APPENDIX 3.A: CREATE TABLE STATEMENTS FOR THE UNIVERSITY DATABASE TABLES

This appendix contains CREATE TABLE statements for the university database tables presented in the body of Chapter 3. The CREATE TABLE statements conform to the SQL standard. In addition, the CREATE TABLE statements execute in both Oracle and PostgreSQL.

```
CREATE TABLE Student
( StdNo           CHAR(11)      CONSTRAINT
                                StdNoRequired NOT NULL,
  StdFirstName    VARCHAR(50)
                  CONSTRAINT StdFirstNameRequired NOT NULL,
  StdLastName     VARCHAR(50)
                  CONSTRAINT StdLastNameRequired NOT NULL,
  StdCity         VARCHAR(50)
                  CONSTRAINT StdCityRequired NOT NULL,
  StdState        CHAR(2)
                  CONSTRAINT StdStateRequired NOT NULL,
  StdZip          CHAR(10)
                  CONSTRAINT StdZipRequired NOT NULL,
  StdMajor        CHAR(6),
  StdClass        CHAR(6),
  StdGPA          DECIMAL(3,2) DEFAULT 0,
  CONSTRAINT PKStudent PRIMARY KEY (StdNo),
  CONSTRAINT ValidGPA CHECK ( StdGPA BETWEEN 0 AND 4 ),
  CONSTRAINT ValidStdClass CHECK
  (StdClass IN ('FR', 'SO', 'JR', 'SR')),
  CONSTRAINT MajorDeclared CHECK ( StdClass IN ('FR','SO')
    OR StdMajor IS NOT NULL ) );

CREATE TABLE Course
( CourseNo        CHAR(6),
  CrsDesc         VARCHAR(250)
                  CONSTRAINT CrsDescRequired NOT NULL,
  CrsUnits        INTEGER,
CONSTRAINT PKCourse PRIMARY KEY (CourseNo),
CONSTRAINT UniqueCrsDesc UNIQUE (CrsDesc)  );

CREATE TABLE Faculty
( FacNo           CHAR(11),
  FacFirstName    VARCHAR(30)
                  CONSTRAINT FacFirstNameRequired NOT NULL,
  FacLastName     VARCHAR(30)
                  CONSTRAINT FacLastNameRequired NOT NULL,
  FacCity         VARCHAR(30)
                  CONSTRAINT FacCityRequired NOT NULL,
  FacState        CHAR(2)
                  CONSTRAINT FacStateRequired NOT NULL,
  FacZipCode      CHAR(10)
                  CONSTRAINT FacZipRequired NOT NULL,
  FacHireDate     DATE,
  FacDept         CHAR(6),
  FacRank         CHAR(4),
  FacSalary       DECIMAL(10,2),
```

```
  FacSupervisor  CHAR(11),
CONSTRAINT PKFaculty PRIMARY KEY (FacNo),
CONSTRAINT FKFacSupervisor FOREIGN KEY (FacSupervisor)
           REFERENCES Faculty );

CREATE TABLE Offering
( OfferNo         INTEGER,
  CourseNo        CHAR(6)       CONSTRAINT OffCourseNoRequired
                                NOT NULL,
  OffLocation     VARCHAR(30),
  OffDays         CHAR(6)       DEFAULT 'MW',
  OffTerm         CHAR(6)
                  CONSTRAINT OffTermRequired NOT NULL,
  OffYear         INTEGER       DEFAULT 2022
                  CONSTRAINT OffYearRequired NOT NULL,
  FacNo           CHAR(11),
  OffTime         DATE,
CONSTRAINT PKOffering PRIMARY KEY (OfferNo),
CONSTRAINT FKCourseNo FOREIGN KEY (CourseNo)
  REFERENCES Course,
CONSTRAINT FKFacNo FOREIGN KEY (FacNo) REFERENCES Faculty );

CREATE TABLE Enrollment
( OfferNo         INTEGER,
  StdNo           CHAR(11),
  EnrGrade        DECIMAL(3,2) DEFAULT 0,
CONSTRAINT PKEnrollment PRIMARY KEY (OfferNo, StdNo),
CONSTRAINT FKOfferNo FOREIGN KEY (OfferNo)
  REFERENCES Offering ON DELETE CASCADE,
CONSTRAINT FKStdNo FOREIGN KEY (StdNo) REFERENCES Student
  ON DELETE CASCADE );
```

APPENDIX 3.B: SQL SYNTAX SUMMARY

This appendix provides a convenient summary of the SQL syntax for the CREATE TABLE statement along with several related statements. The SQL syntax provides rules that define valid SQL statements. For brevity, only the syntax of the most common parts of the statements is described. SQL:2019 is the current version of the SQL standard although the syntax rules in SQL:2019 for the statements described in this appendix are identical to previous SQL standards (SQL:2016, SQL:2011, SQL:2008, SQL:2003, SQL:1999 and SQL-92). For the complete syntax, refer to a SQL:2019 or a SQL-92 reference book such as Groff and Weinberg (2002). The conventions used in the syntax notation are listed before the syntax rules.

- Uppercase words denote reserved words.
- Mixed-case words without hyphens denote names that the user substitutes.
- The asterisk * after a syntax element indicates that a comma-separated list can be used.
- The plus symbol + after a syntax element indicates that a list can be used. No commas appear in the list.
- Names enclosed in angle brackets <> denote definitions defined later in the syntax. The definitions occur on a new line with the element and colon followed by the syntax.
- Square brackets [] enclose optional elements.
- Curly brackets {} enclose choice elements. One element must be chosen among the elements separated by the vertical bars |.
- The parentheses () denote themselves.
- Double hyphens — denote comments that are not part of the syntax.

CREATE TABLE[1] Syntax

```
CREATE TABLE  TableName
 ( <Column-Definition>*  [ ,  <Table-Constraint>*  ]  )

<Column-Definition>: ColumnName DataType
   [ DEFAULT { DefaultValue | USER | NULL } ]
   [ <Column-Constraint>+ ]

<Column-Constraint>:
   { [ CONSTRAINT ConstraintName ] NOT NULL      |
     [ CONSTRAINT ConstraintName ] UNIQUE        |
     [ CONSTRAINT ConstraintName ] PRIMARY KEY   |
     [ CONSTRAINT ConstraintName ] <Check-Constraint> |
     [ CONSTRAINT ConstraintName ] FOREIGN KEY
         REFERENCES TableName [ ( ColumnName ) ]
         [ ON DELETE  <Action-Specification> ]
         [ ON UPDATE  <Action-Specification> ] }

<Table-Constraint>: [ CONSTRAINT ConstraintName ]
     { <Primary-Key-Constraint>  |
       <Foreign-Key-Constraint>  |
       <Uniqueness-Constraint>   |
       <Check-Constraint>   }

<Primary-Key-Constraint>: PRIMARY KEY ( ColumnName* )
```

[1] Chapter 4 defines <Row-Condition> due to its complex structure and usage in the SQL SELECT statement.

```
<Foreign-Key-Constraint>: FOREIGN KEY ( ColumnName* )
     REFERENCES TableName [( ColumnName* )]
     [ ON DELETE  <Action-Specification> ]
     [ ON UPDATE  <Action-Specification> ]

<Action-Specification>: { CASCADE | SET NULL |
     SET DEFAULT | RESTRICT }

<Uniqueness-Constraint>: UNIQUE ( ColumnName* )

<Check-Constraint>: CHECK ( <Row-Condition> )

<Row-Condition>: -- defined in Chapter 4
```

Other related statements

The ALTER TABLE and DROP TABLE statements support modification of a table definition and deleting a table definition. The ALTER TABLE statement is particularly useful because table definitions often change over time. In both statements, the keyword RESTRICT means that the statement cannot be performed if related tables exist. The keyword CASCADE means that the same action will be performed on related tables.

```
ALTER TABLE  TableName
 { ADD { <Column-Definition> |  <Table-Constraint> } |
   ALTER ColumnName { SET DEFAULT DefaultValue  |
          DROP DEFAULT } |
   DROP ColumnName { CASCADE | RESTRICT }  |
   DROP CONSTRAINT ConstraintName {CASCADE | RESTRICT }
}

DROP TABLE  TableName { CASCADE | RESTRICT }
```

Notes on Oracle Syntax

The CREATE TABLE statement in Oracle 21c SQL conforms closely to the SQL:2019 standard. Here is a list of the most significant syntax differences:

- Oracle SQL does not support the ON UPDATE clause for referential integrity constraints.
- Oracle SQL only supports CASCADE and SET NULL as the action specifications of the ON DELETE clause. If an ON DELETE clause is not specified, the deletion is not allowed (restricted) if related rows exist.
- Oracle does not support the BOOLEAN data type. A typical substitution is to use CHAR(1) with text values such as T/F or Y/N.
- Oracle SQL does not support dropping columns in the ALTER statement.
- Oracle SQL supports the MODIFY keyword in place of the ALTER keyword in the ALTER TABLE statement (use MODIFY ColumnName instead of ALTER ColumnName).
- Oracle SQL supports data type changes using the MODIFY keyword in the ALTER TABLE statement.

APPENDIX 3.C: GENERATION OF UNIQUE VALUES FOR PRIMARY KEYS

The SQL standard provides the GENERATED clause to support the generation of unique values for selected columns, typically primary keys. The GENERATED clause is used in place of a default value as shown in the following syntax specification. Typically, a whole number data type such as INTEGER should be used for columns with a GENERATED clause. The START BY and INCREMENT BY keywords can be used to indicate the initial value and the increment value. The ALWAYS keyword indicates that the value is always automatically generated. The BY DEFAULT clause allows a user to specify a value, overriding the automatic value generation.

```
<Column-Definition>: ColumnName DataType
   [ <Default-Specification> ]
   [ <Column-Constraint>+ ]

<Default-Specification>:
    { DEFAULT { DefaultValue | USER | NULL } |
      GENERATED {ALWAYS | BY DEFAULT } AS IDENTITY
       START WITH NumericConstant
     [ INCREMENT BY NumericConstant ] }
```

The following example demonstrates the SQL standard approach for automatic value generation. Note that the primary key constraint is not required for columns with generated values although generated values are mostly used for primary keys. The example executes in both Oracle and PostgreSQL.

SQL GENERATED Clause Example

```
-- Executes in Oracle and PostgreSQL
-- User must have CREATE SEQUENCE privilege in Oracle.
-- Oracle creates a sequence to support identity columns.
CREATE TABLE XStudent
( StdNo INTEGER GENERATED ALWAYS AS IDENTITY
          (START WITH 1 INCREMENT BY 1),
CONSTRAINT PKXStudent PRIMARY KEY (StdNo)     );
```

The second example uses a sequence object, an older approach for generation of unique values. The second example contains two statements: one for the sequence creation and another for the table creation. Because sequences are not associated with columns, Oracle and PostgreSQL use *nextval* and *currval* to generate and reference sequence values when inserting a row into a table. In PostgreSQL, *nextval* and *currval* are functions. In Oracle, *nextval* and *currval* are pseudo columns[2]. In contrast, the usage of extra functions is not necessary in the GENERATED clause of the SQL standard.

Sequence Example

```
-- Executes in Oracle and PostgreSQL
-- Alternative to GENERATED clause
CREATE SEQUENCE StdNoSeq START WITH 1 INCREMENT BY 1;

CREATE TABLE XStudent2
( StdNo INTEGER,
CONSTRAINT PKXStudent2 PRIMARY KEY (StdNo)    );
```

[2] In Oracle, a pseudo column behaves a column of a table but is not stored in the table.

APPENDIX 4.A: SQL SYNTAX SUMMARY

This appendix summarizes SQL syntax for the SELECT, INSERT, UPDATE, and DELETE statements presented in this chapter. The syntax is limited to the simplified statement structure presented in this chapter. Chapters in Part 5 extend the syntax with additional elements. The conventions used in the syntax notation are identical to those used at the end of Chapter 3.

Simplified SELECT Syntax

```
<Select-Statement>: { <Simple-Select> | <Set-Select> }
    [ ORDER BY <Sort-Specification>* ]

<Simple-Select>:
    SELECT  [ DISTINCT ]  <Column-Specification>*
      FROM <Table-Specification>*
      [ WHERE <Row-Condition> ]
      [ GROUP BY ColumnName* ]
      [ HAVING <Group-Condition> ]

<Column-Specification>: { <Column-List> | <Column-Item> }

<Column-List>: { * | TableName.*  }
    -- * is a literal here not a syntax symbol

<Column-Item>: <Column-Expression> [ [ AS ] AliasName ]

<Column-Expression>:
  { <Scalar-Expression> | <Aggregate-Expression> }

<Scalar-Expression>:
  { <Scalar-Item> |
    <Scalar-Item> <Arith-Operator> <Scalar-Item> }


<Scalar-Item>:
  { [ TableName.]ColumnName  |
      Constant   |
      FunctionName [ (Argument*) ] |
      <Scalar-Expression>  |
      ( <Scalar-Expression> ) }


<Arith-Operator>:  { +  |  -  |  *  |  /  }
    -- * and + are literals here not syntax symbols


<Aggregate-Expression>:
 { SUM ( {<Scalar-Expression> | DISTINCT ColumnName } )  |
    AVG ( {<Scalar-Expression> | DISTINCT ColumnName } )  |
    MIN ( <Scalar-Expression> )  |
    MAX ( <Scalar-Expression> )  |
    COUNT ( [ DISTINCT ] ColumnName ) |
    COUNT ( * )  }  -- * is a literal symbol here, not a syntax symbol


<Table-Specification>: { <Simple-Table>  |
                         <Join-Operation> }
```

```
<Simple-Table>: TableName [ [ AS ] AliasName ]


<Join-Operation>:
  { <Simple-Table> [INNER] JOIN <Simple-Table>
     ON <Join-Condition>  |
    { <Simple-Table> | <Join-Operation> } [INNER] JOIN
      { <Simple-Table> | <Join-Operation> }
      ON <Join-Condition>  |
    ( <Join-Operation> )  }


<Join-Condition>: { <Simple-Join-Condition> |
                    <Compound-Join-Condition> }


<Simple-Join-Condition>:
  <Scalar-Expression> <Comparison-Operator>
  <Scalar-Expression>


<Compound-Join-Condition>:
   { NOT <Join-Condition>  |
     <Join-Condition> AND <Join-Condition>  |
     <Join-Condition> OR <Join-Condition>  |
     ( <Join-Condition> )


<Comparison-Operator>: { = | < | > | <= | >= | <> }


<Row-Condition>:
 { <Simple-Condition> | <Compound-Condition> }


<Simple-Condition>:
{ <Scalar-Expression> <Comparison-Operator>
      <Scalar-Experssion>  |
  <Scalar-Expression> [ NOT ] IN ( Constant* )  |
  <Scalar-Expression> BETWEEN <Scalar-Expression> AND
      <Scalar-Expression> |
  <Scalar-Expression> IS [NOT] NULL  |
  ColumnName [ NOT ] LIKE StringPattern  }


<Compound-Condition>:
    { NOT <Row-Condition>  |
      <Row-Condition> AND <Row-Condition>  |
      <Row-Condition> OR <Row-Condition>  |
      ( <Row-Condition> )  }


<Group-Condition>:
 { <Simple-Group-Condition> | <Compound-Group-Condition> }
```

```
<Simple-Group-Condition>:-- permits both scalar and aggregate expressions
{ <Column-Expression> ComparisonOperator
        < Column-Experssion>  |
  <Column-Expression> [ NOT ] IN ( Constant* )  |
  <Column-Expression> BETWEEN <Column-Expression> AND
      <Column-Expression> |
  <Column-Expression> IS [NOT] NULL  |
  ColumnName [ NOT ] LIKE StringPattern  }


<Compound-Group-Condition>:
    { NOT <Group-Condition>  |
      <Group-Condition> AND <Group-Condition>  |
      <Group-Condition> OR <Group-Condition>  |
      ( <Group-Condition> )  }


<Sort-Specification>:
 { ColumnName | ColumnNumber } [ { ASC | DESC } ]


<Set-Select>:
 { <Simple-Select> | <Set-Select> } <Set-Operator>
 { <Simple-Select> | <Set-Select> }


<Set-Operator>: { UNION | INTERSECT | EXCEPT } [ ALL ]
```

INSERT Syntax

```
INSERT INTO TableName ( ColumnName* )
   VALUES ( Constant* )

INSERT INTO TableName [ ( ColumnName* ) ]
   <Simple-Select>
```

UPDATE Syntax

```
UPDATE TableName
  SET <Column-Assignment>*
  [ WHERE <Row-Condition> ]

<Column-Assignment>: ColumnName = <Scalar-Expression>
```

DELETE Syntax

```
DELETE FROM TableName
  [ WHERE  <Row-Condition> ]
```

APPENDIX 4.B: SYNTAX DIFFERENCES AMONG MAJOR DBMS PRODUCTS

Table 4B-1 summarizes syntax differences among PostgreSQL, Oracle, Microsoft SQL Server, and IBM DB2. The differences involve the parts of the SELECT statement presented in the chapter.

TABLE 4B-1
SELECT Syntax Differences among Major DBMS Products

	Product			
Element	***Oracle***	***PostgreSQL***	***MS SQL Server***	***DB2***
Pattern-matching characters	%, _	%, _	%, _	%, _
Case sensitivity in text matching	Yes	Yes	Depends on COLLATE clause[3]	Yes
Date constants	Surround in single quotation marks	Surround in single quotation marks	Surround in single quotation marks	Surround in single quotation marks
Inequality symbol	<>	<>	<>	<>
Join operator style	No in 8i, Yes in 9i and later versions	Yes	Yes	Yes
Difference operations	MINUS keyword	EXCEPT keyword	EXCEPT keyword	EXCEPT keyword

[3] Case insensitive by default but default can be changed for a database or column using the COLLATE clause.

APPENDIX 7.A: SECOND AND THIRD NORMAL FORMS

Second (2NF) and third (3NF) normal forms were proposed before BCNF. Conceptually, BCNF makes 2NF and 3NF obsolete although many books still reference the earlier forms. BCNF provides a simpler definition and covers two special cases not covered by 3NF. The special cases are rare, but the simplified definition makes BCNF superior to the combination of 2NF/3NF. This appendix covers 2NF and 3NF as reference material in case you see these normal forms mentioned in books and websites.

The definitions of 2NF and 3NF distinguish between key and nonkey columns[4]. A column is a key column if it is part of a candidate key or a candidate key by itself. Recall that a candidate key is a minimal set of column(s) that has unique values in a table. Minimality means that none of the columns can be removed without losing the uniqueness property. Essentially, candidate keys do not have extra columns. Nonkey columns are any other columns. In the big university database table shown in Table 7A-1, the combination of (*StdNo*, *OfferNo*) is the only candidate key. Other columns such as *StdCity* and *StdClass* are nonkey columns.

2NF and 3NF involve the relationship between key and nonkey columns and the relationship between nonkey columns. The basic goal of 2NF and 3NF is to produce tables in which every key determines the nonkey columns and nonkey columns do not determine other nonkey columns. An easy way to remember the definitions of both 2NF and 3NF is shown in the following definition.

> **Combined Definition of 2NF and 3NF:** a table is in 3NF if each nonkey column depends on all candidate keys, whole candidate keys, and nothing but candidate keys.[5]

Second Normal Form

To understand this definition, let us break it down to the 2NF and 3NF parts. The 2NF definition uses the first part of the definition as shown in the following definition.

> **2NF Definition:** a table is in 2NF if each nonkey column depends on whole candidate keys, not on a subset of any candidate key.

To see if a table is in 2NF, you should look for FDs that violate the definition. An FD in which part of a key determines a nonkey column violates 2NF. If all candidate keys contain only one column, the table is in 2NF. Looking at the FDs in Table 7A-2, you can easily detect violations of 2NF. For example, *StdCity* is a nonkey column but *StdNo*, not the entire primary key (combination of *StdNo* and *OfferNo*), determines

TABLE 7A-1
Sample Data for the Big University Database Table

StdNo	StdCity	StdClass	OfferNo	OffTerm	OffYear	EnrGrade	CourseNo	CrsDesc
S1	SEATTLE	JUN	O1	FALL	2020	3.5	C1	DB
S1	SEATTLE	JUN	O2	FALL	2020	3.3	C2	VB
S2	BOTHELL	JUN	O3	SPRING	2021	3.1	C3	OO
S2	BOTHELL	JUN	O2	FALL	2020	3.4	C2	VB

[4] In some academic literature, key columns are known as prime, and nonkey columns as nonprime.

[5] You can remember this definition by its analogy to the traditional justice oath: "Do you swear to tell the truth, the whole truth, and nothing but the truth, ...".

StdNo → StdCity, StdClass
OfferNo → OffTerm, OffYear, CourseNo, CrsDesc
CourseNo → CrsDesc
StdNo, OfferNo → EnrGrade

TABLE 7A-2
List of FDs for the Big University Database Table

it. The only FDs that satisfy the 2NF definition are *StdNo*, *OfferNo* → *EnrGrade* and *CourseNo* → *CrsDesc*.

2NF Violation: an FD in which part of key determines a nonkey violates 2NF. A table with only single column candidate keys cannot violate 2NF.

To place the table into 2NF, you split the original table into smaller tables that satisfy the 2NF definition. In each smaller table, the entire primary key (not part of the primary key) should determine the nonkey columns. The splitting process involves the project operator of relational algebra. For the university database table, three projection operations split it so that the underlined primary key determines the nonkey columns in each table below.

UnivTable1 (StdNo, StdCity, StdClass)
UnivTable2 (OfferNo, OffTerm, OffYear, CourseNo, CrsDesc)
UnivTable3 (StdNo, OfferNo, EnrGrade)

The splitting process should preserve the original table in two ways. First, the original table should be recoverable using natural join operations on the smaller tables. Second, the FDs in the original table should be derivable from the FDs in the smaller tables. Technically, the splitting process is known as a nonloss, dependency-preserving decomposition. Some of the references at the end of this chapter explain the theory underlying the splitting process.

After splitting the original table into smaller tables, you should add referential integrity constraints to connect the tables. Whenever a table is split, the splitting column becomes a foreign key in the table in which it is not a primary key. For example, *StdNo* is a foreign key in *UnivTable3* because the original university table was split on this column. Therefore, you should define a referential integrity constraint stating that *UnivTable3.StdNo* refers to *UnivTable1.StdNo*. The *UnivTable3* table is repeated below with its referential integrity constraints.

UnivTable3 (StdNo, OfferNo, EnrGrade)
FOREIGN KEY (StdNo) REFERENCES UnivTable1
FOREIGN KEY (OfferNo) REFERENCES UnivTable2

Third Normal Form

UnivTable2 still has modification anomalies. For example, you cannot add a new course unless the *OfferNo* column value is known. To eliminate the modification anomalies, the definition of 3NF should be applied.

3NF Definition
a table is in 3NF if it is in 2NF and each nonkey column depends only on candidate keys, not on other nonkey columns.

An FD in which a nonkey column determines another nonkey column violates 3NF. In *UnivTable2* above, the FD (*CourseNo* → *CrsDesc*) violates 3NF because both columns, *CourseNo* and *CrsDesc* are nonkey. To fix the violation, split *UnivTable2* into two tables, as shown below, and add a foreign key constraint.

UnivTable2-1 (OfferNo, OffTerm, OffYear, CourseNo)
FOREIGN KEY (CourseNo) REFERENCES UnivTable2-2
UnivTable2-2 (CourseNo, CrsDesc)

An equivalent way to define 3NF is that 3NF prohibits transitive dependencies. A transitive dependency is a functional dependency derived by the law of transitivity.

Transitive Dependency
an FD derived by the law of transitivity. Transitive FDs should not be recorded as input to the normalization process.

The law of transitivity says that if an object A is related to B and B is related to C, then you can conclude that A is related to C. For example, the < operator obeys the transitive law for real numbers: A < B and B < C implies that A < C. Functional dependencies, like the < operator, obey the law of transitivity: A → B, B → C, then A → C. In Figure 7.2, *OfferNo* → *CrsDesc* is a transitive dependency derived from *OfferNo* → *CourseNo* and *CourseNo* → *CrsDesc*.

Because transitive dependencies are easy to overlook, the preferred definition of 3NF does not use transitive dependencies. In addition, you learned in Section 7.2.3 to omit derived dependencies such as transitive dependencies in the list of FDs for a table.

Combined Example of 2NF and 3NF

The big patient table as depicted in Table 7A-3 provides another example for applying your knowledge of 2NF and 3NF. The big patient table contains facts about patients, health care providers, patient visits to a clinic, and diagnoses made by health care providers. The big patient table contains a combined primary key consisting of the combination of *VisitNo* and *ProvNo* (provider number). Like the big university database table depicted in Table 7A-1, the big patient table reflects a poor table design with many redundancies. Table 7A-4 lists the associated FDs. You should verify that the sample rows in Table 7A-3 do not falsify the FDs.

As previously discussed, FDs that violate 2NF involve part of a key determining a nonkey. Many of the FDs in Table 7A-4 violate the 2NF definition because the combination of *VisitNo* and *ProvNo* is the primary key. Thus, the FDs with only *VisitNo* or *ProvNo* in the LHS violate 2NF. To alleviate the 2NF violations, split the big patient table so that the violating FDs are associated with separate tables. In the revised list of tables, *PatientTable1* and *PatientTable2* contain the violating FDs. *PatientTable3* retains the remaining columns.

PatientTable1 (ProvNo, ProvSpecialty)
PatientTable2 (VisitNo, VisitDate, PatNo, PatAge, PatCity, PatZip)
PatientTable3 (VisitNo, ProvNo, Diagnosis)
FOREIGN KEY (VisitNo) REFERENCES PatientTable2
FOREIGN KEY (ProvNo) REFERENCES PatientTable1

PatientTable1 and *PatientTable3* are in 3NF because there are no nonkey columns that determine other nonkey columns. However, *PatientTable2* violates 3NF because the

TABLE 7A-3
Sample Data for the Big Patient Table

VisitNo	VisitDate	PatNo	PatAge	PatCity	PatZip	ProvNo	ProvSpecialty	Diagnosis
V10020	1/13/2021	P1	35	DENVER	80217	D1	INTERNIST	EAR INFECTION
V10020	1/13/2021	P1	35	DENVER	80217	D2	NURSE PRACTITIONER	INFLUENZA
V93030	1/20/2021	P3	17	ENGLEWOOD	80113	D2	NURSE PRACTITIONER	PREGNANCY
V82110	1/18/2021	P2	60	BOULDER	85932	D3	CARDIOLOGIST	MURMUR

TABLE 7A-4
List of FDs for the Big Patient Table

PatNo → PatAge, PatCity, PatZip
PatZip → PatCity
ProvNo → ProvSpecialty
VisitNo → PatNo, VisitDate, PatAge, PatCity, PatZip
VisitNo, ProvNo → Diagnosis

FDs *PatNo* → *PatZip*, *PatAge* and *PatZip* → *PatCity* involve nonkey columns that determine other nonkey columns. To alleviate the 3NF violations, split *PatientTable2* into three tables as shown in the revised table list. In the revised list of tables, *PatientTable2-1* and *PatientTable2-2* contain the violating FDs, while *PatientTable2-3* retains the remaining columns.

PatientTable2-1 (PatNo, PatAge, PatZip)
    FOREIGN KEY (PatZip) REFERENCES PatientTable2-2
PatientTable2-2 (PatZip, PatCity)
PatientTable2-3 (VisitNo, PatNo, VisitDate)
    FOREIGN KEY (PatNo) REFERENCES PatientTable2-1

Using 2NF and 3NF requires two normalization steps. The normalization process can be performed in one step using BCNF and the simple synthesis procedure, as presented in section 7.2.

APPENDIX 9.A: SQL SYNTAX SUMMARY

This appendix summarizes the SQL syntax for nested SELECT statements and outer join operations presented in Chapter 9. For the syntax of other variations of the nested SELECT and outer join operations not presented in Chapter 9, consult an SQL reference book. Nested SELECT statements can be used in the WHERE clause of the SELECT, UPDATE, and DELETE statements as well as the FROM and HAVING clauses. The conventions used in the syntax notation are identical to those used at the end of Chapter 3.

Expanded Syntax for Nested Queries in the FROM Clause

```
<Table-Specification>:
     { <Simple-Table>    |   -- defined in Chapter 4
       <Join-Operation>  |   -- defined in Chapter 4
       <Simple-Select>   [ AS ] AliasName[6]   }
      -- <Simple-Select> is defined in Chapter 4
```

Expanded Syntax for Row Conditions

```
<Row-Condition>:
{ <Simple-Condition> |    -- defined in Chapter 4
   <Compound-Condition> |   -- defined in Chapter 4
   <Exists-Condition>  |
   <Element-Condition> }

<Exists-Condition>:  [ NOT ] EXISTS ( <Simple-Select> )

<Simple-Select>: -- defined in Chapter 4

<Element-Condition>:
 <Scalar-Expression> <Element-Operator>
                        ( <Simple-Select> ) |
 ( ColumnName* ) [ NOT ] IN ( <Simple-Select> )

<Element-Operator>:
 { = | < | > | >= | <= | <> | [ NOT ] IN }

<Scalar-Expression>: -- defined in Chapter 4
```

Expanded Syntax for Group Conditions

```
<Simple-Group-Condition>:   -- Last choice is new
{ <Column-Expression> ComparisonOperator
        <Column-Experssion>   |
  <Column-Expression> [ NOT ] IN ( Constant* )   |
  <Column-Expression> BETWEEN <Column-Expression>
       AND <Column-Expression> |
  <Column-Expression> IS [NOT] NULL   |
  ColumnName [ NOT ] LIKE StringPattern |
  <Exists-Condition>   |
```

[6] PostgreSQL requires an AliasName for nested SELECT statements in the FROM clause. Oracle only requires an alias name for ambiguous references to columns of a nested query.

```
  <Column-Expression> <Element-Operator>
                        ( <Simple-Select> }

<Column-Expression>: -- defined in Chapter 4
```

Expanded Syntax for Outer Join Operations

```
<Join-Operation>:
  { <Simple-Table> <Join-Operator> <Simple-Table>
     ON <Join-Condition>  |
    { <Simple-Table> | <Join-Operation> } <Join-Operator>
      { <Simple-Table> | <Join-Operation> }
      ON <Join-Condition>  |
    ( <Join-Operation> )  }

<Join-Operator>:
{ [ INNER ] JOIN        |
  LEFT [ OUTER ] JOIN   |
  RIGHT [ OUTER ] JOIN  |
  FULL [ OUTER ] JOIN   }
```

APPENDIX 10.A: SQL SYNTAX SUMMARY

This appendix summarizes the SQL standard syntax for the CREATE VIEW and DROP VIEW statements and recursive common table expressions. The conventions used in the syntax notation are identical to those used at the end of Chapter 3.

CREATE VIEW Statement

```
CREATE VIEW ViewName [ ( ColumnName* ) ]
  AS  <Select-Statement>
  [ WITH CHECK OPTION ]
```

<Select-Statement>: -- defined in Chapter 4 and extended in Chapter 9

DROP VIEW Statement

```
DROP VIEW ViewName  [ { CASCADE | RESTRICT } ]
```

-- CASCADE deletes the view and any views that use its definition.

-- RESTRICT means that the view is not deleted if any views use its definition.

Expanded Syntax for Recursive Common Table Expressions

```
WITH CTEName ( ColumnName* )
-- PostgreSQL uses WITH RECURSIVE keywords instead of WITH
AS
-- Anchor member (AM) referencing the hierarchical table.
 ( <Simple-Select> – Using expanded <Table-Specification> in Chapter 9
UNION ALL
-- Recursive member (RM) referencing CTEName.
    <Simple-Select> ) – Using expanded <Table-Specification> in Chapter 9
-- Statement using CTEName
<Select-Statement>; – Using expanded <Table-Specification> in Chapter 9
```

APPENDIX 11.A: SQL SYNTAX SUMMARY

This appendix summarizes the SQL standard syntax for the CREATE TRIGGER statement. The conventions used in the syntax notation are identical to those used at the end of Chapter 3.

CREATE Trigger Statement

```
CREATE TRIGGER TriggerName
 <TriggerTiming> <TriggerEvent> ON TableName
 [ REFERENCING <AliasClause> [ <AliasClause> ] ]
 [ <GranularityClause> [ WHEN ( <Row-Condition> ) ] ]
 <ActionBlock>

<TriggerTiming>: { BEFORE | AFTER }

<TriggerEvent>: { INSERT | DELETE |
  UPDATE [ OF ColumnName* ] }

<AliasClause>: { <RowAlias> | <TableAlias> }

<RowAlias>:
  { OLD [ROW] [AS] AliasName |
    NEW [ROW] [AS] AliasName }

<TableAlias>:
  { OLD TABLE [AS] AliasName |
    NEW TABLE [AS] AliasName }

<GranularityClause>: FOR EACH { ROW | STATEMENT }

<Row-Condition>: -- defined in Chapter 4

<ActionBlock>:
 -- can be a procedure call or an SQL block
```

APPENDIX 11.B: POSTGRESQL TRIGGERS AND TRIGGER FUNCTIONS

PostgreSQL triggers follow the SQL standard in most ways. Like triggers in the SQL standard, PostgreSQL supports statement and row triggers, events for INSERT, UPDATE, and DELETE actions, BEFORE, AFTER, and INSTEAD OF timing, and WHEN conditions. INSTEAD OF triggers only apply to views like SQL standard triggers.

PostgreSQL triggers extend SQL standard triggers in several way as indicated in the following list. The trigger body change is a large difference from the SQL standard.

- PostgreSQL also supports trigger events for the TRUNCATE statement providing unqualified removal of rows from multiple tables.
- PostgreSQL supports constraint triggers for transaction processing not specified in the SQL standard. Constraint triggers involve transaction processing concepts presented in Chapter 17, details beyond the presentation in this appendix.
- PostgreSQL fires BEFORE triggers before a delete action, even cascading deletes. The SQL standard indicates that BEFORE DELETE triggers fire after completion of cascaded deletes.
- PostgreSQL executes overlapping triggers in name order. The SQL standard indicates creation time order.
- PostgreSQL supports data change and event triggers. The SQL standard does not make this distinction. Data change triggers support typical processing of row triggers. Event triggers are more limited for notification of events.
- PostgreSQL triggers only contain the EXECUTE FUNCTION statement in the trigger body. The EXECUTE FUNCTION statement invokes a trigger function usually coded with statements conforming to the PL/pgSQL language. In contrast, SQL standard triggers use database programming language code inside the trigger body.

The remainder of this appendix presents some example triggers to depict features of PostgreSQL triggers and associated user-defined functions. All examples involve data change triggers. The textbook website contains PostgreSQL triggers and associated user-defined functions for all trigger examples in Chapter 11 as well as PostgreSQL solutions for chapter problems involving triggers.

The first PostgreSQL trigger displays details about new and old values using the NEW and OLD keywords. The trigger has just a single statement in the body following the FOR EACH ROW keywords. The EXECUTE FUNCTION keywords precede the name of the trigger function to execute. Usually, the trigger function has the same name as the trigger. The trigger function (*tr_Course_DIUA*) returns a NULL result indicating no data changes in the trigger. The RAISE statement with NOTICE level displays details in the Query Tool window without raising an error. The line following the trigger function indicates that the body of the trigger function uses PL/pgSQL code.

Example 11.P1 – Trigger with a combined event that fires for every action on the *course* table along with testing code to fire the trigger. Equivalent to example 11.25.

```
-- Create Trigger function
CREATE OR REPLACE FUNCTION tr_Course_DIUA()
     RETURNS TRIGGER AS $tr_Course_DIUA$
BEGIN
     raise notice 'Inserted Row';
     raise notice 'CourseNo: %', NEW.CourseNo;
     raise notice 'Course Description: %' ,NEW.CrsDesc;
     raise notice 'Course Units: %', NEW.CrsUnits;
     raise notice 'Deleted Row';
     raise notice 'CourseNo: %', OLD.CourseNo;
```

```
     raise notice 'Course Description: %' , OLD.CrsDesc;
     raise notice 'Course Units: %', OLD.CrsUnits;
     RETURN NULL;
END;
$tr_Course_DIUA$ LANGUAGE plpgsql;
-- Create Trigger
CREATE TRIGGER tr_Course_DIUA
     AFTER INSERT OR UPDATE OR DELETE
     ON Course
     FOR EACH ROW EXECUTE FUNCTION tr_Course_DIUA();
-- Testing Statements
INSERT INTO Course (CourseNo, CrsDesc, CrsUnits)
VALUES ('IS485','Advanced Database Management',4);
UPDATE Course
 SET CrsUnits = 3
 WHERE CourseNo = 'IS485';
DELETE FROM Course
 WHERE CourseNo = 'IS485';
DROP TRIGGER tr_Course_DIUA ON Course;
DROP FUNCTION tr_Course_DIUA;
ROLLBACK;  -- Use reversing statements if auto commit is on
```

Example 11.P2 depicts a complex integrity constraint involving a table (*Offering*) related to the trigger table (*Enrollment*). The DECLARE section in the trigger function uses anchored data types like PL/SQL. The INTO clause of the SELECT statement uses trigger function variables for the statement result. The RAISE statement in the trigger function uses the EXCEPTION level, indicating a constraint violation aborting the action firing the trigger. The last line in the trigger function uses RETURN NEW, indicating no error detected.

Example 11.P2 – Trigger to ensure that a seat remains in an offering. Equivalent to example 11.26.

```
-- Create Trigger function
CREATE OR REPLACE FUNCTION tr_Enrollment_IB()
     RETURNS TRIGGER AS $tr_Enrollment_IB$
DECLARE
     anOffLimit Offering.OffLimit%TYPE;
     anOffNumEnrolled Offering.OffNumEnrolled%TYPE;

BEGIN
  SELECT OffLimit, OffNumEnrolled
     INTO anOffLimit, anOffNumEnrolled
     FROM Offering
     WHERE Offering.OfferNo = NEW.OfferNo;

  IF anOffNumEnrolled >= anOffLimit THEN
     RAISE EXCEPTION 'No seats remaining in offering %. Number
          enrolled: %. Offering limit: %.',
          NEW.OfferNo,anOffNumEnrolled, anOffLimit;
     END IF;
     RETURN NEW;
END;
$tr_Enrollment_IB$ LANGUAGE plpgsql;
-- Create Trigger
-- This trigger ensures that the number of enrolled
-- students is less than the offering limit.
CREATE TRIGGER tr_Enrollment_IB
     BEFORE INSERT
     ON Enrollment
     FOR EACH ROW EXECUTE FUNCTION tr_Enrollment_IB();
```

```
-- Testing Statements
-- Insert the last student
-- Set auto commit off and auto rollback on error.
INSERT INTO Enrollment (RegNo, OfferNo, EnrGrade)
     VALUES (1234,5679,0);
-- update the number of enrolled students
UPDATE Offering
     SET OffNumEnrolled = OffNumEnrolled + 1
     WHERE OfferNo = 5679;
-- See offering limit and number enrolled
SELECT OffLimit, OffNumEnrolled
     FROM Offering
     WHERE Offering.OfferNo = 5679;
-- Insert a student beyond the limit
INSERT INTO Enrollment (RegNo, OfferNo, EnrGrade)
     VALUES (1236,5679,0);
SELECT *
 FROM Enrollment
 WHERE OfferNo = 5679 and RegNo = 1234 ;
-- Remove new Enrollment row and Offering update if auto commit
-- no rollback
DELETE FROM Enrollment
 WHERE RegNo = 1234 AND OfferNo = 5679;
-- Reduced OffNumbEnrolled by 1
UPDATE Offering
     SET OffNumEnrolled = OffNumEnrolled - 1
     WHERE OfferNo = 5679;
ROLLBACK;
```

Example 11.P3 depicts update propagation involving a table (*Offering*) related to the trigger table (*Enrollment*). The EXCEPTION section aborts the action firing the trigger with the RAISE EXCEPTION statement. The last line in the trigger function uses RETURN NULL, the normal return value for a AFTER ROW trigger function.

Example 11.P3 – Trigger to update the number of enrolled students in an offering. Equivalent to example 11.27.

```
-- Create Trigger function
CREATE OR REPLACE FUNCTION tr_Enrollment_IA()
     RETURNS TRIGGER AS $tr_Enrollment_IA$
BEGIN
     BEGIN
            UPDATE Offering
                   SET OffNumEnrolled = OffNumEnrolled + 1
                   WHERE OfferNo = NEW.OfferNo;
            EXCEPTION
                   WHEN OTHERS THEN
                          RAISE EXCEPTION 'Database error';
                   RETURN NULL;
     END;
     RETURN NULL;
END;
$tr_Enrollment_IA$ LANGUAGE plpgsql;

-- This trigger updates the number of enrolled
-- students in the related offering row.

CREATE TRIGGER tr_Enrollment_IA
AFTER INSERT
ON Enrollment
FOR EACH ROW EXECUTE FUNCTION tr_Enrollment_IA();
```

```
-- Testing statements
-- See the offering limit and number enrolled
SELECT OffLimit, OffNumEnrolled
 FROM Offering
 WHERE Offering.OfferNo = 5679;

-- Insert the last student
INSERT INTO Enrollment (RegNo, OfferNo, EnrGrade)
VALUES (1234,5679,0);

-- See the offering limit and number enrolled
SELECT OffLimit, OffNumEnrolled
 FROM Offering
 WHERE Offering.OfferNo = 5679;

ROLLBACK; -- Use a DELETE statement to reverse if auto commit
```

The last two examples involve compound triggers extending triggers in 11.P2 and 11.P3. Triggers in both examples use the OR keyword for compound events including UPDATE OF OfferNo. The trigger function for Example 11.P5 uses PG_OP, a PostgreSQL special variable to determine the action firing the trigger. PostgreSQL provides other special variables for trigger details although PG_OP seems the most useful in practice. Example 11.30 provides the testing script for both Examples 11.P4 and 11.P5.

Example 11.P4 – Trigger to ensure that a seat remains in an offering when inserting or updating an *enrollment* row. Equivalent to Example 11.28.

```
-- Drop the previous triggers to avoid interactions
DROP TRIGGER IF EXISTS tr_Enrollment_IB  ON Enrollment;
DROP FUNCTION IF EXISTS tr_Enrollment_IB;

CREATE OR REPLACE FUNCTION tr_Enrollment_IUB()
     RETURNS TRIGGER AS $tr_Enrollment_IUB$
DECLARE
    anOffLimit Offering.OffLimit%TYPE;
    anOffNumEnrolled Offering.OffNumEnrolled%TYPE;
BEGIN
  SELECT OffLimit, OffNumEnrolled
    INTO anOffLimit, anOffNumEnrolled
    FROM Offering
    WHERE Offering.OfferNo = NEW.OfferNo;

  IF (anOffNumEnrolled >= anOffLimit) THEN
    RAISE EXCEPTION
      'No seats remaining in offering %. Number enrolled: %.
      Offering limit: %.',
      NEW.OfferNo,anOffNumEnrolled, anOffLimit;
  END IF;
  RETURN NEW;
END;
$tr_Enrollment_IUB$ LANGUAGE plpgsql;

-- Create Trigger
-- This trigger ensures that the number of enrolled
-- students is less than the offering limit.
CREATE TRIGGER tr_Enrollment_IUB
BEFORE INSERT OR UPDATE OF OfferNo ON Enrollment
FOR EACH ROW EXECUTE FUNCTION tr_Enrollment_IUB();
```

Example 11.P5 – Trigger to update the number of enrolled students in an offering when inserting, updating, or deleting an *enrollment* row. Equivalent to Example 11.29.

```
-- Drop the previous triggers to avoid interactions
DROP TRIGGER IF EXISTS tr_Enrollment_IA  ON Enrollment;
DROP FUNCTION IF EXISTS tr_Enrollment_IA;

-- Create Trigger function
CREATE OR REPLACE FUNCTION tr_Enrollment_DIUA()
     RETURNS TRIGGER AS $tr_Enrollment_DIUA$
BEGIN
     -- Increment the number of enrolled students for insert and update
     IF (TG_OP = 'UPDATE' or TG_OP = 'INSERT') THEN
        UPDATE Offering
          SET OffNumEnrolled = OffNumEnrolled + 1
          WHERE OfferNo = NEW.OfferNo;
     END IF;
     -- Decrease the number of enrolled students for delete and update
     IF (TG_OP = 'UPDATE' or TG_OP = 'DELETE') THEN
        UPDATE Offering
             SET OffNumEnrolled = OffNumEnrolled - 1
             WHERE OfferNo = OLD.OfferNo;
     END IF;

     RETURN NULL;
END;
$tr_Enrollment_DIUA$ LANGUAGE plpgsql;

-- Create Trigger
-- This trigger updates the number of enrolled
-- students the related offering row.
CREATE TRIGGER tr_Enrollment_DIUA
AFTER INSERT OR DELETE OR UPDATE OF OfferNo ON Enrollment
FOR EACH ROW EXECUTE FUNCTION tr_Enrollment_DIUA();
```

These trigger examples indicate similarity between Oracle and PostgreSQL triggers. The PL/pgSQL language shares syntax with PL/SQL in many statements. PL/pgSQL code in trigger functions looks like PL/SQL code in the body of Oracle triggers. The textbook website contains the entire set of example triggers in the chapter and problem solution triggers. Self-study of these examples and problem solutions should help students write PostgreSQL triggers and related trigger functions.

APPENDIX 13.A: DETAILS OF THE SCHEMA INTEGRATION PROBLEM

This appendix provides details of the schema integration problem not covered in Section 13.4.1. You need to use sample tables (Tables 13A-1 to 13A-4) in the last step of the schema integration process to populate sample tables in the data warehouse design. You also need to use rows in the worksheet for custom product purchases (Table 13A-5).

TABLE 13A-1
Sample Rows for the *Supplier* Table

SuppNo	SuppName	SuppEmail	SuppPhone	SuppDisc
S2029929	ColorMeg, Inc.	custrel@colormeg.com	(720) 444-1231	0.10
S3399214	Connex	help@connex.com	(206) 432-1142	0.12
S4290202	Ethlite	ordering@ethlite.com	(303) 213-2234	0.05
S4298800	Intersafe	orderdesk@intersafe.com	(512) 443-2215	0.10
S4420948	UV Components	custserv@uvcomponents.com	(303) 321-0432	0.08
S5095332	Cybercx	orderhelp@cybercx.com	(212) 324-5683	0.00

TABLE 13A-2
Sample Rows for the *Product* Table

ProdNo	ProdName	SuppNo	ProdQOH	ProdPrice	ProdNextShipDate
P0036566	17 inch Color Monitor	S2029929	12	$169.00	02/20/2020
P0036577	19 inch Color Monitor	S2029929	10	$319.00	02/20/2020
P1114590	R3000 Color Laser Printer	S3399214	5	$699.00	01/22/2020
P1412138	10 Foot Printer Cable	S4290202	100	$12.00	
P1445671	8-Outlet Surge Protector	S4298800	33	$14.99	
P1556678	CVP Ink Jet Color Printer	S3399214	8	$99.00	01/22/2020
P3455443	Color Ink Jet Cartridge	S3399214	24	$38.00	01/22/2020
P4200344	36-Bit Color Scanner	S4420948	16	$199.99	01/29/2020
P6677900	Black Ink Jet Cartridge	S3399214	44	$25.69	
P9995676	Battery Back-up System	S5095332	12	$89.00	02/01/2020

TABLE 13A-3
Sample Rows for the *Purchase* Table

PurchNo	PurchDate	SuppNo	PurchPayMethod	PurchDelDate
P2224040	02/03/2020	S2029929	Credit	02/08/2020
P2345877	02/03/2020	S5095332	PO	02/11/2020
P3249952	02/04/2020	S3399214	PO	02/09/2020
P3854432	02/03/2020	S4290202	PO	02/08/2020
P9855443	02/07/2020	S4420948	PO	02/15/2020

TABLE 13A-4
Sample Rows for the *PurchLine* Table

PurchNo	ProdNo	PLQty	PLUnitCost
P2224040	P0036566	10	$100.00
P2224040	P0036577	10	$200.00
P2345877	P9995676	10	$45.00
P3249952	P1114590	15	$450.00
P3249952	P1556678	10	$50.00
P3249952	P3455443	25	$21.95
P3249952	P6677900	25	$12.50
P3854432	P1412138	50	$6.50
P9855443	P4200344	15	$99.00

TABLE 13A-5
Sample Worksheet for Custom Inventory

ProdCode	ProdDesc	Supp	Qty	Unit Price	PurDate	Amount
CPC1	Pencils	Omart	20	$2.00	13-Feb-2020	$40.00
CPC2	Paper	Smart	10	$3.50	14-Feb-2020	$35.00
CPC3	Folders	Pmart	20	$1.50	11-Feb-2020	$30.00

To estimate grain size, you should use these estimates about cardinalities of tables and unique values of some columns.

- Product rows: 1,000
- Supplier rows: 100
- Purchase rows: 100,000 per year
- PurchLine rows: 500,000 per year
- Spreadsheet rows: 1,000 per month; new spreadsheet each month
- Unique products in a spreadsheet for one year: 100
- Unique suppliers in a spreadsheet for one year: 20

APPENDIX 13.B: SOLUTION FOR THE SCHEMA INTEGRATION PROBLEM

1. The dimensions in the problem are reasonably clear. Supplier, calendar, and product are dimensions. Supplier and product come from the ERD and the sample spreadsheet. The calendar dimension is a standard data warehouse dimension. Calendar is a hierarchical dimension. Phone and email can be parsed to be hierarchical as part of the supplier dimension.
   - Supplier
     - SuppNo: ERD only
     - SuppName (Supplier table) | Supp (spreadsheet)
     - SuppPhone: ERD only; hierarchical (country code → area code → prefix → line)
     - SuppEmail: ERD only; hierarchical (top level domain → second level domain → local part)
   - Calendar
     - Date columns in the ERD (ProdNextShipDate, PurchDelDate, and PurchDate) and spreadsheet (PurchDate); hierarchical (year → month → day)
   - Product:
     - ProdNo: ERD only
     - ProdName (ERD) | ProdDesc (spreadsheet)
     - ProdCode: spreadsheet only
2. The measures mostly come from the *PurchLine* table and supply purchases spreadsheet. Measures from related tables are important to associate with the measures from the *PurchLine* table and Supply Purchases spreadsheet.
   - PLQty (PurchLine table) | Qty (spreadsheet); additive measure
   - Amount of purchase: derived additive measure from the spreadsheet
   - PLUnitCost (PuchLine table) | Unit Price (spreadsheet); snapshot measure
   - ProdQOH (Product table) | Stock (Spreadsheet): Semi-additive across products but not useful to add quantity of different products. Usually average across time periods
   - SuppDisc (Supplier table): supplier discount; snapshot measure
   - ProdPrice: product price; snapshot measure indicating the resale price of the product when the purchase occurs
3. The most detailed grain is the combination of individual supplier, individual product, and date.
   - 1,100 products: sum of product rows and unique products in a spreadsheet
   - 120 suppliers: sum of supplier rows and unique suppliers in a spreadsheet
   - Days per year: 365
   - 512,000 purchases of individual products: sum of PurchLine rows and spreadsheet rows (one year)
   - Fact table size is determined from sum of the rows in the PurchLine table and Spreadsheet. Thus, the individual product purchases per year are 512,000.
   - Sparsity estimate:
     - 1 - (fact table size / product of dimensions)
     - (1 – (512,000 / (1,100*120*365)) = 0.98937
     - The data cube has mostly missing cells with slightly more than 1% of cells with non-zero values.

4. The star schema (Figure 13B.1) should support the dimensions and measures specified in parts 1 and 2. There are two relationships between the *Calendar* and *InvFact* tables to record both the purchase and delivery dates. Product type is a new derived column indicating the data source (merchandise for resale or supply for internal usage). *ProdNextShipDate* was dropped in the data warehouse design. The problem did not indicate a clear usage the data warehouse. It could be added as another relationship from *Calendar* to *InvFact* if the date was useful for business intelligence reasoning. The relationship would be incomplete for the spreadsheet data source.

   The star schema design involves design transformations, flatten and merge. The InvFact table involves a flatten transformation of the Purchase and PurchLine tables. The date relationships (PurchDate and DelDate) group products purchased together. The PurchNo column can be added to the InvFact table to provide a link to source data in the Purchase table.

   The merge transformation was applied to combine (1) Supplier table and Supp column in the spreadsheet, (2) Product table and spreadsheet columns (ProdCode and ProdDesc), and (3) flattened table from the Purchase database (InvFact) with spreadsheet columns (Qty, Unit Price, PurchaseDate, and Stock).

5. The *DelDate* relationship is an incomplete fact-dimension relationship as the delivery date is missing in the supply spreadsheet. It is probably not possible to add to existing data but second data source possibly can be changed in the future so delivery date is collected on the spreadsheet. If delivery date is the same as purchase date for supplies, the same date can be used as a default value.

   There are also missing values for *SuppEmail* and *SuppPhone* for the suppliers from the spreadsheet. Although the Supplies relationship is mandatory, these missing values make the relationship missing for the *SuppEmail* and *SuppPhone* columns. Additional data collection can resolve this incompleteness as no reliable default value exists.

6. The data warehouse tables (Tables 13B-1 to 13B-5) have been derived from the sample rows in the source tables and spreadsheet. The delivery date for the supply purchases uses the default value of the purchase date since the values are missing the source data. New primary key values have been generated for data from the spreadsheet data source.

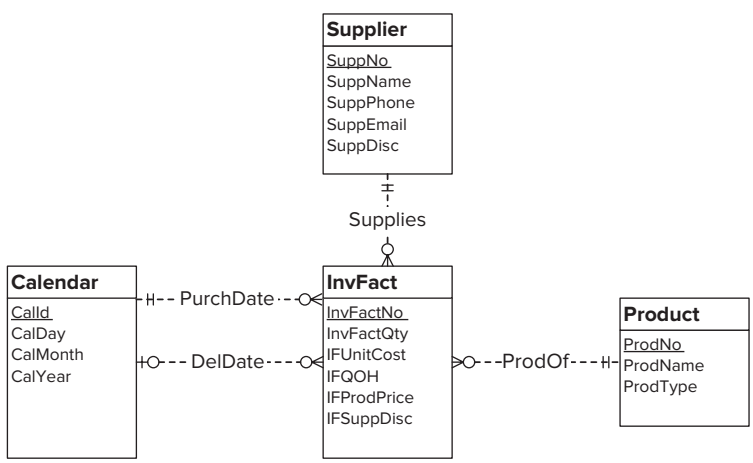

FIGURE 13B.1
ERD for Retail Fitness Database

SuppNo	SuppName	SuppEmail	SuppPhone
S2029929	ColorMeg, Inc.	custrel@colormeg.com	(720) 444-1231
S3399214	Connex	help@connex.com	(206) 432-1142
S4290202	Ethlite	ordering@ethlite.com	(303) 213-2234
S4298800	Intersafe	orderdesk@intersafe.com	(512) 443-2215
S4420948	UV Components	custserv@uvcomponents.com	(303) 321-0432
S5095332	Cybercx	orderhelp@cybercx.com	(212) 324-5683
S1111111	Omart	[null]	[null]
S1111112	Smart	[null]	[null]
S1111113	Pmart	[null]	[null]

TABLE 13B-1
Sample Rows for the *Supplier* Dimension Table

ProdNo	ProdName	ProdType
P0036566	17 inch Color Monitor	Merch
P0036577	19 inch Color Monitor	Merch
P1114590	R3000 Color Laser Printer	Merch
P1412138	10 Foot Printer Cable	Merch
P1445671	8-Outlet Surge Protector	Merch
P1556678	CVP Ink Jet Color Printer	Merch
P3455443	Color Ink Jet Cartridge	Merch
P4200344	36-Bit Color Scanner	Merch
P6677900	Black Ink Jet Cartridge	Merch
P9995676	Battery Back-up System	Merch
P111111	No 2 pencils	Supp
P111112	Copier paper	Supp
P111113	File folders	Supp

TABLE 13B-2
Sample Rows for the *Product* Dimension Table

CalId	CalDay	CalMonth	CalYear
C10000211	1	2	2020
C10000212	2	2	2020
C10000213	3	2	2020
C10000214	4	2	2020
C10000215	5	2	2020
C10000216	6	2	2020
C10000217	7	2	2020
C10000218	8	2	2020
C10000219	9	2	2020
C10000220	10	2	2020
C10000221	11	2	2020
C10000222	12	2	2020
C10000223	13	2	2020
C10000224	14	2	2020
C10000225	15	2	2020
C10000226	16	2	2020
C10000227	17	2	2020

TABLE 13B-3
Sample Rows for the *Calendar* Dimension Table

TABLE 13B-4
Sample Rows for the *InvFact* Measure Table (Part 1)

InvFactNo	ProdNo	SuppNo	IFQty	IFUnitCost	IFQOH	IFProdPrice	IFSuppDisc
I2224040	P0036566	S2029929	10	$100.00	12	$169.00	0.10
I2224041	P0036577	S2029929	10	$200.00	10	$319.00	0.10
I2224042	P9995676	S5095332	10	$45.00	12	$89.00	0.00
I2224043	P1114590	S3399214	15	$450.00	5	$699.00	0.12
I2224044	P1556678	S3399214	10	$50.00	8	$99.00	0.12
I2224045	P3455443	S3399214	25	$21.95	24	$38.00	0.12
I2224046	P6677900	S3399214	25	$12.50	44	$25.69	0.12
I2224047	P1412138	S4290202	50	$6.50	100	$12.00	0.05
I2224048	P4200344	S4420948	15	$99.00	16	$199.99	0.08
I2224049	P111111	S1111111	20	$2.00	1	[null]	[null]
I2224050	P111112	S1111112	10	$3.50	2	[null]	[null]
I2224051	P111113	S1111113	20	$1.50	0	[null]	[null]

TABLE 13B-5
Sample Rows for the *InvFact* Measure Table (Part 2)

InvFactNo	PurchCalNo	DelCalNo
I2224040	C10000213	C10000218
I2224041	C10000213	C10000218
I2224042	C10000213	C10000221
I2224043	C10000214	C10000219
I2224044	C10000214	C10000219
I2224045	C10000214	C10000219
I2224046	C10000214	C10000219
I2224047	C10000213	C10000218
I2224048	C10000217	C10000225
I2224049	C10000223	C10000223
I2224050	C10000224	C10000224
I2224051	C10000221	C10000221

APPENDIX 14.A: CREATE TABLE STATEMENTS FOR SECTION 14.3.5

This appendix contains statements to create tables used in the examples of Section 14.3.5 for the MERGE, multiple table INSERT, and INSERT …. ON CONFLICT statements. A document on the textbook's website contains complete statements to create (CREATE TABLE) and populate (INSERT) these tables.

```
-- Create table statements for MERGE examples in Section
-- 14.3.5 Also used for Examples 14.5 and 14.6
CREATE TABLE SSCustomer
(     CustId      CHAR(8),
      CustName    VARCHAR(30),
      CustPhone   VARCHAR(15),
      CustStreet  VARCHAR(50),
      CustCity    VARCHAR(30),
      CustState   VARCHAR(20),
      CustZip     VARCHAR(10),
      CustNation  VARCHAR(20),
 CONSTRAINT PKSSCustomer PRIMARY KEY (CustId)  );

CREATE TABLE SSCustomerChanges
(     CustId      CHAR(8),
      CustName    VARCHAR(30),
      CustPhone   VARCHAR(15),
      CustStreet  VARCHAR(50),
      CustCity    VARCHAR(30),
      CustState   VARCHAR(20),
      CustZip     VARCHAR(10),
      CustNation  VARCHAR(20),
 CONSTRAINT PKSSCustomerChanges PRIMARY KEY (CustId)  );

CREATE TABLE SSCustomerChanges2
(     CustId      CHAR(8),
      CustName    VARCHAR(30),
      CustPhone   VARCHAR(15),
      CustStreet  VARCHAR(50),
      CustCity    VARCHAR(30),
      CustState   VARCHAR(20),
      CustZip     VARCHAR(10),
      CustNation  VARCHAR(20),
 CONSTRAINT PKSSCustomerChanges2 PRIMARY KEY (CustId)
);

-- Section 14.3.5
-- Create table statements for INSERT examples in
CREATE TABLE ProductSale
( Product_ID      INTEGER NOT NULL,
  ProductName     VARCHAR(50),
  ProductCategory VARCHAR(50),
  Qtr1            INTEGER,
  Qtr2            INTEGER,
  Qtr3            INTEGER,
  Qtr4            INTEGER );
```

```
-- CREATE TABLE statements for Example 14.3
CREATE TABLE QTR1Sale
( Product_ID            INTEGER NOT NULL,
     ProductName        VARCHAR(50),
     ProductCategory    VARCHAR(50),
     Qtr1               INTEGER );

-- Create QTR2Sale table
CREATE TABLE Qtr2Sale
( Product_ID            INTEGER NOT NULL,
     ProductName        VARCHAR(50),
     ProductCategory    VARCHAR(50),
     Qtr2               INTEGER );

-- Create QTR3Sale table
CREATE TABLE Qtr3Sale
( Product_ID            INTEGER NOT NULL,
     ProductName        VARCHAR(50),
     ProductCategory    VARCHAR(50),
     Qtr3               INTEGER  );

-- Create QTR4Sale table
CREATE TABLE Qtr4Sale
( Product_ID            INTEGER NOT NULL,
     ProductName        VARCHAR(50),
     ProductCategory    VARCHAR(50),
     Qtr4               INTEGER );

-- CREATE TABLE statements for Example 14.4
CREATE TABLE ElectronicsSale
( Product_ID            INTEGER NOT NULL,
  ProductName           VARCHAR(50),
  ProductCategory       VARCHAR(50),
  TotalSales            INTEGER );

-- Create BooksSale table
CREATE TABLE BooksSale
( Product_ID            INTEGER NOT NULL,
  ProductName           VARCHAR(50),
  ProductCategory       VARCHAR(50),
  TotalSales            INTEGER );

-- Create MoviesSale table
CREATE TABLE MoviesSale
( Product_ID            INTEGER NOT NULL,
  ProductName           VARCHAR(50),
  ProductCategory       VARCHAR(50),
  TotalSales            INTEGER );
```

APPENDIX 14.B: CREATE TABLE STATEMENTS FOR CHAPTER PROBLEMS

This appendix contains SQL statements to create tables used in problems at the end of the chapter. A document on the textbook's website contains complete statements to create (CREATE TABLE) and populate (INSERT) these tables.

```
-- Create table statements for problems 14.4 and 14.13
CREATE TABLE SSItem
(     ItemId          CHAR(8),
      ItemName        VARCHAR(30),
      ItemBrand       VARCHAR(30),
      ItemCategory    VARCHAR(30),
      ItemUnitPrice   DECIMAL(12,2),
CONSTRAINT PKSSItem PRIMARY KEY (ItemId) );

CREATE TABLE SSItemChanges1
(     ItemId          CHAR(8),
      ItemName        VARCHAR(30),
      ItemBrand       VARCHAR(30),
      ItemCategory    VARCHAR(30),
      ItemUnitPrice   DECIMAL(12,2),
CONSTRAINT PKSSItemChanges1 PRIMARY KEY (ItemId) );

-- New table required for problem 14.5 and 14.14
-- Existing item changes have null values except for PK
-- and changed columns
CREATE TABLE SSItemChanges2
(     ItemId          CHAR(8),
      ItemName        VARCHAR(30),
      ItemBrand       VARCHAR(30),
      ItemCategory    VARCHAR(30),
      ItemUnitPrice   DECIMAL(12,2),
CONSTRAINT PKSSItemChanges2 PRIMARY KEY (ItemId) );

-- Tables in problems 14.6 and 14.7.
CREATE TABLE ProductSale1
( Product_ID          INTEGER NOT NULL,
  ProductName         VARCHAR(50),
  ProductCategory     VARCHAR(50),
  SalesYear           INTEGER,
  SalesAmt            INTEGER );

-- Create ProductSales2018 table
CREATE TABLE ProductSales2018
( Product_ID          INTEGER NOT NULL,
  ProductName         VARCHAR(50),
  ProductCategory     VARCHAR(50),
  SalesAmt            INTEGER );

-- Create ProductSales2019 table
CREATE TABLE ProductSales2019
( Product_ID          INTEGER NOT NULL,
  ProductName         VARCHAR(50),
  ProductCategory     VARCHAR(50),
  SalesAmt            INTEGER );

-- Create ProductSales2020 table
```

```
CREATE TABLE ProductSales2020
( Product_ID           INTEGER NOT NULL,
  ProductName          VARCHAR(50),
  ProductCategory      VARCHAR(50),
  SalesAmt             INTEGER );

-- Create ProductSales2021 table
CREATE TABLE ProductSales2021
( Product_ID           INTEGER NOT NULL,
  ProductName          VARCHAR(50),
  ProductCategory      VARCHAR(50),
  SalesAmt             INTEGER );

-- Tables for problems 14.8 and 14.9
CREATE TABLE ProductSale2
( Product_ID           INTEGER NOT NULL,
  ProductName          VARCHAR(50),
  ProductCategory      VARCHAR(50),
  Qtr1                 INTEGER,
  Qtr2                 INTEGER,
  Qtr3                 INTEGER,
  Qtr4                 INTEGER );

-- Create YEAR_LOW_SALES table
CREATE TABLE YEAR_LOW_SALES
( Product_ID           INTEGER NOT NULL,
  ProductName          VARCHAR(50),
  TotalSales           INTEGER );

-- Create YEAR_MID_SALES table
CREATE TABLE YEAR_MID_SALES
( Product_ID           INTEGER NOT NULL,
  ProductName          VARCHAR(50),
  TotalSales           INTEGER );

-- Create YEAR_HIGH_SALES table
CREATE TABLE YEAR_HIGH_SALES
( Product_ID           INTEGER NOT NULL,
  ProductName          VARCHAR(50),
  TotalSales           INTEGER );

-- Tables for problem 14.10
CREATE TABLE Mobile_Bill
( CustID               INTEGER NOT NULL,
  CurrentAmt           DECIMAL (10,2),
  PastAmt              DECIMAL (10,2),
  CONSTRAINT           MBCustId PRIMARY KEY (CustId) );

CREATE TABLE Mobile_Usage
( CustID               INTEGER NOT NULL,
  MinutesUsed          INTEGER );

-- Tables for problems 14.11 and 14.12

CREATE TABLE Mobile_Customer
( CustID      INTEGER         NOT NULL,
  CustName    VARCHAR(50)     NOT NULL,
```

```
  CustState   VARCHAR(2)      NOT NULL,
  CustType    VARCHAR(15)     NOT NULL,
  CustAge     INTEGER,
  CurrentAmt  DECIMAL(10,2),
  CONSTRAINT  PK_CustID PRIMARY KEY (CustID) );

-- Create three summary tables for customers at three
-- different levels: gold, silver, and bronze.
CREATE TABLE Mobile_Gold
( CustID      INTEGER         NOT NULL,
  CustName    VARCHAR(50)     NOT NULL,
  CustState   VARCHAR(2)      NOT NULL,
  CustType    VARCHAR(15)     NOT NULL,
  CurrentAmt  DECIMAL(10,2)   NOT NULL );

CREATE TABLE Mobile_Silver
( CustID      NUMBER(10)      NOT NULL,
  CustName    VARCHAR(50)     NOT NULL,
  CustState   VARCHAR(2)      NOT NULL,
  CustType    VARCHAR(15)     NOT NULL,
  CurrentAmt  DECIMAL(10,2)   NOT NULL );

CREATE TABLE Mobile_Bronze
( CustID      INTEGER         NOT NULL,
  CustName    VARCHAR(50)     NOT NULL,
  CustState   VARCHAR(2)      NOT NULL,
  CustType    VARCHAR(15)     NOT NULL,
  CurrentAmt  DECIMAL (10,2)  NOT NULL );

-- Create table statements for problem 14.15
CREATE TABLE SSItem
( ItemId       CHAR(8),
  ItemName     VARCHAR(30)    NOT NULL,
  ItemBrand    VARCHAR(30)    NOT NULL,
  ItemCategory VARCHAR(30)    NOT NULL,
  ItemUnitPrice DECIMAL(12,2) NOT NULL,
CONSTRAINT PKSSItem PRIMARY KEY (ItemId) );

-- New table required for problem 14.5 and 14.14
-- Existing item changes have null values except for PK
-- and changed columns
CREATE TABLE SSItemChanges2
( ItemId        CHAR(8),
  ItemName      VARCHAR(30),
  ItemBrand     VARCHAR(30),
  ItemCategory  VARCHAR(30),
  ItemUnitPrice DECIMAL(12,2),
CONSTRAINT PKSSItemChanges2 PRIMARY KEY (ItemId) );
```

APPENDIX 16.A: SQL SYNTAX SUMMARY

This appendix summarizes the SQLstandards syntax for the CREATE/DROP ROLE statements, the GRANT/REVOKE statements, the CREATE DOMAIN statement, and the CREATE ASSERTION statement. The conventions used in the syntax notation are identical to those used at the end of Chapter 3.

CREATE and DROP ROLE Statements

```
CREATE ROLE RoleName
  [ WITH ADMIN UserName { CURRENT_USER | CURRENT_ROLE } ]

DROP ROLE RoleName
```

GRANT and REVOKE Statements

```
-- GRANT statement for privileges
GRANT { <Privilege>* | ALL PRIVILEGES } ON ObjectName
  TO UserName* [ WITH GRANT OPTION ]

<Privilege>:
   { SELECT [ (ColumnName*) ] |
     DELETE  |
     INSERT [ (ColumnName*) ]       |
     REFERENCES [ (ColumnName*) ]   |
     UPDATE [ (ColumnName*) ]
     USAGE  |
     TRIGGER  |
     UNDER  |
     EXECUTE }

-- GRANT statement for roles
GRANT RoleName*
  TO UserName* [ WITH ADMIN OPTION ]

-- REVOKE statement for privileges
REVOKE [ GRANT OPTION FOR ] <Privilege>*
  ON ObjectName FROM UserName*
  [ GRANTED BY { CURRENT_USER | CURRENT_ROLE } ]
  { CASCADE | RESTRICT }

-- REVOKE statement for roles
REVOKE [ ADMIN OPTION FOR ] RoleName*
  FROM UserName*
  [ GRANTED BY { CURRENT_USER | CURRENT_ROLE } ]
  { CASCADE | RESTRICT }
```

CREATE DOMAIN and DROP DOMAIN Statements

```
CREATE DOMAIN DomainName DataType
  [ CHECK ( <Domain-Condition> ) ]

<Domain-Condition>:
  { VALUE <Comparison-Operator> Constant |
    VALUE BETWEEN Constant AND Constant  |
    VALUE IN ( Constant* )  }
```

```
<Comparison-Operator>:
  { =  |  <  |  >  |  <=  |  >=  |  <>  }

DROP DOMAIN DomainName { CASCADE | RESTRICT }
```

CREATE ASSERTION and DROP ASSERTION Statements

```
CREATE ASSERTION AssertionName
  CHECK ( <Group-Condition> )

<Group-Condition>: -- initially defined in Chapter 4 and extended in
                      Chapter 9

DROP ASSERTION AssertionName { CASCADE | RESTRICT }
```

APPENDIX 17.A: SQL SYNTAX SUMMARY

This appendix summarizes SQL syntax for the constraint timing clause, the SET CONSTRAINTS statement, the SET TRANSACTION statement, and the save point statements discussed in the chapter. The conventions used in the syntax notation are identical to those used at the end of Chapter 3.

Constraint Timing Clause

```
CREATE TABLE  TableName
 ( <Column-Definition>*  [ ,  <Table-Constraint>*  ]  )

<Column-Definition>: ColumnName DataType
 [ DEFAULT { DefaultValue | USER | NULL } ]
   [ <Column-Constraint> ]

<Column-Constraint>: [ CONSTRAINT ConstraintName ]
  { NOT NULL   |
    <Foreign-Key-Constraint>   |  -- defined in Chapter 3
    <Uniqueness-Constraint>    |  -- defined in Chapter 3
    <Check-Constraint>   } -- defined in Chapter 16
   [ <Timing-Clause> ]

<Table-Constraint>: [ CONSTRAINT ConstraintName ]
  { <Primary-Key-Constraint>  |  -- defined in Chapter 3
    <Foreign-Key-Constraint>  |  -- defined in Chapter 3
    <Uniqueness-Constraint>   |  -- defined in Chapter 3
    <Check-Constraint> } -- defined in Chapter 3
    [ <Timing-Clause> ]

<Timing-Clause>:
  { NOT  DEFERRABLE  |
    DEFERRABLE { INITIALLY IMMEDIATE |
                 INITIALLY DEFERRED } }

CREATE ASSERTION AssertionName
 CHECK ( <Group-Condition> ) [ <Timing-Clause> ]

<Group-Condition>: -- defined in Chapter 4
```

SET CONSTRAINTS Statement

```
SET CONSTRAINTS  { ALL | ConstraintName* }
 { IMMEDIATE | DEFERRED  }
```

SET TRANSACTION Statement

```
SET [LOCAL] TRANSACTION <Mode>*

<Mode>: { <Isolation-Level> | <Access-Mode> |
          <Diagnostics> }

<Isolation-Level>:  ISOLATION LEVEL
  { SERIALIZABLE    |
```

```
    REPEATABLE READ |
    READ COMMITTED  |
    READ UNCOMMITTED }

<Access-Mode>: { READ WRITE  |  READ ONLY  }

<Diagnostics>: DIAGNOSTICS SIZE Constant
```

Save Point Statements

```
SAVEPOINT <SavePointName>   -- creates a save point

RELEASE <SavePointName>   -- deletes a save point

ROLLBACK TO SAVEPOINT <SavePointName>   -- rollback to a save point
```

APPENDIX 19.A: INSERT STATEMENTS FOR N1QL

This appendix contains the complete set of INSERT statements for the collections (*Agent*, *Home*, *AgentHome*, *Agent2*, and *Home2*) used in examples in Section 19.5.2. Before inserting documents, you need to create a bucket in the Couchbase Dashboard. After setting the bucket and scope in the Query Workbench, you must create collections and primary indexes before inserting or retrieving documents. See the document on the textbook's website for precise details about creating collections and indexes in the Query Workbench of Couchbase.

```
// Create collections
CREATE COLLECTION Agent;
CREATE COLLECTION Home;
CREATE COLLECTION AgentHome;
CREATE COLLECTION Agent2;
CREATE COLLECTION Home2;

// Create indexes
CREATE PRIMARY INDEX ON Agent;
CREATE PRIMARY INDEX ON Home;
CREATE PRIMARY INDEX ON AgentHome;
CREATE PRIMARY INDEX ON Agent2;
CREATE PRIMARY INDEX ON Home2;


// Traditional table design with foreign keys
// INSERT statement adding 3 documents in the Agent
// collection
// INSERT statement can add multiple JSON documents by
// repeating VALUES clause

INSERT INTO Agent (KEY, VALUE)
VALUES ("A871111",
  {"AgentId":"A871111", "AgFirstName":"Willie",
   "AgLastName":"Jones", "AgPhone":"(720)555-1212"} ),

VALUES ("A991111",
  {"AgentId":"A991111", "AgFirstName":"Jorge",
   "AgLastName":"Lopez", "AgPhone":"(303)435-9999"} ),

VALUES ("A999222",
  {"AgentId":"A999222", "AgFirstName":"Aimee",
   "AgLastName":"Chan", "AgPhone": "(303)555-8888"} );

// One INSERT statement for all home documents
INSERT INTO Home (KEY, VALUE)
VALUES ("H111111",
 {"HomeId":"H111111","HomeNoBdrms":3,
  "HomeNoBathrms":2,"HomeAge":15,"AgentId":"A871111",
  "HomeAddr":{"City":"Denver","State":"CO","ZipCode":80113}
} ),
```

```
VALUES ("H222222",
 {"HomeId":"H2222222","HomeNoBdrms":4,
  "HomeNoBathrms":3,"HomeAge":25,"AgentId":"A871111",
  "HomeAddr":{"City":"Centennial","State":"CO",
  "ZipCode":80112} } ),

VALUES ("H333333",
 {"HomeId":"H333333","HomeNoBdrms":2,
  "HomeNoBathrms":2,"HomeAge":3,"AgentId":"A991111",
  "HomeAddr":{"City":"Denver","State":"CO","ZipCode":80104}
} ),

VALUES ("H444444",
 {"HomeId":"H444444","HomeNoBdrms":5,
  "HomeNoBathrms":3,"HomeAge":10,"AgentId":"A999222",
  "HomeAddr":{"City":"Aurora","State":"CO","ZipCode":80107}
} ),

VALUES ("H555555",
 {"HomeId":"H555555","HomeNoBdrms":3,
  "HomeNoBathrms":2,"HomeAge":null,"AgentId":"A999222",
  "HomeAddr":{"City":"Centennial","State":"CO",
  "ZipCode":80112} } ),

VALUES ("HomeH666666",
 {"HomeId":"H666666","HomeNoBdrms":4,
  "HomeNoBathrms":2,"HomeSold":false,"AgentId":"A999222",
  "HomeAddr":{"City":"Aurora","State":"CO","ZipCode":80109}
} );

// Total nested representation with homes nested inside
// agents
// INSERT statement for the AgentHome collection
INSERT INTO AgentHome (KEY, VALUE)
 VALUES ("AH871111",
  {"AgentId":"A871111", "AgFirstName":"Willie",
   "AgLastName":"Jones", "AgPhone":"(720)555-1212",
    "Home":[
     {"HomeId":"H111111","HomeNoBdrms":3,
      "HomeNoBathrms":2,"HomeAge":15,
      "HomeAddr":{"City":"Denver","State":"CO","ZipCo
de":80110}},
     {"HomeId":"H222222","HomeNoBdrms":4,
      "HomeNoBathrms":3,"HomeAge":25,
      "HomeAddr":{"City":"Centennial","State":"CO",
"ZipCode":80112}}] } ),

VALUES ("AH991111",
  {"AgentId":"A991111", "AgFirstName":"Jorge",
   "AgLastName":"Lopez", "AgPhone":"(303)435-9999",
    "Home":[
     {"HomeId":"H333333","HomeNoBdrms":2,
      "HomeNoBathrms":2,"HomeAge":3,
      "HomeAddr":{"City":"Denver","State":"CO",
"ZipCode":80104}}] } ),
```

```
VALUES ("AH999222",
  {"AgentId":"A999222", "AgFirstName":"Aimee",
   "AgLastName":"Chan", "AgPhone":"(303)555-8888",
    "Home":[
     {"HomeId":"H444444","HomeNoBdrms":5,
      "HomeNoBathrms":3,"HomeAge":10,
      "HomeAddr":{"City":"Aurora","State":"CO",
"ZipCode":80107}},
     {"HomeId":"H555555","HomeNoBdrms":3,
      "HomeNoBathrms":2,"HomeAge":null,
      "HomeAddr":{"City":"Centennial","State":"CO",
"ZipCode":80112}},
     {"HomeId":"H666666","HomeNoBdrms":4,
      "HomeNoBathrms":2,"HomeSold":false,
      "HomeAddr":{"City":"Aurora","State":"CO",
"ZipCode":80109} } ] } );

// Partial nested representation with home ids nested inside
// agents
// INSERT statement for the Agent2 collection with array of
// home ids
INSERT INTO Agent2 (KEY, VALUE)
VALUES ("A871111",
  {"AgentId":"A871111", "AgFirstName":"Willie",
   "AgLastName":"Jones", "AgPhone":"(720)555-1212",
   "HomeId":["H111111","H222222"] } ),

VALUES ("A991111",
  {"AgentId":"A991111", "AgFirstName":"Jorge",
   "AgLastName":"Lopez", "AgPhone":"(303)435-9999",
   "HomeId":["H333333"] } ),

VALUES ("A999222",
  {"AgentId":"A999222", "AgFirstName":"Aimee",
   "AgLastName":"Chan", "AgPhone": "(303)555-8888",
   "HomeId":["H444444","H555555","H666666"] } );


// INSERT statement for the Home2 collection
INSERT INTO Home2 (KEY, VALUE)
VALUES ("H111111",
 {"HomeId":"H111111","HomeNoBdrms":3,
  "HomeNoBathrms":2,"HomeAge":15,
  "HomeAddr":{"City":"Denver","State":"CO","ZipCode":80113}
} ),

VALUES ("H222222",
 {"HomeId":"H2222222","HomeNoBdrms":4,
  "HomeNoBathrms":3,"HomeAge":25,
  "HomeAddr":{"City":"Centennial","State":"CO",
"ZipCode":80112} } ),

VALUES ("H333333",
 {"HomeId":"H333333","HomeNoBdrms":2,
  "HomeNoBathrms":2,"HomeAge":3,
```

```
  "HomeAddr":{"City":"Denver","State":"CO","ZipCode":80104}
} ),

VALUES ("H444444",
 {"HomeId":"H444444","HomeNoBdrms":5,
  "HomeNoBathrms":3,"HomeAge":10,
  "HomeAddr":{"City":"Aurora","State":"CO","ZipCode":80107}
} ),

VALUES ("H555555",
 {"HomeId":"H555555","HomeNoBdrms":3,
  "HomeNoBathrms":2,"HomeAge":null,
  "HomeAddr":{"City":"Centennial","State":"CO",
"ZipCode":80112} } ),

VALUES ("H666666",
 {"HomeId":"H666666","HomeNoBdrms":4,
  "HomeNoBathrms":2,"HomeSold":false,
  "HomeAddr":{"City":"Aurora","State":"CO","ZipCode":80109}
} );
```

APPENDIX 19.B: OBJECT DATABASE FEATURES IN POSTGRESQL

PostgreSQL supports new base types with a standard set of features and storage structures for efficient usage. PostgreSQL requires tedious implementation of functions for input, output, storage, and other standard purposes for new base types. Implementation of the standard functions must be done in a programming language such as C. PostgreSQL does not support inheritance for new base types.

Beyond new base types, PostgreSQL supports a mixed set of object database features with some variation from the SQL standard. The following list summarizes PostgreSQL object features noting deviation from the SQL standard and Oracle.

- PostgreSQL supports user-defined composite types but lacks type inheritance as indicated in the SQL standard and implemented in Oracle.
- PostgreSQL supports table inheritance as subtables inherit columns and some constraints from ancestor tables. Subtables do not inherit indexes and foreign key constraints from ancestor tables.
- PostgreSQL supports set inclusion for subtable families as indicated in the SQL standard and Section 19.2. Set inclusion involves propagation of modification operations (INSERT, UPDATE, and DELETE) for subtables to ancestor tables. Support for subtable families in PostgreSQL contrasts with limited support in Oracle through type inheritance.
- PostgreSQL supports object identifiers at the storage level but lacks support in SQL statements. PostgreSQL does not support REF columns so that foreign keys must be used. PostgreSQL does not support the DEREF function because REF columns cannot be defined. Oracle supports object identifiers and REF columns.

The remainder of this appendix demonstrates PostgreSQL object features to contrast with the SQL standard and Oracle. The examples correspond to SQL standard examples in Section 19.2.

Example 19.B.1 demonstrates user-defined composite types. PostgreSQL does not support methods and functions for composite types so only variables can be defined. Since PostgreSQL lacks subtypes, the *Colorpoint* type cannot be defined as a subtype of *PointPG*.

Example 19.B.1 – *PointPG* type in PostgreSQL. Corresponds to example 19.1 except for the name change to avoid conflict with the base Point type in PostgreSQL.

```
CREATE TYPE PointPG AS
( X FLOAT,
  Y FLOAT ) ;
```

Example 19.B.2 demonstrates user-defined composite types for addresses and agents. PostgreSQL supports composite types used with columns and typed tables. A sequence provides primary key values for the typed *Agent* table because PostgreSQL does not support object identifiers for primary key columns.

Example 19.B.2 – *AddressType* and *AgentType* type and typed table *Agent* in PostgreSQL. Corresponds to example 19.6.

```
CREATE TYPE AddressType AS
 (Street    VARCHAR(50),
  City      VARCHAR(30),
  State     CHAR(2),
  Zip       CHAR(9) );

CREATE TYPE AgentType AS
(AgentNo    INTEGER,
 AName      VARCHAR(30),
 Address    AddressType,
 Phone      CHAR(13),
 Email      VARCHAR(50) );

CREATE SEQUENCE AgentNoSeq
START 1 INCREMENT 1;

CREATE TABLE Agent OF AgentType
(AgentNo PRIMARY KEY DEFAULT NEXTVAL('AgentNoSeq') );
```

Example 19.B.3 demonstrates another typed table (*Property*). Because PostgreSQL does not support REF columns with object identifiers, traditional foreign keys must be used. The *PropertyType* contains the *AgentNo* variable. In the typed *Property* table, a foreign key constraint extends the data type (*PropertyType*) used to define the table.

Example 19.B.3 – *PropertyType* and typed table *Property* in PostgreSQL. Corresponds to example 19.7. REF types not supported in PostgreSQL.

```
CREATE TYPE PropertyType AS
(PropNo     INTEGER,
 Address    AddressType,
 SqFt       INTEGER,
 PView      BYTEA, -- BYTEA is PostgreSQL data type for BLOB
 Location   Point,
 AgentNo    INTEGER );

CREATE SEQUENCE PropNoSeq
START 1 INCREMENT 1;

CREATE TABLE Property OF PropertyType
( PropNo PRIMARY KEY DEFAULT NEXTVAL('PropNoSeq'),
  AgentNo REFERENCES Agent );
```

Example 19.B.4 demonstrates table inheritance with the *Residential* and *Industrial* tables. PostgreSQL uses the INHERITS keyword at the end of a CREATE TABLE statement instead of the UNDER keyword in the SQL standard. In the *Residential* table, the ARRAY data type allows a maximum size although PostgreSQL does not check for array limits in INSERT statements.

Example 19.B.4 – *Residential* and *Industrial* subtables inheriting from *Property*. Corresponds to example 19.8.

```
CREATE TABLE Residential
(
 BedRooms    INTEGER,
 BathRooms   INTEGER,
 Assessments DECIMAL(9,2) ARRAY[6]
) INHERITS (Property);

CREATE TABLE Industrial
(  Zoning           VARCHAR(20),
   AccessDesc       VARCHAR(20),
   RailAvailable    BOOLEAN,
   Parking          VARCHAR(10)
) INHERITS (Property);
```

Example 19.B.5 demonstrates INSERT statements into subtables. PostgreSQL automatically inserts into the parent *Property* table. Since the primary keys use default values, the primary key columns do not appear in the INSERT statements. For composite data types, the ROW keyword must be used. The name of the composite data type (*AddressType*) cannot be used in INSERT statements.

Example 19.B.5 – INSERT statements for agents and properties. This example extends examples 19.9 and 19.11.

```
INSERT INTO Agent
(AName, Address, Email, Phone)
VALUES ('Sue Smith',
         ROW('123 Any Street', 'Denver', 'CO', '80217'),
         'sue.smith@anyisp.com', '13031234567');

INSERT INTO Residential
(Address, SqFt, AgentNo, BedRooms, BathRooms, Assessments)
VALUES( ROW('123 Any Street', 'Denver', 'CO', '80217'),
        2000, CurrVal('AgentNoSeq'), 3, 2, ARRAY[190000, 200000] );

INSERT INTO Industrial
(Address, SqFt, AgentNo, Zoning, AccessDesc, RailAvailable, Parking)
VALUES( ROW('123 Big Street', 'Parker', 'CO', '80234'),
   4000, CurrVal('AgentNoSeq'), 'A1', 'Street', FALSE, 'Large lot' );
```

Example 19.B.6 demonstrates INSERT statements into subtables like Example 19.B.5. The current values of the sequences advance after each INSERT statement. The current value of the *AgentNoSeq* sequence is the same in both INSERT statements for the *Residential* table.

Example 19.B.6 – INSERT statements for agents and properties. This example extends examples 19.10 and 19.11.

```
INSERT INTO Agent
(AName, Address, Email, Phone)
VALUES ('John Smith',
         ROW('123 Big Street', 'Boulder', 'CO', '80217'),
          'john.smith@bigisp.com', '13034567123');

INSERT INTO Residential
(Address, SqFt, AgentNo, BedRooms, BathRooms, Assessments)
VALUES( ROW('123 Big Street', 'Denver', 'CO', '80203'),
        2000, CurrVal('AgentNoSeq'), 2, 3, ARRAY[200000, 190000] );

INSERT INTO Industrial
(Address, SqFt, AgentNo, Zoning, AccessDesc, RailAvailable, Parking)
VALUES( ROW('123 Any Road', 'Parker', 'CO', '80238'),
           3000, CurrVal('AgentNoSeq'), 'A2', 'Strip mall', TRUE,
           'Small lot' );
```

Example 19.B.7 demonstrates an UPDATE statement for the *Residential* subtable. The UPDATE statement subtracts 1 from the current value of each sequence to obtain primary key values for the last *Residential* row and first *Agent* row.

Example 19.B.7 – UPDATE statement to change the AgentNo in the last *Residential* row. for agents and properties. This example extends example 19.12.

```
UPDATE Residential
  SET AgentNo = CurrVal('AgentNoSeq') - 1
  WHERE PropNo = CurrVal('PropNoSeq') - 1;
```

Example 19.B.8 retrieves rows to demonstrate row propagation in subtable families. The INSERT statements in Examples 19.B.5 and 19.B.6 use the child tables (*Residential* and *Industrial*) as target tables. Row propagation also inserts rows into the parent table. The SELECT statement for *Residential* also demonstrates the result of the UPDATE statement in Example 19.B.7.

Example 19.B.8 – Retrieve rows for all tables in the table hierarchy. The parent *Property* table contains 4 rows. Each child table (*residential* and *industrial*) contain 2 rows.

```
SELECT * FROM Property;
SELECT * FROM Residential;
SELECT * FROM Industrial;
```

Example 19.B.9 demonstrates path expressions to reference components of composite types. Note that parentheses must surround composite type references including any table aliases such as `(P.Address).City`. Because PostgreSQL does not support REF columns and path expressions to combine tables, a traditional join condition must be used. Path expressions only apply to columns with composite types, not references among tables.

Example 19.B.9 – Retrieve components of columns with composite data types. This example extends example 19.13.

```
SELECT PropNo, (P.Address).City AS PropertyCity,
       (A.Address).City AS AgentCity
 FROM Property P, Agent A
 WHERE A.AName = 'John Smith'
            AND P.AgentNo = A.AgentNo;
```

Example 19.B.10 demonstrates the ONLY keyword to retrieve from selected tables in a subtable family. The SELECT statement retrieves the two rows in the *Residential* table but not the additional two rows in the parent *Property* table.

Example 19.B.10 – Use the ONLY keyword to retrieve from a subtable. This example corresponds to example 19.14.

```
SELECT PropNo, Address, Location, AgentNo
 FROM ONLY (Residential)
          WHERE Sqft > 1500;
```

These examples demonstrate that subtable families are the most important object feature of PostgreSQL for most organizations. Implementing a base user-defined type involves a major software development effort and data type expertise. Most organizations do not possess this level of expertise. Subtable families allow direct conversion of generalization hierarchies in an ERD rather than indirect conversion required without table hierarchies. However, performance of subtable families is uncertain so testing should be used in large databases to check for performance degradation. Composite types and typed tables seem burdensome without type inheritance, so the author advises against using composite types in most cases.