19

Databases

19.1 All in all, we consider Figure E19.1c and Figure E19.1d most desirable. The first two figures are poor models. Here are some observations.

- Figure E19.1a fails to indicate that each *from-to* node pair can have many edges.
- Figure E19.1a may make it difficult to answer queries that specify an edge name. Figure E19.1b may make it difficult to answer queries that specify a node name.
- The symmetry in Figure E19.1b can be confusing; we had to arbitrarily break the symmetry with the *end1* and *end2* designation. Some implementations of Figure E19.1b may require that instances be entered twice or that an edge be searched through both qualifiers in order to find all pertinent instances.
- Figure E19.1b cannot represent the case where only one edge connects to a node. (Useful for an incomplete diagram.)
- Figure E19.1c and Figure E19.1d are better than the first two figures because they give node and edge equal stature. Nodes and edges seem equally important in the construction of directed graphs so they both should be recognized as classes.
- Figure E19.1c and Figure E19.1d can be extended to permit dangling edges and/or nodes by changing the “1” multiplicity to “0..1” multiplicity. This distinction can be important for software that must support partially completed diagrams.
- Figure E19.1c and Figure E19.1d capture more multiplicity constraints than the first two figures. Each edge has exactly one *from* node and one *to* node.
- An implementation of Figure E19.1d must assign the *end* qualifier an enumeration data type with values: *from* and *to*. Enforcing the enumeration may be awkward for some databases and languages.
- Figure E19.1d can most easily find all edges connected to a given node.

19.2

- Tables for Figure E19.1a. The underlying class model is a poor model.

Node table

nodeID	nodeName (ck1)

ToFrom table

toNodeID (references Node)	**toNodeID** (references Node)	edgeName

Figure A19.1 Tables for Figure E19.1a

- Tables for Figure E19.1b. The underlying class model is a poor model.

Edge table

edgeID	edgeName (ck1)

End1End2 table

edge1ID (references Edge)	**end1**	**edge2ID** (references Edge)	**end2**	nodeName

Figure A19.2 Tables for Figure E19.1b

- Tables for Figure E19.1c.

Edge table

edgeID	edgeName (ck1)	fromNodeID (references Node)	toNodeID (references Node)

Node table

nodeID	nodeName (ck1)

Figure A19.3 Tables for Figure E19.1c

- Tables for Figure E19.1d. *EdgeID* + *nodeID* is not a candidate key for the association table because an edge may connect a node with itself.

Edge table

edgeID	edgeName (ck1)

Node table

nodeID	nodeName (ck1)

Edge_Node table

edgeID (references Edge)	**end**	nodeID (references Node)

Figure A19.4 Tables for Figure E19.1d

19.3a. SQL commands to create database tables and indexes for Figure E19.1c.

```
CREATE TABLE Node
( node_ID   NUMBER(30)   CONSTRAINT nn_node1 NOT NULL,
  node_name VARCHAR2(50) CONSTRAINT nn_node2 NOT NULL,
CONSTRAINT pk_node PRIMARY KEY (node_ID),
CONSTRAINT uq_node1 UNIQUE (node_name));
CREATE SEQUENCE seq_node;

CREATE TABLE Edge
( edge_ID      NUMBER(30)   CONSTRAINT nn_edge1 NOT NULL,
  edge_name    VARCHAR2(50) CONSTRAINT nn_edge2 NOT NULL,
  from_node_ID NUMBER(30)   CONSTRAINT nn_edge3 NOT NULL,
  to_node_ID   NUMBER(30)   CONSTRAINT nn_edge4 NOT NULL,
  CONSTRAINT pk_edge PRIMARY KEY (edge_ID),
  CONSTRAINT uq_edge1 UNIQUE (edge_name));
CREATE SEQUENCE seq_edge;
CREATE INDEX index_edge1 ON Edge (from_node_ID);
CREATE INDEX index_edge2 ON Edge (to_node_ID);
ALTER TABLE Edge ADD CONSTRAINT fk_edge1
  FOREIGN KEY from_node_ID
  REFERENCES Node;
ALTER TABLE Edge ADD CONSTRAINT fk_edge2
  FOREIGN KEY to_node_ID
  REFERENCES Node;
```

b. SQL commands to create database tables and indexes for Figure E19.1d. We cascade deletes for *Edge* to *Edge_Node* because an *Edge* involves only two occurrences of *Edge_Node* so the effect would likely be anticipated. In contrast a *Node* could have a large number of *Edge_Node* occurrences, so we block propagation of deletes to avoid accidental side effects.

```
CREATE TABLE Node
( node_ID   NUMBER(30)   CONSTRAINT nn_node1 NOT NULL,
  node_name VARCHAR2(50) CONSTRAINT nn_node2 NOT NULL,
CONSTRAINT pk_node PRIMARY KEY (node_ID),
CONSTRAINT uq_node1 UNIQUE (node_name));
CREATE SEQUENCE seq_node;

CREATE TABLE Edge
( edge_ID   NUMBER(30)   CONSTRAINT nn_edge1 NOT NULL,
  edge_name VARCHAR2(50) CONSTRAINT nn_edge2 NOT NULL,
```

```
CONSTRAINT pk_edge PRIMARY KEY (edge_ID),
CONSTRAINT uq_edge1 UNIQUE (edge_name));
CREATE SEQUENCE seq_edge;

CREATE TABLE Edge_Node
( edge_ID NUMBER(30)  CONSTRAINT nn_edgenode1 NOT NULL,
  end     VARCHAR2(4) CONSTRAINT nn_edgenode2 NOT NULL,
  node_ID NUMBER(30)  CONSTRAINT nn_edgenode3 NOT NULL,
  CONSTRAINT pk_edgenode PRIMARY KEY (edge_ID,end));
CREATE INDEX index_edgenode1 ON Edge_Node (node_ID);
ALTER TABLE Edge_Node ADD CONSTRAINT fk_edgenode1
  FOREIGN KEY edge_ID
  REFERENCES Edge ON DELETE CASCADE;
ALTER TABLE Edge_Node ADD CONSTRAINT fk_edgenode2
  FOREIGN KEY node_ID
  REFERENCES Node;
ALTER TABLE Edge_Node ADD CONSTRAINT enum_edge_node1
  CHECK (end IN ('to', 'from'));
```

19.4 Populated tables for the directed graph in Figure E19.2 using the model of Figure E19.1c.

Node table

node_ID	node_name
1	n1
2	n2
3	n3
4	n4
5	n5

Edge table

edge_ID	edge_name	from_node_ID	to_node_ID
1	e1	n5	n4
2	e2	n3	n4
3	e3	n2	n3
4	e4	n2	n1
5	e5	n1	n5
6	e6	n5	n2

Figure A19.5 Populated database tables for Figure E19.1c

Populated tables for the directed graph in Figure E19.2 using the model of Figure E19.1d.

Edge table

edge_ID	edge_name
1	e1
2	e2
3	e3
4	e4
5	e5
6	e6

Node table

node_ID	node_name
1	n1
2	n2
3	n3
4	n4
5	n5

Edge_Node table

edge_ID	**end**	node_ID
e1	from	n5
e1	to	n4
e2	from	n3
e2	to	n4
e3	from	n2
e3	to	n3
e4	from	n2
e4	to	n1
e5	from	n1
e5	to	n5
e6	from	n5
e6	to	n2

Figure A19.6 Populated database tables for Figure E19.1d

19.5a. Given the name of an edge, determine the two nodes that it connects.

```
SELECT E.edge_name, EN.end, N.node_name
FROM Edge_Node EN, Node N, Edge E
WHERE E.edge_ID = EN.edge_ID AND
      N.node_ID = EN.node_ID AND
      E.edge_name = :anEdgeName;
```

b. Given the name of a node, determine all edges connected to or from it.

```
SELECT E.edge_name, EN.end, N.node_name
FROM Edge_Node EN, Node N, Edge E
WHERE E.edge_ID = EN.edge_ID AND
      N.node_ID = EN.node_ID AND
      N.node_name = :aNodeName;
```

c. Given a pair of nodes, determine the edges that directly connect them.

```
SELECT E.edge_name
FROM Edge_Node EN1, Edge_Node EN2, Edge E
WHERE ((EN1.end = 'from' AND EN2.end = 'to') OR
      (EN1.end = 'to' AND EN2.end = 'from')) AND
```

```
        EN1.edge_ID = EN2.edge_ID AND
        EN1.edge_ID = E.edge_ID AND
        EN1.node_ID = :aNode1  AND
        EN2.node_ID = :aNode2;
```

d. Given a node, determine the nodes connected through transitive closure. Pseudocode is required because SQL provides no intrinsic support for transitive closure.

```
NodeTransitiveClosure (startNode) RETURNS SET OF nodeID;
   visitedNodes := {};
   FindNextNodes (startNode, visitedNodes);
   RETURN (visitedNodes);
END OF NodeTransitiveClosure


FindNextNodes (startNode, visitedNodes);
   tempSet := FindDbNodes (startNode);

   /* do not revisit a node */
   FOR EACH aNode IN tempSet DO
       IF aNode IS IN visitedNodes THEN
          tempSet -= aNode;
       ENDIF
   END FOR EACH


   /* add remaining nodes to visited set */
   FOR EACH aNode IN tempSet DO
       visitedNodes += aNode;
   END FOR EACH


   /* recurse for newly discovered nodes */
   FOR EACH aNode IN tempSet DO
       FindNextNodes (aNode, visitedNodes);
   END FOR EACH
END OF FindNextNodes


FindDbNodes (startNode) RETURNS SET OF node_ID;
   /* The following code shows a SQL query returning a  */
   /* set. Most SQL programming language interfaces do  */
   /* not permit return of a set and would require      */
   /* looping through a cursor to accumulate the answer. */
   SELECT EN2.NodeID INTO nodeSet
   FROM EdgeNode EN1, EdgeNode EN2
   WHERE EN1.nodeID = :startNode AND
        EN1.edgeID = EN2.edgeID AND
        EN1.end = 'from' AND EN2.end = 'to';
```

```
    RETURN (nodeSet);
END OF FindDbNodes
```

19.6 Figure A19.7 shows the tables.

Term table

termID	term Type

Expression table

expressionID (references Term)	binary Operator	firstOperand (references Term)	secondOperand (references Term)

Variable table

variableID (references Term)	name (ck1)

Constant table

constantID (references Term)	value

Figure A19.7 Tables for Figure E19.3

19.7 SQL commands for Figure E19.3. We arbitrarily store values for constants as strings. The strings would be converted to numbers before evaluating expressions.

```
CREATE TABLE Term
( term_ID     NUMBER(30)   CONSTRAINT nn_term1 NOT NULL,
  term_type   VARCHAR2(10) CONSTRAINT nn_term2 NOT NULL,
CONSTRAINT pk_term PRIMARY KEY (term_ID));
CREATE SEQUENCE seq_term;
ALTER TABLE Term ADD CONSTRAINT enum_term1
CHECK (term_type IN ('Expression','Variable','Constant'));

CREATE TABLE Expression
( expression_ID   NUMBER(30)  CONSTRAINT nn_exp1 NOT NULL,
  binary_operator VARCHAR2(10) CONSTRAINT nn_exp2 NOT NULL,
  first_operand   NUMBER(30)  CONSTRAINT nn_exp3 NOT NULL,
  second_operand  NUMBER(30)  CONSTRAINT nn_exp4 NOT NULL,
CONSTRAINT pk_exp PRIMARY KEY (expression_ID));
CREATE INDEX index_exp1 ON Expression (first_operand);
CREATE INDEX index_exp2 ON Expression (second_operand);
ALTER TABLE Expression ADD CONSTRAINT fk_exp1
  FOREIGN KEY expression_ID
  REFERENCES Term ON DELETE CASCADE;
ALTER TABLE Expression ADD CONSTRAINT fk_exp2
  FOREIGN KEY first_operand
  REFERENCES Term;
```

```
ALTER TABLE Expression ADD CONSTRAINT fk_exp3
  FOREIGN KEY second_operand
  REFERENCES Term;

CREATE TABLE Variable
( variable_ID   NUMBER(30)   CONSTRAINT nn_var1 NOT NULL,
  variable_name VARCHAR2(50) CONSTRAINT nn_var2 NOT NULL,
CONSTRAINT pk_var PRIMARY KEY (variable_ID),
CONSTRAINT uq_var1 UNIQUE (variable_name));
ALTER TABLE Variable ADD CONSTRAINT fk_var1
  FOREIGN KEY variable_ID
  REFERENCES Term ON DELETE CASCADE;

CREATE TABLE Constant
( constant_ID  NUMBER(30)   CONSTRAINT nn_const1 NOT NULL,
  const_value  VARCHAR2(50) CONSTRAINT nn_const2 NOT NULL,
CONSTRAINT pk_const PRIMARY KEY (constant_ID));
ALTER TABLE Constant ADD CONSTRAINT fk_const1
  FOREIGN KEY constant_ID
  REFERENCES Term ON DELETE CASCADE;
```

19.8 Populated tables for Figure E19.3

Term table

term_ID	termType
1	expression
2	expression
3	expression
4	variable
5	expression
6	expression
7	variable
8	constant
9	constant

Variable table

variable_ID	variable_name
4	X
7	Y

Constant table

constant_ID	const_value
8	2
9	3

Expression table

expression_ID	binary_operator	first_operand	second_operand
1	/	2	3
2	+	4	5
3	-	6	7
5	/	7	8
6	/	4	9

Figure A19.8 Populated database tables for Figure E19.3

19.9 We include tables for *Polyline* and *ObjectGroup* even though they have no attributes. As a matter of style we prefer to apply standard mapping rules unless there is a performance bottleneck; it is easier to define foreign keys if you do not elide tables. A real problem would contain more attributes than shown in the exercise. In general, most of these additional attributes would further describe classes and could be null.

Note that we show separate tables for *Ellipse* and *Rectangle* even though they have the same attributes, because they really are two different things. A full model would most likely have additional attributes that would distinguish *Ellipse* and *Rectangle*.

We show *document_ID* + *page_number* as a candidate key for the page table. This constraint is based on our understanding of desktop publishing and was not specified in the class model.

Document table

document ID	page Width	page Height	left Margin	right Margin

Page table

page ID	page Number (ck1)	documentID (ck1) (references Document)

DrawingObject table

drawing ObjectID	line Thickness	drawing ObjectType	pageID (references Page)	objectGroupID (references ObjectGroup)

Ellipse table

ellipseID (references DrawingObject)	boundingBoxID (ck1) (references BoundingBox)

Rectangle table

rectangleID (references DrawingObject)	boundingBoxID (ck1) (references BoundingBox)

Polyline table

polylineID (references DrawingObject)

Point table

pointID	x	y	polylineID (references Polyline)

Textline table

textlineID (references DrawingObject)	alignment	text	pointID (ck1) (references Point)	fontID (references Font)

Figure A19.9 Tables for Figure E19.4

ObjectGroup table

objectGroupID (references DrawingObject)

BoundingBox table

boundingBoxID	leftEdge	topEdge	width	height

Font table

fontID	fontSize	fontFamily	isBold	isItalic	isUnderlined

Figure A19.9 (continued) Tables for Figure E19.4

19.10 The table is the same as that for Exercise 19.9 except for the following change.

Point table

pointID	x	y	polylineID (ck1) (references Polyline)	sequenceNumber (ck1)

Figure A19.10 Change to table from Exercise 19.9

Note that a relational DBMS stores the rows of a table in an arbitrary order, and not necessarily in the order specified by *sequenceNumber*. To retrieve points in the proper order an *ORDER BY* clause must be included in the SQL query, such as:

```
SELECT point_ID
FROM Point
WHERE polyline_ID = :aPolyline
ORDER BY sequence_Number;
```

19.11 The tables are the same as that for Exercise 19.9 except for the following changes and cutting the *pointID* field from the *Textline* table. It is arbitrary whether we bury the foreign key for the one-to-one association in *Textline* or *TextlinePoint*.

Point table

pointID	x	y	pointType

PolylinePoint table

polylinePointID (references Point)	polylineID (references Polyline)

TextlinePoint table

textlinePointID (references Point)	textlineID (references Textline)

Figure A19.11 Changes to tables from Exercise 19.9

The proposed revision adds a structural constraint. Instead of just stating that a point may or may not associate with a textline or polyline, we can be more specific. We can state that each point associates with exactly one polyline or textline. However the change clutters the model. The merits of the revision depends on the importance of the constraint.

Unfortunately current RDBMS do not support generalization. Thus it is difficult to enforce the exclusive ‘or’ nature of a generalization in RDBMS tables. There are basically two alternatives.

- Forget about trying to enforce the generalization . (Disadvantage: loses the constraint.)
- Enforce the generalization with application code. (Disadvantage: error prone—may be omitted due to oversight. Also it is time consuming.)

19.12 We assume a real coordinate system in assigning data types. Note that the SQL code does not enforce the exclusive-or aspect of generalization.

```
CREATE TABLE Document
( document_ID   NUMBER(30)    CONSTRAINT nn_doc1 NOT NULL,
  page_width    NUMBER(20,10) CONSTRAINT nn_doc2 NOT NULL,
  page_height   NUMBER(20,10) CONSTRAINT nn_doc3 NOT NULL,
  left_margin   NUMBER(20,10) CONSTRAINT nn_doc4 NOT NULL,
  right_margin  NUMBER(20,10) CONSTRAINT nn_doc5 NOT NULL,
CONSTRAINT pk_doc PRIMARY KEY (document_ID));

CREATE SEQUENCE seq_doc;

CREATE TABLE Page
( page_ID      NUMBER(30)    CONSTRAINT nn_page1 NOT NULL,
  page_number  NUMBER(10)    CONSTRAINT nn_page2 NOT NULL,
  document_ID  NUMBER(30)    CONSTRAINT nn_page3 NOT NULL,
CONSTRAINT pk_page PRIMARY KEY (page_ID),
CONSTRAINT uq_page1 UNIQUE (document_ID, page_number));

CREATE SEQUENCE seq_page;

ALTER TABLE Page ADD CONSTRAINT fk_page1
  FOREIGN KEY document_ID
  REFERENCES Document;

CREATE TABLE Drawing_Object
( drawing_object_ID  NUMBER(30) CONSTRAINT nn_do1 NOT NULL,
  line_thickness     NUMBER(10) CONSTRAINT nn_do2 NOT NULL,
  drawing_object_type VARCHAR2(20)
    CONSTRAINT nn_do3 NOT NULL,
  page_ID           NUMBER(30) CONSTRAINT nn_do4 NOT NULL,
  object_group_ID    NUMBER(30),
CONSTRAINT pk_do PRIMARY KEY (drawing_object_ID));
```

```
CREATE SEQUENCE seq_do;

CREATE INDEX index_do1 ON Drawing_Object (page_ID);

CREATE INDEX index_do2 ON Drawing_Object (object_group_ID);

ALTER TABLE Drawing_Object ADD CONSTRAINT fk_do1
  FOREIGN KEY page_ID
  REFERENCES Page;

ALTER TABLE Drawing_Object ADD CONSTRAINT fk_do2
  FOREIGN KEY object_group_ID
  REFERENCES Object_Group;

ALTER TABLE Drawing_Object ADD CONSTRAINT enum_do1
  CHECK (drawing_object_type IN ('Ellipse', 'Rectangle',
  'Polyline', 'Textline', 'Object_Group'));

CREATE TABLE Ellipse
( ellipse_ID      NUMBER(30) CONSTRAINT nn_ell1 NOT NULL,
  bounding_box_ID NUMBER(30) CONSTRAINT nn_ell2 NOT NULL,
CONSTRAINT pk_ell PRIMARY KEY (ellipse_ID),
CONSTRAINT uq_ell1 UNIQUE (bounding_box_ID));

CREATE INDEX index_ell1 ON Ellipse (bounding_box_ID);

ALTER TABLE Ellipse ADD CONSTRAINT fk_ell1
  FOREIGN KEY ellipse_ID
  REFERENCES Drawing_Object ON DELETE CASCADE;

ALTER TABLE Ellipse ADD CONSTRAINT fk_ell2
  FOREIGN KEY bounding_box_ID
  REFERENCES Bounding_Box;

CREATE TABLE Rectangle
( rectangle_ID    NUMBER(30) CONSTRAINT nn_rect1 NOT NULL,
  bounding_box_ID NUMBER(30) CONSTRAINT nn_rect2 NOT NULL,
CONSTRAINT pk_rect PRIMARY KEY (rectangle_ID),
CONSTRAINT uq_rect1 UNIQUE (bounding_box_ID));

CREATE INDEX index_rect1 ON Rectangle (bounding_box_ID);

ALTER TABLE Rectangle ADD CONSTRAINT fk_rect1
  FOREIGN KEY rectangle_ID
  REFERENCES Drawing_Object ON DELETE CASCADE;

ALTER TABLE Rectangle ADD CONSTRAINT fk_rect2
  FOREIGN KEY bounding_box_ID
  REFERENCES Bounding_Box;

CREATE TABLE Polyline
( polyline_ID NUMBER(30) CONSTRAINT nn_polyline1 NOT NULL,
 CONSTRAINT pk_polyline PRIMARY KEY (polyline_ID));
```

```
ALTER TABLE Polyline ADD CONSTRAINT fk_polyline1
  FOREIGN KEY polyline_ID
  REFERENCES Drawing_Object ON DELETE CASCADE;

CREATE TABLE Point
( point_ID    NUMBER(30)    CONSTRAINT nn_point1 NOT NULL,
  x           NUMBER(20,10) CONSTRAINT nn_point2 NOT NULL,
  y           NUMBER(20,10) CONSTRAINT nn_point3 NOT NULL,
  polyline_ID NUMBER(30),
CONSTRAINT pk_point PRIMARY KEY (point_ID));

CREATE SEQUENCE seq_point;

CREATE INDEX index_point1 ON Point (polyline_ID);

ALTER TABLE Point ADD CONSTRAINT fk_point1
  FOREIGN KEY polyline_ID
  REFERENCES Polyline;

CREATE TABLE Textline
( textline_ID NUMBER(30)    CONSTRAINT nn_tline1 NOT NULL,
  alignment   VARCHAR2(10),
  text        VARCHAR2(255) CONSTRAINT nn_tline2 NOT NULL,
  point_ID    NUMBER(30)    CONSTRAINT nn_tline3 NOT NULL,
  font_ID     NUMBER(30)    CONSTRAINT nn_tline4 NOT NULL,
CONSTRAINT pk_tline PRIMARY KEY (textline_ID));

CREATE INDEX index_tline1 ON Textline (point_ID);

CREATE INDEX index_tline2 ON Textline (font_ID);

ALTER TABLE Textline ADD CONSTRAINT fk_tline1
  FOREIGN KEY textline_ID
  REFERENCES Drawing_Object ON DELETE CASCADE;

ALTER TABLE Textline ADD CONSTRAINT fk_tline2
  FOREIGN KEY point_ID
  REFERENCES Point;

ALTER TABLE Textline ADD CONSTRAINT fk_tline3
  FOREIGN KEY font_ID
  REFERENCES Font;

CREATE TABLE Object_Group
( object_group_ID NUMBER(30) CONSTRAINT nn_og1 NOT NULL,
 CONSTRAINT pk_og PRIMARY KEY (object_group_ID));

ALTER TABLE Object_Group ADD CONSTRAINT fk_og1
  FOREIGN KEY object_group_ID
  REFERENCES Drawing_Object ON DELETE CASCADE;
```

```
CREATE TABLE Bounding_Box
( bounding_box_ID NUMBER(30)  CONSTRAINT nn_bbox1 NOT NULL,
  left_edge      NUMBER(20,10) CONSTRAINT nn_bbox2 NOT NULL,
  top_edge       NUMBER(20,10) CONSTRAINT nn_bbox3 NOT NULL,
  width          NUMBER(20,10) CONSTRAINT nn_bbox4 NOT NULL,
  height         NUMBER(20,10) CONSTRAINT nn_bbox5 NOT NULL,
CONSTRAINT pk_bbox PRIMARY KEY (bounding_box_ID));
CREATE SEQUENCE seq_bbox;

CREATE TABLE Font
( font_ID       NUMBER(30)    CONSTRAINT nn_font1 NOT NULL,
  font_size     NUMBER(20,10) CONSTRAINT nn_font2 NOT NULL,
  font_family   VARCHAR2(30)  CONSTRAINT nn_font3 NOT NULL,
  is_bold       VARCHAR2(1)   CONSTRAINT nn_font4 NOT NULL,
  is_italic     VARCHAR2(1)   CONSTRAINT nn_font5 NOT NULL,
  is_underlined VARCHAR2(1)   CONSTRAINT nn_font6 NOT NULL,
CONSTRAINT pk_font PRIMARY KEY (font_ID));
CREATE SEQUENCE seq_font;
```

19.13 This exercise is similar to the edge-node problem from Exercise 19.1. *City* is analogous to *Node* and *Route* is analogous to *Edge*. We infer that a *Route* has 2 *Cities* from the problem statement. We could not deduce that from the SQL code alone.

Figure A19.12 Class model for Figure E19.6

19.14 SQL code to determine distance between two cities for Figure E19.6. This code is similar to that for Exercise 19.5c that determines the edge for a pair of nodes.

```
SELECT distance
FROM Route R, City C1, City C2,
     City_Distance CD1, City_Distance CD2
WHERE C1.city_ID = CD1.city_ID AND
      CD1.route_ID = R.route_ID AND
      R.route_ID = CD2.route_ID AND
      CD2.city_ID = C2.city_ID  AND
      C1.city_name = :aCityName1  AND
      C2.city_name = :aCityName2;
```

19.15 The class model in Figure A19.13 corresponds to the SQL code in Figure E19.7. Note that this model is similar to Figure E19.1a. Figure A19.13 is a better model than Figure E19.1a; each pair of cities has a single value of distance as Figure A19.13 correctly states; however each pair of nodes corresponds to many edges and not one edge as Figure E19.1a shows.

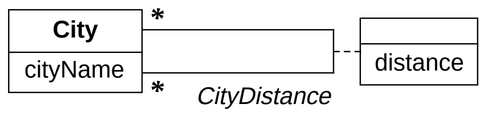

Figure A19.13 Class model for Figure E19.7

19.16 SQL code to determine distance between two cities for Figure E19.7. We don't know which name is *1* and which name is *2*, so the SQL code allows for either possibility.

```
SELECT distance
FROM City C1, City C2, City_Distance CD
WHERE C1.city_ID  = CD.city1_ID AND
      CD.city2_ID = C2.city_ID  AND
      ((C1.cityName = :aCityName1 AND
        C2.cityName = :aCityName2) OR
       (C1.cityName = :aCityName2 AND
        C2.cityName = :aCityName1));
```

19.17 We make the following observations about Figure A19.12 and Figure A19.13.

- Figure A19.12 has an additional table. Figure A19.12 could store multiple routes between the same cities with different distances. Given the lack of explanation about route in the problem statement (is it a series of roads with different distances or is it the distance by air?) this may or may not be a drawback.
- Figure A19.13 is awkward because of the symmetry between city1 and city2. Either data must be stored twice with waste of storage, update time, and possible consistency problems, or special application logic must enforce an arbitrary constraint.

We need to know more about the requirements to choose between the models.

19.18 The city-route problem is essentially an undirected graph. Two cities relate to a route and there is no natural way to distinguish the cities. We normally discourage symmetrical models for undirected graphs (such as Figure A19.13) because they are confusing, lead to possible redundancy, and complicate search and update code.

In contrast, the *Edge-Node* problem is essentially a directed graph. In a directed graph there is a *from* node and a *to* node.

As specified, a given pair of cities has only a single value of distance. In contrast, two nodes may have any number of edges between them. Thus Figure A19.13 is correct (though discouraged). The class model in Figure E19.1a is wrong.

19.19 Figure A19.14 shows our tables.

Meet table

meetID	date	location

RawScore table

trialID (references Trial)	**judgeID** (references Judge)	value

Event table

eventID	startingTime	meetID (references Meet)	stationID (references Station)

Judge table

judgeID	name

Station_Judge table

stationID (references Station)	**judgeID** (references Judge)

Station table

stationID	location	meetID (references Meet)

Trial table

trialID	netScore	eventID (references Event)	competitorID (references Competitor)

Competitor table

competitorID	name	age	address	telephoneNumber

Figure A19.14 Tables for scoring system problem

SQL commands to create an empty database.

```
CREATE TABLE Meet
( meet_ID       NUMBER(30)    CONSTRAINT nn_meet1 NOT NULL,
  meet_date     DATETIME,
  location      VARCHAR2(255),
CONSTRAINT pk_meet PRIMARY KEY (meet_ID));
CREATE SEQUENCE seq_meet;

CREATE TABLE RawScore
( trial_ID      NUMBER(30) CONSTRAINT nn_rs1 NOT NULL,
  judge_ID      NUMBER(30) CONSTRAINT nn_rs2 NOT NULL,
```

```
  raw_score    NUMBER(10),
CONSTRAINT pk_rs PRIMARY KEY (trial_ID, judge_ID));

CREATE INDEX index_rs1 ON RawScore (judge_ID);

ALTER TABLE RawScore ADD CONSTRAINT fk_rs1
  FOREIGN KEY trial_ID
  REFERENCES Trial;

ALTER TABLE RawScore ADD CONSTRAINT fk_rs2
  FOREIGN KEY judge_ID
  REFERENCES Judge;


CREATE TABLE Event
( event_ID      NUMBER(30) CONSTRAINT nn_event1 NOT NULL,
  starting_time DATETIME,
  meet_ID       NUMBER(30) CONSTRAINT nn_event2 NOT NULL,
  station_ID    NUMBER(30) CONSTRAINT nn_event3 NOT NULL,
CONSTRAINT pk_event PRIMARY KEY (event_ID));

CREATE SEQUENCE seq_event;

CREATE INDEX index_event1 ON Event (meet_ID);

CREATE INDEX index_event2 ON Event (station_ID);

ALTER TABLE Event ADD CONSTRAINT fk_event1
  FOREIGN KEY meet_ID
  REFERENCES Meet;

ALTER TABLE Event ADD CONSTRAINT fk_event2
  FOREIGN KEY station_ID
  REFERENCES Station;


CREATE TABLE Judge
( judge_ID    NUMBER(30)    CONSTRAINT nn_judge1 NOT NULL,
  judge_name  VARCHAR2(50),
CONSTRAINT pk_judge PRIMARY KEY (judge_ID));

CREATE SEQUENCE seq_judge;


CREATE TABLE Station
( station_ID NUMBER(30)   CONSTRAINT nn_station1 NOT NULL,
  location   VARCHAR2(255),
  meet_ID    NUMBER(30)   CONSTRAINT nn_station2 NOT NULL,
CONSTRAINT pk_station PRIMARY KEY (station_ID));

CREATE SEQUENCE seq_station;

CREATE INDEX index_station1 ON Station (meet_ID);

ALTER TABLE Station ADD CONSTRAINT fk_station1
  FOREIGN KEY meet_ID
  REFERENCES Meet;
```

```
CREATE TABLE Station_Judge
( station_ID    NUMBER(30) CONSTRAINT nn_sj1 NOT NULL,
  judge_ID      NUMBER(30) CONSTRAINT nn_sj2 NOT NULL,
CONSTRAINT pk_sj PRIMARY KEY (station_ID, judge_ID));

CREATE INDEX index_sj1 ON Station_Judge (judge_ID);

ALTER TABLE Station_Judge ADD CONSTRAINT fk_sj1
  FOREIGN KEY station_ID
  REFERENCES Station;

ALTER TABLE Station_Judge ADD CONSTRAINT fk_sj2
  FOREIGN KEY judge_ID
  REFERENCES Judge;


CREATE TABLE Trial
( trial_ID      NUMBER(30) CONSTRAINT nn_trial1 NOT NULL,
  net_score     NUMBER(10),
  event_ID      NUMBER(30) CONSTRAINT nn_trial2 NOT NULL,
  competitor_ID NUMBER(30) CONSTRAINT nn_trial3 NOT NULL,
CONSTRAINT pk_trial PRIMARY KEY (trial_ID));

CREATE SEQUENCE seq_trial;

CREATE INDEX index_trial1 ON Trial (event_ID);

CREATE INDEX index_trial2 ON Trial (competitor_ID);

ALTER TABLE Trial ADD CONSTRAINT fk_trial1
  FOREIGN KEY event_ID
  REFERENCES Event;

ALTER TABLE Trial ADD CONSTRAINT fk_trial2
  FOREIGN KEY competitor_ID
  REFERENCES Competitor;


CREATE TABLE Competitor
( competitor_ID NUMBER(30)   CONSTRAINT nn_comp1 NOT NULL,
  comp_name     VARCHAR2(50) CONSTRAINT nn_comp2 NOT NULL,
  age           NUMBER(3)    CONSTRAINT nn_comp3 NOT NULL,
  address       VARCHAR2(255),
  telephone_number VARCHAR2(20),
CONSTRAINT pk_comp PRIMARY KEY (competitor_ID));

CREATE SEQUENCE seq_comp;
```

19.20 Figure A19.15 is the same class model that we used in our answer in Chapter 3.

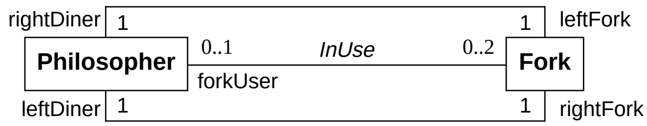

Figure A19.15 Class model for dining philosopher's problem

The mapping rules yield Figure A19.16. and the following SQL commands. (Alternatively, you could have buried the *leftDiner* and *rightDiner* in the *Fork* table.)

Philosopher table

philosopherID	leftForkID (ck1) (references Fork)	rightForkID (ck2) (references Fork)

Fork table

forkID	forkUser (references Philosopher)

Figure A19.16 Tables for dining philosopher's problem

```
CREATE TABLE Philosopher
( philosopher_ID NUMBER(30) CONSTRAINT nn_phil1 NOT NULL,
  left_fork_ID   NUMBER(30) CONSTRAINT nn_phil2 NOT NULL,
  right_fork_ID  NUMBER(30) CONSTRAINT nn_phil3 NOT NULL,
CONSTRAINT pk_phil PRIMARY KEY (philosopher_ID),
CONSTRAINT uq_phil1 UNIQUE (left_fork_ID),
CONSTRAINT uq_phil2 UNIQUE (right_fork_ID));

CREATE SEQUENCE seq_phil;

ALTER TABLE Philosopher ADD CONSTRAINT fk_phil1
  FOREIGN KEY left_fork_ID
  REFERENCES Fork;

ALTER TABLE Philosopher ADD CONSTRAINT fk_phil2
  FOREIGN KEY right_fork_ID
  REFERENCES Fork;


CREATE TABLE Fork
( fork_ID    NUMBER(30) CONSTRAINT nn_fork1 NOT NULL,
  fork_user  NUMBER(30),
CONSTRAINT pk_fork PRIMARY KEY (fork_ID));

CREATE SEQUENCE seq_fork;

CREATE INDEX index_fork1 ON Fork (fork_user);
```

```
ALTER TABLE Fork ADD CONSTRAINT fk_fork1
  FOREIGN KEY fork_user
  REFERENCES Philosopher;
```

Figure A19.17 shows database contents for the case where each philosopher has the left fork.

Philosopher table

philosopher_ID	left_fork_ID	right_fork_ID
1	15	11
2	11	12
3	12	13
4	13	14
5	14	15

Fork table

fork_ID	fork_user
11	2
12	3
13	4
14	5
15	1

Figure A19.17 Populated database table for dining philosopher's problem

19.21 Figure A19.18 shows our tables.

Here are SQL commands to create an empty database.

```
CREATE TABLE Customer
( customer_ID NUMBER(30)   CONSTRAINT nn_cust1 NOT NULL,
  cust_name   VARCHAR2(50) CONSTRAINT nn_cust2 NOT NULL,
CONSTRAINT pk_cust PRIMARY KEY (customer_ID));

CREATE SEQUENCE seq_cust;

CREATE INDEX index_cust1 ON Customer (cust_name);

CREATE TABLE Mailing_Address
( mailing_addr_ID NUMBER(30) CONSTRAINT nn_ma1 NOT NULL,
  address         VARCHAR2(255),
  phone_number    VARCHAR2(20),
CONSTRAINT pk_ma PRIMARY KEY (mailing_addr_ID));

CREATE SEQUENCE seq_ma;

CREATE TABLE Customer__Mailing_Address
( account_holder  NUMBER(30) CONSTRAINT nn_cma1 NOT NULL,
  mailing_addr_ID NUMBER(30) CONSTRAINT nn_cma2 NOT NULL,
CONSTRAINT pk_cma PRIMARY KEY
  (account_holder, mailing_addr_ID));
```

Customer table

customerID	name

MailingAddress table

mailingAddressID	address	phoneNumber

Customer_MailingAddress table

accountHolder (references Customer)	**mailingAddressID** (references MailingAddress)

Institution table

institutionID	name	address	phone Number

CreditCardAccount table

creditCard AccountID	maximum Credit	current Balance	mailingAddressID (references MailingAddress)	institutionID (ck1) (references Institution)	account Number (ck1)

Statement table

statementID	payment DueDate	finance Charge	minimum Payment	creditCardAccountID (ck1) (references CreditCardAccount)	statement Date (ck1)

Transaction table

transac tionID	transaction Date	explan ation	amount	transaction Type	statement ID (ck1)	transaction Number (ck1)

CashAdvance table

cashAdvanceID (references Transaction)

Interest table

interestID (references Transaction)

Purchase table

purchaseID (references Transaction)	merchantID (references Merchant)

Fee table

feeID (references Transaction)	fee Type

Adjustment table

adjustmentID (references Transaction)

Merchant table

merchantID	name

Figure A19.18 Tables for managing credit card accounts

```
CREATE INDEX index_cma1 ON Customer__Mailing_Address
  (mailing_addr_ID);
ALTER TABLE Customer__Mailing_Address
  ADD CONSTRAINT fk_cma1
  FOREIGN KEY account_holder
  REFERENCES Customer ON DELETE CASCADE;
ALTER TABLE Customer__Mailing_Address
  ADD CONSTRAINT fk_cma2
  FOREIGN KEY mailing_addr_ID
  REFERENCES Mailing_Address;

CREATE TABLE Institution
( institution_ID NUMBER(30)   CONSTRAINT nn_inst1 NOT NULL,
  inst_name      VARCHAR2(50) CONSTRAINT nn_inst2 NOT NULL,
  address        VARCHAR2(255),
  phone_number   VARCHAR2(20),
CONSTRAINT pk_inst PRIMARY KEY (institution_ID));
CREATE SEQUENCE seq_inst;

CREATE TABLE Credit_Card_Account
( ccard_account_ID NUMBER(30) CONSTRAINT nn_cca1 NOT NULL,
  maximum_credit   NUMBER(12,2),
  current_balance  NUMBER(12,2),
  mailing_addr_ID  NUMBER(30) CONSTRAINT nn_cca2 NOT NULL,
  institution_ID  NUMBER(30)   CONSTRAINT nn_cca3 NOT NULL,
  account_num     VARCHAR2(20) CONSTRAINT nn_cca4 NOT NULL,
CONSTRAINT pk_cca PRIMARY KEY (ccard_account_ID),
CONSTRAINT uq_cca1 UNIQUE (institution_ID, account_num));
CREATE SEQUENCE seq_cca;
CREATE INDEX index_cca1 ON Credit_Card_Account
  (mailing_addr_ID);
ALTER TABLE Credit_Card_Account ADD CONSTRAINT fk_cca1
  FOREIGN KEY mailing_addr_ID
  REFERENCES Mailing_Address;
ALTER TABLE Credit_Card_Account ADD CONSTRAINT fk_cca2
  FOREIGN KEY institution_ID
  REFERENCES Institution;

CREATE TABLE Statement
( statement_ID     NUMBER(30) CONSTRAINT nn_stat1 NOT NULL,
  payment_due_date DATETIME   CONSTRAINT nn_stat2 NOT NULL,
  finance_charge NUMBER(12,2) CONSTRAINT nn_stat3 NOT NULL,
  minimum_paymt  NUMBER(12,2) CONSTRAINT nn_stat4 NOT NULL,
  ccard_account_ID NUMBER(30) CONSTRAINT nn_stat5 NOT NULL,
```

```
  statement_date  DATETIME   CONSTRAINT nn_stat6 NOT NULL,
CONSTRAINT pk_stat PRIMARY KEY (statement_ID),
CONSTRAINT uq_stat1
  UNIQUE (ccard_account_ID, statement_date));

CREATE SEQUENCE seq_statemt;

ALTER TABLE Statement ADD CONSTRAINT fk_statemt1
  FOREIGN KEY ccard_account_ID
  REFERENCES Credit_Card_Account;

CREATE TABLE Transaction
( transact_ID   NUMBER(30)   CONSTRAINT nn_trans1 NOT NULL,
  transact_date DATETIME     CONSTRAINT nn_trans2 NOT NULL,
  explanation   VARCHAR2(255),
  amount       NUMBER(12,2) CONSTRAINT nn_trans3 NOT NULL,
  transact_type VARCHAR2(20) CONSTRAINT nn_trans4 NOT NULL,
  statement_ID  NUMBER(30)  CONSTRAINT nn_trans5 NOT NULL,
  transact_num  VARCHAR2(20) CONSTRAINT nn_trans6 NOT NULL,
CONSTRAINT pk_trans PRIMARY KEY (transact_ID),
CONSTRAINT uq_trans1 UNIQUE (statement_ID, transact_num));

CREATE SEQUENCE seq_trans;

ALTER TABLE Transaction ADD CONSTRAINT fk_trans1
  FOREIGN KEY statement_ID
  REFERENCES Statement;

ALTER TABLE Transaction ADD CONSTRAINT enum_trans1
  CHECK (transact_type IN ('Cash_Advance', 'Interest',
  'Purchase', 'Fee', 'Adjustment'));

CREATE TABLE Cash_Advance
( cash_advance_ID NUMBER(30) CONSTRAINT nn_casha1 NOT NULL,
CONSTRAINT pk_casha PRIMARY KEY (cash_advance_ID));

ALTER TABLE Cash_Advance ADD CONSTRAINT fk_casha1
  FOREIGN KEY cash_advance_ID
  REFERENCES Transaction ON DELETE CASCADE;

CREATE TABLE Interest
( interest_ID  NUMBER(30) CONSTRAINT nn_interest1 NOT NULL,
CONSTRAINT pk_interest PRIMARY KEY (interest_ID));

ALTER TABLE Interest ADD CONSTRAINT fk_interest1
  FOREIGN KEY interest_ID
  REFERENCES Transaction ON DELETE CASCADE;

CREATE TABLE Adjustment
( adjustment_ID NUMBER(30) CONSTRAINT nn_adjust1 NOT NULL,
CONSTRAINT pk_adjust PRIMARY KEY (adjustment_ID));
```

```
ALTER TABLE Adjustment ADD CONSTRAINT fk_adjust1
  FOREIGN KEY adjustment_ID
  REFERENCES Transaction ON DELETE CASCADE;

CREATE TABLE Purchase
( purchase_ID  NUMBER(30) CONSTRAINT nn_purchase1 NOT NULL,
  merchant_ID  NUMBER(30) CONSTRAINT nn_purchase2 NOT NULL,
CONSTRAINT pk_purchase PRIMARY KEY (purchase_ID));
CREATE INDEX index_purchase1 ON Purchase (merchant_ID);
ALTER TABLE Purchase ADD CONSTRAINT fk_purchase1
  FOREIGN KEY purchase_ID
  REFERENCES Transaction ON DELETE CASCADE;
ALTER TABLE Purchase ADD CONSTRAINT fk_purchase2
  FOREIGN KEY merchant_ID
  REFERENCES Merchant;

CREATE TABLE Fee
( fee_ID        NUMBER(30)   CONSTRAINT nn_fee1 NOT NULL,
  fee_Type      VARCHAR2(20) CONSTRAINT nn_fee2 NOT NULL,
CONSTRAINT pk_fee PRIMARY KEY (fee_ID));
ALTER TABLE Fee ADD CONSTRAINT fk_fee1
  FOREIGN KEY fee_ID
  REFERENCES Transaction ON DELETE CASCADE;

CREATE TABLE Merchant
( merchant_ID   NUMBER(30)   CONSTRAINT nn_merch1 NOT NULL,
  merchant_name VARCHAR2(50) CONSTRAINT nn_merch2 NOT NULL,
CONSTRAINT pk_merch PRIMARY KEY (merchant_ID));
CREATE SEQUENCE seq_merch;
CREATE INDEX index_merch1 ON Merchant (merchant_name);
```

19.22 Here are the SQL queries.

- What transactions occurred for a credit card account within a time interval?

```
SELECT transact_ID
FROM Credit_Card_Account CCA, Statement S, Transaction T
WHERE CCA.ccard_account_ID = S.ccard_account_ID AND
      S.statement_ID = T.statement_ID AND
      T.transact_date >= :aStartDate AND
      T.transact_date <= :anEndDate;
```

- What volume of transactions were handled by an institution in the last year?

```
SELECT sum (T.amount)
FROM Institution I, Credit_Card_Account CCA, Statement S,
     Transaction T
```

```
WHERE I.institution_ID = CCA.institution_ID AND
      CCA.ccard_account_ID = S.ccard_account_ID AND
      S.statement_ID = T.statement_ID AND
      T.transact_date >= :aStartDate AND
      T.transact_date <= :anEndDate;
```

- What customers patronized a merchant in the last year by any kind of credit card?

```
SELECT DISTINCT customer_ID
FROM Merchant M, Purchase P, Transaction T, Statement S,
     Credit_Card_Account CCA, Mailing_Address MA,
     Customer__Mailing_Address CMA, Customer C
WHERE M.merchant_ID = P.merchant_ID AND
      P.purchase_ID = T.transaction_ID AND
      T.statement_ID = S.statement_ID AND
      S.ccard_account_ID = CCA.ccard_account_ID AND
      CCA.mailing_addr_ID = MA.mailing_addr_ID AND
      MA.mailing_addr_ID = CMA.mailing_addr_ID AND
      CMA.customer_ID = C.customer_ID AND
      T.transact_date >= :aStartDate AND
      T.transact_date <= :anEndDate;
```

- How many credit card accounts does a customer currently have?

```
SELECT COUNT(*)
FROM Customer C, Customer__Mailing_Address CMA,
     Mailing_Address MA, Credit_Card_Account CCA
WHERE C.customer_ID = CMA.customer_ID AND
      CMA.mailing_addr_ID = MA.mailing_addr_ID AND
      MA.mailing_addr_ID = CCA.mailing_addr_ID;
```

- What is the total maximum credit for a customer?

```
SELECT sum (maximum_credit)
FROM Customer C, Customer__Mailing_Address CMA,
     Mailing_Address MA, Credit_Card_Account CCA
WHERE C.customer_ID = CMA.customer_ID AND
      CMA.mailing_addr_ID = MA.mailing_addr_ID AND
      MA.mailing_addr_ID = CCA.mailing_addr_ID;
```