RDM Reference Manual

Grant E. Weddell

Department of Computer Science
University of Waterloo
Waterloo, Canada, N2L 3G1

Research Report CS-89-41

RDM Reference Manual

Grant E. Weddell

Research Report CS-89-41
September 1989

ABSTRACT

The *Resident Database Manager* (RDM) is a software tool set for applications that manipulate memory-resident databases which is currently under development in the multimedia laboratory of the University of Waterloo. At present, the tool set consists of two compilers. The first produces access and representation code in a language called PDM from source that specifies the organization of data, and the manner in which it is accessed and changed. The input source for this compiler is expressed in a language called LDM. PDM and LDM are acronyms for *Physical Data Model* and *Logical Data Model* respectively. Output from this first compiler can be input to a second compiler, together with applications written in an extended C language called C/DB. C/DB has additional language constructs that permit the direct use of data access specifications written in LDM. The result of this second compiler is *pure* C code that can then be compiled with a standard C compiler.

The first section begins with an overview of LDM, and then illustrates its use in implementing a reachability algorithm for directed graphs. Section 1 concludes with a summary of limitations existing on the present tool set. The remaining two sections define the LDM language and the extensions to C in C/DB respectively.

1. INTRODUCTION

1.1. Overview

Almost any software system will have components that access and update information residing in main-memory. For example, a language compiler has procedures to maintain the so-called *symbol table*, which is essentially a memory-resident database of parsed source code, intermediate code, and so on. This manual is about a software tool set that helps with the development of such components in a way similar to how Yacc, for example, helps with the development of other components responsible for parsing.

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada, Bell-Northern Research Ltd., and the University of Waterloo.

The tool set presently consists of two compilers, and is called RDM, an acronym for *Resident Database Manager*. Input to the first compiler is a list of specifications in an object-oriented database language called LDM, another acronym for *Logical Data Model*. Most specifications are an instance of one of four sublanguages that characterize four categories of information about a database. Their main features are as follows:

- A data definition sublanguage (DDL) is used to describe the logical organization of data. The DDL manifests a data model that generalizes the relational model in two ways. First, the notions of *relation* and *domain* are combined into a notion of *class* by introducing surrogate keys for tuples, and by allowing attributes to be tuple-valued. Second, classes can be organized in a generalization taxonomy, whereby more specialized classes will automatically inherit attributes of more general classes. The taxonomy is established by declaring any number of immediate superclasses for each class (LDM supports so-called *multiple inheritance*).
- A data manipulation sublanguage (DML) is used to describe how the data is used. Data access requests are expressed in the query language component of the DML, while data update requests are expressed in another transaction language component. The query language is a SQL-like language that has been generalized for access to classes. The transaction language allows the user to specify simple combinations of update operations on the database. One operation allows the user to change the *identity* of an object (effectively changing its type).
- A data statistics language (DSL) is used to specify statistical information about a database. Using the DSL, a user can supply estimates of the number of objects in a class, how often a query or transaction is invoked, the relative cost of space and time, and so on. The statistical information is used by the component of the compiler responsible for performance prediction.
- A storage definition language (SDL) can be used to override some of the decisions made by the compiler on internal encoding of data. In particular, a user can specify a selection of the indices to be maintained in order to support searching within the database, and a selection of storage managers for managing the space used by objects.

Output from the first compiler is access and representation code in a language called PDM (for *Physical Data Model*). PDM code can then be input to the second compiler along with applications written in an extended C language called C/DB. C/DB has additional language constructs that permit the direct use of data access specifications originally written in LDM. The result of this second compiler is *pure* C code that can then be compiled with a standard C compiler.

Any number of C/DB source files can access and update a common database by including the same PDM source file, and then by linking their object code files. Also, any number of different databases can be accessed by a single C/DB source file. Thus a rudimentary capability exists for so-called *separate compilation*; that is, an ability to simultaneously develop separate parts of a software system. A summary of overall dataflow for the tool set is given in Figure 1.

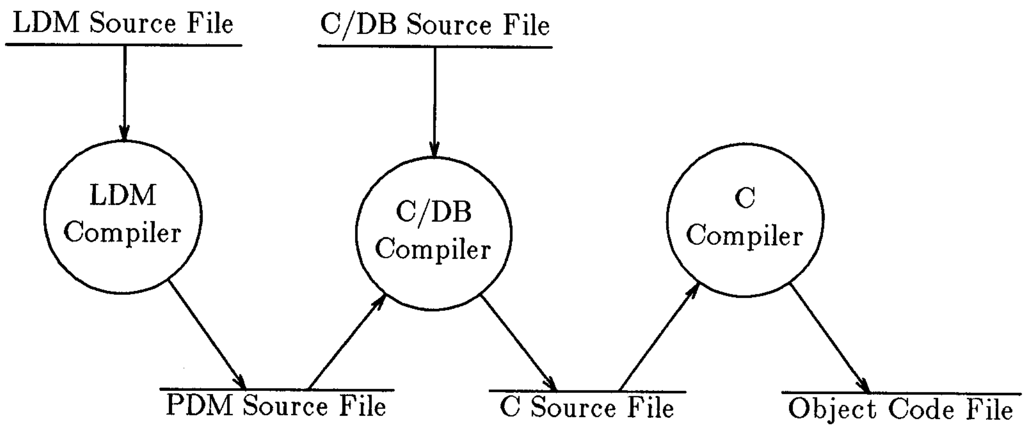

Figure 1. Basic Dataflow in RDM

1.2. Example

Assume a company is using RDM to develop a software system called *GraphLab*, which will consist of a number of programs for manipulating graphs. The specification for one of the programs, to be called *FindRV*, is as follows:

> **Functional Overview** — *FindRV* inputs a directed graph G together with a distinguished start vertex v. Output is a list of all vertices in G reachable from v.
>
> **Input/Output Format** — Each vertex is labelled with a unique identifier consisting of a non-blank sequence of up to twenty characters. The first line of input is the label $l(v)$ of the start vertex v. Each remaining line of input consists of a pair of identifiers "Id_1 Id_2" representing a new arc $u \rightarrow v$ in G where $l(u)=Id_1$ and $l(v)=Id_2$. No two lines have the same pair of identifiers. Output is a list of identifiers on separate lines.
>
> **Performance Requirements** — No limitations should exist on the size of G, beyond those imposed by the size of main memory. Running time should be at most $O(|A|\log|A|)$ where $|A|$ is the number arcs in G.

Also assume that architectural design for *FindRV* has resulted in the selection of a marking algorithm for finding the reachable vertices. The following is a description of the algorithm by Barstow [1]:

> **Algorithmic Details** — Mark the start vertex v as a boundary vertex and mark the rest of the vertices in G as unexplored. If there are any vertices marked as boundary vertices, select one, mark it as explored, and mark each of its unexplored successors as a boundary vertex. Repeat until there are no more boundary vertices. The set of vertices marked as explored is the desired set of reachable vertices.

The LDM specifications and C/DB source code that implement *FindRV* are given in Figures 2 and 3 respectively. In the former case, keywords are given in a boldface type. Also, all lines have been numbered to help with their referral in the following discussion.

Almost all the LDM specifications fall in one of the four categories of information outlined above. The only exception is the **schema** statement in line 1, which serves to name the specifications that follow. (There is further significance to the statement that will become clear during our discussion of the C/DB source.) The DDL source, lines 5 to 16, indicates that vertices and arcs in G are represented as instances of the classes Vertex and Arc respectively. Each vertex will have two property values encoding its label and mark (the latter indicating its *explored* status), and each arc will also have two property values encoding its source and destination vertices. The constraints (lines 6 and 13) imply that each vertex has a unique label, and that no two arcs have the same source and destination vertices.

```
1.   schema FindRV
2.
3.   % DDL Specification
4.
5.   class Vertex properties Label, Mark
6.   constraints Id determined by Label
7.
8.   property Label on String maxlen 20
9.
10.  class Mark
11.
12.  class Arc properties FromVertex, ToVertex
13.  constraints Id determined by FromVertex, ToVertex
14.
15.  property FromVertex on Vertex
16.  property ToVertex on Vertex
17.
18.  % DML Specification
19.
20.  query VerticesWithMark given M from Mark
21.  select V from Vertex where V.Mark = M
22.
23.  query VertexWithMark given M from Mark
24.  select one V from Vertex where V.Mark = M
25.
26.  query VertexWithLabel given L from Label
27.  select one V from Vertex where V.Label = L
28.
29.  query ConnectedVertices
30.  given VFrom, M from Vertex, Mark
31.  select VTo from Vertex
32.  where VTo.Mark = M and
33.          Arc {VFrom as FromVertex, VTo as ToVertex}
34.
35.  transaction NewMark
36.  declare M from Mark
37.  insert M
38.  return M
39.
40.  transaction ChgMark given V, M from Vertex, Mark
41.  V.Mark := M
42.
43.  transaction NewVertex given L, M from Label, Mark
44.  declare V from Vertex
45.  insert V (V.Label := L; V.Mark := M)
46.  return V
47.
48.  transaction NewArc
49.  given VFrom, VTo from Vertex, Vertex
50.  declare A from Arc
51.  insert A (A.FromVertex := VFrom; A.ToVertex := VTo)
52.
53.  % DSL Specification
54.
55.  size Vertex 100
56.  size Mark 3
57.  size Arc 200
58.
59.  % SDL Specification
60.
61.  index VertexList on Vertex
62.  of type distributed list on Mark
63.
64.  index VertexTree on Vertex
65.  of type binary tree ordered by Label asc
66.
67.  index ArcList on Arc
68.  of type distributed list on FromVertex
69.
70.  store VertexStore of type dynamic storing Vertex
71.  store ArcStore of type dynamic storing Arc
72.  store MarkStore of type static 3 storing Mark
```

Figure 2. LDM Source for *FindRV*

The DML source lists specifications for four queries and four transactions, which altogether constitute the database access and update requirements of *FindRV.* Note that each query and transaction is named. An outline of their definitions follows.

VerticesWithMark
(lines 20 and 21) The first query accepts an instance of the Mark class as input, and returns all vertices with this instance as the value of their Mark property.

VertexWithMark
(lines 23 and 24) This query also accepts an instance the Mark class as input, but in this case returns an arbitrarily chosen vertex satisfying the same constraint on the value of its Mark property (if one exists).

VertexWithLabel
(lines 26 and 27) The query returns a vertex with a given label value if such a vertex exists. Note that no more than one such vertex ever exists since Label is a key property of the class Vertex — as specified in the data definition.

ConnectedVertices
(lines 29 to 33) The last query accepts two things as input: a source vertex VFrom, and an instance M of the Mark class. The query returns all vertices VTo that satisfy two constraints. First, each must have M as the value of its Mark property. And second, there must exist an arc with VFrom as the value of its FromVertex property and VTo as the value of its ToVertex property.

NewMark
(lines 35 to 38) The first of the transactions creates and returns a new instance of the Mark class.

ChgMark
(lines 40 and 41) Transaction ChgMark changes the value of the Mark property for an input vertex to an input mark.

NewVertex
(lines 43 to 46) Transaction NewVertex creates a new vertex object, initializes the values of its Mark and Label properties, and then returns the new vertex.

NewArc
(lines 48 to 51) Transaction NewArc creates a new arc object and initializes its property values to the input vertices.

The DSL source (lines 55 to 57) are estimates of the expected number of instances of each class for a typical invocation of *FindRV.* This information is of particular use when the compiler invokes the query optimizer on the query ConnectedVertices. At some point in optimization, a choice must be make between an evaluation strategy that involves scanning all vertices with a given mark value, and an evaluation strategy that involves scanning all vertices connected by an arc to another vertex. The statistics indirectly establish that the latter strategy is to be preferred, since 200/100 vertices will be estimated to qualify in comparison to 100/3 vertices with the former strategy.

The SDL source (lines 61 to 72) specifies a selection of three indices and three storage managers. The first index on the Vertex class can be used for finding all vertices, or for finding all vertices with a given mark as the value of their Mark property. The second index, also on the Vertex class, can be used like the first for finding all vertices, or for finding a vertex with a given string as the value of its Label property. The third index on the class Arc can be used for finding all arcs, or for finding all arcs with a given vertex as the value of their FromVertex property. Note that any number of indices can be declared for a class. In this case, two are specified for the Vertex class, one for the Arc class and none for the Mark class.

The first two storage managers manage storage for vertex and arc objects respectively. Since they are declared to be dynamic, no limits will exist on the number of such objects, beyond those implied by the amount of main memory. The third storage manager is declared to be static with enough room for at most three mark objects — exactly the number needed by the algorithm for computing reachable vertices. One advantage of static storage for a class is that it permits a more compact encoding of values for properties defined on the class. In this case, encoding values for the Mark property of vertex objects will require at most two bits.

The *FindRV* program is implemented as a single **main** function in Figure 3. The schema statement in line 3 essentially serves as a place holder for the global type and data declarations chosen by the LDM compiler as the means of encoding a database. Any number of different databases can be manipulated by including a schema statement for each. Also, any number of C/DB source files can access the same database by prefixing the keyword **extern** to the keyword **schema** in all but one of the files. (An **extern schema** is eventually replaced by global type declarations only.)

Lines 7 and 8 in the function body declare a number of variables for referring to instances of the Vertex and Mark classes. Line 9 declares three variables for referring to the built-in String class. This follows since the property name following the **prop** keyword (i.e. *Label*) is string-valued.

Lines 11 to 13 create three instances of the Mark class, and bind values for their surrogate keys to the three mark variables. The remainder of the body consists of three parts: lines 17 to 29 for inputing the graph, lines 33 to 37 for computing reachable vertices according to the above algorithm, and line 41 for producing a list of the labels of all reachable vertices. We conclude our discussion of the example with a few comments that should suffice to clarify the C/DB source.

- An invocation of an LDM transaction that returns a value occurs in lines 11, 12, 13, 18, 23 and 26. An invocation of an LDM transaction that does not return a value occurs in lines 28, 35 and 36. The values returned are the surrogate keys for newly created objects.
- An invocation of an LDM query that returns at most a single result occurs in lines 22, 25 and 33 with new forms of C **if** and **while** statements. A new form of C **for** statement is used to invoke queries returning a set of values in lines 36 and 41.
- A new @ operator is used in line 41 as the means of property value access for property variables. In this case, the expression V@Label denotes the value of the Label property for the object having V bound to its surrogate key value.

1.3. Limitations

There are several limitations with the data model, and with the capabilities of the present version of the C/DB compiler. With respect to the data model, LDM presently assumes complete knowledge of the database. For example, all values for object properties are assumed to be known — no *null-unknown* values are permitted.

```
#include <stdio.h>
#include <string.h>
schema FindRV;

main()
{
  prop Vertex VStart, VFrom, VTo, V;
  prop Mark Unexplored, Boundary, Explored;
  prop Label VStartLabel, VFromLabel, VToLabel;

  Unexplored = NewMark();
  Boundary = NewMark();
  Explored = NewMark();

  /* input the graph */

  scanf("%s", VStartLabel);
  VStart = NewVertex(VStartLabel, Boundary);

  while (scanf("%s %s", VFromLabel, VToLabel) != EOF)
  {
    if V in VertexWithLabel(VFromLabel) VFrom = V;
    else VFrom = NewVertex(VFromLabel, Unexplored);

    if V in VertexWithLabel(VToLabel) VTo = V;
    else VTo = NewVertex(VToLabel, Unexplored);

    NewArc(VFrom, VTo);
  }

  /* find all reachable vertices */

  while V in VertexWithMark(Boundary)
  {
    ChgMark(V, Explored);
    for VTo in ConnectVertices(V, Unexplored) ChgMark(VTo, Boundary);
  }

  /* print the reachable vertices */

  for V in VerticesWithMark(Explored) printf("%s\n", V@Label);
}
```

Figure 3. C/DB Source for *FindRV*

Both compilers assume DML specifications include only *trusted transactions.* This means that neither compiler will generate code for consistency checking of either inherent or explicit constraints. (However, it is still very worthwhile for a user to declare constraints in the DDL. The constraints are used extensively by the query optimizer.) The tool set does not at present have any built-in support for managing concurrency. It is incumbent on the user to manage concurrent database access by multiple processes.

The current C/DB compiler supports only the **list** and **distributed list** index types, and only the **dynamic** store type. As a consequence, any query with an **order by** clause is currently not supported. In the full LDM transaction language, the identity of an object can be changed with an assignment of the form

<VarName>.**Id** := <Term>

This form of assignment is also not currently supported by the existing C/DB compiler.

2. THE LDM LANGUAGE

This part of the manual is an informal definition of LDM. We begin with the data definition language (DDL), and then define the query and transaction languages which comprise the data manipulation language (DML). The final two parts of this section define the data statistics language (DSL) available for expressing statistical estimates and cost model parameters for a database, and the storage definition language (SDL) for specifying a selection of indices and storage managers for data objects.

Throughout this section, examples will refer to a hypothetical software system that manages information about students, teachers and courses at some university. An *enterprise view* of the relevant data is illustrated by the entity-relationship diagram in Figure 4. Two features of the diagram are worth noting. First, the diagram has *existence constraints* for Course and GradStudent objects with respect to TaughtBy and Supervisor relationships respectively. This implies in the former case, for example, that each course object is always *TaughtBy related* to one and only one Teacher object. And second, the diagram suggests in several places that entity types can be declared as subtypes of other entity types (by using *Isa* triangles). For example, the Isa link between Student and GradStudent implies that some Student objects may also be GradStudent objects. The Isa link between Person, Student and Teacher implies three things: that some Person objects may also be Student objects, that some Person objects may also be Teacher objects, and that no object is just a Person. This latter condition is a consequence of there existing more than one incoming arc to the Isa link. If this is not desired, then two separate Isa links can be used.

Language syntax will be specified using BNF, with the following additional conventions: square brackets "[...]" are used to indicate optional arguments, braces "{...,<marker>}" to indicate options that may be repeated one or more times, separated by <marker> (either a blank, comma or semicolon), and keywords are indicated in a boldface type.

2.1. LDM Program Format

All LDM programs have the following form

<LDMProgram> ::=
  schema <SchemaName> {<DDLSpec>," "} {<NonDDLSpec>," "}

<DDLSpec> ::= <ClassDefn> | <PropertyDefn>

<NonDDLSpec> ::= <DMLSpec> | <DSLSpec> | <SDLSpec>

<DMLSpec> ::= <QueryDefn> | <TransactionDefn>

where <SchemaName> is an identifier naming the schema. Note that all DDL specifications must precede any other specifications. The LDM source for the University database begins with

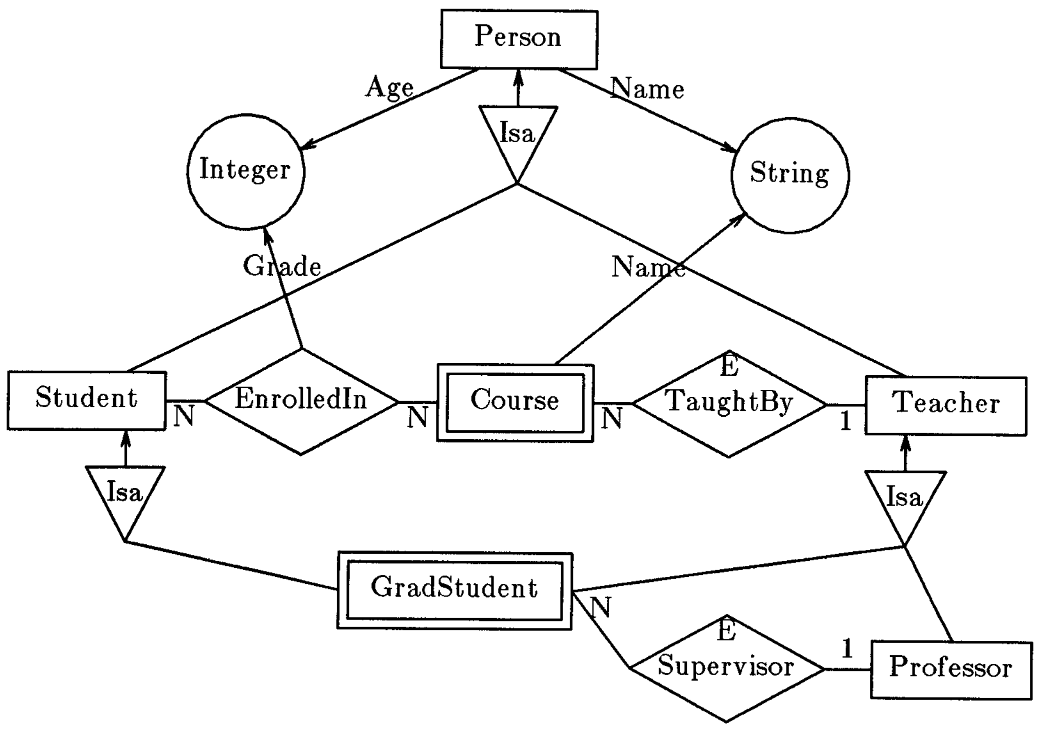

Figure 4. ER Diagram of a University Schema

schema University
. . .

2.2. Data Definition

Data is described by specifying a number of *classes* and *properties.*

```
<ClassDefn> ::=
    class <ClassName> [isa {<ClassName>,","}]
    [properties {<PropertyName>,","}]
    [constraints {<Constraint>," "}]

<PropertyDefn> ::=
    property <PropertyName> on <ClassName> |
    property <PropertyName> on String maxlen <Integer> |
    property <PropertyName> on Integer range <Integer> to <Integer> |
    property <PropertyName> on Real |
    property <PropertyName> on DoubleReal
```

Note that properties are declared separately in LDM, and that all definitions may occur in any order. For a given schema, no two classes can have the same name, and no two properties can have the same name. As a convenience, a property definition of the form

property C **on** C

is assumed for each class C, whenever such a property is not already declared by the user.

There are two kinds of constraints that may be specified when defining a class: path functional dependencies (PFDs), and covers.

```
<Constraint> ::=
        <PathFunction> determined by {<PathFunction>,","} |
        cover by {<ClassName>,","}
```

The meaning of PFD and cover constraints will be explained by example below. For the university application, class and property definitions corresponding to the above E-R diagram are as follows.

```
class Person
properties Name, Age
constraints
        Id determined by Name
        cover by Student, Teacher

class Student isa Person

class Teacher isa Person
constraints cover by GradStudent, Professor

class Professor isa Teacher

class GradStudent isa Student, Teacher
properties Supervisor

class Course
properties Name, TaughtBy
constraints Id determined by Name

class EnrolledIn
properties Student, Course, Grade
constraints Id determined by Student, Course

property Name on String maxlen 20
property Age on Integer range 16 to 75
property Supervisor on Professor
property TaughtBy on Teacher
```

The PFD and cover constraints for the Person class assert that no two Person objects have the same value for their Name property, and that each Person object must also be a Student or Teacher object (or both). For a more thorough discussion of PFD constraints, see [3,4].

2.3. Data Manipulation — The Query Language

We start by giving the full grammar for the query language, and then explain its constructs by giving example queries on the University schema. In general, syntax for both queries and transactions has been chosen so as to resemble the SQL query language wherever possible.

A query has the general form

```
<QueryDefn> ::=
      query <QueryName> [given <VarDecl>]
      select [one] [<VarDecl>]
      [where <Predicate>]
      [order by {<OrderItem>,","}]
      [precomputed]

<OrderItem> ::= <Term> asc | <Term> desc

<VarDecl> ::= {<VarName>,","} from {<PropertyName>,","}
```

Note that each <PropertyName> must associate one-to-one with each <VarName>. A declaration of the form

V1, V2 **from** P1, P2

defines two variables V1 and V2 that may have any values that are also legal for properties P1 and P2 respectively. Terms and predicates are defined as follows.

```
<Term> ::=
      <Integer> |
      <Real> |
      "<String>" |
      ["-"] <VarName> ["." <PathFunction>] |
      <Term> <ArithmeticOperator> <Term> |
      (<Term>)

<PathFunction> ::= Id | {<PropertyName>,"."}

<ArithmeticOperator> ::= + | - | * | / | mod

<Predicate> ::=
      <Term> <ComparisonOperator> <Term> |
      <VarName> has <MaxOrMin> <PathFunction> [where <Predicate>] |
      <ClassName> "{" {<Term> as <PathFunction>,","} "}" |
      not <Predicate> |
      exist <VarDecl> [where <Predicate>] |
      for all <VarDecl> <Predicate> |
      <Predicate> <LogicalOperator> <Predicate> |
      (<Predicate>)
```

<ComparisonOperator> ::= = | < | <= | >= | > | <>

<MaxOrMin> ::= **max** | **min**

<LogicalOperator> ::= **implies** | **and** | **or**

The standard precedence for operators in terms and predicates is assumed. For term operators this order of precedence is: binary addition "+" and subtraction "-" (weakest binding); multiplication "*", division "/" and modulus **mod**; unary minus "-"; and finally property value access "." (strongest binding). For predicate operators the order is: existential quantification **exist** and universal quantification **for all** (weakest binding); implication **implies**; disjunction **or**; conjunction **and**; negation **not**; and finally the arithmetic comparison operators and the two special forms for expressing maximum and minimum value criteria **has max** and **has min**, and for expressing *atomic predicate* conditions "<ClassName>{ · · · }" (strongest binding). Here are some example queries on the University schema.

Ex 1. (getting all objects in a class extension) Retrieve all people objects.

```
query People
select P
from Person
```

Ex 2. (specifying conditions and query parameters) Retrieve all student objects older than 30 that are enrolled in a given course.

```
query OldStudentsInCourse given C from Course
select S from Student
where S.Age > 30 and EnrolledIn {S as Student, C as Course}
```

This query can also be specified as follows.

```
query OldStudentsInCourse given C from Course
select S from Student
where S.Age > 30 and (
        exist E from EnrolledIn
        where E.Student = S and E.Course = C)
```

Ex 3. (subqueries) Retrieve all integers that occur as the age value of some student.

```
query StudentAges
select A from Age
where exist S from Student where S.Age = A
```

Ex 4. (use of path functions and nondeterminism) Retrieve a graduate student object that is supervised by some professor with a given name.

```
query GradWithSupervisorName given N from Name
select one G from GradStudent where G.Supervisor.Name = N
```

Ex 5. (sorted retrieval) Retrieve all graduate student objects in major order by their supervisor's name, and minor order by their own name.

```
query Graduates
select G from GradStudent
order by G.Supervisor.Name asc, G.Name asc
```

Ex 6. (use of max and min) Retrieve all undergraduate objects who received the highest grade in some course.

```
query SmartUndergrads
select S from Student
where not GradStudent {S as Id} and (
        exist C, E from Course, EnrolledIn
        where E.Student = S and
                E has max Grade where E.Course = C)
```

The query can also be specified in either of the following two ways.

```
query SmartUndergrads
select S from Student
where not (exist G from GradStudent where G = S) and (
        exist C, E from Course, EnrolledIn
        where E.Student = S and
                E.Course = C and (
                for all E1 from EnrolledIn (
                E1.Course = C implies E1.Grade <= E.Grade)))

query SmartUndergrads
select S from Student
where not (exist G from GradStudent where G = S) and (
        exist C, E from Course, EnrolledIn
        where E.Student = S and
                E.Course = C and not (
                exist E1 from EnrolledIn
                where E1.Course = C and E1.Grade > E.Grade))
```

Ex 7. (complex queries) Retrieve an undergraduate object who received a grade in a course higher than any graduate student also enrolled in the course.

```
query PossibleGrad
select one S from Student
where not GradStudent {S as Id} and (
        exist E1 from EnrolledIn
        where E1.Course = S and (
                for all E2 from EnrolledIn
                (E2.Course = E1.Course and GradStudent {E2.Student as Id})
                implies E2.Grade < E1.Grade))
```

Ex 8. (forcing projections) Retrieve and temporarily store all student objects enrolled in a course taught by a teacher with a given name.

```
query StudentsTaughtByTeacher given N from Name
select S from Students
where EnrolledIn {S as Student, N as Course.TaughtBy.Name}
precomputed
```

When specified, a **precomputed** clause will force the results of a query to be precomputed and temporarily stored before any action on each result is permitted. This may be necessary if the action intended for a result can invoke a transaction that interferes with a query evaluation strategy (referred to as the *Halloween problem*). An example with the above is a transaction that deletes each student object in the returned result.

2.4. Data Manipulation — The Transaction Language

Again, we start by giving the full grammar for the transaction language, and then illustrate its use with example transactions for the University schema.

```
<TransactionDefn> ::=
    transaction <TransactionName> [given <VarDecl>]
    [declare <VarDecl>]
    {<Statement>,";"}
    [return <Term>]

<Statement> ::=
    insert {<VarName>,","} ["(" {<InitStatement>,";"} ")"] |
    delete {<VarName>,","} |
    <Term> ":=" <Term>

<InitStatement> ::= <VarName> "." <PropertyName> ":=" <Term>
```

Here are some example transactions for the University schema.

Ex 9. (updating property values) Change the teacher assigned to a given course object to another given teacher.

```
transaction AssignTeacher given T, C from Teacher, Course
C.TaughtBy := T
```

Ex 10. (creating new objects) Enroll a given student object in given course object.

```
transaction EnrollStudent given S, C from Student, Course
declare E from EnrolledIn
insert E (E.Student := S; E.Course := C)
```

Ex 11. (creating and returning objects) Create and return a new course object with a given name and teacher.

```
transaction NewCourse given T, N from Teacher, Name
declare C from Course
insert C (C.TaughtBy := T; C.Name := N)
return C
```

Ex 12. (deleting an object from the database) Delete a given student object from the database.

> **transaction** RemStudent **given** S **from** Student
> **delete** S

Ex 13. (changing the type of an object) Enter a given student object in graduate school, assigning a given professor object as an initial supervisor.

> **transaction** BecomeGrad **given** S, P **from** Student, Professor
> **declare** G **from** GradStudent
> **insert** G (G.Name := S.Name; G.Age := S.Age; G.Supervisor := P);
> G.**Id** := S
> **return** P

Note in this last example that the assignment statement "G.**Id** := S" will cause the identity of the newly created G object to be changed to the identity of the S object. As a consequence, the S object is deleted from the database, and any previously existing references to S will now be to G.

2.5. Data Statistics

At present, a single form of statistic can be specified for classes.

> <DSLSpec> ::= **size** <ClassName> <Integer>

A size statistic for a class corresponds to an estimate of the expected number of objects in the class that are not also in any subclasses. For example, assume size estimates for the University database have been specified as follows.

> **size** Student 500
> **size** GradStudent 100
> **size** Course 200
> **size** EnrolledIn 4000
> **size** Professor 50

The statistics imply that one can expect a total of 600 student objects: 100 that are GradStudent objects, and 500 that are not. Note that size estimates for classes having one or more cover constraints are therefore nonsensical.

2.6. Storage Definition — Store Management

In LDM, each class must be associated with a store manager from which space is allocated when objects are created for the class, and to which space is released when objects that were created for that class are deleted. The user is currently responsible for declaring store managers using the following language.

> <SDLSpec> ::=
>   **store** <StoreName> **of type** <StoreType>
>   **storing** {<ClassName>,","}
>
> <StoreType> ::= **dynamic** | **static** <Integer>

There are two types of store managers that may declared: dynamic store, and static store. A class associated with a dynamic store manager will have no limit on the number of objects that may be created, beyond limits imposed by the available memory. This is not true of static store managers; the total number of objects that may be created for all associated classes is limited by the static store manager's size. However, the advantage in this case is that encoding property values for properties defined on any of the associated classes will usually require much less space. To ensure such an encoding is possible, a constraint on a static store specification is that the set of associated classes must satisfy the condition that all subclasses of any element of the set are also in the set.

Storage management for the University database might be specified as follows.

store PersonStore **of type dynamic storing** Student, GradStudent
store ProfStore **of type static** 60 **storing** Professor
store EnrollStore **of type dynamic storing** EnrolledIn
store CourseStore **of type dynamic storing** Course

Note that store for all Student and GradStudent objects are managed in a common pool. Also note that the specification of ProfStore implies that no more than 60 professor objects will exist at one time. This permits a more compact encoding of Supervisor property values: only 6 bits of store are required for each, in comparison to the number of bits necessary to encode pointer values.

Free space managed by store managers satisfies two conditions. First, all blocks of memory associated with a particular manager are the same size. This implies internal fragmentation (or memory loss) if more that one class is associated with the manager, since smaller objects are still allocated enough space for the largest possible object. And second, space once allocated to a given store manager becomes unavailable for use by any other store manager. This can cause external fragmentation, for example, in a case where a large number of objects for one class are created, then deleted, and then a large number of objects for another class associated with a different store manager are created. The specification of store managers therefore requires balancing possible internal and external memory fragmentation, and the need for data compaction.

2.7. Storage Definition — Indices

Access to class extensions is achieved by declaring a number of indices, which at present is also the responsibility of the user. Each index is associated with a unique class and is declared to be of a particular type. For example, in the University database, a linked list of person objects can be declared with the form

index PersonList **on** Person **of type list**

The index, called PersonList, establishes the existence of a doubly linked list of all person objects at run-time. Note that the list will include all objects created in any subclasses of Person, such as Student objects, Teacher objects, and so on.

Any number of indices (including none) may be declared for a given class. The language for specifying an index is as follows.

```
<SDLSpec> ::=
     index <IndexName> on <ClassName>
     of type <IndexType>

<IndexType> ::=
     list |
     array <Integer> ordered by {<SearchCond>,","} |
     binary tree ordered by {<SearchCond>,","} |
     distributed list on <PathFunction> |
     distributed binary tree on <PathFunction> ordered by {<SearchCond>,","}

<SearchCond> ::= <PathFunction> asc | <PathFunction> desc | <ClassName>
```

As the language suggests, there are currently five types of index that may be declared. The list and binary tree types result in two additional pointer values for each object, which encode a doubly linked list in the first case, and a tree in the second. An array index is a static index corresponding to a FORTRAN-like fixed sized array of object identifiers. In this case, a binary search is used to find entries that satisfy given search conditions. The distributed list and distributed tree indices require the user to specify an additional path function, which must also satisfy the constraint that its *range* class is user-defined. The two distributed types behave like their undistributed counterparts when *distributed* among the objects in the range class of this path function.

There are three kinds of ordered search conditions that may be specified for an array, a binary tree or a distributed binary tree index type. The first has the form "<PathFunction> **asc**", and represents an ordering in which index entries occur in ascending order of their value for <PathFunction>. An ascending order for integer values, real values and string values has the obvious interpretation. An ascending order for all other kinds of objects is defined internally (and therefore legal), but is not meaningful to a user. The second kind of ordered search condition has the form "<PathFunction> **desc**", and represents an ordering in which index entries occur in descending order of their value for <PathFunction>. The third kind of ordered search condition has the form "<ClassName>", and is referred to as a *subclass sort*. A subclass sort on class C is two-valued: zero if an object in the index is not also in class C, and one otherwise.

Examples of indices that might be declared for the University database are as follows.

```
index PersonTree on Person of type binary tree
ordered by Student, GradStudent, Supervisor.Name asc

index TeacherTree on Teacher of type binary tree
ordered by Name asc

index EDistList1 on EnrolledIn of type distributed list on Course
index EDistList2 on EnrolledIn of type distributed list on Student

index CDistList on Course of type distributed list on TaughtBy
```

Five indices are declared, of which two are binary trees and three are distributed lists. The first index, called PersonTree, illustrates the use of subclass sort conditions. For example, the first subclass sort on Student is zero-valued for Person objects that are not also Student objects, and one-valued otherwise. The query optimizer will choose index PersonTree as the best possible means of evaluating any of the following four queries.

> **query** Q1 **select** P **from** Person
> **query** Q2 **select** S **from** Student
> **query** Q3 **select** G **from** GradStudent
> **query** Q4 **select** G **from** GradStudent **where** G.Supervisor.Name = "Fred"

For a more complete discussion of memory-resident indices, see [5].

3. THE C/DB LANGUAGE

In order to access a database, the C language has been extended to include a number of additional constructs with the following purposes:

- declaring access to a schema,
- declaring object-valued variables,
- accessing the value of an object property, and
- invoking queries.

This extended language is called C/DB. Note that no extensions to syntax were needed to support invoking LDM transactions. These eventually become separate C functions, and are invoked in the same way as any other C functions.

Our discussion of C/DB will center on defining the extensions to the C grammar given in [2]. The syntax notation we use adheres to the notation adopted in the reference (and therefore differs from the conventions used in the previous section). In particular, syntactic categories (non-terminals) are indicated by *italic* type, and keywords in **bold** type. An optional keyword is indicated by subscripting with *opt*. The necessary extensions are straightforward, and the reader is encouraged to reexamine the C/DB source in Figure 3 (in the first section) for examples of their use.

Access to an LDM schema is accomplished with the use of an additional form of *data-definition* in an *external-definition* for a *program.*

> *data-definition:*
>
> . . .
>
> **extern**$_{opt}$ **schema** *identifier* ;

Subsequent access to properties, classes, queries and transactions defined in the LDM schema with the name *identifier* is then enabled. The **extern** modifier may be used if more than one program accesses the same LDM schema.

Object-valued variables are declared with a new form of *type-specifier*.

> *type-specifier:*
>
> . . .
>
> **prop** *identifier*

The *identifier* must correspond to the name of an LDM property. Translation of this form of *type-specifier* by the C/DB compiler depends on the declaration of the LDM property itself. A property declared on a user defined class

property <PropertyName> **on** <ClassName>

translates to a variety of special forms (e.g. pointer types), which cannot be assumed by the user. A property declared on the built-in **String**, **Integer**, **Real** or **DoubleReal** classes translates in the obvious manner to other forms of *type-specifiers*, such as **short**, **int**, **long**, **float**, **double**, and (array of) **char**.

For variables referring to objects in a user defined class, an additional form of *primary* is available for accessing property values.

primary:
  . . .
  primary @ *identifier*

Note that *primary* must be an expression with a type "**prop** P" for which *identifier* is a legal property of the class on which P is defined.

New forms of **if**, **while** and **for** *statements* have been added to support the invocation of queries.

statement:
  . . .
  if *query-call statement*
  if *query-call statement* **else** *statement*
  while *query-call statement*
  for *query-call statement*

query-call:
  $identifier_1$
  $identifier_1$ (*expression-list*)
  identifier-list **in** $identifier_1$
  identifier-list **in** $identifier_1$ (*expression-list*)

identifier-list:
  $identifier_2$
  $identifier_2$, *identifier-list*

In a *query-call*, $identifier_1$ is the name of the query, *expression-list* a sequence of argument expressions that are bound in sequence to the **given** variables of the named query (if any), and *identifier-list* a sequence of $identifier_2$ which are bound in sequence to the **select** variables of the named query (if any).

The named query in **if** and **while** statements must have the form

query · · · **select one** · · ·

in which a single solution to the query is non-deterministically selected. If such a solution exists, then the first form of **if** binds each $identifier_2$ in *identifier-list* to the **select** values of the solution, and evaluates the argument *statement*. The second form of **if** operates similarly, except that the second argument *statement* is evaluated if no solution to the query is found. If such a solution exists in the case of a **while**, then each $identifier_2$ in *identifier-list* is bound to the **select** values of the solution, the argument *statement* is evaluated, and then this process is repeated.

The new form of **for** may take any query as an argument. In this case, the argument *statement* is evaluated for each query solution. The order in which solutions are considered will satisfy the **"order by"** clause of a query (if specified).

Finally, all forms of *query-call* must satisfy some typing conditions on *expression-list* and *identifier-list*. To illustrate, consider an LDM query of the form

query Q
given $V_{1,1}, \cdots, V_{1,m}$ **from** $P_{1,1}, \cdots, P_{1,m}$
select $V_{2,1}, \cdots, V_{2,n}$ **from** $P_{2,1}, \cdots, P_{2,n}$
where $\cdots$

together with a C/DB **for** statement of the form

for $V_{3,1}, \cdots, V_{3,n}$ **in** Q ($Exp_1, \cdots, Exp_m$) $\cdots$

Also assume the type of variable $V_{3,i}$ and expression Exp_j is "**prop** $P_{3,i}$" and "**prop** $P_{4,j}$" respectively. The typing conditions are as follows.

(a) The class on which property $P_{4,j}$ is defined must be a subclass of the class on which property $P_{1,j}$ is defined, $1 \leq j \leq m$.

(b) The class on which property $P_{2,i}$ is defined must be a subclass of the class on which property $P_{3,i}$ is defined, $1 \leq i \leq n$.

For built-in classes, such as **Integer**, the subclassing constraints correspond to assignment compatibility.

4. References

1. D. R. Barstow, *Knowledge Based Program Construction,* North Holland (1979).
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall (1978).
3. G. E. Weddell, Reasoning about functional dependencies generalized for semantic data models, Research Report CS-89-14, Department of Computer Science, University of Waterloo (1989).
4. G. E. Weddell, A theory of functional dependencies for object-oriented data models, *to be presented at First International Conference on Deductive and Object-Oriented Database,* (December 1989).
5. G. E. Weddell, Selection of indices to memory-resident entities for semantic data models, *to appear in IEEE Transactions on Knowledge and Data Engineering,* (June 1989).