will focus on topics relating to index type extension. The necessary information for RDM on an index type and a language used to express this information will be described. The fifth chapter will outline detailed algorithms and procedures for code generation in RDM.

Chapter 2

Overview of RDM

The *Resident Database Manager* (RDM) is a software tool set for applications that manipulate memory-resident databases. Almost all application software has components that access and update data residing in main memory. RDM helps with the development of such components in application software.

The sections in this chapter provide an overview of RDM. Section 2.1 introduces the two compilers in RDM and describes how they are used in developing software systems. The input languages of the two compilers and the basic data flow in RDM are also described. Section 2.2 describes the internal structure of RDM; some internal components of RDM are introduced. These components enable RDM users to have more control over the software development process. In Section 2.3, a design information language which is used among internal RDM components is also discussed.

2.1 User's View of RDM: Architectural Design

To use RDM to develop a software system, the software developer specifies a *database schema*, which consists of data definition and data manipulation requests, in an object-oriented language called LDM (LDM is an acronym for *Logical Data Model*). The remainder of the software system is written in an extended C language called C/DB. C/DB has additional language constructs which permit direct use of data manipulation requests written in LDM. RDM compiles LDM and C/DB sources, and generates C code as output which can then be compiled with existing C compilers.

Architecturally, RDM consists of two compilers, namely the LDM compiler and the C/DB compiler. The LDM compiler takes an LDM source file as input and produces the source code of a dedicated database management system (DDBMS), which is capable of managing memory-resident databases of the database schema specified in the LDM source. The C/DB compiler takes a C/DB source file as input and replaces all the data manipulation requests, which are expressed in extended C constructs, by ANSI C code. Output of the C/DB compiler, which is called the *Application Source*, and the DDBMS source can then be compiled with a C compiler to produce an object code file. Any number of object code files can be linked to form the final executable code. A summary of the architectural view of RDM is depicted by the data-flow diagram in Figure 2.1. (Data-flow diagrams in this chapter assume the following conventions: activities are represented by circles, data sources are represented by boxes, group activities are represented by dotted boxes, and data-flow between activity and data source is represented by an arrow with a cardinality. The cardinality specifies the number of data sources that the activity takes or produces.)

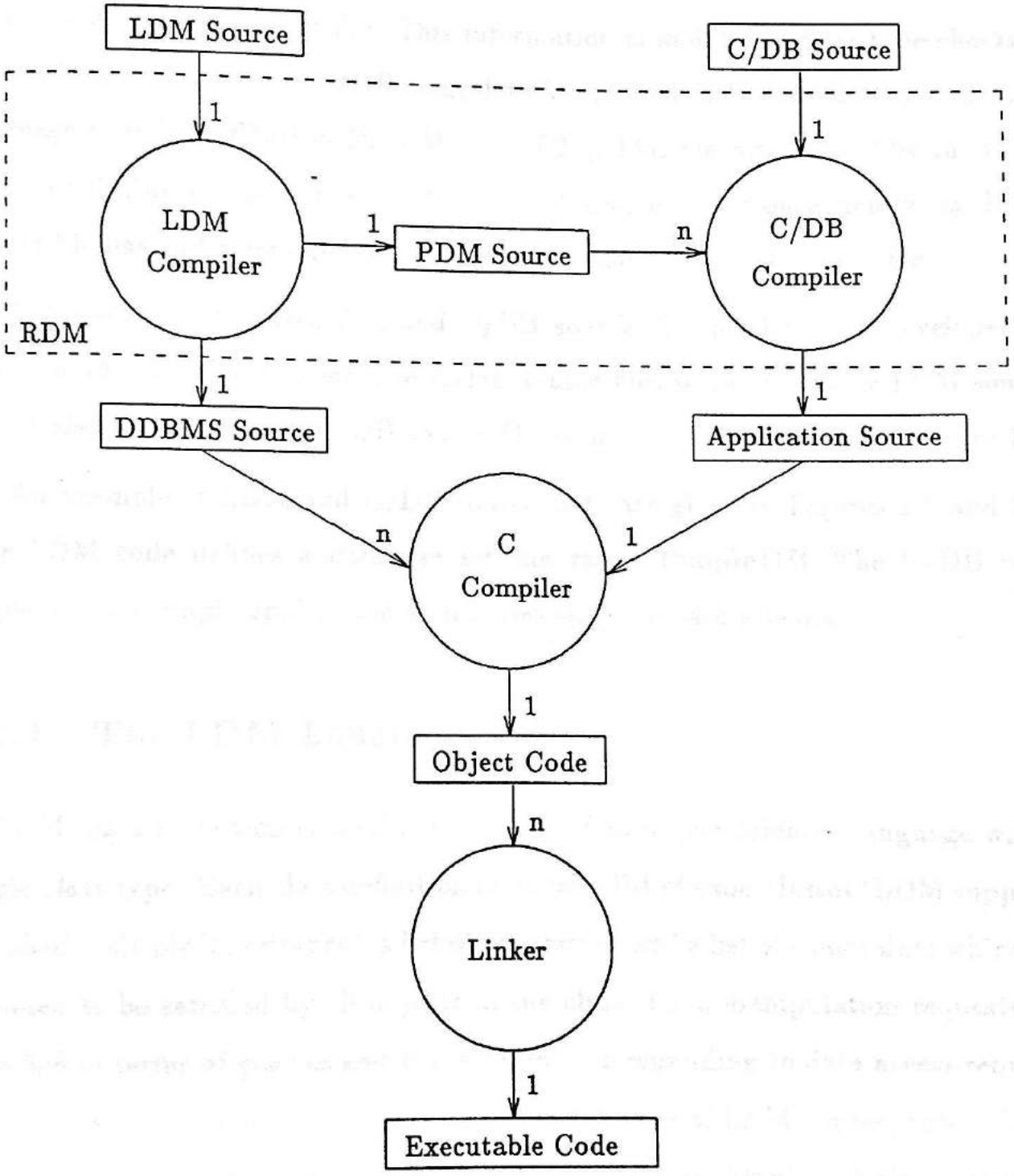


Figure 2.1: Basic Data-flow in RDM

The LDM compiler and C/DB compiler work dependently. In fact, to compile a C/DB source file, the C/DB compiler has to have some information about the corresponding LDM source file. This information is mainly used for type checking on the C/DB source. The C/DB compiler obtains such information from a file in a language called the *Physical Data Model* (PDM). This file is generated by the LDM compiler during the compilation of the LDM source file. Consequently, an LDM source file has to be compiled before the corresponding C/DB source file.

Moreover, LDM source files and C/DB source files need not be developed in pairs. RDM allows any number of C/DB source files to use the same LDM source file; it also allows a single C/DB source file to use more than one LDM source file.

An example of LDM and C/DB source code are given in Figures 2.2 and 2.3. The LDM code defines a database schema called **PeopleDB**. The C/DB code implements a simple application which uses this database schema.

2.1.1 The LDM Language

In LDM, data definition is specified in terms of an object-oriented language with a single *class* type. Each class definition includes a list of superclasses (LDM supports so-called multiple inheritance), a list of properties, and a list of constraints which are assumed to be satisfied by all objects in the class. Data manipulation requests are specified in terms of *queries* and *transactions* corresponding to data access requests and data update requests respectively. In the sample of LDM source, three classes are defined in lines 4 to 12, namely class 'Person', class 'Man', and class 'Woman'. The two queries and two transactions specified in lines 23 to 37 behave as follows:

AllPeople — This query returns all the Person objects (objects of the class Person) that are stored in the database.

```
1.  schema PeopleDB
2.
3.
4.  class Person
5.  properties Name, Age
6.  constraints
7.  Id determined by Name
8.  cover by Man, Woman
9.
10. class Man isa Person
11.
12. class Woman isa Person
13.
14. property Name on String maxlen 20
15. property Age on Integer range 16 to 75
16.
17. size Man 500
18. size Woman 500
19.
20. store PersonStore of type dynamic storing Man, Woman
21.
22.
23. query AllPeople
24. select P
25. from Person
26.
27. query ManWithName
28. given N from Name
29. select M from Man where M.Name = N
30.
31. transaction EnterMan given N, A from Name, Age
32. declare M from Man
33. insert M (M.Name := N; M.Age := A)
34.
35. transaction EnterWoman given N, A from Name, Age
36. declare W from Woman
37. insert W (W.Name := N; W.Age := A)
```

Figure 2.2: A LDM Source Sample

```
1.  #include <stdio.h>
2.
3.  schema PeopleDB DB1, DB2;
4.
5.  main()
6.  {
7.      prop Person P1, P2;
8.      prop Name N;
9.      prop Age A;
10.
11.     int DBNum;
12.     char Type[10];
13.
14.     invoke InitPeopleDB() in DB1;
15.     invoke InitPeopleDB() in DB2;
16.
17.     /* input */
18.
19.     while (scanf("%d %s %s %d", &DBNum, Type, N, &A) != EOF)
20.         if (DBNum == 1)
21.             if (strcmp(Type, "Man") == 0)
22.                 invoke EnterMan(N, A) in DB1;
23.             else
24.                 invoke EnterWoman(N, A) in DB1;
25.         else
26.             if (strcmp(Type, "Man") == 0)
27.                 invoke EnterMan(N, A) in DB2;
28.             else
29.                 invoke EnterWoman(N, A) in DB2;
30.
31.     /* query */
32.
33.     for P1 in AllPeople in DB1
34.     {
35.         printf("%s %d\n", P1@Name, P1@Age);
36.         if P2 in ManWithName(P1@Name) in DB2
37.             printf("----------\n");
38.     }
39. }
```

Figure 2.3: A C/DB Source Sample

ManWithName — This query returns a Man object with a given Name (Name is a property of class Man) only if such a Man object exists.

EnterMan — This transaction creates a new Man object, initializes its Name and Age properties and inserts the object in the database.

EnterWoman — This transaction creates a new Woman object, initializes its Name and Age properties and inserts the object in the database.

The reader is referred to [23] for further details concerning the syntax and semantics of LDM.

2.1.2 The C/DB Language

The C/DB language is an extension of the C language that includes additional constructs to enable:

- declaring memory-resident databases from database schemas,
- declaring object-valued variables,
- accessing object properties,
- invoking queries, and
- invoking transactions.

Examples of these extented features appear in the C/DB source example in Figure 2.3. The example implements a trivial application based on the **PersonDB** database schema. When executed, the application will input and store Person objects in on of two memory-resident databases DB1 and DB2. Then a list of

names and ages for all Person objects in the database DB1 will be printed out. If the name of a Person object in this list matches the name of a Man object in the database DB2, the name of this Person object will be underlined in the output list.

The schema statement in line 3 of the C/DB source example declares the two databases for the application. Both of these databases are declared to be on the database schema PersonDB. Thus, they are capable of storing objects that have been defined in this schema; and the application can manipulate objects in these databases by the data manipulation requests that have been specified in this schema.

Lines 7 to 9 declare four object-valued variables. Line 7 declares two variables named P1 and P2 for referring to object instances of the class Person. Line 8 and 9 each declares a variable for referring to an instance of Name object and Age object respectively.

Object properties are accessed by means of a new '@' operator. The operator is used in lines 35 and 36 to access the Name property and Age property of a Person object referenced by the variable P1.

Invocations of database transactions occur in lines 14, 15, 22, 24, 27 and 29 of the C/DB source. The transaction InitPeopleDB which is invoked in lines 14 and 15 initializes the two databases DB1 and DB2. Database initialization is required before a database can be used. The transaction for initializing databases of a schema is automatically created by RDM when the corresponding LDM source file is compiled. Transactions EnterMan and EnterWoman are invoked in lines 22, 24, 27, and 29 to create new Man objects and Woman objects in the two databases.

In general, transactions are invoked with for form

invoke *Transaction-Name*(*Expression-List*)[**in** *Database-Name*]

where *Transaction-Name* is the name of the transaction, the *Expression-List* is a sequence of input argument expressions for the transaction, and the **in** clause specifies the database on which the transaction is to be executed. The **in** clause in a transaction invocation is optional (as indicated by the square brackets), and it can be omitted when only one database has been declared in the application (so the objective is well defined).

An invocation of a transaction can return a value (object) in the same way as any other C function. In fact, transaction invocations can be placed in all locations where ordinary function calls are allowed in the C language.

Invocations of database queries occur in lines 33 and 36. The invocation of the query AllPeople returns all the Person objects from the database DB1; and the invocation to the query ManWithName, which is nested inside the **for** statement that supports the invocation to the query AllPeople, returns one Man object with a given name from the database DB2.

The two query invocations are specified in terms of new forms of **for** and **if** statements. There is also a new form of **while** statement that supports a query invocation. The forms of these statements are as follows:

if *Query-Invocation Statement*$_1$ [**else** *Statement*$_2$]

while *Query-Invocation Statement*

for *Query-Invocation Statement*

The query invocation specified in **if** and **while** statements must return a single solution. (A query solution consists of a set of objects. Each of them is referenced by an object-valued variable.) If such a solution exists, the if statement will evaluate the argument *Statement*$_1$. If the **if** statement is supplied with an else clause, the argument *Statement*$_2$ will be evaluated when no solution to the query is found.

If a solution exists in the case of a while statement, the argument *Statement* will be evaluated, and the whole process, including the Query-Invocation, will be repeated.

The new form of for statement may take queries that return more than one solution. Its argument *Statement* is evaluated for each query solution.

A Query-Invocation has the format

[*Identifier-List* in] *Query-Name* [(*Expression-List*)] [in *Database-Name*]

where *Query-Name* is the name of the query and the *Expression-List* is a sequence of argument expressions which supply input parameter values to the query. In the same way as a transaction invocation, a database name is required when more than one database has been declared in the application. The *Identifier-List* is a sequence of object-valued variable names. The variables are bound to each component of a query solution. They serve as the means of query result communications.

2.2 User's View of LDM Compiler: Detailed Design

In a detailed design user's view, the LDM compiler is composed of three compilers, referred to as the DML compiler, the C/PDM compiler, and the C/ITS compiler. The data-flow among the three compilers is outlined in Figure 2.4.

The DML compiler (DML stands for Data Manipulation Language) is responsible for generating a PDM source file based on an input LDM source file together with some files that contain performance information on the available user defined index types. The PDM source encodes a number of decisions:

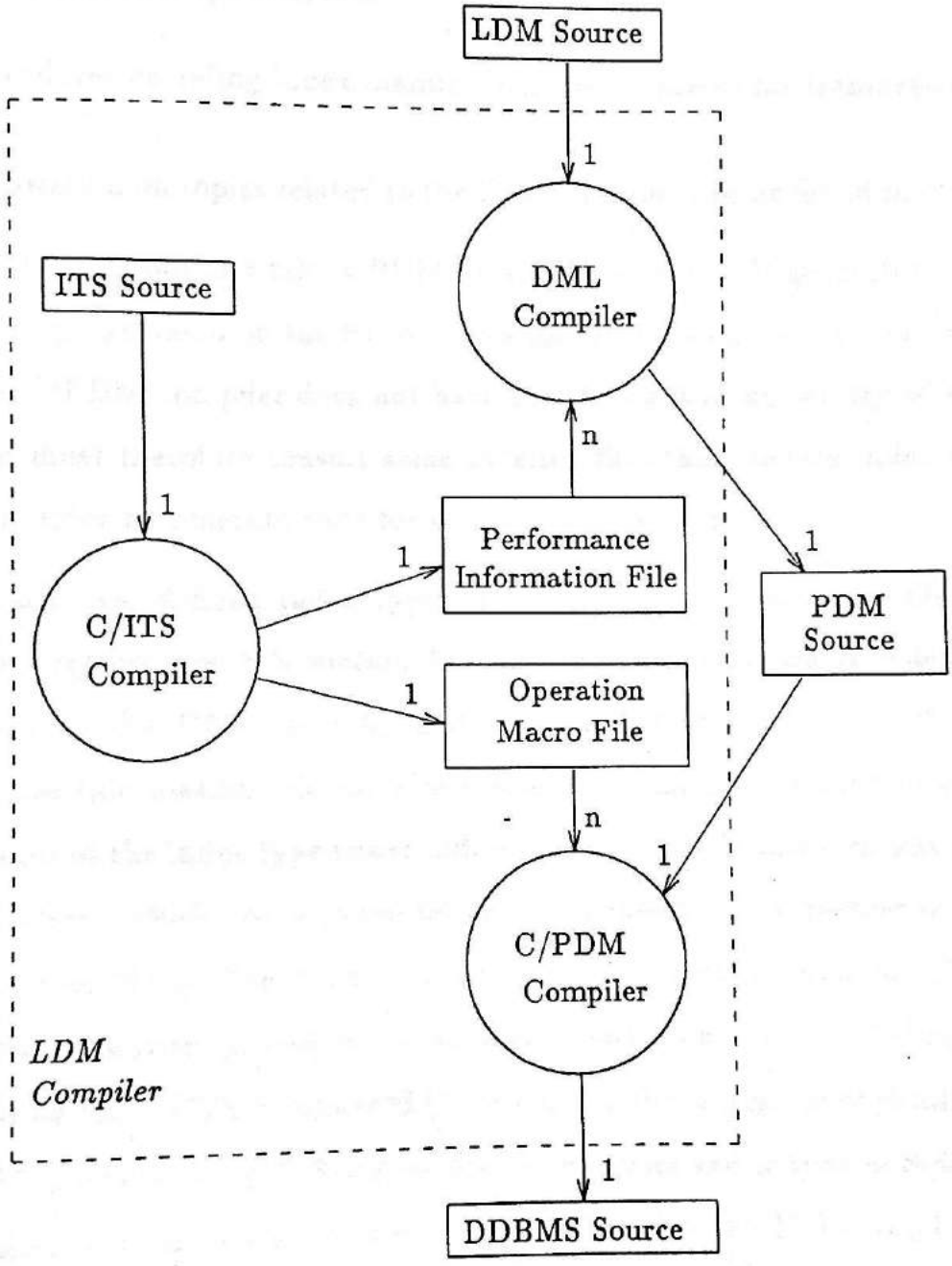


Figure 2.4: Data-flow in the LDM Compiler

- an internal encoding of the database (including a selection of indices),
- access plans for queries, and
- procedures encoding index maintenance requirements for transactions.

More elaboration on topics related to the DML compiler can be found in [21, 22, 24].

The C/PDM compiler take a PDM source file as input. It generates a DDBMS in the C language based on the PDM source file. In the same way as the DML compiler, the C/PDM compiler does not have specific built-in knowledge of any index type, and must therefore consult some external files that contain index operation macros in order to generate code for index related operations.

For each user defined index type, there are two corresponding files referred to as the *Performance Information File* and the *Operation Macro File* which are for the use of the DML compiler and the C/PDM compiler respectively. The Performance Information File contains information that can be used to predict the performance of the index type under different situations. It also contains additional data definitions which are required for the implementation of indices of this type. The Operation Macro File contains a set of generic operation macros. Each macro defines the execution procedure of an index operation. Both of these files are generated by the C/ITS compiler (ITS stands for *Index Type Specification*). Input to the ITS compiler is an ITS source file, in which an index type is defined. After a new index type has been defined and compiled with the ITS compiler, RDM is then able to create and use indices based on the new index type.

In the detailed design level of the LDM compiler, users can define index types which better suit their needs. Moreover, they can overrule the decisions made by the DML compiler on the DDBMS implementation by making appropriate changes

in the PDM source. In the extreme case, users can by-pass the DML compiler and code in PDM directly.

2.3 The PDM Language

The Physical Data Model (PDM) is used to express detailed design information relating to a particular DDBMS. PDM source is normally generated by the DML compiler from LDM source, and is mainly used by the C/PDM compiler to produce DDBMS source. The C/DB compiler also uses information encoded in PDM for type checking C/DB source.

For each class definition in a LDM source file, there is a corresponding class definition in the PDM source file. The class definition has a set of field names and a *reference method* added into it. This supplementary information basically determines how objects in the class are encoded, and translate more-or-less directly as a *structure type* in C[1].

For each query or transaction in an LDM source file, there is a corresponding *compiled query* or *compiled transaction* stated in the PDM source file. A compiled query or compiled transaction is eventually translated to a *function* in C (functions that implement queries are called *query functions*; whereas functions that implement transactions are called *transaction functions*). The forms of compiled queries and compiled transactions are similar in structure. Both have constructs to specify input information and detailed execution procedures. These execution procedures are specified in terms of database objects, and typically assume the existence of some database indices and storage managers.

[1]PDM is actually independent of high-level programming languages. C is used for illustration simply because it is the language that the current version of RDM has adopted.

Database indices and storage managers are also defined in the PDM source. Database indices are chosen by the DML compiler according to their support for query evaluation. Storage managers are logical components responsible for memory management.

A PDM source generated by the DML compiler is listed in Figures 2.5 and 2.6. The PDM source is generated from the LDM source listed in Figure 2.2.

Four **class** definitions appear in lines 3 to 27 of the PDM source. The first three correspond to class definitions in the LDM source. The last one, PeopleDBStruct, is added by the DML compiler to represent the database schema. Each object of this class represents a resident database for the PeopleDB database schema. (The term *schema structure type* is used in the rest of this thesis for the C structure type that represents this special class.)

The first class definition corresponds to the class Person. Logical properties of the class Person are stated in the **properties** clause in line 4. Six fields are listed in lines 7 to 12. The first fields ('Msc') encodes a type identifier for the objects in this class. The second, third, and fourth fields ('PersonTreeMark', 'PersonTreeRSon', and 'PersonTreeLSon') encode an index for objects of this class. The last two fields ('Name' and 'Age') encode the logical properties of the class Person.

The **reference** clause in line 5 indicates that the reference method for Person objects is *direct*. This indicates that an internal reference to a Person object can correspond directly to the address of a structure encoding field values for the object. The other possible reference method option is *indirect*. A class requires this reference method if objects in the class are allowed to change their *type* to other classes. This suggests that each structure encoding field values for an object in the class must be referred to indirectly through a pointer, which we call an *indirect cell* (ICell). When an object changes its type, a new structure is assigned to represent

```
1.  schema PeopleDB
2.
3.  class Person
4.  properties Name, Age
5.  reference direct
6.  fields
7.      Msc;
8.      PersonTreeMark;
9.      PersonTreeRSon;
10.     PersonTreeLSon;
11.     Name;
12.     Age
13.
14. class Woman isa Person
15. reference direct
16. extension of Person
17.
18. class Man isa Person
19. reference direct
20. extension of Person
21.
22. class PeopleDBStruct
23. reference direct
24. fields
25.     PersonStore;
26.     PersonTreeHead;
27.     PersonTreeLock
28.
29. property PersonStore on StoreTemplate
30. property PersonTreeMark on Integer range 0 to 1
31. property PersonTreeRSon on Person
32. property PersonTreeLSon on Person
33. property PersonTreeHead on Person
34. property PersonTreeLock on Integer range 0 to 1
35. property Msc on Integer range 1 to 2
36. property PeopleDBStruct on PeopleDBStruct
37. property Person on Person
38. property Man on Man
39. property Woman on Woman
40. property Age on Integer range 16 to 75
41. property Name on String maxlen 20
42.
43. index PersonTree on Person
```

Figure 2.5: A PDM Source Sample (a)

```
44.  of type BINTREE
45.  ordered by Man, Name asc
46.
47.  store PersonStore of type dynamic
48.  storing Woman, Man
49.
50.  query AllPeople
51.  select P from Person
52.  nest
53.      assign P as each of PersonTree;
54.      end
55.
56.  query ManWithName
57.  given N from Name
58.  select M from Man
59.  nest
60.      assign M as first of PersonTree where
61.      M IN Man,
62.      verify M.Name = N;
63.      end
64.
65.  transaction EnterMan
66.  given N, A from Name, Age
67.  declare M from Man
68.  actions
69.      allocate M from PersonStore;
70.      M.Name := N;
71.      M.Age := A;
72.      insert M in PersonTree
73.
74.  transaction EnterWoman
75.  given N, A from Name, Age
76.  declare W from Woman
77.  actions
78.      allocate W from PersonStore;
79.      W.Name := N;
80.      W.Age := A;
81.      insert W in PersonTree
82.
83.  transaction InitPeopleDB
84.  actions
85.      init store PersonStore;
86.      init index PersonTree
```

Figure 2.6: A PDM Source Sample (b)

this object, and only the value in the ICell has to be updated to keep all references to the object valid.

The second and third class definitions correspond to the classes Woman and Man. They are indicated as subclasses of the class Person by the keyword isa. Moreover, they are specified as *extension* of the class Person by the **extension** clauses in lines 16 and 20. When a class is specified as an extension of another class, the former class will inherit all the fields from the latter class. In this particular example, the class Woman and the class Man will have the same set of fields as the class Person.

For each class name and field name mentioned in the class definitions, there is a corresponding **property** statement given in the PDM source. These property statements specify the *type* of each class name and field name. A type can be one of the user defined classes, or it can be one of the four basic classes: Integer, Real, DoubleReal, and String. Also, there is a special class for storage managers called StoreTemplate. In the sample PDM source, property statements are listed in lines 29 to 41.

An index for the Person objects called PersonTree is specified in the index statement in lines 43 to 45[2], and the type of this index is specified to be BINTREE (which stands for *binary tree*). An index statement may include three types of supplementary clauses: *order by*, *distributed on*, and *maximum size*. These clauses supply additional information for indices that carry special characteristics (which will be discussed in chapter 4). For this particular index statement, an *order by* clause appears in line 45.

A storage manager called PersonStore is specified in lines 47 to 48 for the classes

[2]The present version of DML compiler does not support automatic index selection. Thus, the same index declaration is required in the LDM source.

Woman and Man. PersonStore is then responsible for memory space allocation for objects of these two classes.

Two compiled queries, which correspond to the queries AllPeople and ManWithName, and three compiled transactions, which correspond to the transactions EnterMan, EnterWoman, and InitPeopleDB, are stated in lines 50 to 86. The compiled transaction InitPeopleDB is added by the DML compiler for database initialization. Input to queries and transactions is specified by given clauses in their compiled forms. Given clauses for query ManWithName, transaction EnterMan, and transaction EnterWoman appear in lines 57, 66, and 75 respectively. Objects that are expected for a query solution are specified in the select clause. Select clauses for queries AllPeople and ManWithName appear in lines 51 and 58. Access plans for queries are expressed in terms of an *object-oriented algebra* [4, 10, 13, 14]. Access plans for queries AllPeople and ManWithName are specified in lines 52 to 54 and lines 59 to 63 respectively. Execution procedures for transactions are expressed by means of *transaction operation statements*. Transaction operation statements in lines 69 to 72, lines 78 to 81, and lines 85 to 86 specify the execution procedures for transactions EnterMan, EnterWoman, and InitPeopleDB respectively.

2.3.1 The Transaction Operations

Every compiled transaction contains a sequence of transaction operation statements. Each statement is in turn any of the operations list in Table 2.1, or a control-flow **if-else** statement with the following form.

if *predicate* **then**

Transaction-Operation-Statements

[**else**

Transaction Operation Statement Syntax	Operation Name
v.p := property-value	Assignment
remove v from I	Remove
insert v in I	Insert
create v for I	Create
destroy v for I	Destroy
copy v_1 to v_2 for I	Copy
allocate v from S	Allocate
free v to S	Free
allocate indirect v from S	Allocate Indirect
free indirect v to S	Free Indirect
v_1 id := v_2	ICell Assignment
alloc id v	ICell Allocate
free id v	ICell Free
init index I	Init Index
init store S	Init Store

Note:
- v, v_1, v_2 are object-valued variable names
- p is a property name
- I is an index name
- S is a store name

Table 2.1: Transaction Operation Statement Syntax

Transaction-Operation-Statements]

endif

The *predicate* in the **if-else** statement has the following form

term operator Class-Name

which expresses a *type* check on a *term*. A *term* always refers to an object in the form

Variable-Name [. *Property-Name*]*

which consists of an object-valued variable name followed by (possible empty) sequence of valid property names separated by dots. For example, in line 70 of the PDM source in Figure 2.6, 'M.Name' is a term which specifies a *String* object.

There are two possible operators which can occur in a predicate: **is** and **in**. The operator **is** checks if the *most specialized class* of the object denoted by *term* is *Class-Name*. The operator **in** checks if the *most specialized class* of the object denoted by *term* is a subclass of *Class-Name*. Both kinds of type checking operators are needed. For example, the **is** operator is used for selecting an appropriate storage manager. The **in** operator is often used in the role of a *subclass* check in the case of index scans. The predicate 'P **in** Man', for example, can be used to select a subset of all Person objects.

There are fifteen different kinds of transaction operations. A brief description for most of them is given as below. The five operations that have been left out (Remove, Insert, Create, Destroy, and Copy) are operations that are used to manipulate database indices, and are discussed in Chapter 3.

Assignment — Assign a property value to a property of an object.

Allocate — Allocate memory space for encoding property values of an object from a storage manager, and assign the starting address to the contents of an object-valued variable.

Free — Return the memory space used for encoding property values of an object to the storage manager. The object is referenced by an object-valued variable.

Allocate Indirect — Allocate memory space for encoding property values of an object from a storage manager, and assign the starting address to the contents of the ICell referenced by an object-valued variable.

Free Indirect — Return the memory used for encoding property values for an object to the storage manager. (The contents of an ICell determines the memory address.)

ICell Assignment — Assign the contents of one ICell to the contents of another.

ICell Allocate — Allocate memory space for an ICell from a storage manager (called ICellStore) and assign its starting address to an object-valued variable.

ICell Free — Return the memory space used by an ICell to ICellStore. The ICell is referenced by an object-valued variable.

Init Index — Initialize a database index.

Init Store — Initialize a storage manager.

2.3.2 The Query Access Plan

Query access plans are expressed in terms of an object-oriented algebra. Expressions in the algebra specify strategies based on both index scans and property value navigation; and may therefore considered *low level* or *procedural*. We refer to expressions in this algebra as *access plan expressions* (APEs).

An APE is an expression tree with internal and leaf nodes corresponding to one of the operations listed in Table 2.2. An diagrammatic conventions for APE expression trees are indicated in Figures 2.7 and 2.8. We view each operation as an abstract data type consisting of an *init* procedure and a *next* procedure. Both procedures return either *success* or *failure* depending on whether or not the operation has finished producing a *result*. An operational semantics for each of the APE operations is specified in Appendix B.

The communication of results among operations in an APE is accomplished by means of a *work area* specific to the query, called a *query environment* (QE). The C code eventually generated for an APE will access fields within a structure encoding a QE for a given query. The QE for an APE also manages any storage required for necessary temporary results (as in the case of **Sort** and **Projection** operations).

The following APE

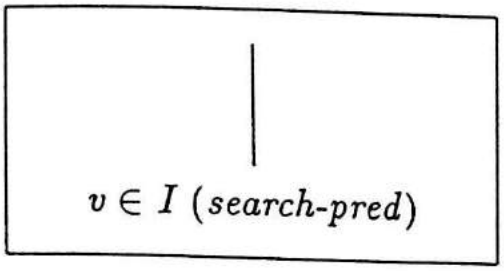
```
nest
    assign T as each of PersonIndex;
    assign M as T in Man;
    verify M.Name = N;
    cut T;
    end
```

specifies the access plan for the query

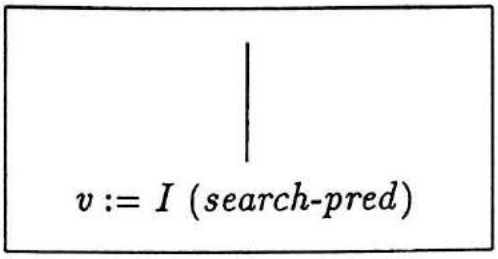
APE Syntax	APE Operations
assign v as each of I [where *search-pred*]	Index Scan
assign v as first of I [where *search-pred*]	Index Lookup
assign v as t in C	Conditional Assignment
assign v as t	Assignment
verify *selection-pred sub-expression*	Selection
compliment *sub-expression*	Compliment
cut v *sub-expression*	Cut
project $v_1,\ldots,v_n$ *sub-expression*	Projection
sort *sort-info sub-expression*	Sort
nest *sub-expression*; *sub-expression-list*; end	Nested Cross Product

Note:
- v, $v_1,\ldots,v_n$ are object-valued variable names
- t is a term
- I is an index name
- C is a class name
- *sub-expression* is an algebraic sub-expression
- *sub-expression-list* is a list of algebraic sub-expressions separated by semi-colons

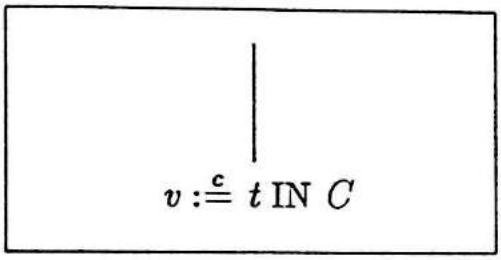
Table 2.2: APE Syntax



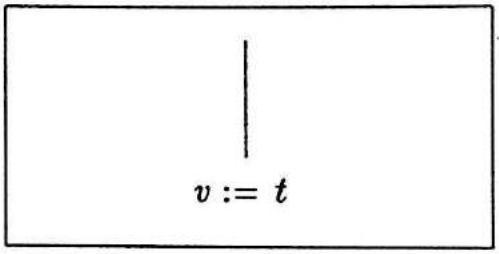
Index Scan



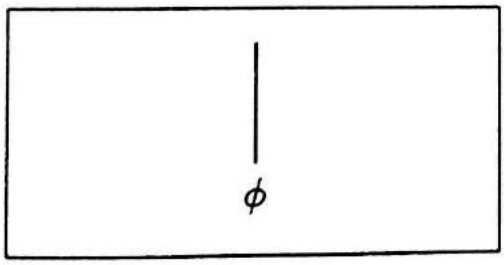
Index Lookup



Conditional Assignment



Assignment

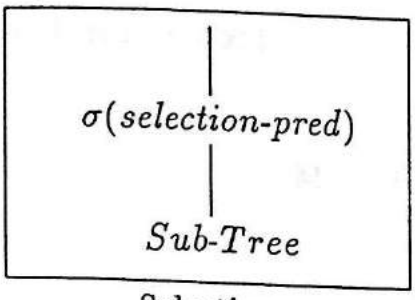


Null

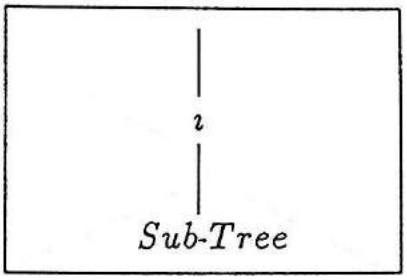
Note :

- v is a variable name
- t is a term
- I is an index name
- C is a class name

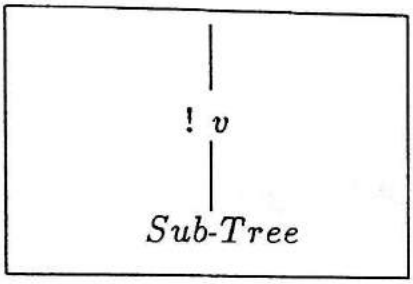
Figure 2.7: Diagrammatic Representation of APE Operations (a)



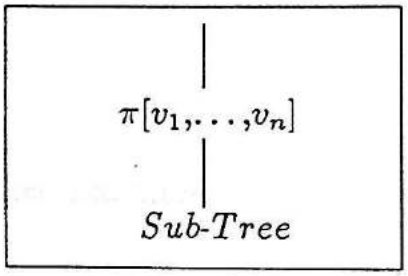
Selection



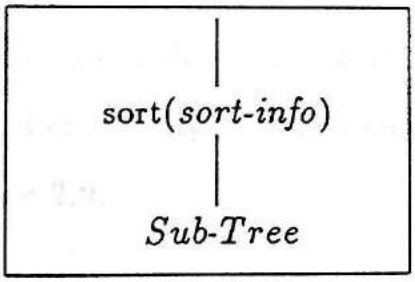
Complement



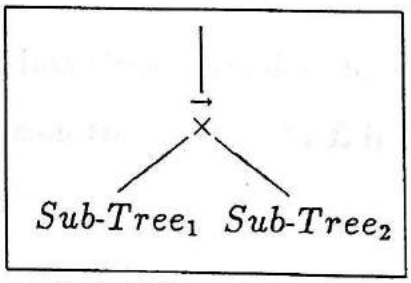
Cut



Projection



Sort



Nested Cross Product

Note: – v, $v_1,\ldots,v_n$ are variable name
– *Sub-Tree*, $Sub\text{-}Tree_1$, $Sub\text{-}Tree_2$ represent expression sub-trees

Figure 2.8: Diagrammatic Representation of APE Operations (b)

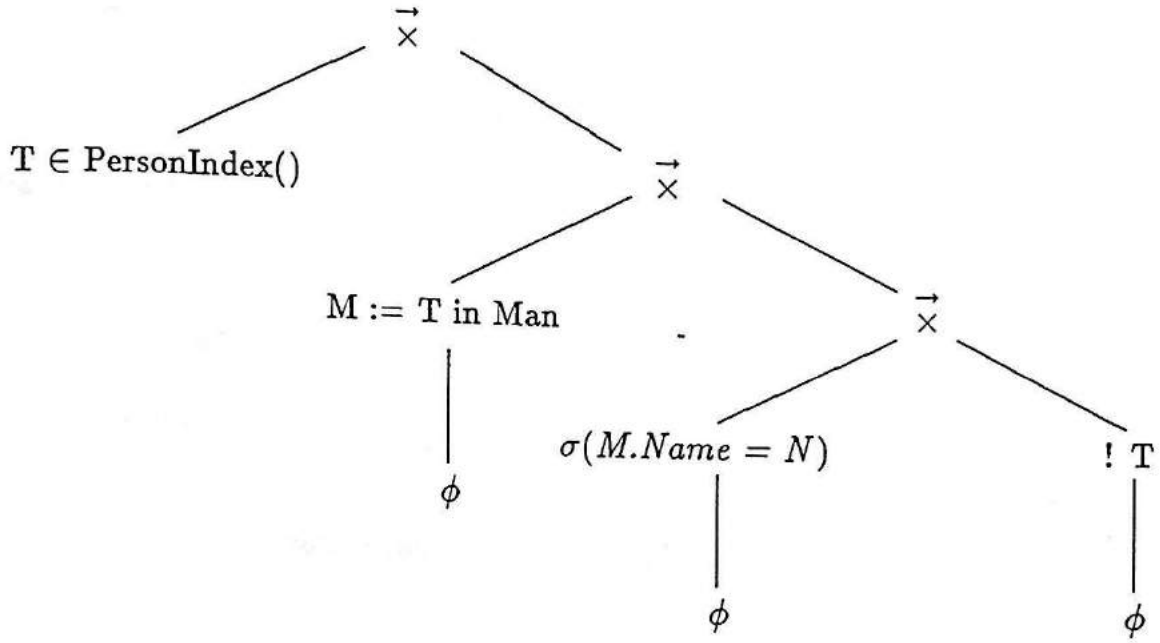


Figure 2.9: An Expression Tree Example

Select one Man object with a given name.

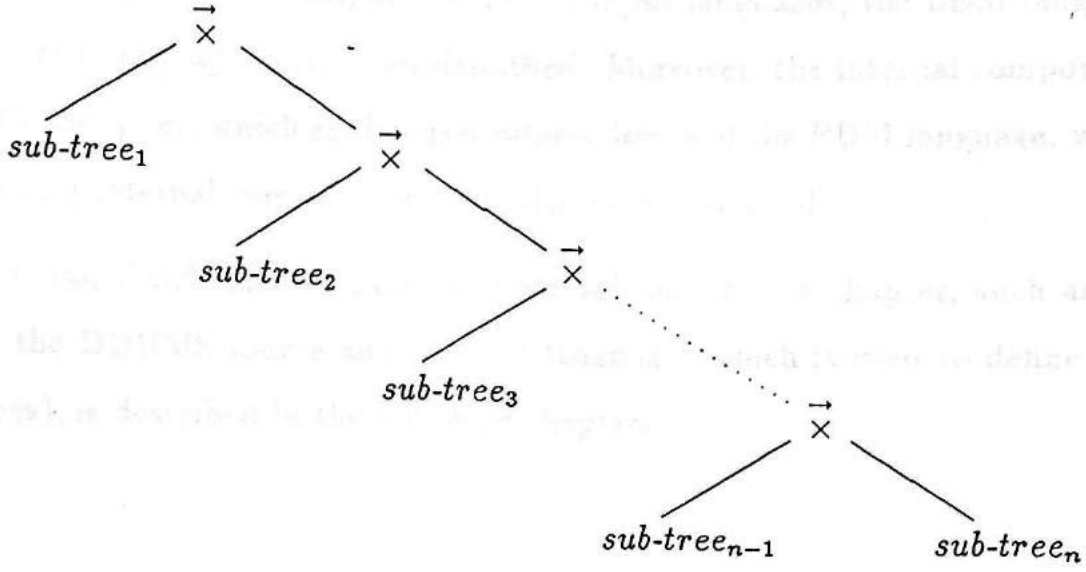
where the non-ordered index 'PersonIndex' on the class Person provides an ability to *visit* all Person objects. The corresponding expression tree for the APE is specified in Figure 2.9.

Note that when an expected sub-expression is omitted from an APE, a **Null** operation is assumed to represent the missing sub-expression in the expression tree. Also, the APE syntax that corresponds to the Nested Cross Product operation can take one or more sub-expressions. In general, APEs of the form

nest
sub-expression$_1$;
sub-expression$_2$;
sub-expression$_3$;
:
sub-expression$_{n-1}$;

sub-expression$_n$;
end

are translated to



where *sub-tree*$_i$ is the corresponding expression tree of the *sub-expression*$_i$. The expression tree has one or more Nested Cross Product operations depending on the number of sub-expressions in the APE.

A rough semantics for the expression tree example in Figure 2.9 is as follows. Objects associated to the index 'PersonIndex' are retrieved by an index scan. Each retrieved object is passed through a type check for Man objects. Then the Name of each Man object is compared to the given name. Once a qualifying Man object is found, a cut operation is executed to terminate the index scan (since only one qualified Man object is required for the query).

2.4 Summary

In this chapter, a general overview of the structure and usage of RDM has been given. In particular, the functions of the two compilers that compose the RDM, the LDM compiler and C/DB compiler, and their input languages, the LDM language and the C/DB language, have been described. Moreover, the internal components of the LDM compiler, which enable user interaction, and the PDM language, which is used among internal components, have also been discussed.

Some detailed information that has been left out in this chapter, such as the design of the DDBMS source and the ITS language (which is used to define user index types), is described in the following chapters.

Chapter 3

Internal Design of DDBMS

Software development with the use of RDM results in two sets of C source code. The first set implements dedicated database management systems (DDBMSs) for the application. The second set consists of the remainder of the application source code which contains calls to the DDBMSs for the purpose of managing memory-resident databases (MRDBs). Thus software systems developed with the use of RDM have a general structure shown in Figure 3.1.

In this chapter, the internal design of the DDBMS is presented. Section 3.1 introduces the three-level hierarchy in the internal DDBMS architecture. Sections 3.2 to 3.4 describe each of the three levels in detail.

3.1 Internal structure of DDBMS

DDBMSs generated by RDM share a common internal architecture. This internal architecture is organized into a three-level hierarchy where each level comprises a set of C functions. The C functions in one level of the hierarchy offer certain

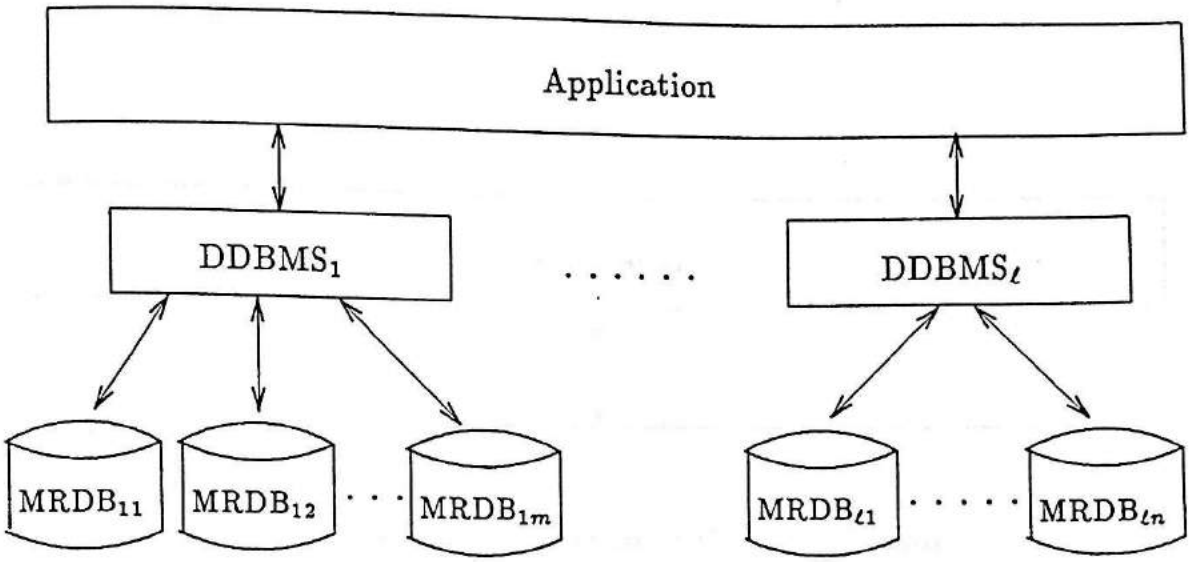


Figure 3.1: Structure of a Software System

services to the higher-levels (the C functions in the top level of the hierarchy offer services to the application), shielding those levels from the details of how the offered services are actually implemented. The internal architecture of a typical DDBMS is depicted in Figure 3.2.

The first or the top level of the hierarchy interacts with the application itself. This level consists of a set of query functions and transaction functions which can be invoked directly by the application. The second level of the hierarchy consists of a set of index functions which supports index operations for each index that is defined in the DDBMS. The third level of the hierarchy consists of a set of access functions and update functions. This set of functions is responsible for physical access and update to the memory-resident database.

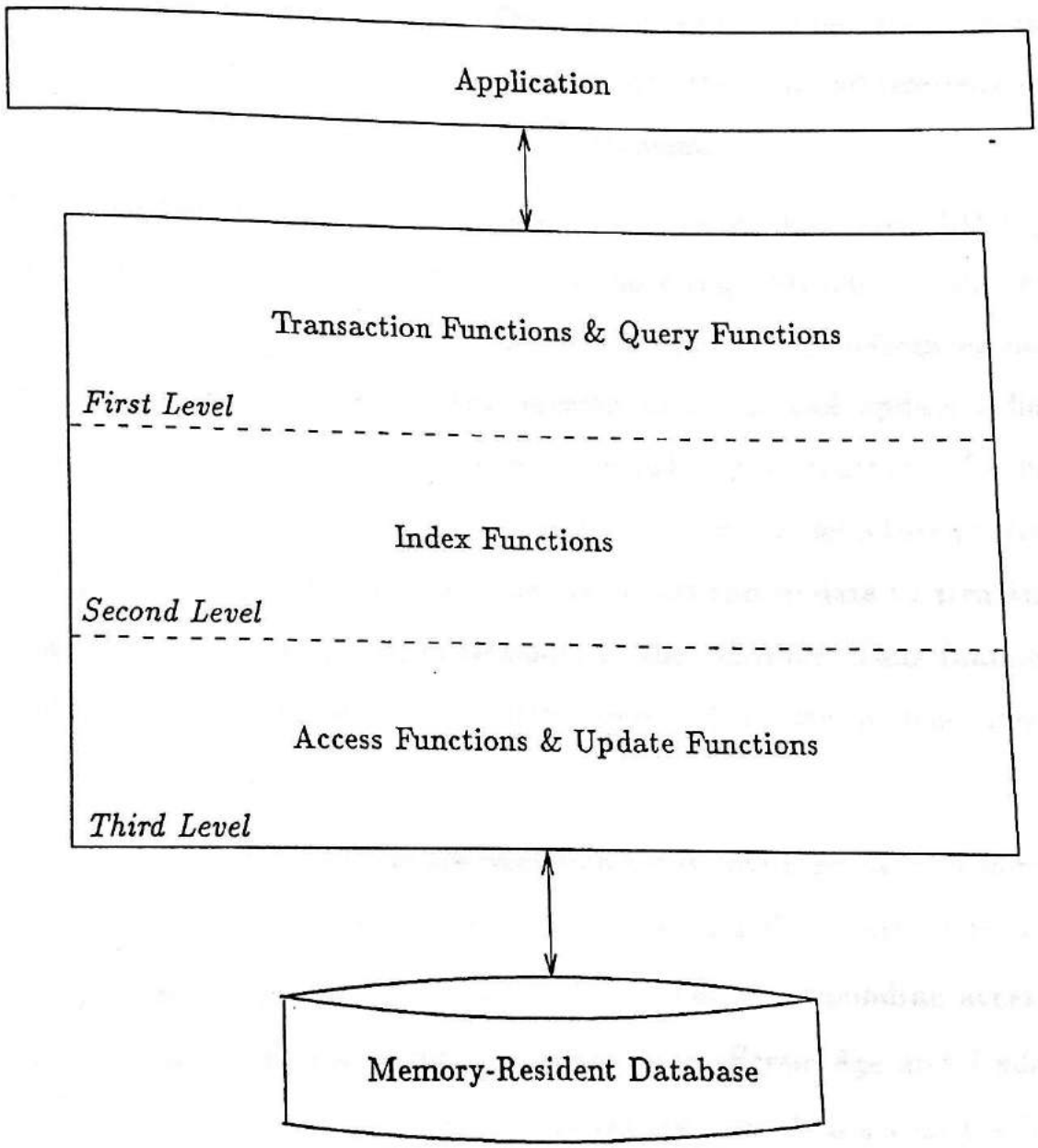


Figure 3.2: DDBMS Internal Architecture

3.2 Access Functions and Update Functions

In a memory-resident database, objects of a defined class are represented by instances of the same C structure type. This structure type adopts the class name as it structure type name and is declared according to the field and reference method information in a corresponding PDM class statement.

For a class that adopts the *direct* reference method, structures that represent objects of this class are referenced by direct addressing. Structures that represent objects of a class which adopts *indirect* reference method are referenced indirectly through pointers called ICells. Consequently, to access and update a field in a structure, one has to know the reference method of the structure. To hide this reference detail, the third level hierarchy of the DDBMS architecture provides a set of access functions and update functions for access and update to structure fields of all the defined structure types (classes) in the DDBMS. Thus functions from higher-levels in the hierarchy can do field access and update to structures without knowing their reference methods.

Access and update functions are named in a way that reflects their function. For example, the PDM source in Figure 2.5 indicates that the structure type that represents the class Person has a field named Age. The corresponding access function and update function for this field are named *AccessPersonAge* and *UpdatePersonAge*, respectively. The function AccessPersonAge is used to access the field Age of structures that represent Person objects; likewise, the function UpdatePersonAge is used to update the field Age of these structures.

To show how these functions are used, suppose that there is a Person object which is referenced by an object-valued variable P (either *directly* or *indirectly*). To use the access function to access the Age of this Person object, the function

AccessPersonAge is called with P as its parameter

AccessPersonAge(P)

and the Age value will be returned by the function. To use the update function to update the Age of the Person object, the function UpdatePersonAge is called with P and the new Age value as its parameters:

UpdatePersonAge(P, Value)

The function will perform the update and nothing will be returned.

3.3 Index Functions

In the top level hierarchy of the DDBMS architecture (which consists of query functions and transaction functions), each database index defined in the DDBMS is viewed as a set of index operations. Index manipulation in query functions and transaction functions is accomplished solely in terms of these index operations. The second level hierarchy of the DDBMS architecture provides a set of index functions to support index operations for all the defined database indices in the DDBMS.

For each database index, there is a logical ordering among objects that are associated with the index. That is, one object can always be distinguished to be either 'smaller' or 'bigger' than another object in the index. This ordering may be imposed either by the data structure that implements the index or according to the *ordered search conditions* which have been specified for the index (or both).

The ordered search conditions of an index is a list of expressions (the order is significant) which determine how objects are ordered in the index. The primary

search condition is given first and determines the major sort order of index entries. The secondary search condition follows.

There are three kinds of ordered search conditions, namely *ascending sort*, *descending sort* and *subclass sort*. Ascending sort and descending sort specify an ascending or descending ordering on a *path function*[1] of the indexed class. A subclass sort condition is used for indexed classes that have at least one immediate subclass, and instructs the objects in an immediate subclass of the indexed class to have a higher priority in the ordering. For example, consider an index on the class Person (which has Name and Age as its properties) specified with three search conditions. Assume that the first is a subclass sort on the class Man (recall that the class Person has two immediate subclass: the class Man and the class Woman), that the second specifies a descending sort on the Age property, and that the third specifies a ascending sort on the Name property. Person objects in the index will then have an ordering in which all Man objects are located in the front. The group of Man objects are in turn stored in descending order of their Age values; and for Man objects that have same Age value are arranged in ascending order according to their Name values. The reader is referred to [24] for further details concerning search conditions on indices.

Based on the ordering among indexed objects, five primitive index operations[2] which apply uniformly to all database indices (called *the primitive index operations*) are defined as follows.

The operation

SetUp()

[1] A path function is a property sequence which navigates among classes.

[2] Every index in a DDBMS will have its own set of C functions which implement the index operations. The names of the C functions signify their corresponding indices and index operations.

is used to initialize an index in preparation for insertion operations. An index must be initialized before it is used.

The operation

Init(ρ) or **Init**()

scans the index and returns the 'smallest' object, according to the ordering, that satisfies the given search predicate ρ. A search predicate is a list of expressions for the ordered search conditions of the index. The first supplies a search value to the first search condition, the second supplies a search value to the second search condition, and so on. The number of search values given in a search predicate can be less than the total number of search conditions, but search values must be given in the order of search conditions starting from the first search condition. For the Person index mentioned above, [Man], [Man, Age = 20], [Man, Age = v] and [Man, Age = 55, Name = 'Fred'] are four valid search predicates. Note that the third search predicate involves an object-valued variable v. If no search condition has been specified for the index, then no search predicate is allowed for the operation **Init** [3]. For a call to the operation **Init** that does not supply a search predicate, the operation will return the 'smallest' object in the index. In any case, if no object in the index satisfies the search predicate or the index is empty, then a special value[4] is returned.

The operation

Next(o, ρ) or **Next**(o)

[3]Distributed indices do not allow their **Init** and **Next** operations to have a search predicate. There are other operations, called **DistInit** and **DistInit**, that must be used instead.

[4]Our implementation adopts the value '0' as this special value.

continues an index scan. Given an object o in the index and a search predicate ρ, the operation Next returns the Next 'smallest' object (after the given object in the index ordering) that satisfies the given search predicate ρ. For a call to the operation Next that does not supply a search predicate, the operation will return the next object (the object right after the given object) in the index. Again, a special value is returned if no such object exists.

The operation

Add(o)

inserts the object o into an index. If ordered search conditions have been specified for the index, the given object will be inserted according to the ordered search conditions. Otherwise, the given index will be inserted in an arbitrary position of the index (according to the data structure that implements the index).

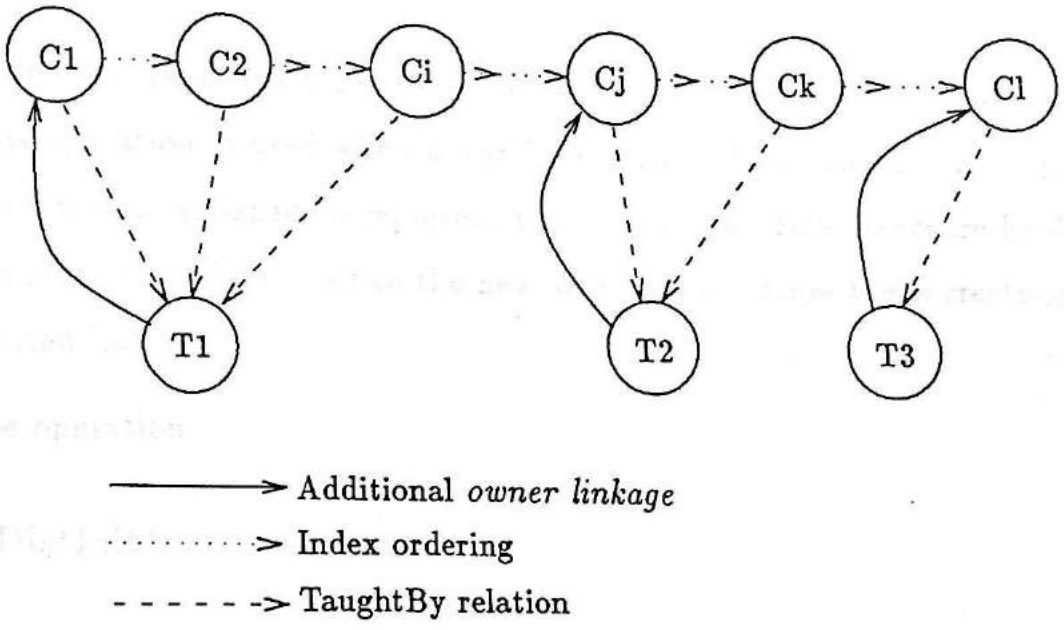
The operation

Sub(o)

removes the object o from an index.

For a subset of database indices called *distributed indices*, five extra index operations (in addition to the five primitive index operations), called *the distributed index operations*, are defined to support their special capability. Indexed objects in a distributed index are distributed among other objects in a class called the *distributed class*. (Distributed indices resemble *owner coupled sets* in DBTG network model [6].) The distributed class of a distributed index contains objects that can be reached from indexed objects through a fixed path function. This path function represents a relation between the indexed object class and the distributed

class. Since a distributed class object can be reached by more than one indexed object through the path function, and an indexed object can only reach a particular distributed class object, the relation represented by the path function is a *many-to-one relationship* between the indexed objects and the distributed class objects. A distributed index groups the indexed objects by the distributed class objects and provides an efficient means for locating indexed objects that are related to a distributed class object. For example, a distributed index on Course objects distributed by Teacher objects through the TaughtBy relation can be visualized as follows (note that how the Course objects are clustered).



For each distributed index, five extra index operations are defined to support its special properties.

Each distributed class object of a distributed index will require a special data structure in order to access *member objects* (as suggested by the solid arcs in the above diagram). Given a distributed class object τ, the index operation

Create(τ)

initializes this data structure.

Given a distributed class object τ of a distributed index, the index operation

Destroy(τ)

clears the corresponding data structure (by releasing all the memory space held by the data structure) originally *created* for the distributed index.

Given two distributed class objects τ_1 and τ_2, the index operation

Copy(τ_1, τ_2)

duplicates the data structure for the distributed index in object τ_1 to the object τ_2. This operation is used when a distributed class object changes its type and a new structure is assigned to represent the object. The data structure held by the old structure must be copied to the new structure to ensure the correctness of the distributed index.

The operation

DistInit(ρ)

initializes a scan on indexed objects related to a particular distributed class object, and returns the 'smallest' object satisfying the search predicate ρ. The distributed class object must be appear as the first search instance of the search predicate ρ. (In an distributed index, to distribute the indexed objects on the distributed class is considered to be the first search condition. This is also the reason why the **Init** operation and **Next** operation do not allow a search predicate for distributed indices.) In any case, if no object in the index satisfies the search predicate or the index is empty, then a special value is returned.

Given an indexed objected (which is assumed to be related to the distributed class object specified in the search predicate) and a search predicate ρ, the operation

DistNext(o, ρ)

returns the next 'smallest' object satisfying the predicate ρ. Again, a special value is returned if no such object exists.

Index operations for each defined index in the DDBMS are implemented as functions (called index functions) in the second level of the DDBMS. Thus an ordinary index will have five corresponding index functions to support its five primitive index operations and a distributed index will have ten corresponding index functions to support its primitive index operations and distributed index operations.

Our naming convention for index functions is straightforward. An index function name is formed by the concatenation of the index operation name and the index name. For example, a distributed index called PersonIndex will have the following index functions:

SetUpPersonIndex
InitPersonIndex
NextPersonIndex
AddPersonIndex
SubPersonIndex
CreatePersonIndex
DestroyPersonIndex
CopyPersonIndex
DistInitPersonIndex
DistNextPersonIndex

3.4 Query Functions and Transaction Functions

The first level of the DDBMS architecture consists of a set of query functions and transaction functions. These functions implement data manipulation requests (queries and transactions) that are defined to the DDBMS. Query functions and transaction functions are named exactly as their corresponding query names and transaction names. When invoked by the application, they perform their specific tasks on the database using functions lower in the hierarchy. We discuss the nature of these functions in detail in chapter 5.

3.5 Summary

The internal architecture of a DDBMS is organized into a three-level hierarchy, where the first level comprises a set of query functions and transaction functions, the second level a set of index functions, and the third level a set of field access and update functions. We have presented an outline of the operational abstraction assumed by RDM indices. This operational abstraction suggests the design of our index type specification language discussed in the next chapter.

Chapter 4

Index Type Extensibility

Indices are a basic part of RDM, which will automatically select indices and then generate code to manage these indices for the dedicated database management system. Index types themselves, however, are defined externally by RDM users. Thus, RDM selects and implements indices from the set of user defined index types, while new index types can be added to RDM by the users. In this way, RDM users can design index types which suit their own needs. This results in greater flexibility in software development with RDM.

In this chapter, we focus on topics that relate to the definition by RDM users of index types. Section 4.1 describes the necessary information required by RDM for each index type. Section 4.2 presents the general structure of an ITS source file (ITS is an acronym for *Index Type Specification*), in which the necessary information for RDM on an index type can be specified. Section 4.3 discusses in detail how code templates for index operations are specified in ITS source files. Section 4.4 presents a full example of an ITS source file, which specifies a simple index type—DISTLIST.

4.1 An Abstraction of Index Types

RDM requires two kinds of information on each index type: *selection information* and *implementation information*. Selection information provides a means for RDM to compare different index types and therefore enables RDM to choose the most suitable type of index for a particular situation. The implementation information provides implementation details for indices of different index types. This information enables RDM to generate index specific code as part of the DDBMS.

In terms of access and update, each RDM index is abstracted as a set of index operations. (Recall that there are five primitive index operations for all RDM indices, and five extra distributed index operations for distributed indices.) Even though the functions of index operations apply uniformly to all RDM indices, their implementation details vary from one index type to another. For instance, the procedure for inserting an object into an index[1] of type *list* is obviously different to the procedure that performs the same function to another index of type *binary tree*. Beside the implementation details of index operations, implementing an index may require additional fields in some related structure types (such as the structure type that represents the index class). These additional fields support the data structure which implements the index, and indices of different index types have different field extension requirements on their related structure types. RDM requires both kinds of implementation information—implementation details on index operations and field extension requirements—for all user defined index types.

RDM indices are distinguished by five selection characteristics. An RDM index can either possess or not possess each of these five characteristics according to its index type. Four of these characteristics do not affect the operational abstraction

[1]This corresponds to the index operation **Add**.

on indices (the only exception is the *distributed* characteristic—distributed indices have five extra distributed index operations on top of the five primitive index operations), but the set of characteristics that an index possesses affects the way it is implemented. Also, RDM considers the characteristics of each index type in the index selection process. So, RDM has to know the characteristics of each user defined index type.

The five characteristics that an index can possess are:

distributed characteristic— A distributed index has an associated distributed class and indexed objects are distributed among objects of the distributed class. The index captures a many-to-one relation between the indexed objects and the distributed class objects; with a distributed class object, the index provides efficient retrieval to all related indexed objects.

static characteristic— A static index has a preset upper bound on the maximum number of indexed objects that the index can handle. This upper bound is fixed in the index implementation and it cannot be changed during execution.

ordered characteristic— An ordered index allows ordered search conditions to be specified explicitly. Index objects are then be ordered according to the ordered search conditions, and the index makes use of this object ordering to benefit indexed object retrieval.

exogenous characteristic— An exogenous index is an index which does not require additional fields in other structure types for its implementation. As a result, exogenous indices can be dynamically added to a DDBMS during execution time, without any need for database reformatting.

explicit scan state characteristic— An index with the explicit scan state characteristic requires the index functions which implement its index operations Init and Next (and also DistInit and DistNext if the index also possesses the distributed characteristic) to maintain and use a scan indicator. This scan indicator is used to store the information of a suspended index scan. When the index scan resumes[2], the index function Next (or DistNext) consults the scan indicator for the necessary information to continue the index scan.

(Actually, the last-reached indexed object in an index scan is always provided as a parameter to the index functions Next and DistNext, and the data structure that implements the index usually enables the index function to continue the index scan based on the information encoded in the structure that represents the last-reached indexed object. But for some index types, their data structures do not required this information to be encoded in the indexed-object structures. Storing this information explicitly in the indexed-object structures may impose a memory space overhead. For example, in order to continue an index scan on an index of type Array[3], the index function Next has to know from the last-reached indexed object its corresponding position in the array. In order to provide this information, it requires each indexed-object structure to have an extra field for storing its array position. But this extra requirement on memory space can be saved by using the scan indicator to store the array position of the last-reached indexed object.)

Other than the characteristics of each index type, another kind of selection information required by RDM is the performance of index operations for each index

[2]An index scan always starts by the index operation Init (or DistInit) and continues successively by the index operation Next (or DistNext).

[3]This index type is analogous to the common array type in most high-level languages.

type. Since index operations for indices of different index types are implemented differently, so the cost (execution time) of each index operation at a particular load (the total number of indexed objects associated with an index) varies for indices of different index types. However, indices of the same index type under the same load are assumed to have the same cost on the same index operation. In the index selection process, if it arises that more than one index type satisfies the an index characteristics of a situation, then the index will be selected from the index type which has the best performance (minimum cost) on the index operations.

For each user defined index type, RDM requires the index type specifier to supply an estimate of the cost of each index operation. For doing so, RDM has adopted the following simple cost formula for describing the cost of index operations.

$$C = aS + b\log_c S + d$$

In this formula, C represents the cost; a, b, c, and d are Real number constants; and S is an estimate of the indexed class size. By choosing the appropriate constants, this formula can express various kinds of cost functions. For example, to express a cost function which is linearly proportional to the indexed class size, we can choose two positive Real numbers as a and c, and set b and d as zero. The index type specifier just has to supply the four Real number constants for each index operation. (This constants should be chosen with respect to other constants that have been chosen for index operations of other index types.)

To summarize, RDM needs the following information for each user defined index type:

1. Implementation details on each index operation.

2. Field extension requirements.

3. Selection characteristics that the index type possesses.

4. Cost estimate of each index operation.

4.2 Structure of an ITS Source File

To extend RDM with a new index type, a user must supply all the required information on the index type in an ITS source file. The general structure file of an ITS source file is shown in Figure 4.1 (where user supplied information is shown in italics).

In the ITS source file, the index type specifier can use a set of predefined generic names to represent symbolic information which depends on the actual index instances of the index type. There are a total of five generic names that can be used in an ITS source file. Their notations and corresponding symbolic information are as follows

'I — The name of the index.

'S — The name of the database schema that the index works on.

'C — The name of the indexed class.

'D — The name of the distributed class.

'M — The integer which represents the maximum number of indexed objects that the index can handle.

Note that all the generic names begin with a backward quote ('), followed by a single character. To illustrate how a generic name is used: the index function name

index type
Type-Name

characteristics
Characteristic$_1$
Characteristic$_2$
:
:

representation
Field extension statments

cost

Init cost	a_1	b_1	c_1	d_1
Next cost	a_2	b_2	c_2	d_2
Add cost	a_3	b_3	c_3	d_3
Sub cost	a_4	b_4	c_4	d_4

:
:

code template
code template for index functions

Figure 4.1: General Structure of an ITS Source File

(which involves the actual index name) for the index function **Init** can be expressed in the ITS source file as

```
Init'I
```

The use of the generic name 'I in the function name indicates that the index function name depends on the actual index name; and the actual function name is obtained by replacing the generic name with the actual index name. So, if a particular index instance of the index type has a name called 'PersonIndex', then the index function name of the index function **Init** for this index will be called 'InitPersonIndex'.

There are five sections in an ITS source file. Each section begins with its section identifier. The five section identifiers are as follows.

- index type
- characteristics
- representation
- cost
- code template

In the first section, the name of the index type is specified. This name is used for identification and reference purpose and must be different from the names of all other index types defined to RDM. This name has no significance to the index selection process.

The second section of an ITS source file lists the characteristics that the index type possesses. To recall from last section, there are five characteristics that an index type can possesses. Their corresponding names that are used in this section are as follows.

- distributed

- static
- ordered
- exogenous
- explicit scan state

In the third section, field extension requirements for implementing indices of the index type are specified. To implement an RDM index, field extensions are permitted on three related structure types:

1. the schema structure type,
2. the indexed class structure type, and
3. the distributed class structure type.

Field extension requirement information for each of these structure types is expressed in a field extension statement of the following format.

extension to *Structure-Type* **properties** $Field\text{-}Name_1$, ..., $Field\text{-}Name_n$
property $Field\text{-}Name_1$ **on** *Class*
:
property $Field\text{-}Name_1$ **on** *Class*

The first clause of the statement specifies the target structure type (one of **schema**, **indexed class**, or **distributed class**) together with a list of field names that are required to be added to the specified structure type. Each of the field names mentioned in the first clause requires a type to be declared in a subsequent property clause. The *Class* in a property clause can be specified to be on the indexed class (indicated by using the generic name 'C), the distributed class (indicated by using the generic name 'D), or one of the four basic classes: **Integer**, **Real**, **DoubleReal**, and **String**. If the *Class* of a field is specified to be on the basic class **String**, the maximum length for the string has to be provided with the keyword **maxlen**.

The *Class* of a field name can also be specified to be on a user defined class by using the keywords **user class** in the property clause

property *Field-Name* **on user class** *User-Class-Name*

In this case, the index type specifier (the person who specifies the index type) has to specify the user class as a structure type in the code template section of the ITS source file; and the name of this structure type must match the *User-Class-Name*.

The fourth section of the ITS source file supplies the cost information for the index operations. (Recall that there are five index operations for a non distributed index type and ten index operations for a distributed index type). For each index operation, the index type specifier has to provide an estimate of its cost by choosing the four constants for the cost formula $C = aS + b\log_c S + d$. These constants are specified in this section with the following format.

operation-name **cost** a b c d

In the final section (fifth section), C code templates of index functions which implement the index operations of the index type are specified. If user defined classes have been specified in the representation section of the ITS source file, their corresponding structure types have to be specified in the code template as well. More detail on code template specifications is given in the next section.

4.3 Details on Code Template Specification

To specify the code template for the index functions of an index type, the index type specifier must follow a set of rules and conventions that has been adopted by RDM. This set of rules and conventions governs

1. the names of the index functions,
2. the types of the index functions,
3. the values that can be returned by the index functions,
4. the formal parameters of the index functions, and
5. the usage of a set of special functions that can be used by the indexed functions.

The rules and conventions are described in the following subsections.

4.3.1 The Names of the Index Functions

There is a one-to-one correspondence between the index operations and the index functions. The name of each index function is formed by the concatenation of the corresponding index operation name and the actual index name. So, for each index type, five index functions must be specified in the code template with the following names.

SetUp'I
Init'I
Next'I
Add'I
Sub'I

For distributed index types, five extra index functions must be specified with the following names.

Create'I
Destroy'I
Copy'I
DistInit'I
DistNext'I

4.3.2 Types and Return Values of Index Functions

The following six index functions must be declared to have the type 'void':

SetUp'I
Add'I
Sub'I
Create'I
Destroy'I
Copy'I

which means that they are functions that do not return any value. The remaining four index functions:

Init'I
Next'I
DistInit'I
DistNext'I

must be declared to have the type 'int'. The values that they can return are either '0' or '1'. If the value '0' is returned by any of these functions, then RDM considers that the index function has failed to find a qualifying object during the index scan (the index operations **Init**, **Next**, **DistInit**, and **DistNext** are used for index scan); otherwise, the value '1' is returned (signifying a qualifying object has been found).

4.3.3 Formal Parameters of Index Functions

The function header of each index function is listed as follows, where formal parameters are represented by *P1*, *P2*, ..., *P6*. The index type specifier can choose his/her own formal parameter names in the code template.

Index function: SetUp'I

```
int SetUp'I(P1)
struct 'SStruct *P1;
```

The index function **SetUp'I** must have one formal parameter. This formal parameter is a pointer to the schema structure type. Note that the name of the schema structure type is formed by the schema name and the string 'Struct'.

Index function: Init'I

```
int Init'I(P1, P2, P3, P4, P5)
struct 'SStruct *P1;
prop 'C *P2;
int (* P3) ();
void *P4;
struct 'IScanState *P5;
```

The index function **Init'I** can have a maximum of five formal parameters (which depends on the characteristics of the index type). The first parameter *P1* is a pointer to the schema structure type. The second parameter *P2* is a pointer to an object-valued variable on the indexed class 'C. (In specifying index functions in the code template, object-valued variables can be declared and used in the same way as in C/DB). This pointer provides a way for the index function to return a qualifying object during an index scan. The third and fourth parameters are required only if: (1) the index type possesses the ordered characteristic, and (2) the index type does not possess the distributed characteristic. The third parameter *P3* is a pointer to a integer function called the *comparison function*, the use of

this function will be discussed later in this section. The fourth parameter *P4* is a pointer of the type 'void'. This pointer is used solely as an actual parameter to the comparison function: it passes the comparison information from the invoker (a query function) of the index function to the comparison function. The fifth parameter *P5* is required only if the index type possesses the explicit scan state characteristic. This parameter is a pointer to a scan state structure type which contains fields for storing necessary information for resuming an index scan. The name of the scan state structure type is formed by the index name and the string 'ScanState'. This structure type has to be specified in the code template by the index type specifier (of course, this is required only if the index type possesses the explicit scan state characteristic).

In specifying an index function, if a parameter is not required due to the characteristics of the index type, the unnecessary parameter is omitted in the parameter list. For example, if an index type possess the explicit scan state characteristic but does not possess the ordered characteristic, then the function header for its index function **Init'I** will be as follows.

```
int Init'I(P1, P2, P3)
struct 'SStruct *P1;
prop 'C *P2;
struct 'IScanState *P3;
```

Index function: Next'I

```
int Next'I(P1, P2, P3, P4, P5)
struct 'SStruct *P1;
prop 'C *P2;
```

```
int (* P3) ();
void *P4;
struct 'IScanState *P5;
```

The formal parameter requirements for the index function **Next'I** are basically the same as for the index function **Init'I**. However, the second parameter *P2* is used for more than returning the qualifying object from the index scan: its input value will be the last-reached indexed object in the index scan.

Index function: DistInit'I

```
int DistInit'I(P1, P2, P3, P4, P5, P6)
struct 'SStruct *P1;
prop 'C *P2;
prop 'D P3;
int (* P4) ();
void *P5;
struct 'IScanState *P6;
```

The index function **DistInit'I** can have a maximum of six formal parameters. As in the case of **Init'I**, the first parameter *P1* is a pointer to the schema structure type and the second parameter *P2* is a pointer to an object-valued variable on the index class 'C. The third parameter for the index function is an object-valued variable on the distributed class. This parameter supplies the distributed object for the index function. The fourth, fifth, and sixth parameters of the index function **DistInit'I** are the same as the third, fourth, and fifth parameters of the index function **Init'I** respectively.

Index function: DistNext'I

```
int DistNext'I(P1, P2, P3, P4, P5, P6)
struct 'SStruct *P1;
prop 'C *P2;
prop 'D P3;
int (* P4) ();
void *P5;
struct 'IScanState *P6;
```

The formal parameter requirements for the index function **DistNext'I** are the same as for the index function **DistInit'I**. Also in this case, the second parameter *P2* is not just used to return the qualifying object from the index scan, it also supplies the last-reached indexed object in the index scan for use by the index function.

Index function: Add'I

```
int Add'I(P1, P2, P3)
struct 'SStruct *P1;
prop 'C P2;
int (* P3) ();
```

The index function **Add'I** can have a maximum of three formal parameters. The first parameter *P1* is a pointer to the schema structure type. The second parameter *P2* is an object-valued variable on the index class 'C. This parameter provides the object for the index function to add into the index. The third parameter *P3* is required only if the index type possesses the ordered characteristic. This parameter is a pointer to a *comparison function*. The nature of this comparison function is

slightly different from the one that has been mentioned before, and it will also be discussed later.

Index function: Sub'I

```
int Sub'I(P1, P2, P3)
struct 'SStruct *P1;
prop 'C P2;
int (* P3) ();
```

The formal parameter requirements for the index function **Sub'I** are the same as the index function **Add'I**. However, in this case the second parameter *P2* provides an indexed object for the index function to remove from the index.

Index function: Copy'I

```
int Copy'I(P1, P2, P3)
struct 'SStruct *P1;
prop 'D P2;
prop 'D P3;
```

The index function **Copy'I** must have three formal parameters. The first parameter *P1* is a pointer to the schema structure type. The second and third parameters *P2* and *P3* are object-valued variables on the distributed class 'D. The index function **Copy'I** duplicates the index data structure in the distributed object provided by *P2* to the distributed object provided by *P3*.

Index function: Create'I

```
int Create'I(P1, P2)
struct 'SStruct *P1;
prop 'D P2;
```

The index function **Create'I** must have two formal parameters. The first parameter *P1* is a pointer to the schema structure type. The second parameter *P2* is an object-valued variable on the distributed class 'D. The index function **Create'I** initializes the data structure which is required by the distributed index in the distributed object provided by *P2*.

Index function: Destroy'I

```
int Destroy'I(P1, P2)
struct 'SStruct *P1;
prop 'D P2;
```

The index function **Destroy'I** must have two formal parameters. The first parameter *P1* is a pointer to the schema structure type. The second parameter *P2* is an object-valued variable on the distributed class 'D. The index function **Destroy'I** clears the data structure which has been set up by the distributed index in the distributed object provided by *P2*.

4.3.4 The Special Functions

There are three types of special functions which can be used by the index functions:

1. assign and update functions,

2. distributed path function, and

3. comparison functions.

The assign and update functions are used to access and update structure fields of the structure types that represent the indexed class and distributed class. These functions have been described in Chapter 3.

The index functions for each distributed index type are allowed to use a distributed path function. This function is used to obtain the corresponding distributed class object from an indexed object. The name of the function is formed by the string 'DistPath' and the distributed index name, and it is used in the format

DistPath'I(*exp*)

where the *exp* is an expression which can be evaluated to give an indexed object. The function will return the corresponding distributed class object of the indexed object.

For an ordered index type, a comparison function is provided as a parameter to some of its indexed functions. The use of comparison functions varies for different index functions. The comparison function that is provided to the index functions **Init'I, Next'I, DistInit'I,** and **DistNext'I** helps to decide the location (based on the ordering of the indexed objects) of the qualifying object(s) in an index scan. It takes an indexed object and a pointer (which is also provided as a parameter of the index function) as arguments, and returns one of three possible integer values to indicate one of the following three possible situations:

- the value '-1' if the indexed object argument is 'bigger' (based on the index ordering) than the qualifying object(s),
- the value '0' if the indexed object argument is one of the qualifying object(s), and
- the value '1' if the indexed object argument is 'smaller' (based on the index ordering) than the qualifying object(s).

The comparison function that is provided to the index functions **Add'I** and **Sub'I** compares two indexed object arguments based on the index ordering; and returns one of the three possible integer values to indicate one of the following three possible situations:

- the value '-1' if the first indexed object argument is 'smaller' than the second indexed object argument,
- the value '0' if the first indexed object argument is 'equal' to the second indexed object argument, and
- the value '1' if the first indexed object argument is 'bigger' than the second indexed object argument.

In summary, index functions can use other functions that are designed by the index type specifier. But these functions must be included in the code template. There is no restriction on the names of these functions, as long as they do not conflict with other index function names in the code template.

4.4 An Example of ITS Source File

An example of ITS source file is given in Figures 4.2 to 4.5. This ITS source file specifies a simple distributed index type called DISTLIST (which stands for distributed list).

In an index of this index type, indexed objects are organized in a double linked list. Two extra fields encoding the previous and next pointers are added to the indexed class structure type. This is specified in the field extension statement in lines 12 to 14. Indexed objects related to the same distributed class object are clustered together in the double linked list. Each distributed class object has a pointer to the first related object in the double linked list. Thus an extra field for this 'first' pointer is required in the distributed class structure type. This is specified in the field extension statement in lines 16 and 17. Also, an extra field for the head pointer of the double linked list is required in the schema structure type. This is specified in the field extension statement in lines 9 and 10.

Since the index type possesses the distributed characteristic, there are a total of ten index functions specified in the code template section in lines 34 to 147. This index type required no physical operation to be done for the index operation **Destroy**, and therefore the index function **Destroy'I** specified in lines 55 to 59 does not have any executable statement. Note that all index function are declared to be 'inline'. This is a compiler directive for a C compile called 'GNU CC'. This directive directs the GNU CC to integrate the function's code into the code of its callers. This makes execution faster by eliminating the function-call overhead. Further information on the GNU CC can be found in [16, 17]. This directive is suppressed by RDM (due to the fact that it is not a standard C feature) unless the user explicitly requests it. The index type specifier has to indicate those functions

```
1.  index type
2.      DISTLIST
3.
4.  characteristics
5.      distributed
6.
7.
8.  representation
9.      extension to schema properties 'IHead
10.     property 'IHead on 'C
11.
12.     extension to indexed class properties 'IPrev, 'INext
13.     property 'IPrev on 'C
14.     property 'INext on 'C
15.
16.     extension to distributed class properties 'IFirst
17.     property 'IFirst on 'C
18.
19.
20. cost
21.     Init cost        0 0 2 2
22.     Next cost        0 0 2 2
23.     Add cost         1 0 2 3
24.     Sub cost         0 0 2 3
25.     DistInit cost    0 0 2 3
26.     DistNext cost    0 0 2 3
27.     Create cost      0 0 2 1
28.     Destroy cost     0 0 2 0
29.     Copy cost        0 0 2 1
30.     SetUp cost       0 0 2 1
31.
32. code template
33.
34. inline void SetUp'I(DBName)
35. struct 'SStruct *DBName;
36. {
37.     DBName- >'IHead = NULL;
38. }
```

Figure 4.2: An ITS Source File (a)

```
39.
40.  inline void Copy'I(DBName, A, B)
41.  struct 'SStruct *DBName;
42.  prop 'D A;
43.  prop 'D B;
44.  {
45.      Update'D'IFirst(B, Access'D'IFirst(A));
46.  }
47.
48.  inline void Create'I(DBName, A)
49.  struct 'SStruct *DBName;
50.  prop 'D A;
51.  {
52.      Update'D'IFirst(A, NULL);
53.  }
54.
55.  inline void Destroy'I(DBName, A)
56.  struct 'SStruct *DBName;
57.  prop 'D A;
58.  {
59.  }
60.
61.  inline void Add'I(DBName, A)
62.  struct 'SStruct *DBName;
63.  prop 'C A;
64.  {
65.      if (DBName- >'IHead == NULL) {
66.          Update'C'IPrev(A, NULL);
67.          Update'C'INext(A, NULL);
68.          Update'D'IFirst(DistPath'I(A), A);
69.          DBName- >'IHead = A;
70.      } else if (Access'D'IFirst(DistPath'I(A)) == NULL) {
71.          Update'C'INext(A, DBName- >'IHead);
72.          Update'C'IPrev(A, NULL);
73.          Update'D'IFirst(DistPath'I(A), A);
74.          Update'C'IPrev(Access'C'INext(A), A);
75.          DBName- >'IHead = A;
76.      } else {
77.          Update'C'INext(A, Access'D'IFirst(DistPath'I(A)));
```

Figure 4.3: An ITS Source File (b)

```
78.         Update'C'IPrev(A, Access'C'IPrev(Access'D'IFirst(DistPath'I(A))));
79.         Update'C'IPrev(Access'C'INext(A), A);
80.         Update'D'IFirst(DistPath'I(A), A);
81.         if (Access'C'IPrev(A) != NULL)
82.             Update'C'INext(Access'C'IPrev(A), A);
83.         else
84.             DBName- >'IHead = A;
85.     }
86. }
87.
88. inline void Sub'I(DBName, A)
89. struct 'SStruct *DBName;
90. prop 'C A;
91. {
92.     if (Access'D'IFirst(DistPath'I(A)) == A)
93.         if (Access'C'INext(A) != NULL &&
94.             DistPath'I(Access'C'INext(A)) == DistPath'I(A))
95.             Update'D'IFirst(DistPath'I(A), Access'C'INext(A));
96.         else
97.             Update'D'IFirst(DistPath'I(A), NULL);
98.     if (Access'C'IPrev(A) == NULL)
99.         DBName- >'IHead = Access'C'INext(A);
100.    else
101.        Update'C'INext(Access'C'IPrev(A), Access'C'INext(A));
102.    if (Access'C'INext(A) != NULL)
103.        Update'C'IPrev(Access'C'INext(A), Access'C'IPrev(A));
104. }
105.
106. inline int DistInit'I(DBName, A, D)
107. struct 'SStruct *DBName;
108. prop 'C *A;
109. prop 'D D;
110. {
111.     if (Access'D'IFirst(D) == NULL)
112.         return(0);
113.     else
114.         (*A) = Access'D'IFirst(D);
115.     return(1);
116. }
```

Figure 4.4: An ITS Source File (c)

```
117.
118.  inline int DistNext'I(DBName, A, D)
119.  struct 'SStruct *DBName;
120.  prop 'C *A;
121.  prop 'D D;
122.  {
123.      (*A) = Access'C'INext(*A);
124.      if ((*A) != NULL && DistPath'I(*A) == D)
125.          return(1);
126.      return(0);
127.  }
128.
129.  inline int Init'I(DBName, A)
130.  struct 'SStruct *DBName;
131.  prop 'C *A;
132.  {
133.      if (DBName- >'IHead == NULL)
134.          return(0);
135.      (*A) = DBName- >'IHead;
136.      return(1);
137.  }
138.
139.  inline int Next'I(DBName, A)
140.  struct 'SStruct *DBName;
141.  prop 'C *A;
142.  {
143.      (*A) = Access'C'INext(*A);
144.      if ((*A) == NULL)
145.          return(0);
146.      return(1);
147.  }
```

Figure 4.5: An ITS Source File (d)

in the code template that can take advantage of this directive by actually declaring those functions to be 'inline'.

Another example of ITS source file which specifies a much more complicated index type—binary tree—is given in Appendix D.

4.5 Summary

In this chapter, we have outlined RDM's view of an index type. Index type information can be classified into two categories: (1) selection information, which is used for index selection, and (2) implementation information, which provides implementation details for indices based on the index type. We have also described the structure of the ITS source file, in which the necessary information of an index type can be specified. Finally, a sample ITS source file which specifies a simple index type called DISTLIST has been presented.

Chapter 5

Code Generation

There are two kinds of output source code generated by RDM: source comprising the DDBMS, and application source generated from C/DB programs. The two sections of this chapter discuss each respectively. In the former case, we outline the general structure of the DDBMS, and also present detailed algorithms for access plan code generation. In the latter case, we give the translation procedures for the extended C constructs in C/DB source code.

5.1 Code Generation for DDBMS

5.1.1 Overall Structure

A DDBMS source file implements a dedicated database management system. Each DDBMS source file is generated from a PDM source file, which essentially specifies a detailed design of the DDBMS. A typical DDBMS source file has eight logical sections as illustrated in Figure 5.1. Each logical section consists of C declarations or C functions (or both).

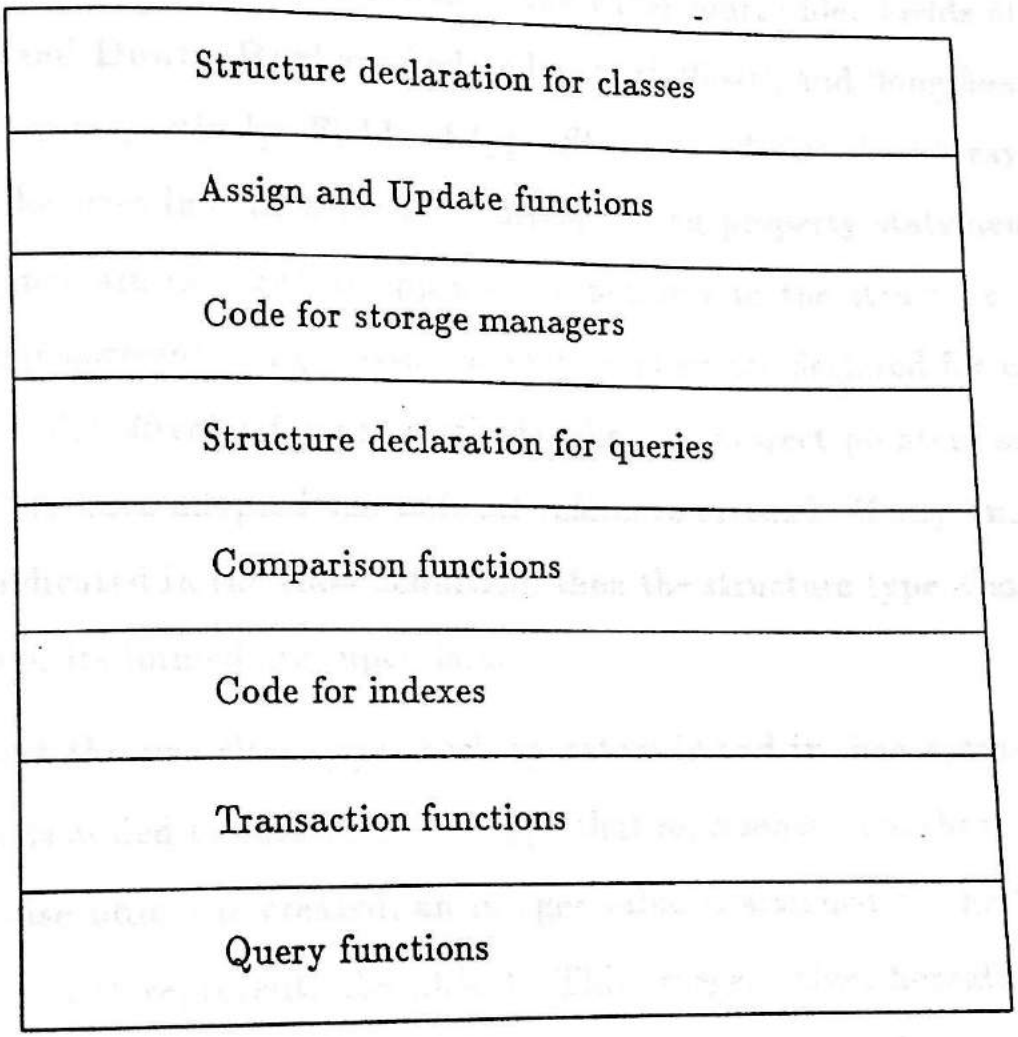


Figure 5.1: Structure of a DDBMS Source

The first section consists of structure type declarations representing the defined classes in the database schema. A structure type is generated for each class definition in the PDM source file. The generated structure type adopts the class name as its structure type name, and has the fields indicated in the class definition. The type and reference method of each field is acquired from the corresponding property statement and class definition in the PDM source file. Fields of type **Integer**, **Real** and **DoubleReal** are declared as 'int', 'float', and 'long float' in their structure types respectively. Fields of type *String* are declared as arrays of characters with the sizes indicated in their corresponding property statements. Fields with *class* types are declared as appropriate pointers to the structure types that represent their corresponding classes. Direct pointers are declared for classes that have adopted the *direct* reference method; whereas indirect pointers are declared for classes that have adopted the *indirect* reference method. If any immediate superclass is indicated in the class definition, then the structure type also inherits all the fields from its immediate superclass.

To support the two class-type check operators **is** and **in**, one extra integer field named 'Msc' is added to each structure type that represents a database class. When a new database object is created, an integer value is assigned to the 'Msc' field of the structure that represents the object. This integer value, hereafter referred to the *Msc value*, identifies the type of the object and also enables class-type check. The Msc value for each class in a database schema is assigned by the following algorithm which is a simplified version of the method described in [1].

> The set of classes is first maximally partitioned satisfying the condition that any given class and superclass are in the same partition. Then for each partition with n classes, $C_1,\ldots,C_n$, the integer value 2^{i-1} will be assigned as the Msc value for the class C_i, where i is in the range

of 1 to n. Note that the number of words needed for encoding the Msc value will depend on the maximum partition size.

The predicate

o **is** C

with the class-type check operator is is implemented in the following way. The Msc value held in the Msc field of the structure that represents the object o is compared to the Msc value of the class C. The predicate is true if and only if the two Msc values are equal.

Since each Msc value is a power of 2, it follows that there is only one '1' (others are all '0's) in the binary representation of each Msc value. Therefore, to implement the predicate

o **in** C

with the class-type check operator **in**, Msc values of all the subclasses of C are added together, then a bitwise AND operation is performed on the sum and the Msc value of the object o. The predicate is true if and only if the result of the bitwise AND operation is non-zero.

The second section consists of access and update functions. For each field in each structure type that has been generated in the first section of a DDBMS source, one access and update function pair is generated for the access and update of this structure field. Access and update functions have been fully discussed in Chapter 3.

The third section contains miscellaneous declarations for the implementation of storage managers. At present the declarations corresponds to the following code.

```
int *PDMICellStore = NULL;

struct StoreTemplate {
    struct StoreTemplate *Next;
};
```

An integer pointer named 'PDMICellStore' is declared and initialized to NULL. This pointer is the head pointer for a storage manager called *ICellStore* which manages unused ICell pointers in a linked list. When an ICell pointer is *freed* by the dedicated database management system, it will be cast as an integer pointer and stored in the ICell store for future use. A structure type called 'StoreTemplate' (this structure type represents the special class 'StoreTemplate' that is used in the PDM source) is also declared. It is the only structure type that can be handled by all storage managers other than the *ICellStore*. All object structures that are *freed* by the system are cast to this structure type and stored in the appropriate storage manager. Note that the only field in this structure type, 'Next', enables unused object structures to be stored as a linked list in their storage managers.

For each storage manager declared in the PDM source file, a symbolic name is defined in this section to represent the memory size of structures that the storage manager can handle. The symbolic name is formed by the corresponding storage manager name and the string 'Size'. For example, if there is a storage manager called 'CourseStore' which stores unused Course object structures, then the following line will be generated in this section:

```
#define CourseStoreSize( sizeof( struct Course ) )
```

The use of the symbolic names is an effort to make the generated code more readable.[1] They are used in the later sections of the DDBMS source.

The fourth section contains a set of structure types called *query structure types*. Each of them corresponds to a compiled query in the PDM source file. As each compiled query is implemented by a C function in the DDBMS, the corresponding query structure type defines the type of structures which serve to communicate information between the query function and its caller. When a query function is called, information required by the function is loaded into a query structure and a pointer to the query structure is then passed as a parameter to the query function. Query *solutions* are returned to the application in the query structure.

The name of a query structure type is formed by the concatenation of its corresponding query name and the string 'Struct'. The first two fields of each query structure type are always the same, and as follows.

```
int First;
int Result;
```

'First' indicates the type of the query call. Recall in Chapter 2 that expression trees that express query execution procedures can be invoked by *init call* and *next call* protocols. Query functions implement the protocol as follows: if 'First' has the value '1' then an init call to the query function is assumed, and the function will return the *first* solution of the query (if there is any); otherwise, a next call to the query function is assumed, and the function will return the *next* solution of the query (if there is any). In both cases if the query function successfully finds a solution to the query, 'Result' is assigned '1', otherwise 'Result' is assigned '0'.

[1] We believe some applications will require RDM users to read and understand the generated DDBMS source code.

Each of the *cut variables* that is defined in the expression tree of a query is also implemented as a field in its query structure type. The field name of the field that represents the cut variable is the same as the cut variable name that is used in the expression tree. Moreover, a query structure type also contains fields that represent object-valued variables occurring in the *given clause*, *select clause* and *declare clause* of the corresponding compiled query. (Recall from Chapter 2 that variables stated in the given clause are used to supply input to the query; variables stated in the select clause are used to store one solution of the query; and variables stated in the declare clause are temporary variables that are used to store intermediate results.) As a result, a query structure contains the necessary storage for a query function to operate; query structures also provide a means for query functions to save their operational status, thus allowing a query function to resume an index scan when a 'next' call arrives.

The fifth section contains comparison functions which support the index functions of the ordered indices. There are two types of comparison functions. A comparison function of the first type decides the order of two indexed objects based on the ordered search conditions of an *ordered index*, and a comparison function of the second type helps to decide the location (based on the indexed object ordering) of the qualifying objects in an index scan.

For each *ordered* index that has been defined in the PDM source, a comparison function of the first type is generated. The name of the comparison function is formed by the concatenation of the index name and the string 'Compare'. The function takes two parameters, named 'P1' and 'P2', both declared as appropriate pointers (with respect to the reference method) to the structure type which represents the indexed class. The function body code (which decides the order of two indexed objects) is generated according to the ordered search conditions specified

in the index declaration. There are four cases listed as follows.

(1) For a search condition which expresses a subclass sort on an immediate subclass C′ (with Msc value i) of an indexed class C, the following code will be generated:

```
if (((AccessCMsc(P1) & i) == 0) || ((AccessCMsc(P2) & i) == 0)){
    if ((AccessCMsc(P1) & i) != 0)
        return(-1);
    if ((AccessCMsc(P2) & i) != 0)
        return(1);
    if (P1 < P2)
        return(-1);
    if (P1 > P2)
        return(1);
    return(0);
} else {
    <Code for the next search condition>
}
```

The first if statement checks if at least one of the given objects is not an instance of class C′. In this case, the second and the third if statements determine if this is true for only one of the two arguments. If either one of the two tests performed by these two if statements is true, then only one of the two given objects is in the class C′ and the other one is not. So the order between these two objects can be decided and the appropriate value (which has been defined in Chapter 4) is returned. If both of the two tests are false, it means that both of the given objects are not in the class C′, then the order between the two objects is decided by the pointer values held in the pointers P1 and P2. (This is done to ensure a total ordering among all indexed objects.) If the pointer values held in both of the pointers are the same, in which case the two given objects must be the same object and the

value '0' is returned. If the test performed by the first if statement is false, it means that both of the given objects are in class C', and the code for the next search condition is executed to conduct a more extended check.

(2) For a search condition which expresses an ascending sort on the property F of the indexed class, the following code will be generated:

```
if (AccessCF(P1) < AccessCF(P2))
    return(-1);
if (AccessCF(P1) > AccessCF(P2))
    return(1);
<Code for the next search condition>
```

The code that is generated for an ascending sort varies depending upon the situation. For example, the string comparison function 'strcmp' will be used to compare property values which are of string type. Alternatively, for an ascending sort on a property of a subclass of the indexed class, code with a more complicated use of access functions will be generated. However the logical structure of the generated code is the same.

(3) For a search condition which express a descending sort on the property F of the indexed class, the following code will be generated:

```
if (AccessCF(P1) < AccessCF(P2))
    return(1);
if (AccessCF(P1) > AccessCF(P2))
    return(-1);
<Code for the next search condition>
```

The code generated for a descending sort is similar to the code that is generated for an ascending sort. The difference is that the opposite values are returned (i.e. '1' is replaced by '-1', and '-1' is replaced by '1').

(4) When the code for each of the ordered search conditions has been generated, the following piece of code will replace the '<Code for the next search condition>' that has been generated for the last search condition:

```
if P1 < P2
    return(-1);
if P1 > P2
    return(1);
return(0);
```

This code is generated to ensure a total ordering among all the indexed objects in an ordered index. When the order of two indexed objects cannot be determined from the specified ordered search conditions, the pointer values held in the two pointers 'P1' and 'P2' are used to decide the order of the two indexed objects.

Index scans and index lookups specified in APEs are supported by four index operations: Init, Next, DistInit, and DistNext. Each of these index operations can take a *search predicate* (which has been defined in Chapter 3), and are implemented as index functions in the DDBMS source. For each index scan or index lookup on an ordered index, a comparison function of the second type is generated to help these index functions in locating the qualifying objects in the ordered index. The name of the comparison function is formed by the string 'PDMCCompare' and an integer which serves to identify the comparison function. The function takes two parameters named 'P' and 'PDMCQStruct'. The parameter 'P' is declared as a pointer to the structure type which represents the indexed class. It represents the indexed object that is supplied to the comparison function. The parameter 'PDMCQStruct' is declared as a pointer to the query structure type of the query for which the index scan or index lookup applies. When the comparison function is

invoked, this parameter will receive a query structure which has fields that represent variables that may be referenced by the search predicate. The function body of the comparison function is generated according to each search instance in the search predicate, based on the following procedure.

(1) For a search instance that corresponds to a subclass sort on on an immediate subclass (whose Msc value is i) of the indexed class C, the following code will be generated:

```
if ((AccessCMsc(P1) & i) == 0)
    return(-1);
<Code for the next search instance>
```

The if statement checks if the given index object does not belong to the class C. If that is the case, the qualifying object(s) should be 'smaller' than the given indexed object in the index ordering, and the value '-1' is returned; otherwise, the code for the next search instance is executed to conduct a more extended check.

(2) For a search instance that corresponds to an ascending sort on the property F of the indexed class, the following code will be generated:

```
if (AccessCF(P1) < value)
    return(-1);
if (AccessCF(P1) > value)
    return(1);
<Code for the next search instance>
```

In the code, the *value* is a property value which is specified by the search instance. The value can be specified as a constant, or it can be specified as the value held in a field of the given query structure. The code that is

generated for a search instance of an ascending sort varies, but the logical structure of the generated code is the same.

(3) Similarly, for a search instance that corresponds to a descending sort on the property F of the indexed class, the following code will be generated:

```
if (AccessCF(P1) < value)
    return(1);
if (AccessCF(P1) > value)
    return(-1);
<Code for the next search instance>
```

(4) When the code for each of the search instances has been generated, the following piece of code will replace the '<Code for the next search instance>' that has been generated for the last search instance:

```
return(0);
```

If the given indexed object satisfies all the search instances, then the value '0' is returned indicating that a qualifying object has been found.

The sixth section of a DDBMS source file contains code for the implementation of indices. This includes the index functions and supplementary declarations required for the indices. For each index that is defined in the PDM source, a copy of the code template defined in the ITS source file of its corresponding index type is placed in this section, with all the generic names in the code template replaced by the actual names. Object-valued variable declarations in the code template are also changed to corresponding C declarations.

The seventh and eighth sections of a DDBMS source contain transaction functions and query functions. Code generation for transaction functions and query functions is more involved, and is discussed in the following two sections.

5.1.2 Code Generation for Transaction Functions

For each compiled transaction in a PDM source file, a transaction function is generated in the DDBMS source file according to the information stated in the compiled transaction. The generated transaction function uses the transaction name as its function name. The first parameter of the function is named 'PDMCSchema' and is declared as a pointer to the schema structure type. Other parameters of the function are defined according to the *given clause* in the compiled transaction. The function body is generated by translating each of the transaction operation statements sequentially to their corresponding C code. If the *declare clause* is present in the compiled transaction, appropriate declaration statements are generated at the beginning of the function body.

There are fifteen types of transaction operation statements. The translation procedure for each of them is specified in Tables 5.1 to 5.4. Most types of transaction operation statements are translated to function calls to their corresponding index functions. We present some remarks on the remaining statements.

- The generated code for **Remove** and **Insert** statements first checks if the corresponding index is locked by an index scan before the function calls to the corresponding index functions. (For each index in a DDBMS, RDM maintains a *lock* field in the schema structure. The name of this additional field is formed by the concatenation of the index name and the string 'Lock'. The purpose of the lock field is to prevent index update to the index while it is being scanned. When an index scan starts, the corresponding *lock* field will be set to '1', and it will reset to '0' when the scan terminates.)

- The generated code for **allocate** statement checks if the store managed by S is empty. If that is the case, memory will be allocated for the object structure

Operation Name Corresponding Code	Statement Syntax
(1) **Assignment**	$v.p := property\text{-}value$
Update$Cp(v, property\text{-}value)$;	-
(2) **Remove**	remove v from I

if (PDMCSchema−>ILock) {
printf("Update Violation.");
exit(0);
}
Sub$I$(PDMCSchema, v);	

(3) **Insert**	insert v in I

if (PDMCSchema−>ILock) {
printf("Update Violation.");
exit(0);
}
Add$I$(PDMCSchema, v);	

(4) **Create**	create v for I
Create$I$(PDMCSchema, v);	

Table 5.1: Translation Procedure for Transaction Operation Statements (a)

Operation Name Corresponding Code	Statement Syntax
(5) **Destroy** Destroy$I$(PDMCSchema, v);	destroy v for I
(6) **Copy** Copy$I$(PDMCSchema, v_1, v_2);	copy v_1 to v_2 for I

(7) **Allocate**
if (PDMCSchema−>S == NULL)
v = (struct C *)malloc(SSize);
else {
v = (struct C *)(PDMCSchema−>S);
PDMCSchema−>S = PDMCSchema−>S−>Next;
}
Assign$C$Msc(v, i);
	allocate v from S

(8) **Free** ((struct StoreTemplate *)v)−>Next = PDMCSchema−>S; PDMCSchema−>S = (struct StoreTemplate *)v;	free v to S

Table 5.2: Translation Procedure for Transaction Operation Statements (b)

Operation Name
Corresponding Code — Statement Syntax

(9) **Allocate Indirect** — allocate indirect v from S

```
if (PDMCSchema->S == NULL)
    (*v) = (struct C *)malloc(SSize);
else {
    (*v) = (struct C *)(PDMCSchema->S);
    PDMCSchema->S = PDMCSchema->S->Next;
}
AssignCMsc(v , i);
```

(10) **Free Indirect** — free indirect v to S

```
((struct StoreTemplate *)(*v))->Next = PDMCSchema->S;
PDMCSchema->S = (struct StoreTemplate *)(*v);
```

(11) **ICell Assignment** — v_1 id := v_2

```
(*v1) = (struct C *)(*v2);
```

(12) **ICell Allocate** — alloc id v

```
if (PDMCellStore == NULL)
    v = (struct C **)malloc( sizeof(int *) );
else {
    v = (struct C **)PDMCellStore;
    PDMCellStore = (int *) * PDMCellStore;
}
```

Table 5.3: Translation Procedure for Transaction Operation Statements (c)

Operation Name Corresponding Code	Statement Syntax
(13) **ICell Free**	free id v
$*v$ = (struct C $*$)PDMCellStore; PDMCellStore = (int $*$)v;	
(14) **Init Index**	init index I
Setup$I$(PDMCSchema); PDMCSchema$->I$Lock = 0;	
(15) **Init Index**	init store S
PDMCSchema$->S$ = NULL;	

Note: – v, v_1, and v_2 are object-valued variables
– p is a property name
– I is an index name
– S is a storage manager name

Table 5.4: Translation Procedure for Transaction Operation Statements (d)

by the function 'malloc'; otherwise, a free object structure will be allocated from the linked list where the storage manager stores the free structures. The 'Msc' field of the new object structure will then be initialized to indicate its type.

- The generated code for **free** statement puts the unused object structure into the store managed by S. The unused structure is stored in the beginning of the linked list where the storage manager stores free structures.
- The generated code for **allocate indirect** statement is similar to the one that is generated for the allocate statement, but allocate indirect statement is used for object classes that adopt *indirect* reference method. The address of the allocated structure in this case is assigned to $*v$ instead of v.
- The generated code for **free indirect** statement is similar to the one that is generated for the free statement, but free indirect statement is used for object classes that adopt *indirect* reference method. So $*v$ is used instead of v in the generated code.
- The generated code for **ICell allocate** statement allocates one ICell from the *ICellStore* to the variable v. If the *ICellStore* is empty, then memory of the ICell will be allocated by the function 'malloc'.
- The generated code for **ICell free** statement stores one unused ICell into the *ICellStore*.

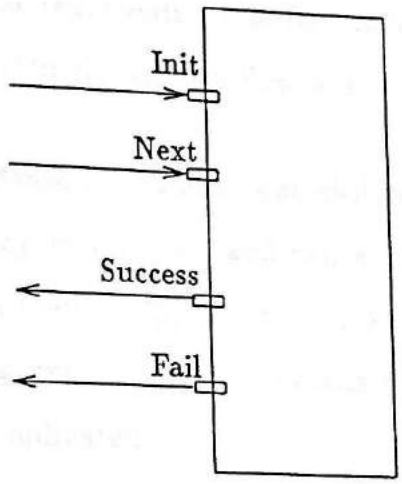
5.1.3 Code Generation for Query Functions

A C function is also generated for each compiled query in a PDM source file. A query function adopts its corresponding query name as its function name and has

two parameters named 'PDMCSchema' and 'PDMCQStruct'. The first parameter 'PDMCDchema' is declared as a pointer to the schema structure type. The second parameter 'PDMCQStruct' is declared as a pointer to the query structure type that corresponds to the query. However, code generation for a query function body is not as straightforward as for a transaction function body.

The list of transaction operation statements that define the execution procedure of a transaction expresses a simple sequential control flow. Thus, code generation for a transaction function body is easily accomplished by generating the corresponding code for each transaction operation statement successfully in this sequential order. However, the execution procedure of a query is expressed by an APE which is operationally interpreted by an expression tree. Each expression tree node represents an APE operation and can be invoked from its parent by two kinds of call (*init call* and *next call*), and returns two kinds of result (*success* and *fail*) to its parent. Its parent then proceeds according to the returned result. This operational abstraction on expression tree nodes requires a more complex control-flow structure: control transfers among generated code for each operation in the expression.

Control flow for all nodes in an expression tree can therefore be viewed as general *control-flow box* or *circuit* with two input and two output as follows.



Since tree nodes representing the same APE operation have the same internal control-flow structure, a standard control-flow box with *internal circuit* suffices for each. The control-flow box for each of the APE operations is given in Appendix C (Note that sort and projection are not currently supported by RDM, and they are considered beyond the scope of this chapter.)

A few remarks should clarify our diagrammatic conventions for control-flow boxes.

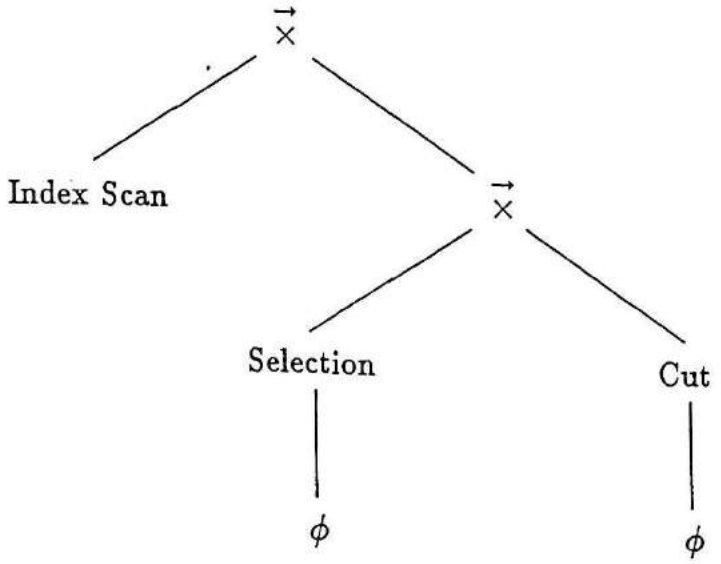
1. Operations that a tree node performs are represented by circles inside a control-flow box.

2. All operations have one control-flow entry point, and one or two control-flow exit points. For operations that have two control-flow exit points, the actual control flow that is used depends on the result of the operations. The one labeled 'Y' corresponds to a successful operation; whereas the one labeled 'N' corresponds to a unsuccessful operation.

3. Control flow among operations is represented by lines with arrows (which show the directions) inside a control-flow box.

4. When a control-flow box represents a non-leaf node, each subtree of the node occurs as a *sub-box* within the control-flow box.

An example should illustrate how these control-flow boxes can be used to derive the control flow in an expression tree, and hence simplify the code generation process for a query function body. Suppose there is a compiled query whose APE corresponds to the following expression tree (details concerning the **index scan**, **selection**, and **cut** are not indicated).



Assembling the control-flow boxes for this expression tree will result in the flowchart depicted in Figure 5.2. Although initially complicated, simple post-processing (logically removing all the boxes) results in the much simplified control-flow diagram depicted in Figure 5.3. The query function body is then generated according to this control-flow diagram in the following manner. The corresponding code for each operation in the control-flow diagram is generated. This code is preceded by a **label** statement. Each control-flow exit point of the operation is then implemented by a **goto** statement, which directs the control to a label that precedes the code of another operation.

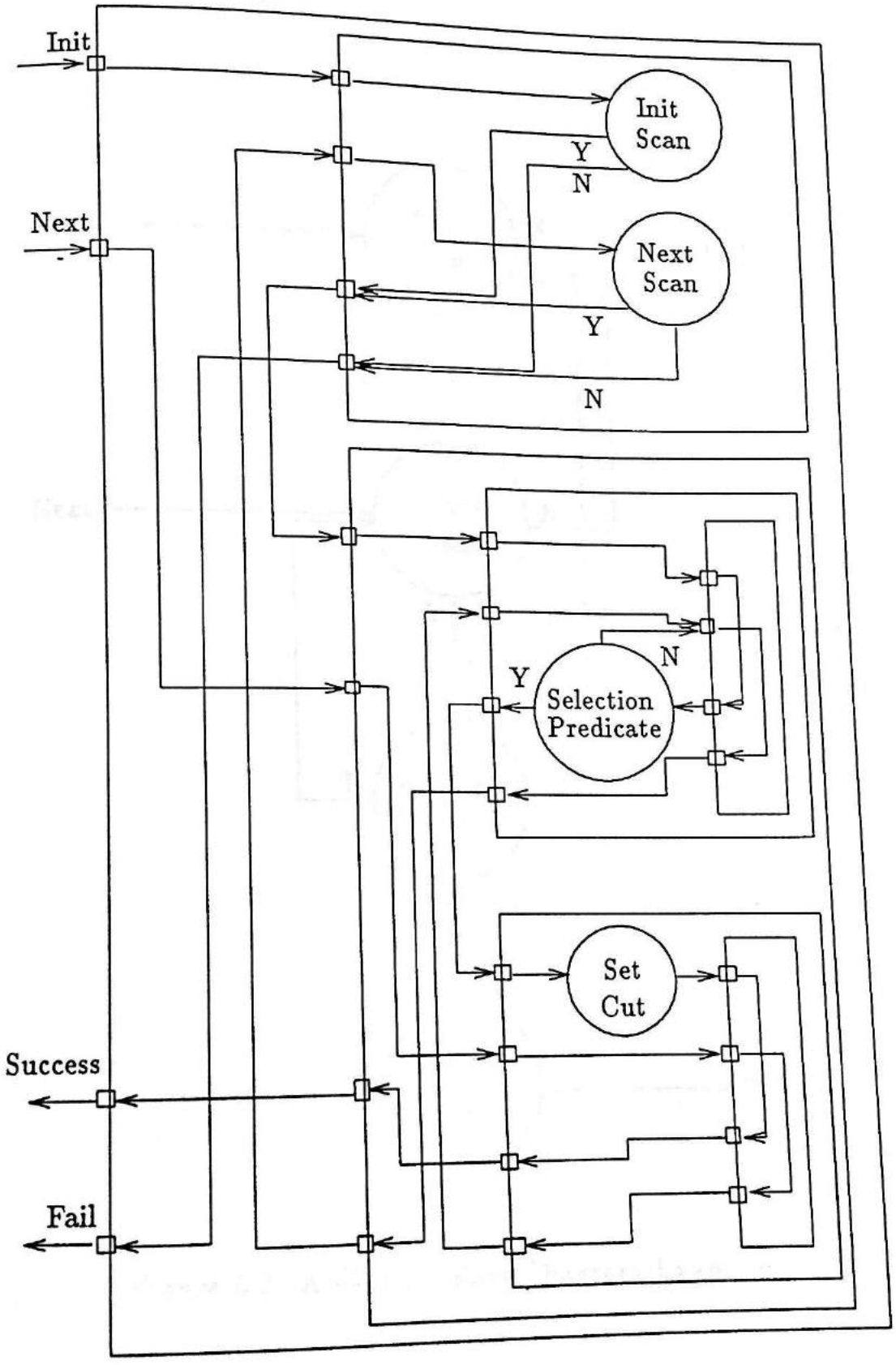


Figure 5.2: A Control-Flow Box Example

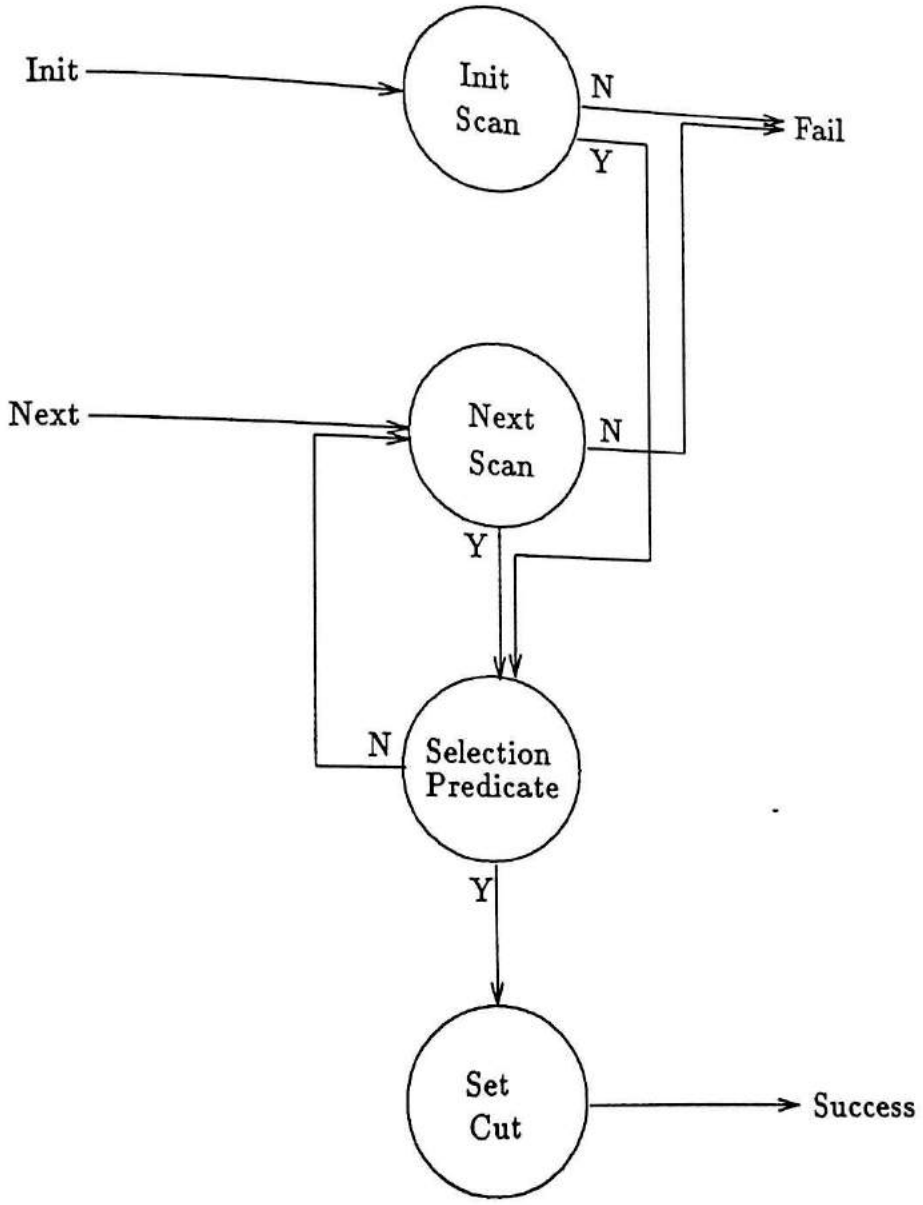


Figure 5.3: A Control-Flow Diagram Example

The implementation of the four special entities **Init**, **Next**, **Success**, and **Fail** are as follows: at the beginning of a query function body, an **if** statement

```
if(PDMCQStruct->First == 0) goto Next;
```

is generated. This **if** statement determines if the call represents an **Init** or **Next** entry. (Recall that this is indicated by the value in the field 'First' of the query structure.) If the call represents a **Next** entry, then control is transferred by a **goto** statement to the control-flow entry point of the code that represents the operation from which **Next** should be started. Following the **if** statement is a list of assignment statements which initialize and update the fields in the query structure. The field 'First' and fields representing the cut variables are assigned '0'. Then the code (including the **label** statement if necessary) for the operation from which **Init** should be started is generated. So, if the check made by the **if** statement fails, this code will be immediately evaluated. **Success** and **Fail** are both represented by a **label** statement, and the following piece of code is generated after the **label** statement that represents the entity **Success**.

```
PDMCQStruct->Result = 1;
return;
```

This code sets the field 'Result' in the query structure to the value '1' (which indicates that a solution to the query has been successfully found), and the control is then returned to the caller of the query function. The following piece of code is generated for the the entity **Fail** as follows.

```
PDMCQStruct->Result = 0;
return;
```

Code generation for each type of operations is outlined in Tables 5.5 and 5.6. Some remarks concerning the generated code are given as follows.

Operation Name
Corresponding Code

Corresponding APE

(1) **Init Scan**

case 1:

assign v as first of I [where *search-pred*]

```
if (fcall)
    goto Y;
goto N;
```

case 2:

assign v as each of I [where *search-pred*]

```
PDMCSchema->ILock = 1;
if (fcall)
    goto Y;
PDMCSchema->ILock = 0;
goto N;
```

(2) **Next Scan**

assign v as each of I [where *search-pred*]

```
if (PDMCQStruct->Cutv)
    goto N;
if (fcall)
    goto Y;
PDMCSchema->ILock = 0;
goto N;
```

(3) **Conditional Assignment**

assign v as t in C

```
if (p) {
    Assign-Statement;
    goto Y;
}
goto N;
```

Table 5.5: Code Generation for Operations in Control-Flow Diagram (a)

Operation Name Corresponding Code	Corresponding APE
(4) **Assignment** *Assign-Statement*; goto Label;	assign v as t
(5) **Test Predicate** if (p) goto Y; goto N;	verify *predicate sub-expression*
(6) **Set Cut** PDMCSchema−>ILock = 0; PDMCQStruct−>Cut$v$ = 1; goto Label;	cut v *sub-expression*

Note: – v is an object-valued variable name
– t is a term
– C is a class name
– I is an index name

Table 5.6: Code Generation for Operations in Control-Flow Diagram (b)

- In the generated code for the operation **Init Scan**, *fcall* is a function call to the index function **Init'I** or **DistInit'I** with appropriate parameters. (The formal parameters of these index function have been discussed in chapter 4.) The labels 'Y' and 'N' correspond to the two control-flow outlets of the operation in the control-flow diagram.

- For the operation **Next Scan**, the corresponding cut variable is first checked to ensure that the index scan has not been terminated. The *fcall* is a function call to the index function **Next'I** or **DistNext'I** with appropriate parameters.

- For the operation **Conditional Assignment**, p implements a class-type check which tests if the object expressed by the term t is an instance to the class C, and *Assign-Statement* is an assignment statement which binds the object expressed by the term t to the object-valued variable v.

- For the operation **Assignment**, *Assign-Statement* is an assignment statement which binds the object expressed by the term t to the object-valued variable v, and 'Label' corresponds to the control-flow outlet of the operation in the control-flow diagram.

- For the operation **Selection Predicate**, p is a test to check if the *predicate* is true or false.

- For the operation **Set Cut**, the field that represents the cut variable in the query structure is set.

5.1.4 An Example of DDBMS Source file

A complete example of DDBMS source file is given in Appendix F. The reader is referred to this file for an example of the code generation process discussed above.

5.2 Code Generation for Application Source

An application source file is generated from a C/DB source file with extended C constructs replaced by ANSI C code. Extended C constructs in C/DB language are in one of five categories:

- **schema** statements which declare memory-resident databases from database schemas,
- **prop** statements which declare object-valued variables,
- use of the special operator @ to access object properties,
- invocations to transactions, and
- invocations to queries which are supported by new forms of **if**, **for**, and **while** statements.

The following subsections discuss code generation for each.

5.2.1 Schema Statements

A schema statement that has the form

schema *Schema-Name Declarator-List*;

is translated to

#include "*Schema-Name*.h";
struct *Schema-Name*Struct *Declarator-List*;

The include statement in the first line includes the corresponding DDBMS source of the specified database schema. (The file name of a DDBMS source is formed by its corresponding schema name with the extension '.h'.) The *Declarator-List* is declare to the structure type that represents the database schema. The structure type is defined in the DDBMS source.

5.2.2 Prop Statements

A prop statement that has the form

prop *Class-Name Declarator-List*;

is translated according to the type of the class named *ClassName*. The possible cases are as follows.

Case 1: if the type of the class named *ClassName* is on the basic class **Integer**, the prop statement is translated to

int *Declarator-List*;

Case 2: if the type is on the basic class **Real**, the prop statement is translated to

float *Declarator-List*;

Case 3: if the type is on the basic class **DoubleReal**, the prop statement is translated to

double *Declarator-List*;

Case 4: if the type is on the basic class String with a maximum size i, and the *Declarator-List* contains n identifiers, namely $id_1,\ldots,id_n$, the prop statement is translated to

char $id_1[i],\ldots,id_n[i]$;

Case 5: if the type is on a user defined class which adopts the *direct* reference method, and the *Declarator-List* contains n identifiers, namely $id_1,\ldots,id_n$, the prop statement is translated to

struct *Class-Name* $*id_1,\ldots,*id_n$;

Case 6: if the type is on a user defined class which adopts the *indirect* reference method, and the *Declarator-List* contains n identifiers, namely $id_1,\ldots,id_n$, the prop statement is translated to

struct *Class-Name* $**id_1,\ldots,**id_n$;

5.2.3 The Special Operator @

Each use of the special operator @ in a C/DB source is translated to a function call to an access function. Thus, when the operator is used in the following way

v @ p

to access the property p of an object-valued variable v which is on the class C, the expression will be translated to the following function call

Access$Cp(v)$

In general, an expression of the form

$$v \ @ \ p_1 \ @ \ p_2 \ \ldots @ \ p_n$$

where v is an object-valued valued on the class C, and p_1, p_2,...,p_n are property names for properties on the classes C_1, C_2,...,C_n respectively. The expression is translated to

$$\text{Access}C_{n-1}p_n(\ldots(\text{Access}C_1p_2(\text{Access}Cp_1(v)))\ldots)$$

5.2.4 Invocations to Transactions

A transaction invocation that has the form

invoke *Transaction-Name*(*Expression-List*) **in** *Database-Name*

or

invoke *Transaction-Name*(*Expression-List*)

where the **in** clause is absent when the *Database-Name* is clear in context. (A *Database-Name* is an identifier of a database structure which has been defined in a schema statement). The transaction invocation is translated to the following function call

Transaction-Name(&*Database-Name*, *Expression-List*)

with &*Database-Name* and *Expression-List* as parameters.

5.2.5 Invocations to Queries

Invocations to queries are supported by new forms of while, for, and if statements. A query invocation supported by the while statement in the form

> while $Id_1,\ldots,Id_m$ in *Query-Name*$(expr_1,\ldots,expr_n)$ in *Database-Name*
> *Statements*

is translated to the following piece of code (the same piece of code is generated when *Database-Name* is supplied implicitly, i.e. the in clause which supplies the *Database-Name* is absent):

```
{
   struct Query-NameStruct Generated-Id;
   Generated-Id.Result = 1;
   while (Generated-Id.Result) {
      Generated-Id.First = 1;
      Generated-Id.Given-Id_1 = expr_1;
          :
      Generated-Id.Given-Id_n = expr_n;
      Query-Name(&Database-Name, &Generated-Id);
      if (Generated-Id.Result) {
         Id_1 = Generated-Id.Select-Id_1;
             :
         Id_m = Generated-Id.Select-Id_m;
         Statements
      }
   }
}
```

In the generated code, a system generated variable named 'Generated-Id' is declared as a query structure of the query. This query structure is then used as a communication bridge between the application source and the query function.

Inputs to the query ($expr_1$,...,$expr_n$) are first evaluated and stored in their corresponding fields (*Given-Id*$_1$,...,*Given-Id*$_n$) in the query structure. The string function 'strcpy' will be used appropriately for copying character strings. Then the query function is called with the schema structure (held by the variable *Database-Name*) and query structure as its parameters. If the query function has successfully found a solution (which is indicated by the value '1' in the field 'Result' of the query structure), the solution of the query is then copied from the fields in the query structure (*Select-Id*$_1$,...,*Select-Id*$_m$) to the object-valued variables (Id_1,...,Id_m) that are specified in the query invocation statement, then the *Statements* are evaluated, and the whole process is repeated.

A query invocation supported by the **for** statement in the form

for Id_1,...,Id_m **in** *Query-Name*($expr_1$,...,$expr_n$) **in** *Database-Name*
Statements

is translated to the following piece of code

```
{
    struct Query-NameStruct Generated-Id;
    Generated-Id.First = 1;
    do {
        Generated-Id.Given-Id_1 = expr_1;
            :
        Generated-Id.Given-Id_n = expr_n;
        Query-Name(&Database-Name, &Generated-Id);
        if (Generated-Id.Result) {
            Id_1 = Generated-Id.Select-Id_1;
                :
            Id_m = Generated-Id.Select-Id_m;
            Statements
        }
    } while (Generated-Id.Result);
}
```

In this generated code, a do-while loop is used to get all the solutions from the query; and the *Statements* is evaluated for each solution.

A query invocation supported by the if statement in the form

if $Id_1,\ldots,Id_m$ in $QueryName(expr_1,\ldots,expr_n)$ in *Database-Name*
$Statements_1$ else $Statements_2$

is translated to the following piece of code

```
{
    struct Query-NameStruct Generated-Id;
    Generated-Id.First = 1;
    while (Generated-Id.Result) {
        Generated-Id.First = 1;
        Generated-Id.Given-Id1 = expr1;
            :
        Generated-Id.Given-Idn = exprn;
        Query-Name(&Database-Name, &Generated-Id);
        if (Generated-Id.Result) {
            Id1 = Generated-Id.Select-Id1;
                :
            Idm = Generated-Id.Select-Idm;
            Statements1
        }
        else
            Statements2
    }
}
```

In this generated code, an if-else statement is used to decide whether $Statements_1$ or $Statements_2$ is evaluated. The $Statements_1$ is evaluated if the query function has successfully found an solution; otherwise the $Statements_2$ is evaluated. If the else clause is omited from the query invocation statement, then the else clause in the generated code will be omitted accordingly.

5.2.6 An Example of Application Source File

A complete example of application source file is given in Appendix E. The reader is referred to this source file for examples related to code translation procedures discussed in this section.

5.3 Summary

In this chapter, we have described the process of code generation for RDM's output sources: the DDBMS source and the Application source. For the DDBMS source, its general structure has been described together with discussion on how each part of a DDBMS source is generated. An internal abstraction called control-flow boxes which are used to derive the overall control flow of an expression tree. For the Application source, the translation procedure for each kind of extended C construct in the C/DB source to its corresponding C code has been presented.

Chapter 6

Conclusions

6.1 Summary

RDM is a software tool set which helps with the development of application software components that access and update information residing in main memory. In this thesis, RDM (Resident Database Manager) has been described. Discussion has concentrated on how index type extension and code generation are done in RDM.

We have presented the operational abstraction that RDM applies to all indices. Index manipulation is accomplished solely in terms of a fixed set of index operations. Based on that, we have addressed the information that is required for RDM on an user defined index type. These include two types of information: selection information and implementation information. Selection information provides a means for RDM to compare the index type to other index types and thus enabling RDM to perform index selection. The implementation information provides detailed information on implementing index instances of the index type. We have proposed a language called ITS in which the required information on an index type

can be specified. By using this language, information on index types can be supplied externally to RDM and new index types can be extended to RDM at any moment.

We have described the code generation process for the two kinds of RDM output source files: application source file and DDBMS source file. An application source file is generated form a source file in an extended C language called C/DB. The translation procedure for each kind of extended C construct allowed in C/DB to its corresponding C code has been outlined. A DDBMS source file contains the source code of a dedicated database management system. The general structure of a DDBMS source file has been described together with discussion on how each of its part is generated. We have proposed a tool called control-flow boxes. Control-flow boxes assist the code generation process for query access plans which involve complicated control-flow structures.

6.2 Future Directions

To specify an index type in terms of the ITS language, the index type has to be adapted to the abstraction that is assumed by ITS. The abstraction includes the set of index operations and the set of index type characteristics. Although these index operations and index type characteristics are rather fundamental and they capture the essential manipulation operation and characteristics of most index types, difficulty will arise when we try to specify index types that possess special characteristics and abilities. For example, the current set of index operations and index type characteristics is not adequate to capture the hash index types. One way to overcome this is by allowing a new index type characteristic which is mutually excluded form the *ordered* characteristic. An index with this characteristic is specified

with a set of search conditions together with a hash function. The index must be accessed with an instance for each of the search conditions, and the hash function will be used to located the qualifying object. Further work can be done on selecting a better set of index operations and index type characteristics, so that index types in a wider spectrum can be captured.

The formula used for describing costs of index operations is an initial attempt. Since cost information for index operations pays a crucial role in the index selection process, a better cost formula is certainly required.

The two unimplemented APE operations, **Sort** and **Projection**, can be implemented easily by designing a data structure to encode the TS (temporary store) in the query structure. Corresponding code would be generated in the DDBMS source to encode the functions 'initialize_TS', 'store_TS', 'release_TS', 'project_TS', and 'sort_TS' (which are defined in Appendix B).

Appendix A

Table of Acronyms

APE	Access Plan Expression
DDBMS	Dedicated Database Management System
DML	Data Manipulation Language
ITS	Index Type Specification
LDM	Logical Data Model
MRDB	Memory-Resident Database
PDM	Physical Data Model
RDM	Resident Database Manager

Appendix B

APE Operation Semantics

In this appendix, we specify an operational semantics for each of the APE operations listed in Table 2.2. Recall that each APE is associated with a unique QE. Part of our operational specification indicates how an APE operation affects the associated QE.

Operation: **Index Scan**

APE syntax: assign v as each of I [where *search-pred*]

Parameters: v, I, *search-pred*

Effect on QE: a field named 'Cut$v$' is added

Procedures:

```
init:   Cutv := false;
        if InitI(v, search-pred) = 1 then
          return success
        else
          return failure;

next:   if Cutv = true then
          return failure;
        if NextI(v, search-pred) = 1 then
          return success
        else
          return failure;
```

Remarks:

- The operation **Index Scan** scans a specified index for indexed objects that satisfy a given search predicate. (Search predicate has been described in chapter 3.)
- The field added to the QE implements a cut variable whose name is formed by concatenating the string 'Cut' with the the object-valued variable name instance 'v'.
- The two functions 'Init$I$' and 'Next$I$' implements two index operations (note that the index name instances participate part of both function names) which access the index. Index operations have been described in chapter 3.

Operation: **Index Lookup**

APE syntax: assign v as first of I [where *search-pred*]

Parameters: v, I, *search-pred*

Effect on QE: Nil

Procedures:

```
init:   if InitI(v, search-pred) = 1 then
          return success
        else
          return failure;

next:   return failure;
```

Remarks:

- The operation **Index Scan** scans a specified index for ONE indexed object that satisfies a given search predicate.

Operation: **Conditional Assignment**

APE syntax: assign v as t in C

Parameters: v, t, C

Effect on QE: Nil

Procedures:

```
init:   if t in C then
        begin
          v := t;
          return success;
        end
        else
          return failure;

next:   return failure;
```

Operation: **Assignment**

APE syntax: assign v as t

Parameters: v, t

Effect on QE: Nil

Procedures:

```
init:   v := t;
        return success;

next:   return failure;
```

Operation: **Null**

APE syntax: Nil

Parameters: Nil

Effect on QE: Nil

Procedures:

init: **return** success;

next: **return** failure;

Remarks:

- The operation **Null** does not have its corresponding APE syntax. Its existence is implied implicitly by the absent of an expected sub-expressions in an APE statement.

Operation: **Selection**

APE syntax: verify *selection-pred sub-expression*

Parameters: *selection-pred sub-expression*

Effect on QE: Nil

Procedures:

```
init:   if I(sub-expression) = failure then
            return failure;
        while true do
        begin
            if selection-pred then
                return success;
            if N(sub-expression) = failure then
                return failure;
        end

next:   while true do
        begin
            if N(sub-expression) = failure then
                return failure;
            if selection-pred then
                return success;
        end
```

Remarks:

- The function calls '$\mathcal{I}$(*sub-expression*)' and '$\mathcal{N}$(*sub-expression*)' signify calls to the *init* and *next* procedures of the APE operation that corresponds to the *sub-expression*.
- A selection predicate '*selection-pred*' expresses a database object comparison in terms of variable names and class property names.

Operation: **Complement**

APE syntax: complement *sub-expression*

Parameters: *sub-expression*

Effect on QE: Nil

Procedures:

```
init:   if I(sub-expression) = failure then
            return success
        else
            return failure;

next:   return failure;
```

Operation: **Cut**

APE syntax: cut *v sub-expression*

Parameters: *v*, *sub-expression*

Effect on QE: Nil

Procedures:

```
init:   Cutv := true;
        if I(sub-expression) = failure then
            return failure
        else
            return success;

next:   if N(sub-expression) = failure then
            return failure
        else
            return success;
```

Operation: **Projection**

APE syntax: project $v_1,\ldots,v_n$ *sub-expression*

Parameters: $v_1,\ldots,v_n$, *sub-expression*

Effect on QE: fields are added to support a temporary store (TS) for storing the temporary results

Procedures:

```
init:   initialize_TS();
        if I(sub-expression) = success then
        begin
            store_TS(v1,...,vn);
            while N(sub-expression) = success do
                store_TS(v1,...,vn);
        end
        else
            return failure;
        project_TS(v1,...,vn);
        if release_TS() = 1 then
            return success
        else
            return failure;

next:   if release_TS() = 1 then
            return success
        else
            return failure;
```

Remarks:

- The function 'initialize_TS()' initializes the TS in preparation for storing temporary results.

- The function 'store_TS()' stores a temporary result into the TS. A temporary result is the contents of a set of object-valued variables that are defined in the QE.

- The function 'release_TS()' releases a stored result. The stored object-valued contents are copied back to their original object-valued variables. The function will return the value '1', if a stored result has been successfully released.

- The function 'project_TS()' performs a projection on the stored results to make each result unique in contents. In other words, only one result is retained for a set of results that have the same contents.

Operation: **Sort**

APE syntax: sort *sort-info sub-expression*

Parameters: *sort-info*, *sub-expression*

Effect on QE: fields are added to support a temporary store (TS) for storing temporary results

Procedures:

```
init:   initialize_TS();
        if I(sub-expression) = success then
        begin
            store_TS(v1,...,vn);
            while N(sub-expression) = success do
                store_TS(v1,...,vn);
        end
        else
            return failure;
        sort_TS(sort-info);
        if release_TS() = 1 then
            return success
        else
            return failure;

next:   if release_TS() = 1 then
            return success
        else
            return failure;
```

Remarks:

- $v_1,\ldots,v_n$ are object-valued variables involved in the sorting condition provided by the *sort-info*.

- The function 'sort_TS()' sorts the stored results in an order that is specified by the *sort-info*, and the function 'release_TS()' will release the results according to the sorted order.

Operation: **Nested Cross Product**

APE syntax: nest *sub-expression*; *sub-expression-list*; end

Parameters: *sub-expression*, *sub-expression-list*

Effect on QE: Nil

Procedures:

```
init:   if I(sub-expression) = failure then
            return failure;
        while true do
        begin
            if I(sub-expression-list) = success then
                return success;
            if N(sub-expression) = failure then
                return failure;
        end

next:   if N(sub-expression-list) = success then
            return success;
        while true do
        begin
            if N(sub-expression) = failure then
                return failure;
            if I(sub-expression-list) = success then
                return success;
        end
```

Remarks:

- The Nested Cross Product operation is the only APE operation that has the ability to combine results from more than one APE operations.

Appendix C

Control-Flow Boxes for the APE Operations

This appendix presents the control-flow boxes for each of the APE operations. Diagrammatic conventions of these control-flow boxes have been discussed in chapter 5. Control-flow boxes for

- Index Scan,
- Index Lookup,
- Conditional Assignment,
- Assignment,
- Null,
- Selection,
- Complement,
- Cut, and
- Nested Cross Product

are depicted in Figures C.1 to C.9 respectively.

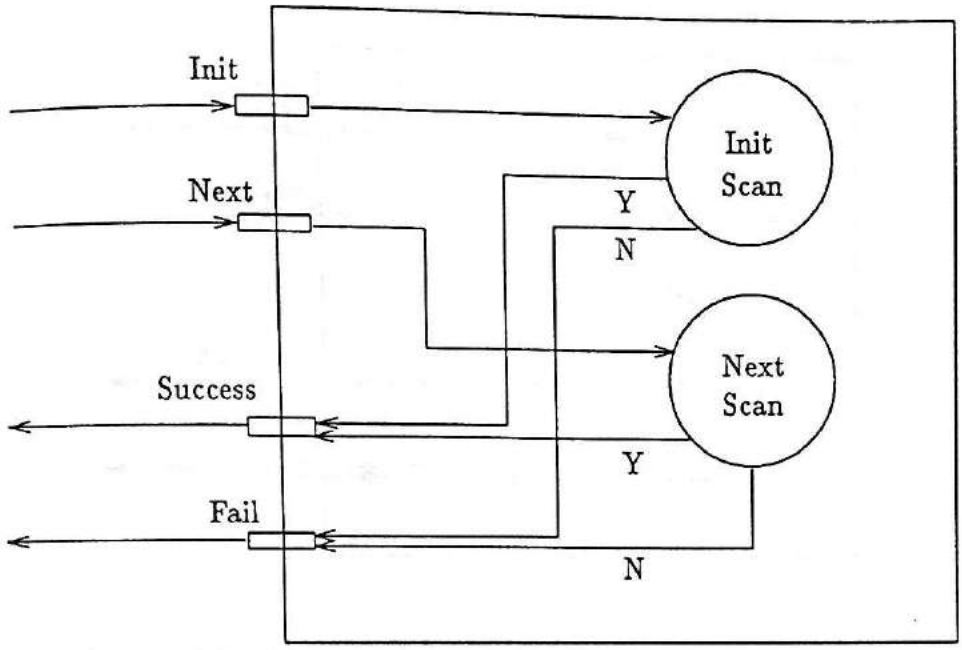


Figure C.1: Control-Flow Box for Index Scan

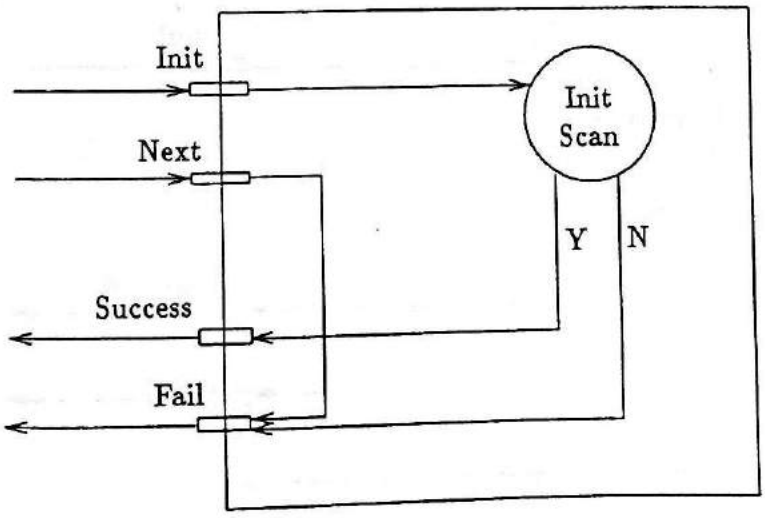


Figure C.2: Control-Flow Box for Index Lookup

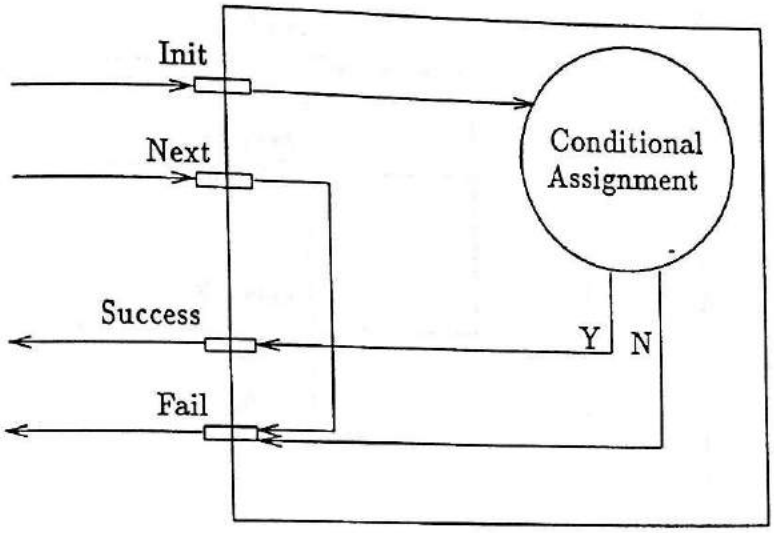


Figure C.3: Control-Flow Box for Conditional Assignment

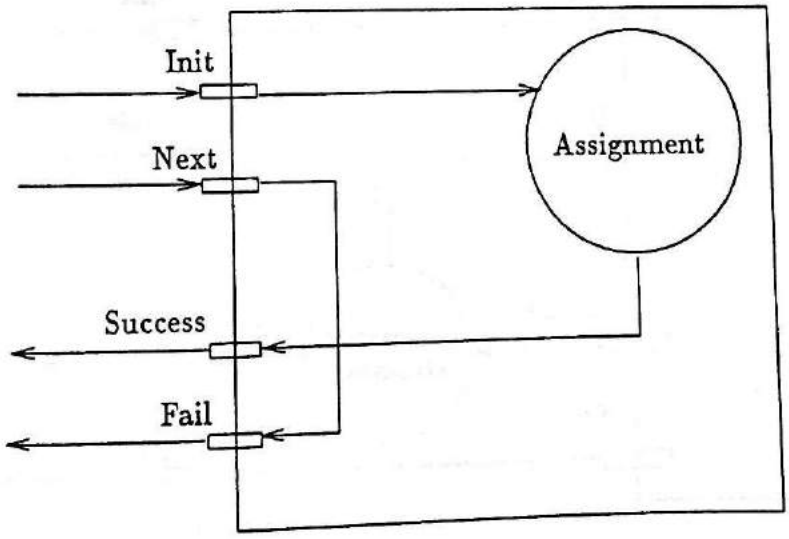


Figure C.4: Control-Flow Box for Assignment

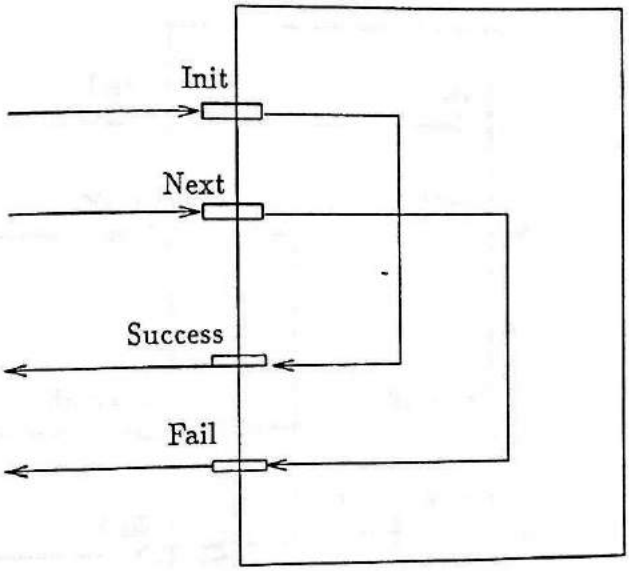


Figure C.5: Control-Flow Box for Null

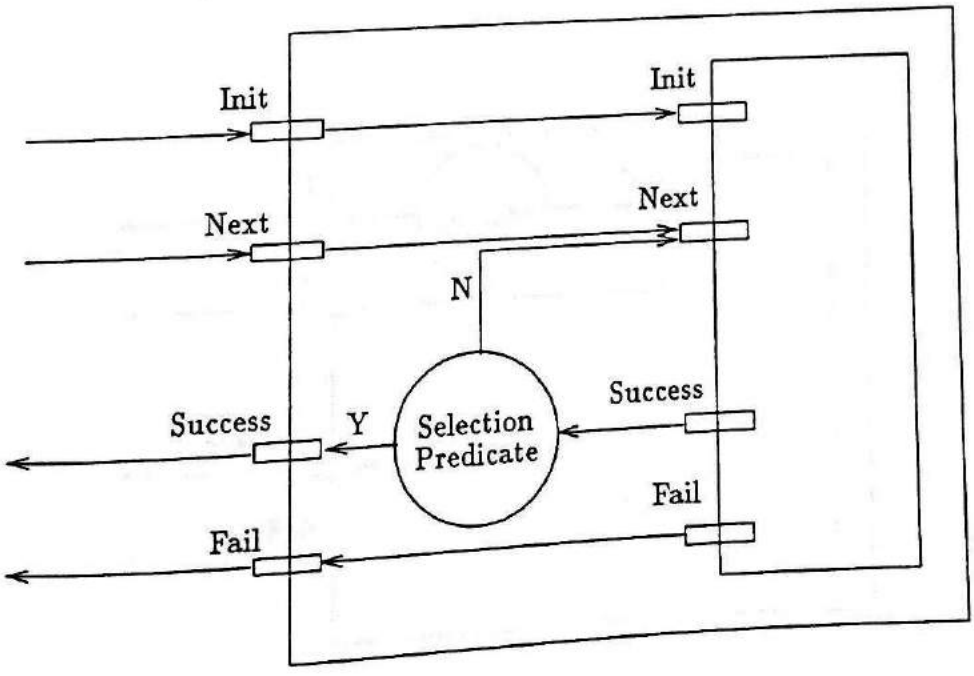


Figure C.6: Control-Flow Box for Selection

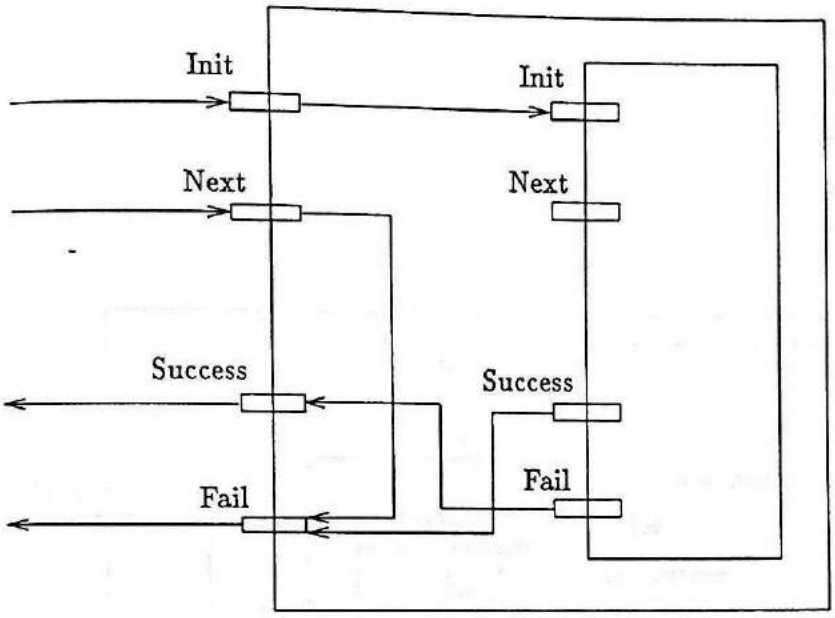


Figure C.7: Control-Flow Box for Complement

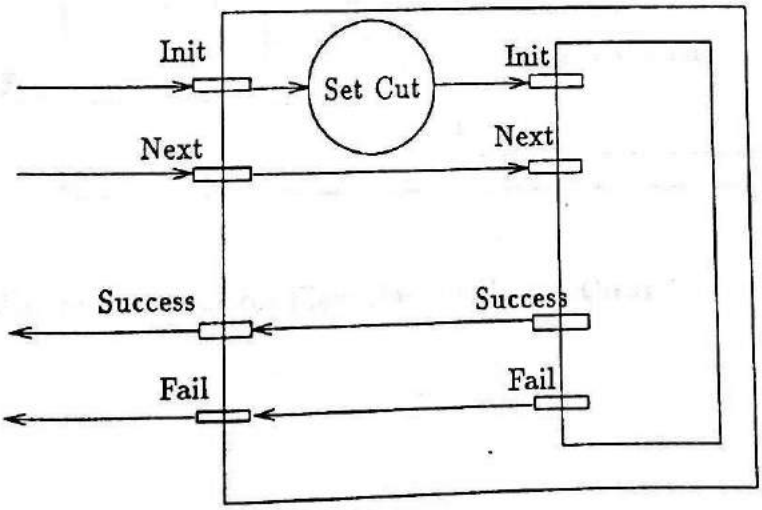


Figure C.8: Control-Flow Box for Cut

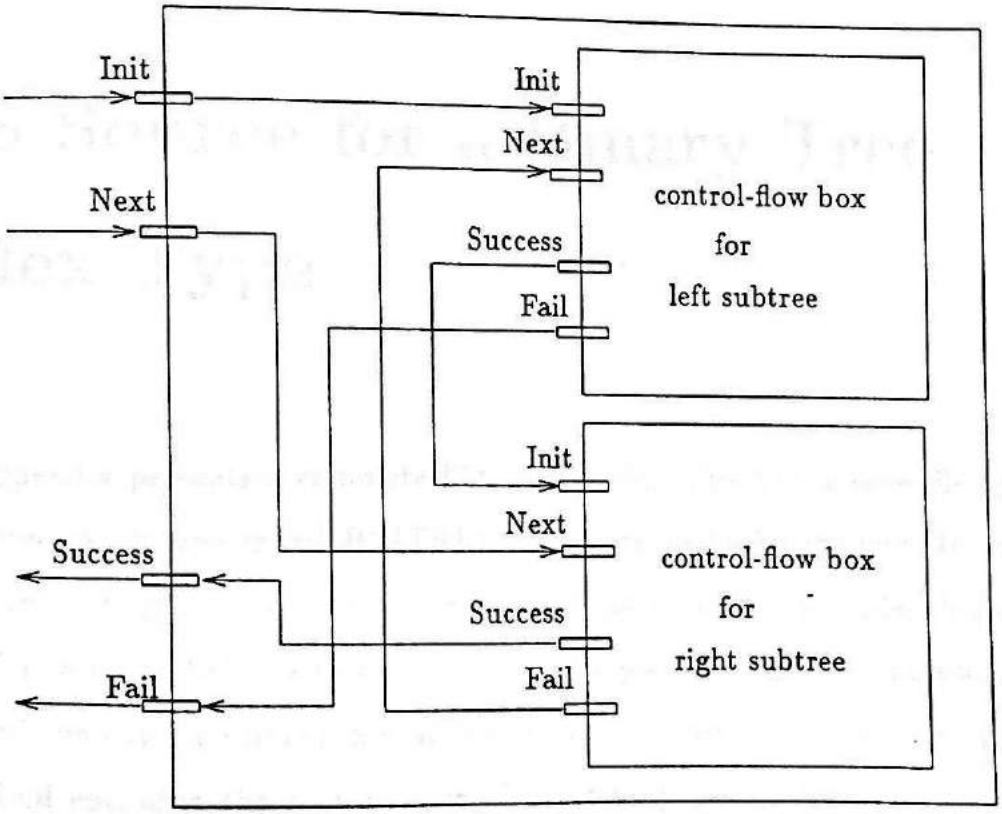


Figure C.9: Control-Flow Box for Nested Cross Product

Appendix D

ITS Source for a Binary Tree Index Type

This appendix presents a complete ITS source file. This ITS source file specifies an ordered index type called 'BINTREE' which stands for binary tree. In an index of this index type, indexed objects are organized in a right-threaded binary tree [20]. Three extra fields, encoding the left son pointer, right son pointer, and a threaded indicator ('IMark), are added to the indexed class structure type. An extra field encoding the root pointer of the binary tree is added in the schema structure type. Other than the required index functions, a number of additional function templates are defined in the code template section of this ITS source file. These extra functions are designed to facilitate the index functions.

```
index type
   BINTREE

characteristics
   ordered


representation
   extension to schema properties 'IHead
   property 'IHead on 'C

   extension to indexed class properties 'IRSon, 'ILSon, 'IMark
   property 'ILSon on 'I
   property 'IRSon on 'I
   property 'IMark on Integer


cost
   SetUp cost   0  0  2  1
   Init cost    0  1  2  0
   Next cost    0  0  2  4
   Add cost     0  1  2  0
   Sub cost     0  1  2  2


code template

inline void SetUp'I(DBName)
struct 'SStruct *DBName;
{
   DBName->'IHead = NULL;
}

inline void 'ITreeInsert(new, pointer, CompFunc)
prop 'C new, pointer;
int (* CompFunc) ();
{
   switch ((* CompFunc) (new, pointer)) {
   case -1:   if (Access'C'ILSon(pointer) == NULL) {
                 Update'C'IMark(new, 1);
```

```
                Update'C'IRSon(new, pointer);
                Update'C'ILSon(pointer, new);
            } else
                'ITreeInsert(new, Access'C'ILSon(pointer), CompFunc);
            break;
    case 1 : if (Access'C'IMark(pointer) == 1) {
                Update'C'IMark(new, 1);
                Update'C'IMark(pointer, 0);
                Update'C'IRSon(new, Access'C'IRSon(pointer));
                Update'C'IRSon(pointer, new);
            }
            else if (Access'C'IRSon(pointer) == NULL)
                Update'C'IRSon(pointer, new);
            else
                'ITreeInsert(new, Access'C'IRSon(pointer), CompFunc);
            break;
    case 0 : printf("error : new key already exists.\n");
            exit(0);
            break;
    }
}

inline prop 'C 'ITreeNext(pointer)
prop 'C pointer;
{
    if (Access'C'IMark(pointer) == 0 && Access'C'IRSon(pointer) != NULL) {
        pointer = Access'C'IRSon(pointer);
        while (Access'C'ILSon(pointer) != NULL)
            pointer = Access'C'ILSon(pointer);
            return (pointer);
    }
    else
        return(Access'C'IRSon(pointer));
}

inline prop 'C 'ITreeSearch(pointer, CompFunc, P)
prop 'C pointer;
int (* CompFunc) ();
void *P;
{
    if (pointer == NULL) return(NULL);
    if ((* CompFunc) (pointer, P) == 1)
```

```
    if (Access'C'IMark(pointer) == 0)
        return('ITreeSearch(Access'C'IRSon(pointer), CompFunc, P));
    else
        return('ITreeNext(pointer));
    if (Access'C'ILSon(pointer) != NULL)
        return('ITreeSearch(Access'C'ILSon(pointer), CompFunc, P));
  return(pointer);
}

inline prop 'C 'IFindParent(pointer,child, CompFunc)
prop 'C pointer, child;
int (* CompFunc) ();
{
  switch ((* CompFunc)(child, pointer)) {
    case -1:  if (Access'C'ILSon(pointer) == child) return(pointer);
              else return('IFindParent(Access'C'ILSon(pointer), child, CompFunc));
    case 1 :  if (Access'C'IRSon(pointer) == child) return(pointer);
              else return('IFindParent(Access'C'IRSon(pointer), child, CompFunc));
  }
}

inline void 'ITreeRemove(DBName, pointer, CompFunc)
struct 'SStruct *DBName;
prop 'C pointer;
int (* CompFunc) ();
{
  prop 'C parent, temp;
  if (Access'C'ILSon(pointer) != NULL) {
    temp = Access'C'ILSon(pointer);
    while (Access'C'IMark(temp) != 1)
      temp = Access'C'IRSon(temp);
    Update'C'IRSon(temp, 'ITreeNext(pointer));
  }
  if (Access'C'IMark(pointer) == 0)
    if (Access'C'ILSon(pointer) != NULL) {
      Update'C'IMark(temp, 0);
      Update'C'IRSon(temp, Access'C'IRSon(pointer));
    }
    else
      Update'C'ILSon(pointer, Access'C'IRSon(pointer));
  if (DBName->'IHead == pointer)
    DBName->'IHead = Access'C'ILSon(pointer);
```

```
    else {
        parent = 'IFindParent(DBName->'IHead, pointer, CompFunc);
        if (Access'C'ILSon(parent) == pointer)
            Update'C'ILSon(parent, Access'C'ILSon(pointer));
        else {
            if (Access'C'ILSon(pointer) == NULL && Access'C'IMark(pointer) == 1) {
                Update'C'IMark(parent, 1);
                Update'C'IRSon(parent, Access'C'IRSon(pointer));
            }
            else
                Update'C'IRSon(parent, Access'C'ILSon(pointer));
        }
    }
}

inline void Add'I(DBName, A, CompFunc)
struct 'SStruct *DBName;
prop 'C A;
int (* CompFunc) ();
{
    Update'C'ILSon(A, NULL);
    Update'C'IRSon(A, NULL);
    Update'C'IMark(A, 0);
    if (DBName->'IHead == NULL)
        DBName->'IHead = A;
    else
        'ITreeInsert(A, DBName->'IHead, CompFunc);
}

inline void Sub'I(DBName, A, CompFunc)
struct 'SStruct *DBName;
prop 'C A;
int (* CompFunc) ();
{
    'ITreeRemove(DBName, A, CompFunc);
}

inline int Init'I(DBName, A, CompFunc, P)
struct 'SStruct *DBName;
prop 'C *A;
int (* CompFunc) ();
void *P;
```

```
{
   (*A) = 'ITreeSearch(DBName->'IHead, CompFunc, P);
   if ((*A) == NULL || (* CompFunc) (*A, P) != 0)
      return(0);
   else
      return(1);
}

inline int Next'I(DBName, A, CompFunc, P)
struct 'SStruct *DBName;
prop 'C *A;
int (* CompFunc) ();
void *P;
{
   (*A) = 'ITreeNext((*A));
   if (((*A) == NULL) || (* CompFunc) (*A, P) != 0)
      return(0);
   else
      return(1);
}
```

Appendix E

A Sample Application Source

This appendix presents a sample application source. This application source file is generated from the C/DB source file listed in Figure 2.3. Note how each of the extended language contructs is translated to its corresponding C code.

```
#include <stdio.h>
#include "PeopleDB.h"
struct PeopleDBStruct DB1, DB2;


main()
{
    struct Person *P1;
    struct Person *P2;
    char N[20] ;
    int A;
    int DBNum;
    char Type[10] ;
    InitPeopleDB(&DB1);
    InitPeopleDB(&DB2);
    while (scanf("%d %s %s %d", &DBNum, Type, N, &A) != EOF)
        if (DBNum == 1)
            if (strcmp(Type, "Man") == 0)
                EnterMan(&DB1, N, A);
            else
                EnterWoman(&DB1, N, A);
        else if (strcmp(Type, "Man") == 0)
            EnterMan(&DB2, N, A);
        else
            EnterWoman(&DB2, N, A);
    {
        struct AllPeopleStruct CDBSchemaVar1;
        CDBSchemaVar1.First = 1;
        do {
            AllPeople(&DB1, &CDBSchemaVar1);
            if (CDBSchemaVar1.Result) {
                P1 = CDBSchemaVar1.P;
                printf("%s %d \n", AccessPersonName(P1), AccessPersonAge(P1));
                {
                    struct ManWithNameStruct CDBSchemaVar2;
                    CDBSchemaVar2.First = 1;
                    strcpy(CDBSchemaVar2.N, AccessPersonName(P1));
                    ManWithName(&DB2, &CDBSchemaVar2);
                    if (CDBSchemaVar2.Result) {
                        P2 = (struct Person *) CDBSchemaVar2.M;
```

```
                printf("----------\n");
            }
        }
    }
  } while (CDBSchemaVar1.Result);
 }
}
```

Appendix F

A Sample DDBMS Source

This appendix presents a complete source code of a DDBMS. This DDBMS source file is generated from the PDM source file listed in Figures 2.5 and 2.6. Each of the eight logical sections of the DDBMS source starts with a comment line. Note that the compiler directive 'inline' for the functions has been suppressed in this DDBMS source.

```
/* === Section 1: structure declaration for classes === */

struct PeopleDBStruct {
   struct StoreTemplate *PersonStore;
   struct Person *PersonTreeHead;
   int PersonTreeLock;
};

struct Person {
   int Msc;
   int PersonTreeMark;
   struct Person *PersonTreeRSon;
   struct Person *PersonTreeLSon;
   char Name[20] ;
   int Age;
};

struct Woman {
   int Msc;
   int PersonTreeMark;
   struct Person *PersonTreeRSon;
   struct Person *PersonTreeLSon;
   char Name[20] ;
   int Age;
};

struct Man {
   int Msc;
   int PersonTreeMark;
   struct Person *PersonTreeRSon;
   struct Person *PersonTreeLSon;
   char Name[20] ;
   int Age;
};

/* === Section 2: assign and update functions === */

struct StoreTemplate *AccessPeopleDBStructPersonStore(P)
struct PeopleDBStruct *P;
```

```
{
    return(P->PersonStore);
}

void UpdatePeopleDBStructPersonStore(P, Val)
struct PeopleDBStruct *P;
struct StoreTemplate *Val;
{
    P->PersonStore = Val;
}

struct Person *AccessPeopleDBStructPersonTreeHead(P)
struct PeopleDBStruct *P;
{
    return(P->PersonTreeHead);
}

void UpdatePeopleDBStructPersonTreeHead(P, Val)
struct PeopleDBStruct *P;
struct Person *Val;
{
    P->PersonTreeHead = Val;
}

int AccessPeopleDBStructPersonTreeLock(P)
struct PeopleDBStruct *P;
{
    return(P->PersonTreeLock);
}

void UpdatePeopleDBStructMsc(P, Val)
struct PeopleDBStruct *P;
int Val;
{
    P->PersonTreeLock = Val;
}

int AccessPersonMsc(P)
struct Person *P;
{
    return(P->Msc);
}
```

```
void UpdatePersonMsc(P, Val)
struct Person *P;
int Val;
{
   P->Msc = Val;
}

int AccessPersonPersonTreeMark(P)
struct Person *P;
{
   return(P->PersonTreeMark);
}

void UpdatePersonPersonTreeMark(P, Val)
struct Person *P;
int Val;
{
   P->PersonTreeMark = Val;
}

struct Person *AccessPersonPersonTreeRSon(P)
struct Person *P;
{
   return(P->PersonTreeRSon);
}

void UpdatePersonPersonTreeRSon(P, Val)
struct Person *P;
struct Person *Val;
{
   P->PersonTreeRSon = Val;
}

struct Person *AccessPersonPersonTreeLSon(P)
struct Person *P;
{
   return(P->PersonTreeLSon);
}

void UpdatePersonPersonTreeLSon(P, Val)
struct Person *P;
```

```
struct Person *Val;
{
   P->PersonTreeLSon = Val;
}

char *AccessPersonName(P)
struct Person *P;
{
   return(P->Name);
}

void UpdatePersonName(P, Val)
struct Person *P;
char Val[20] ;
{
   strcpy(P->Name, Val);
}

int AccessPersonAge(P)
struct Person *P;
{
   return(P->Age);
}

void UpdatePersonAge(P, Val)
struct Person *P;
int Val;
{
   P->Age = Val;
}

int AccessWomanMsc(P)
struct Woman *P;
{
   return(P->Msc);
}

void UpdateWomanMsc(P, Val)
struct Woman *P;
int Val;
{
   P->Msc = Val;
```

```
}
int AccessWomanPersonTreeMark(P)
struct Woman *P;
{
    return(P->PersonTreeMark);
}
void UpdateWomanPersonTreeMark(P, Val)
struct Woman *P;
int Val;
{
    P->PersonTreeMark = Val;
}
struct Person *AccessWomanPersonTreeRSon(P)
struct Woman *P;
{
    return(P->PersonTreeRSon);
}
void UpdateWomanPersonTreeRSon(P, Val)
struct Woman *P;
struct Person *Val;
{
    P->PersonTreeRSon = Val;
}
struct Person *AccessWomanPersonTreeLSon(P)
struct Woman *P;
{
    return(P->PersonTreeLSon);
}
void UpdateWomanPersonTreeLSon(P, Val)
struct Woman *P;
struct Person *Val;
{
    P->PersonTreeLSon = Val;
}
char *AccessWomanName(P)
```

```
struct Woman *P;
{
   return(P->Name);
}

void UpdateWomanName(P, Val)
struct Woman *P;
char Val[20] ;
{
   strcpy(P->Name, Val);
}

int AccessWomanAge(P)
struct Woman *P;
{
   return(P->Age);
}

void UpdateWomanAge(P, Val)
struct Woman *P;
int Val;
{
   P->Age = Val;
}

int AccessManMsc(P)
struct Man *P;
{
   return(P->Msc);
}

void UpdateManMsc(P, Val)
struct Man *P;
int Val;
{
   P->Msc = Val;
}

int AccessManPersonTreeMark(P)
struct Man *P;
{
   return(P->PersonTreeMark);
```

```
}

void UpdateManPersonTreeMark(P, Val)
struct Man *P;
int Val;
{
   P->PersonTreeMark = Val;
}

struct Person *AccessManPersonTreeRSon(P)
struct Man *P;
{
   return(P->PersonTreeRSon);
}

void UpdateManPersonTreeRSon(P, Val)
struct Man *P;
struct Person *Val;
{
   P->PersonTreeRSon = Val;
}

struct Person *AccessManPersonTreeLSon(P)
struct Man *P;
{
   return(P->PersonTreeLSon);
}

void UpdateManPersonTreeLSon(P, Val)
struct Man *P;
struct Person *Val;
{
   P->PersonTreeLSon = Val;
}

char *AccessManName(P)
struct Man *P;
{
   return(P->Name);
}

void UpdateManName(P, Val)
```

```
struct Man *P;
char Val[20] ;
{
   strcpy(P->Name, Val);
}

int AccessManAge(P)
struct Man *P;
{
   return(P->Age);
}

void UpdateManAge(P, Val)
struct Man *P;
int Val;
{
   P->Age = Val;
}

/* === Section 3: code for storage managers === */

int *PDMCellStore = NULL;

struct StoreTemplate {
   struct StoreTemplate *Next;
};

#define PersonStoreSize (sizeof(union {struct Woman PDMCDummy1;
struct Man
PDMCDummy2;}))

/* === Section 4: structure declaration for queries === */

struct AllPeopleStruct {
   int First;
   int Result;
   int CutP;
   struct Person *P;
};
```

```
struct ManWithNameStruct {
   int First;
   int Result;
   int CutM;
   char N[20] ;
   struct Man *M;
};

/* === Section 5: comparison functions === */

int PersonTreeCompare(P1, P2)
struct Person *P1;
struct Person *P2;
{
    if (((AccessPersonMsc(P1) & 1) == 0) || ((AccessPersonMsc(P2) & 1) == 0)) {
        if ((AccessPersonMsc(P1) & 1) != 0)
            return( -1);
        if ((AccessPersonMsc(P2) & 1) != 0)
            return(1);
        if (P1 < P2)
            return( -1);
        if (P1 > P2)
            return(1);
        return(0);
    } else {
        if (strcmp(AccessManName(P1), AccessManName(P2)) < 0)
            return( -1);
        if (strcmp(AccessManName(P1), AccessManName(P2)) > 0)
            return(1);
        if (P1 < P2)
            return( -1);
        if (P1 > P2)
            return(1);
        return(0);
    }
}

int PDMCCompare1(P, PDMCQStruct)
struct Person *P;
struct AllPeopleStruct *PDMCQStruct;
{
```

```
    return(0);
}

int PDMCCompare2(P, PDMCQStruct)
struct Person *P;
struct ManWithNameStruct *PDMCQStruct;
{
    if ((AccessPersonMsc(P) & 1) == 0)
        return( -1);
    if (strcmp(PDMCQStruct->N, AccessManName(P)) < 0)
        return( -1);
    if (strcmp(PDMCQStruct->N, AccessManName(P)) > 0)
        return(1);
    return(0);
}


/* === Section 6: code for indices === */

void SetUpPersonTree(DBName)
struct PeopleDBStruct *DBName;
{
    DBName->PersonTreeHead = NULL;
}

void PersonTreeTreeInsert(new, pointer, CompFunc)
struct Person *new;
struct Person *pointer;
int (*CompFunc) ();
{
    switch ((*CompFunc)(new, pointer)) {
    case -1:
        if (AccessPersonPersonTreeLSon(pointer) == NULL) {
            UpdatePersonPersonTreeMark(new, 1);
            UpdatePersonPersonTreeRSon(new, pointer);
            UpdatePersonPersonTreeLSon(pointer, new);
        } else
            PersonTreeTreeInsert(new, AccessPersonPersonTreeLSon(pointer),
            CompFunc);
        break;
    case 1:
        if (AccessPersonPersonTreeMark(pointer) == 1) {
```

```
            UpdatePersonPersonTreeMark(new, 1);
            UpdatePersonPersonTreeMark(pointer, 0);
            UpdatePersonPersonTreeRSon(new, AccessPersonPersonTreeRSon(pointer));
            UpdatePersonPersonTreeRSon(pointer, new);
        } else if (AccessPersonPersonTreeRSon(pointer) == NULL)
            UpdatePersonPersonTreeRSon(pointer, new);
        else
            PersonTreeTreeInsert(new, AccessPersonPersonTreeRSon(pointer),
            CompFunc);
        break;
    case 0:
        printf("error : new key already exists.\n");
        exit(0);
        break;
    }
}

struct Person *PersonTreeTreeNext(pointer)
struct Person *pointer;
{
    if (AccessPersonPersonTreeMark(pointer) == 0
        && AccessPersonPersonTreeRSon(pointer) != NULL) {
        pointer = AccessPersonPersonTreeRSon(pointer);
        while (AccessPersonPersonTreeLSon(pointer) != NULL)
            pointer = AccessPersonPersonTreeLSon(pointer);
        return(pointer);
    } else
        return(AccessPersonPersonTreeRSon(pointer));
}

struct Person *PersonTreeTreeSearch(pointer, CompFunc, P)
struct Person *pointer;
int (*CompFunc) ();
void *P;
{
    if (pointer == NULL)
        return(NULL);
    if ((*CompFunc)(pointer, P) == 1)
        if (AccessPersonPersonTreeMark(pointer) == 0)
            return(PersonTreeTreeSearch(AccessPersonPersonTreeRSon(pointer),
            CompFunc, P));
        else
```

```
        return(PersonTreeTreeNext(pointer));
    if (AccessPersonPersonTreeLSon(pointer) != NULL)
        return(PersonTreeTreeSearch(AccessPersonPersonTreeLSon(pointer),
        CompFunc, P));
    return(pointer);
}

struct Person *PersonTreeFindParent(pointer, child, CompFunc)
struct Person *pointer;
struct Person *child;
int (*CompFunc) ();
{
    switch ((*CompFunc)(child, pointer)) {
    case -1:
        if (AccessPersonPersonTreeLSon(pointer) == child)
            return(pointer);
        else
            return(PersonTreeFindParent(AccessPersonPersonTreeLSon(pointer),
            child, CompFunc));
    case 1:
        if (AccessPersonPersonTreeRSon(pointer) == child)
            return(pointer);
        else
            return(PersonTreeFindParent(AccessPersonPersonTreeRSon(pointer),
            child, CompFunc));
    }
}

void PersonTreeTreeRemove(Schema, pointer, CompFunc)
struct PeopleDBStruct *Schema;
struct Person *pointer;
int (*CompFunc) ();
{
    struct Person *parent;
    struct Person *temp;
    if (AccessPersonPersonTreeLSon(pointer) != NULL) {
        temp = AccessPersonPersonTreeLSon(pointer);
        while (AccessPersonPersonTreeMark(temp) != 1)
            temp = AccessPersonPersonTreeRSon(temp);
        UpdatePersonPersonTreeRSon(temp, PersonTreeTreeNext(pointer));
    }
    if (AccessPersonPersonTreeMark(pointer) == 0)
```

```
        if (AccessPersonPersonTreeLSon(pointer) != NULL) {
            UpdatePersonPersonTreeMark(temp, 0);
            UpdatePersonPersonTreeRSon(temp, AccessPersonPersonTreeRSon
            (pointer));
        }
        else
            UpdatePersonPersonTreeLSon(pointer, AccessPersonPersonTreeRSon
            (pointer));
    if (Schema->PersonTreeHead == pointer)
        Schema->PersonTreeHead = AccessPersonPersonTreeLSon(pointer);
    else {
        parent = PersonTreeFindParent(Schema->PersonTreeHead, pointer, CompFunc);
        if (AccessPersonPersonTreeLSon(parent) == pointer)
            UpdatePersonPersonTreeLSon(parent, AccessPersonPersonTreeLSon(pointer));
        else {
            if (AccessPersonPersonTreeLSon(pointer) == NULL
                && AccessPersonPersonTreeMark(pointer) == 1) {
                UpdatePersonPersonTreeMark(parent, 1);
                UpdatePersonPersonTreeRSon(parent, AccessPersonPersonTreeRSon
                (pointer));
        } else
            UpdatePersonPersonTreeRSon(parent, AccessPersonPersonTreeLSon(pointer));
        }
    }
}

void AddPersonTree(Schema, A, CompFunc)
struct PeopleDBStruct *Schema;
struct Person *A;
int (*CompFunc) ();
{
    UpdatePersonPersonTreeLSon(A, NULL);
    UpdatePersonPersonTreeRSon(A, NULL);
    UpdatePersonPersonTreeMark(A, 0);
    if (Schema->PersonTreeHead == NULL)
        Schema->PersonTreeHead = A;
    else
        PersonTreeTreeInsert(A, Schema->PersonTreeHead, CompFunc);
}

void SubPersonTree(Schema, A, CompFunc)
struct PeopleDBStruct *Schema;
```

```
struct Person *A;
int (*CompFunc) ();
{
   PersonTreeTreeRemove(Schema, A, CompFunc);
}

int InitPersonTree(Schema, A, CompFunc, P)
struct PeopleDBStruct *Schema;
struct Person **A;
int (*CompFunc) ();
void *P;
{
   (*A) = PersonTreeTreeSearch(Schema->PersonTreeHead, CompFunc, P);
   if ((*A) == NULL || (*CompFunc)(*A, P) != 0)
      return(0);
   else
      return(1);
}

int NextPersonTree(Schema, A, CompFunc, P)
struct PeopleDBStruct *Schema;
struct Person **A;
int (*CompFunc) ();
void *P;
{
   (*A) = PersonTreeTreeNext((*A));
   if (((*A) == NULL) || (*CompFunc)(*A, P) != 0)
      return(0);
   else
      return(1);
}


/* === Section 7: transation functions === */

void EnterMan(PDMCSchema, N, A)
struct PeopleDBStruct *PDMCSchema;
char N[20] ;
int A;
{
   struct Man *M;
   if (PDMCSchema->PersonStore == 0)
```

```
        M = (struct Man *)malloc(PersonStoreSize);
    else {
        M = (struct Man *)(PDMCSchema->PersonStore);
        PDMCSchema->PersonStore = PDMCSchema->PersonStore->Next;
    }
    UpdateManMsc(M , 1);
    UpdateManName(M , N);
    UpdateManAge(M , A);
    if (PDMCSchema->PersonTreeLock) {
        printf("Update Violation.");
        exit(0);
    }
    AddPersonTree(PDMCSchema , M , PersonTreeCompare);
}

void EnterWoman(PDMCSchema, N, A)
struct PeopleDBStruct *PDMCSchema;
char N[20] ;
int A;
{
    struct Woman *W;
    if (PDMCSchema->PersonStore == 0)
        W = (struct Woman *)malloc(PersonStoreSize);
    else {
        W = (struct Woman *)(PDMCSchema->PersonStore);
        PDMCSchema->PersonStore = PDMCSchema->PersonStore->Next;
    }
    UpdateWomanMsc(W , 2);
    UpdateWomanName(W , N);
    UpdateWomanAge(W , A);
    if (PDMCSchema->PersonTreeLock) {
        printf("Update Violation.");
        exit(0);
    }
    AddPersonTree(PDMCSchema , W , PersonTreeCompare);
}

void InitPeopleDB(PDMCSchema)
struct PeopleDBStruct *PDMCSchema;
{
    PDMCSchema->PersonStore = 0;
    SetUpPersonTree(PDMCSchema);
```

```
    PDMCSchema->PersonTreeLock = 0;
}

/* === Section 8: query functions === */

void AllPeople(PDMCSchema, PDMCQStruct)
struct PeopleDBStruct *PDMCSchema;
struct AllPeopleStruct *PDMCQStruct;
{
    if (PDMCQStruct->First == 0)
        goto PDMCLabel2;
    PDMCQStruct->First = 0;
    PDMCQStruct->CutP = 0;
    PDMCSchema->PersonTreeLock = 1;
    if (InitPersonTree(PDMCSchema, &(PDMCQStruct->P), PDMCCompare1,
    PDMCQStruct))
        goto PDMCLabel3;
    PDMCSchema->PersonTreeLock = 0;
    goto PDMCLabel4;
PDMCLabel2:
    if (PDMCQStruct->CutP)
        goto PDMCLabel4;
    if (NextPersonTree(PDMCSchema, &(PDMCQStruct->P), PDMCCompare1,
    PDMCQStruct))
        goto PDMCLabel3;
    PDMCSchema->PersonTreeLock = 0;
    goto PDMCLabel4;
PDMCLabel3:
    PDMCQStruct->Result = 1;
    return;
PDMCLabel4:
    PDMCQStruct->Result = 0;
    return;
}

void ManWithName(PDMCSchema, PDMCQStruct)
struct PeopleDBStruct *PDMCSchema;
struct ManWithNameStruct *PDMCQStruct;
{
    if (PDMCQStruct->First == 0)
        goto PDMCLabel6;
```

```
    PDMCQStruct->First = 0;
    PDMCQStruct->CutM = 0;
    if (InitPersonTree(PDMCSchema, &(PDMCQStruct->M), PDMCCompare2,
    PDMCQStruct))
        goto PDMCLabel7;
    goto PDMCLabel6;
PDMCLabel7:
    PDMCQStruct->Result = 1;
    return;
PDMCLabel6:
    PDMCQStruct->Result = 0;
    return;
}
```