CHAPTER II – IMPLEMENTATION DATA MODELS

Section 1. The Relational Data Model

In this section we discuss the data model that is the most widely used in current commercial database management systems -- the relational model. The relational data model was proposed by E.F. Codd in 1970\(^1\) and is based on the mathematical theory of relations; Codd adapted this theory for use in databases. In the relational model entities/classes and relationships/associations between classes are represented by mathematical relations\(^2\). Recall that a mathematical relation between sets \(D_1, \ldots, D_n\) is a subset \(R \subseteq D_1 \times D_2 \times \ldots \times D_n\); that is

\[
R \subseteq D_1 \times D_2 \times \ldots \times D_n = \{(d_1, d_2, \ldots, d_n) : d_j \in D_j, \ j=1, \ldots, n\}
\]

Here each \(D_i\) is a domain of the relation and the number \(n\) is known as the \textit{degree} (or \textit{arity}) of the relation. Each element \((d_1, d_2, \ldots, d_n)\) of \(R\) is known as an \textit{n-tuple} (or simply \textit{tuple}).

- The number of tuples in a relation is known as the \textit{cardinality} of the relation. For a given tuple \(r = (r_1, \ldots, r_n)\) of a relation \(R\), the value \(r_j\) is called the \textit{jth component} (or \textit{jth element}) of \(r\).

When the cardinality of a relation is finite (and relatively small!) it is common to represent a relation \(R\) as a table, with each tuple \(r \in R\) appearing as a row in the table, and the \(j\)th component being listed in the \(j\)th column of the table.

\[
\begin{array}{|c|c|c|c|c|}
\hline
  & D_1 & D_2 & D_3 & D_4 & D_5 \\
\hline
r_{11} & r_{12} & r_{13} & r_{14} & r_{15} \\
\hline
r_{21} & r_{22} & r_{23} & r_{24} & r_{25} \\
\hline
r_{31} & r_{32} & r_{33} & r_{34} & r_{35} \\
\hline
r_{41} & r_{42} & r_{43} & r_{44} & r_{45} \\
\hline
\end{array}
\]

For conciseness, we shall use the notation \(R(D_1, D_2, \ldots, D_n)\) to represent a relation between domains \(D_1, D_2, \ldots, D_n\).

Applications of Relations to Data Modeling – The Relational Model

In the relational data model all classes and associations between classes are represented as relations. When using a relation to represent a class, each attribute corresponds to a component of the relation and each instance of a class is represented as a tuple of the relation. It is common to use attribute

\[\text{---}
\]


\(^{2}\) To avoid using awkward constructs such as entity/class or relationship/association, we shall use one of each pair to represent the pair, say “class” instead of “entity/class” and “association” instead of “relationship/association.”
names for components rather than the names of the associated domain, and to give the relation the name of the class being represented. Whenever it is necessary to refer to the associated domain for an attribute A, we shall use the notation \( \text{dom}(A) \) to reference this domain.

**Example:** In a data model for a college or university, a class \( \text{STUDENT} \) having attributes \( \text{StID} \), \( \text{Name} \), \( \text{Address} \), \( \text{Major} \), and \( \text{GPA} \) would be represented in the relational model by the relation

\[
\text{STUDENT(StID, Name, Address, Major, GPA)}
\]

If we represent this relation with a table it would have the structure

<table>
<thead>
<tr>
<th>StID</th>
<th>Name</th>
<th>Address</th>
<th>Major</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It is common to refer to the notation \( R(D_1, D_2, \ldots, D_n) \) as the *relational schema* for relation \( R \).

By the definition of a relation, there exists an ordering among the domains of a relation. Consequently, if we rearrange the way the domains are given for a relation, we get another relation. Thus the relations

\[
\begin{array}{cc}
\text{M} & \text{N} \\
\Gamma_1^M & \Gamma_1^N \\
\Gamma_2^M & \Gamma_2^N \\
\Gamma_3^M & \Gamma_3^N \\
\end{array}
\quad
\begin{array}{cc}
\text{N} & \text{M} \\
\Gamma_1^N & \Gamma_1^M \\
\Gamma_2^N & \Gamma_2^M \\
\Gamma_3^N & \Gamma_3^M \\
\end{array}
\]

are not the same. In a relational data model, however, columns represent attributes and hence are usually referenced by the attribute’s name rather than by position. Consequently, column ordering generally is not regarded as significant. One can rearrange the columns of a relation and still retain the same relation.

In the relational data model, every relation must have an attribute or set of attributes that serve as a primary key. In representing a relation in the form \( R(A_1, A_2, \ldots, A_n) \) or as a table, we shall always underline the names of the primary key attributes.

**Section 2. A Definition Language for Relational Data Models – Data Definition in SQL**

SQL (Structured Query Language; pronounced “sequel” by some, or simply “S-Q-L” by others) is a data language that has become the standard language for working with data in relational databases. Although the word “query” is prominent in the name of SQL, the language has constructs that support data definition (that is defining the structure of a relational data model) and data manipulation (inserting, deleting, modifying, and querying the data in the database). At this time we are going to focus on some of the most structures for data definition in SQL – those for specifying domains and relations (known as *tables* in SQL).
Fundamental Data Types

The main data types available for attributes and domain definition in SQL include numeric, character string, date and time types:

- **Numeric types** include various sizes of integers (INTEGER, INT, or SMALLINT), various precisions of real numbers (FLOAT, REAL, and DOUBLE PRECISION), and formatted numbers (DECIMAL(i,j), DEC(i,j), or NUMERIC(i,j), where i, the precision, is the total number of decimal digits and j, the scale, is the number of digits after the decimal point. The default scale is 0).

- **Character string** data types include fixed length character strings (CHAR(n) or CHARACTER(n), where n is the number of characters), or varying length (VARCHAR(n), or CHAR VARYING(n), or CHARACTER VARYING(n), where n is the maximum length).

- For **date** and **time** types, the type DATE has ten positions in the form YYYY-MM-DD and the type TIME has eight positions in the form HH:MM:SS. There are other types for working with dates and times, but these two will be sufficient for our purposes.

Domain Definition in SQL

Domains can be created using the CREATE DOMAIN construct:

```
CREATE DOMAIN domain-name AS data-type;
```

**Example:** To create a domain NAME_DOM for representing names in a relational data model we can use the construct

```
CREATE DOMAIN NAME_DOM AS VARCHAR(30);
```

Simple Table Definition in SQL

The CREATE TABLE construct can be used to specify relations in SQL. In its simplest form one specifies the name of the relation/table and gives its attributes (name and type/domain) as follows:

```
CREATE TABLE relation-name

   (attribute_1 type/domain,
    attribute_2 type/domain,
    ...
    attribute_n type/domain);
```

**Example:** Suppose we are given a relation STUDENT with attributes StID, StName, Address, Major, and GPA. Suppose further that we have domains ID_DOM, NAME_DOM, ADDRESS_DOM, and MAJOR_DOM defined as follows

```
CREATE DOMAIN ID_DOM AS VARCHAR(11);
CREATE DOMAIN NAME_DOM AS VARCHAR(30);
CREATE DOMAIN ADDRESS_DOM AS VARCHAR(40);
CREATE DOMAIN MAJOR_DOM AS VARCHAR(4);
```

The relation STUDENT can then be created via the statement

---

3 Although the CREATE DOMAIN construct has appeared as a construct in “standard” SQL since 1992, in fact few databases actually implement it. For example it is not supported in Oracle, MS Access or MS SQLServer.
CREATE TABLE STUDENT
(
StID ID_DOM,
StName NAME_DOM,
Address ADDRESS.Dom,
Major MAJOR.Dom,
GPA numeric(4,3)
);

Note that for GPA, rather than associate a domain with the attribute we specified its type directly.

Attribute and Table Constraints – Designation of Primary Keys

In the relational data model, constraints are conditions that are imposed on various structures in the model. In some cases the constraints are imposed to make the data model represent its enterprise more accurately; in other cases, especially when the data model is realized as a database, the constraints are imposed to ensure that the data in the database does not become inconsistent as new data are introduced into the database, as data are removed from the database, or as data are modified.

In representing relational data models in SQL it is possible to associate constraints with attributes, with tables, or with the data model as a whole. For now, our major use of attribute and tables constraints will be to designate a primary key for a table. Later we shall describe other types of constraints.

1. **Attribute constraints**:

   a. **Primary Key constraint**: For a table that has a single attribute for its primary key, this can be specified via the PRIMARY KEY constraint on the attribute. The general format for such a constraint is:

   
   \[
   \text{Attribute-name type/domain \quad \text{PRIMARY KEY}},
   \]

   **Example**: in the table declaration for STUDENT we can specify that the attribute StID be the primary key for STUDENT as follows

   
   StID ID_DOM \quad \text{PRIMARY KEY},

   b. **NOT NULL constraint**: The value null is a special value for an attribute in a tuple that indicates that the value for that attribute of a tuple is either not known yet or that a value is not applicable for the attribute for a given tuple. SQL allows null values, but supports a constraint NOT NULL that can be applied to an attribute if the attribute should not be allowed to have the value null. For example, in the table declaration for STUDENT we can specify that the attribute Name not allow null values as follows

   
   Name NAME_DOM \quad \text{NOT NULL},

   c. **Default Values**: When an attribute is being specified, one can also specify a default value to be included in any tuple if an explicit value is not provided for the attribute. If not default

---

4 For example, in a relation STUDENT(StID, StName, MajorAdvisorID), it would be appropriate let MajorAdvisorID be null for a student who has not yet declared a major. Likewise in a relation EMPLOYEE(EmployeeID, Name, SpouseName) it would be appropriate to let SpouseName be null for an employee who is not married.
value is specified, a default value of null is used. For example, we can specify that GPA be given a default value of 0.000 as follows

\[
\text{GPA NUMERIC(4,3) DEFAULT 0.000,}
\]

2. **Table constraints**: Table constraints are constraints that are specified after the attribute declarations. Among the constraints that can be imposed in this way are primary and alternate key constraints. Although not required, table constraints can be given a name in order that it can be identified later in case it is necessary to drop the constraint and replace it with another.

a. **Primary key**: The primary key for a relation can be specified with a *primary key clause*, which assumes one of the following forms, depending on whether one wants to name the constraint or not

\[
\text{PRIMARY KEY(PKatt}_1,\ldots,\text{PKatt}_n) \quad \text{or} \quad \text{CONSTRAINT constraint-name PRIMARY KEY(PKatt}_1,\ldots,\text{PKatt}_n)
\]

Typically one would use a table constraint to specify a primary key when the primary key is comprised of more than one attribute. For single attribute primary keys one can use an attribute constraint.

b. **Alternate keys**: An alternate key can be specified with a **UNIQUE clause**

\[
\text{UNIQUE(AKatt}_1,\ldots,\text{AKatt}_n) \quad \text{or} \quad \text{CONSTRAINT constraint-name UNIQUE(AKatt}_1,\ldots,\text{AKatt}_n)
\]

Although there are other types of table constraints that are supported in SQL, we shall be content with these now and will introduce the others on an as-needed basis. We now turn our attention to transforming conceptual data models into a relational data models.

**Section 3. Transformation of Conceptual Data Models into a Relational Data Model**

**Representation of Classes in the Relational Data Model**

1. **Simple classes**: A class C whose attributes A\(_1\),\ldots,A\(_n\) are all simple and have domains D\(_1\), D\(_2\), \ldots, D\(_n\) is represented by the relation C(A\(_1\),\ldots,A\(_n\))

Each tuple in the relation represents an instance of the class. When the relation is represented as a table, each attribute will appear as a column of the table and each tuple will be a row of the table.

When transforming to the relational model from an EER model the primary key of an entity will be the primary key of the relation. When transforming from an object-oriented model, each instance of a class has an identity independent of its attribute values. In representing this in the relational model one needs to define a new attribute to serve as the primary key for the table.

---

5 **UNIQUE** also can be used in an attribute specification.
In SQL, the associated table would be defined (except for primary key designation) by:

```sql
CREATE TABLE C
(
    A1 D1,
    A2 D2,
    ...
    An Dn
);
```

**Example:** Consider a database for a college. If we had a class **STUDENT** having attributes **STID** (for a student id. number), **StName**, **Major**, and **CreditHrs** then we can represent the **STUDENT** by the relation

```sql
STUDENT(STID, StName, Major, CreditHrs)
```

Note that STID has been underlined, indicating its role as a primary key. This relation can be represented physically by a table with the following structure

<table>
<thead>
<tr>
<th>STID</th>
<th>StName</th>
<th>Major</th>
<th>CreditHrs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When data has been supplied for the class, it might appear something like:

<table>
<thead>
<tr>
<th>STID</th>
<th>StName</th>
<th>Major</th>
<th>CreditHrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2323</td>
<td>Pothering</td>
<td>Mathematics</td>
<td>62</td>
</tr>
<tr>
<td>5555</td>
<td>Leclerc</td>
<td>Computer Science</td>
<td>96</td>
</tr>
<tr>
<td>9876</td>
<td>Pharr</td>
<td>Philosophy</td>
<td>45</td>
</tr>
</tbody>
</table>

2. **Classes with a Multi-valued Simple Attribute:** Assume class C has attributes A₁, A₂, ..., Aₙ, and M, where M is a multi-valued simple attribute and A₁ is the primary key of C.⁶ To map C into a relational model we form two relations:

```sql
C(A₁, ..., Aₙ) and M(A₁, MVal)
```

where MVal gives each of the possible values of M that can be associated with a given object of class C. Note that the primary key of M is compound.

⁶ In object-oriented modeling, the class C would not have a primary key, so instead we shall assume that A₁ is an attribute that serves as the primary key when C is mapped to a relation in the relational model.
Example: Consider a class `COURSE` with attributes `CourseName`, `CreditHrs` and `ReqBooks`. Here `ReqBooks` may be a multi-valued attribute if a course has more than one required textbook assigned for it. As entities in the ER model, we may have the following two instances of `COURSE`:

<table>
<thead>
<tr>
<th>CourseName</th>
<th>CreditHrs</th>
<th>ReqBooks</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCI234</td>
<td>4</td>
<td>BookA, BookB, BookC</td>
</tr>
<tr>
<td>CSCI432</td>
<td>3</td>
<td>Book1, BookC</td>
</tr>
</tbody>
</table>

If we could enter them in tables that admitted multiple-valued attributes, an associated table might look as follows:

<table>
<thead>
<tr>
<th>CourseName</th>
<th>CreditHrs</th>
<th>ReqBooks</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCI234</td>
<td>4</td>
<td>BookA, BookB, BookC</td>
</tr>
<tr>
<td>CSCI432</td>
<td>3</td>
<td>Book1, BookC</td>
</tr>
</tbody>
</table>

Instead we use the following relations:

- `COURSE(CourseName, CreditHours)`
- `REQBOOKS(CourseName, Book)`

and the associated tables:

<table>
<thead>
<tr>
<th>CourseName</th>
<th>CreditHrs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCI234</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CSCI432</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CourseName</th>
<th>Book</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCI234</td>
<td>BookA</td>
</tr>
<tr>
<td>CSCI234</td>
<td>BookB</td>
</tr>
<tr>
<td>CSCI234</td>
<td>BookC</td>
</tr>
<tr>
<td>CSCI432</td>
<td>Book1</td>
</tr>
<tr>
<td>CSCI432</td>
<td>BookC</td>
</tr>
</tbody>
</table>

3. **Classes with a Compound Attribute:** Assume class `C` has attributes $A_1, \ldots, A_n$, and $G$, where $G$ is a compound attribute with components $G_1$ and $G_2$ that are both simple attributes. Assume also that attribute $A_1$ is the primary key for `C`. To map `C` into a relational model we consider two cases:

- a. If $G$ is a single-valued attribute we map `C` into the relation

\[C(A_1, \ldots, A_n, G_1, G_2)\]
b. If G is multi-valued, then following the example of multi-valued simple attributes we map C to the relational structure

\[ C(A_1, \ldots, A_n) \text{ and } G(A_1, G_1, G_2) \]

**Example:** Consider a class Course with attributes *CourseName*, *CreditHrs* and *AssignedRoom*. Here *AssignedRoom* is a compound attribute with components *Building* and *Room*.

a. If a course cannot have more than one section then for purposes of this discussion we shall assume that a course has only one classroom in which it meets. We may have the following instances of **COURSE**:

<table>
<thead>
<tr>
<th>CourseName</th>
<th>CreditHrs</th>
<th>AssignedRoom</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCI234</td>
<td>4</td>
<td>Building A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Room 1</td>
</tr>
<tr>
<td>MATH246</td>
<td>4</td>
<td>Building A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Room 3</td>
</tr>
<tr>
<td>HIST345</td>
<td>3</td>
<td>Building A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Room 1</td>
</tr>
<tr>
<td>CSCI432</td>
<td>3</td>
<td>Building B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Room 3</td>
</tr>
</tbody>
</table>

If we could enter them in tables that admitted structured attributes, an associated table might look as follows:

<table>
<thead>
<tr>
<th>CourseName</th>
<th>CreditHrs</th>
<th>AssignedRoom</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCI234</td>
<td>4</td>
<td>Building A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Room 1</td>
</tr>
<tr>
<td>MATH246</td>
<td>4</td>
<td>Building A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Room 3</td>
</tr>
<tr>
<td>HIST345</td>
<td>3</td>
<td>Building A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Room 1</td>
</tr>
<tr>
<td>CSCI432</td>
<td>3</td>
<td>Building B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Room 3</td>
</tr>
</tbody>
</table>

In this case we create the relation **COURSE**(*CourseName*, *CreditHrs*, *Building*, *Room*). Implementing this relation with a table we have the following

<table>
<thead>
<tr>
<th>CourseName</th>
<th>CreditHrs</th>
<th>Building</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCI234</td>
<td>4</td>
<td>BuildingA</td>
<td>Room1</td>
</tr>
<tr>
<td>MATH 246</td>
<td>4</td>
<td>BuildingA</td>
<td>Room3</td>
</tr>
<tr>
<td>HIST 345</td>
<td>3</td>
<td>BuildingA</td>
<td>Room1</td>
</tr>
<tr>
<td>CSCI432</td>
<td>3</td>
<td>BuildingB</td>
<td>Room3</td>
</tr>
</tbody>
</table>

b. If a course can have more than one section we may have the following instances of **COURSE**:
If we could enter them in tables that admitted structured and multiple-valued attributes, an associated table might look as follows:

<table>
<thead>
<tr>
<th>COURSE</th>
<th>AssignedRoom</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCI234 4</td>
<td>BuildingA Room1</td>
</tr>
<tr>
<td></td>
<td>BuildingB Room2</td>
</tr>
<tr>
<td>MATH246 4</td>
<td>BuildingA Room3</td>
</tr>
<tr>
<td>HIST345 3</td>
<td>BuildingA Room1</td>
</tr>
<tr>
<td>CSCI432 3</td>
<td>BuildingA Room3</td>
</tr>
<tr>
<td></td>
<td>BuildingB Room4</td>
</tr>
</tbody>
</table>

In this case we create the relation

\[ \text{COURSE(CourseName, CreditHrs)} \]

and the relation

\[ \text{ASSIGNEDROOM(CourseName, Building, Room)} \]

Implementing this relation with tables we have the following

<table>
<thead>
<tr>
<th>COURSE</th>
<th>ASSIGNED ROOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>CourseName</td>
<td>CreditHrs</td>
</tr>
<tr>
<td>CSCI234</td>
<td>4</td>
</tr>
<tr>
<td>CSCI234</td>
<td>4</td>
</tr>
<tr>
<td>MATH234</td>
<td>4</td>
</tr>
<tr>
<td>HIST345</td>
<td>3</td>
</tr>
<tr>
<td>CSCI432</td>
<td>3</td>
</tr>
<tr>
<td>CSCI432</td>
<td>3</td>
</tr>
</tbody>
</table>
4. **Representation of Superclasses and Subclasses**: Let the class $G$ with attributes $A_1, \ldots, A_n$ be a superclass of the class $S$. Suppose in addition to the attributes it inherits from $G$, the class $S$ also has attributes $B_1, \ldots, B_m$. For simplicity sake, let us assume all of the attributes for $G$ and $S$ are simple. If we assume $A_1$ is a primary key for $G$, then we would represent $G$ via the relational schema

$$G(A_1, \ldots, A_n)$$

and we represent $S$ by the relational schema

$$S(A_1, B_1, \ldots, B_m)$$

**Example**: Consider the following hierarchy in an EER model.

```
PERSON(ID, Name, Address)
EMPLOYEE(ID, Title)
CUSTOMER(ID, CreditLimit)
```

5. **Representation of Weak Entity Sets**: Let $W$ be a weak entity set that depends on an entity set $E$. Suppose $E$ has primary key $A$, along with other (simple) attributes, and suppose $W$ uses the attribute $B$ to distinguish among instances that are dependent upon the same instance from $E$. $W$ may also have other attributes, which we assume to be simple. We then map $E$ and $W$ to the relational model via the following relational schemas:

$$E(A, \text{other attributes of } E)$$

$$W(A, B, \text{other attributes of } W)$$

**Example**: Consider the entities EMPLOYEE and DEPENDENT. While the EMPLOYEE entity may use an attribute SSN as a key attribute, it may not be important for the model to give the entity DEPENDENT a similar attribute. Instead we may use an attribute Name as a partial identifier for an instance of DEPENDENT and rely on the fact that a dependent will only exist in the model if it is associated with an instance of EMPLOYEE. This makes DEPENDENT a weak entity set with a dependency on EMPLOYEE.
We map this representation to the relational model as follows:

- **EMPLOYEE**: (ID, other attributes)
- **DEPENDENT**: (ID, Name, other attributes)

**Foreign Keys**

In our relational models for classes with multi-valued attributes, generalization/specialization, and weak entity sets we have situations where the primary key P of a relation R1 appears as an attribute A in another relation R2. In this case we say that A is a foreign key in R2. More precisely, an attribute or set of attributes FK in a relation R1 is a foreign key if it satisfies both of the following conditions:

1. The attributes in FK have the same domain as the primary key attributes, PK, of another relation R2; the attributes of FK reference the relation R2.
2. A value of FK in a tuple t1 of R1 either occurs as the value of PK for some tuple t2 in R2, or is null. In the first case we say that the tuple t1 references the tuple t2.

It is not necessary for the foreign key and the primary key to which it refers to have the same name/names, though in the examples we have seen to this point we have made this the case. Now that the concept of a foreign key has been defined, we can more readily adopt different names for foreign keys and the primary keys they reference.

In SQL one can specify foreign keys via a table constraint. The general syntax for this constraint is:

```
FOREIGN KEY(FKatt) REFERENCES table-name(att)
```

**Examples**: In these examples we take each of the mappings of classes to the relational model that required a foreign key and specify the resulting relational schemas in SQL.

1. Use SQL to specify the relations that result from representing the class **COURSE**, with attributes **CourseName**, **CreditHrs**, and **ReqBooks** to the relational model. Here **CourseName** can be a primary key, while **ReqBooks** is a multi-valued attribute.

   In our discussion on how to map entities with multi-valued attributes, we saw that **COURSE** should be mapped to the following relational schemas:

   - **COURSE**: (CourseName, CreditHours)
   - **REQBOOKS**: (CourseName, Book)

   We can specify these schemas in SQL by

   ```sql
   CREATE TABLE COURSE
   (
   CourseName COURSE_NAME_DOM PRIMARY KEY,
   CreditHours SMALLINT);

   CREATE TABLE REQBOOKS
   (
   CourseName COURSE_NAME_DOM,
   Book BOOK_DOM,
   PRIMARY KEY(CourseName, Book),
   FOREIGN KEY (CourseName) REFERENCES COURSE(CourseName)
   );
   ```

CSIS601 - Fall, 2003  Chapter 2  Page - 11
2. Use SQL to specify the relations that result from representing the class **COURSE** with attributes **CourseName** CreditHrs and **AssignedRoom**. Here **AssignedRoom** is a compound attribute with components **Building** and **Room**.

In our discussion on how to map entities with multi-valued compound attributes, we saw that **COURSE** should be mapped to the following relational schemas:

\[
\begin{align*}
&\text{COURSE(CourseName, CreditHrs)} \\
&\text{ASSIGNEDROOM(CourseName, Building, Room)}
\end{align*}
\]

We can specify these schemas in SQL by

```
CREATE TABLE COURSE
(
    CourseName     COURSE_NAME_DOM PRIMARY KEY,
    CreditHours    SMALLINT
);

CREATE TABLE ASSIGNEDROOM
(
    CourseName     COURSE_NAME_DOM,
    Building       BUILDING_DOM,
    Room           ROOM_DOM,

    PRIMARY KEY(CourseName, Building, Room),
    FOREIGN KEY (CourseName) REFERENCES COURSE(CourseName)
);
```

3. Use SQL to specify the relations that result from representing the class hierarchy for the **PERSON**, **EMPLOYEE**, and **CUSTOMER** classes in a previous example.

In our discussion on how to map generalization/specialization hierarchies, we saw that **PERSON**, **EMPLOYEE**, and **CUSTOMER** should be mapped to the following relational schemas:

\[
\begin{align*}
&\text{PERSON(PERSONID, Name, Address)} \\
&\text{EMPLOYEE(EMPID, Title)} \\
&\text{CUSTOMER(CUSTID, CreditLimit)}
\end{align*}
\]

Note that instead of using the attribute name ID in each relation we have changed to a name appropriate to each relation. We can specify these schemas in SQL by

```
CREATE TABLE PERSON
(
    PersonID    ID_DOM PRIMARY KEY,
    Name        NAME_DOM,
    Address     ADDRESS_DOM
);

CREATE TABLE EMPLOYEE
(
    EmpID       ID_DOM PRIMARY KEY,
    Title       TITLE_DOM,

    FOREIGN KEY (EmpID) REFERENCES PERSON(PersonID)
);

CREATE TABLE CUSTOMER
```
4. Use SQL to specify the relations that result from representing the entity sets \texttt{DEPENDENT} and \texttt{EMPLOYEE} used in our discussion of weak entity sets.

We saw that \texttt{EMPLOYEE} and \texttt{DEPENDENT} should be mapped to the following relational schemas:

\texttt{EMPLOYEE(\texttt{EMPI}, other attributes)}
\texttt{DEPENDENT(\texttt{EMPI}, Name, other attributes)}

We can specify these schemas in SQL by

\begin{verbatim}
CREATE TABLE EMPLOYEE 
( 
  EmpID  ID_DOM PRIMARY KEY,
  other attributes,
); 

CREATE TABLE DEPENDENT 
( 
  EmpID ID_DOM,
  Name NAME_DOM,
  other attributes,

  PRIMARY KEY (EmpID, Name),
  FOREIGN KEY (EmpID) REFERENCES EMPLOYEE(EmpID)
); 
\end{verbatim}

\textbf{Representation of Binary Associations in the Relational Model}

1. \textit{Representation of one-to-one associations:} Suppose the classes $O_1$ and $O_2$ are classes that have a one-to-one association with each other. To map this association in the relational model one merely inserts the primary key of one relation into the relational schema of the other as a foreign key. In our case we would have the relations

$O_1(P_1, \text{other attributes}, P_2)$
$O_2(P_2, \text{other attributes})$

or alternatively

$O_1(P_1, \text{other attributes})$
$O_2(P_2, \text{other attributes}, P_1)$

\textbf{Example:} Consider a data model for a health club which uses classes \texttt{PATRON} and \texttt{LOCKER} which are in one-to-one association, \texttt{LockerAssignment}, with each other. If we assume that \texttt{PATRON} is represented in the relational model with a primary key \texttt{PatronID} and that \texttt{LOCKER} has the primary key \texttt{LockerNo}, then we might represent the classes and the association with the relational schemas

\begin{verbatim}
PATRON(PatronID, other attributes)
LOCKER(LockerNo, other attributes, AssignedTo)
\end{verbatim}
Or alternatively

```
PATRON(PatronID, other attributes, LockerAssigned)
LOCKER(LockerNo, other attributes)
```

We can specify these schemas in SQL by

```
CREATE TABLE PATRON
(
    PatronID ID_DOM PRIMARY KEY,
    other attributes,
);

CREATE TABLE LOCKER
(
    LockerNo SMALLINT PRIMARY KEY,
    other attributes,
    AssignedTo ID_DOM,
    FOREIGN KEY (AssignedTo) REFERENCES PATRON(PatronID)
);
```

2. **Representation of one-to-many associations:** Suppose the classes O and M are classes that have a one-to-many association with each other. To map this association in the relational model one includes the primary key of the representation of the "one" class as a foreign key in the "many" class. In our case we would have the relations

```
O(P1,other attributes)
M(P2, other attributes, P1)
```

**Example:** Consider a data model for a sports league that has classes ATHLETE and TEAM and an association **Roster** between TEAM and ATHLETE to represent the composition of the teams in terms of the athletes on those teams. If we assume that no athlete can belong to more than one team, then **Roster** is a one-to-many association between TEAM and ATHLETE. If we assume that ATHLETE is represented in the relational model with primary key AthleteID and TEAM is represented with primary key TeamName then the classes and the Roster association can be represented by the following relational schemas:

```
ATHLETE(AthleteID,other attributes, TeamName)
TEAM(TeamName, other attributes)
```

We can specify these schemas in SQL by

```
CREATE TABLE ATHLETE
(
    AthleteID ID_DOM PRIMARY KEY,
    other attributes,
    TeamName TEAM_DOM
    FOREIGN KEY (TeamName) REFERENCES TEAM(TeamName)
);

CREATE TABLE TEAM
(
    TeamName TEAM_DOM PRIMARY KEY,
    other attributes,
);
3. *Representation of many-to-many associations*: Suppose the classes M1 and M2 are classes that have a many-to-many association R with each other. (*many-to-many relationship*). To map this association in the relational model we create a third relation R. We then use the following relational schemas for M1, M2, and R:

\[
\begin{align*}
M1(P1, \text{other attributes}) \\
M2(P2, \text{other attributes}) \\
R(P1, P2)
\end{align*}
\]

Note here that relation R has a compound primary key comprised of the primary keys of the classes participating in the association. Each of the components of this compound primary key is also a foreign key. Although we have not shown it here, if the association R also had attributes, they would be included in the relation R.

**Example**: Consider the association *Enrollment* between the classes **STUDENT** and **COURSE** that represents the enrollment of students in courses. Suppose also that this association has an attribute *Grade* representing the grade a student earned in the course. This is a many-to-many relationship since normally a student takes more than one course and a given course enrolls more than one student. If we assume the attribute **STID** is the primary key for **STUDENT** in its relational implementation and that **CourseName** is the primary key for **COURSE** in its relational implementation, then the relational schema for the association is:

\[
\begin{align*}
\text{STUDENT}(\text{STID}, \text{other attributes}) \\
\text{COURSE}(\text{CourseName}, \text{other attributes}) \\
\text{ENROLLMENT}(\text{STID}, \text{CourseName}, \text{Grade})
\end{align*}
\]

We can specify these schemas in SQL by

```sql
CREATE TABLE STUDENT
(
    STID ID_DOM PRIMARY KEY,
    other attributes,
);

CREATE TABLE COURSE
(
    CourseName COURSE_DOM PRIMARY KEY,
    other attributes,
);

CREATE TABLE ENROLLMENT
(
    STID ID_DOM,
    CourseName COURSE_DOM,
    Grade GRADE_DOM,
    PRIMARY KEY(STID, CourseName),
    FOREIGN KEY(STID) REFERENCES STUDENT(STID),
    FOREIGN KEY(CourseName) REFERENCES COURSE(CourseName)
);
```

4. *Representation of Ternary and Higher Associations*: An association R among classes C₁,..., Cₙ (n > 2) is represented by a using relation that employs the attributes in the primary keys of the representations for each Cᵢ. We assume, however, that by renaming attributes if necessary, no two classes in the list have attributes with the same name. Any attributes which belong to the association also appear in the attributes in the relation.
Example: In a data model for an airline, consider the association **FlightAssignment** among the classes **FLIGHT**, **PLANE** and **PILOT** to represent the plane and the individuals that were assigned to a given flight (identified by flight number and date). A possible relation schema for the classes and the association is the following:

```plaintext
FLIGHT(FlightNumber, FlightDate, other attributes)
PLANE(TailNumber, other attributes)
PILOT(PilotID, other attributes)
FLIGHT_ASSIGNMENT(FlightNumber, FlightDate, TailNumber, PilotID)
```

**Participation Constraints and Relational Data Models**

For one-to-one and one-to-many associations it may be possible to represent either total participation of some classes in the association when the association is represented by a relational model by using the NOT NULL attribute with a foreign key.

Example: In the example for a one-to-many association between the classes **TEAM** and **ATHLETE**, suppose that every athlete must be assigned exactly one team but a team does not have to have an athlete assigned to it (for example it may be a new team). We could represent this association with the following ER diagram:

```
AthleteID  N
           Athlete
           -----------------------
Roster   | 1  |
           TEAM
           -----------------------
           TeamName
```

We can represent the total participation of **ATHLETE** in this association by the following SQL definitions:

```sql
CREATE TABLE ATHLETE
(
    AthleteID ID_DOM PRIMARY KEY,
    other attributes,
    TeamName TEAM_DOM NOT NULL

    FOREIGN KEY (TeamName) REFERENCES TEAM(TeamName)
);

CREATE TABLE TEAM
(
    TeamName TEAM_DOM PRIMARY KEY,
    other attributes,
);
```

*(end of example)*

By default every attribute that participates in a primary key of a relation cannot accept **null** values as this would violate the intent of a primary key – that its values are sufficient to distinguish one tuple of the relation from another. One consequence of this is that it may not always be possible to represent a partial participation of a class in an association. This can be seen in the previous example where there is nothing in the SQL specification that would allow one to deduce that a team does not have to have an athlete associated with it.
**Example:** Consider the association *AdvisorIs* between classes *STUDENT* and *FACULTY* in a data model for a university. If we assume that a student can have at most one advisor (but need not have an advisor) and that a faculty member can advise several students, or maybe none, then we can represent this association in an ER model by

![ER Diagram](image)

When we represent this association in SQL we get

```sql
CREATE TABLE FACULTY
(
    FacID ID_DOM PRIMARY KEY,
    other attributes,
);

CREATE TABLE STUDENT
(
    STID ID_DOM PRIMARY KEY,
    other attributes,
    AdvisorID ID_DOM,
    FOREIGN KEY (AdvisorID) REFERENCES FACULTY(FacID)
);
```

This specification allows *AdvisorID* to be null, and hence that a student need not have an advisor. On the other hand these specifications still cannot represent the provision that a faculty member need not advise a student.

**Section 4. Normalization**

So far our approach to relational data modeling has been to take what we get when we map an EER object model to the relational model. In this section we shall investigate some qualitative issues of relational models; in particular we shall see that given two relational models of the same enterprise, one may be superior to another when it comes to making changes to the data in the relations.

A *relation scheme*, R, for a relation R is a specification of the attributes, \( A_i \), \( 1 \leq i \leq \text{arity of } R \), where each attribute \( A_i \) is defined on a domain \( D_i \). If we let S denote the set \( \{ A_1, \ldots, A_n \} \), where n is the arity of R, then we denote the relation scheme for R by \( R(S) \), or alternatively \( R(A_1, \ldots, A_n) \).

A *database scheme* is the collection of all relation schemes for the relations in a database. In this section we shall consider some of the desirable properties of relation schemes and a database scheme, and consider processes for obtaining schemes with these properties.

Before considering how to design a good database however we shall first consider an example of a poor database scheme to see how such a scheme might lead to problems.
Modification Anomalies

Suppose a database scheme for a manufacturer consists of the following relation scheme giving information about parts and suppliers

\[
\text{PARTSUPPLIED}(\text{Part\#}, \text{PartDesc}, \text{Supplier\#}, \text{SupplierName}, \text{SupplierLocation}, \text{Price})
\]

The primary key for this relation is the composite key \((\text{Part\#}, \text{Supplier\#})\).

Suppose the following table represents a sample of the data in the relation:

<table>
<thead>
<tr>
<th>Part#</th>
<th>PartDesc</th>
<th>Supplier#</th>
<th>SupplierName</th>
<th>SupplierLocation</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>aaa</td>
<td>1000</td>
<td>JONES</td>
<td>N. CHARLESTON</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>aaa</td>
<td>1500</td>
<td>ABC</td>
<td>CHARLESTON</td>
<td>28</td>
</tr>
<tr>
<td>1</td>
<td>aaa</td>
<td>2050</td>
<td>XYZ</td>
<td>COLUMBIA</td>
<td>22</td>
</tr>
<tr>
<td>1</td>
<td>aaa</td>
<td>1900</td>
<td>P&amp;H</td>
<td>ATLANTA</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>bbb</td>
<td>3100</td>
<td>ALLEN</td>
<td>ATLANTA</td>
<td>520</td>
</tr>
<tr>
<td>2</td>
<td>bbb</td>
<td>1000</td>
<td>JONES</td>
<td>NORTH CHARLESTON</td>
<td>500</td>
</tr>
<tr>
<td>2</td>
<td>bbb</td>
<td>2050</td>
<td>XYZ</td>
<td>COLUMBIA</td>
<td>590</td>
</tr>
<tr>
<td>3</td>
<td>ccc</td>
<td>2050</td>
<td>XYZ</td>
<td>COLUMBIA</td>
<td>1000</td>
</tr>
<tr>
<td>4</td>
<td>ddd</td>
<td>1000</td>
<td>JONES'</td>
<td>N CHARLESTON</td>
<td>80</td>
</tr>
<tr>
<td>4</td>
<td>ddd</td>
<td>3100</td>
<td>ALLEN</td>
<td>ATLANTA</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>ddd</td>
<td>1900</td>
<td>P&amp;H</td>
<td>ATLANTA</td>
<td>95</td>
</tr>
<tr>
<td>5</td>
<td>eee</td>
<td>1500</td>
<td>ABC</td>
<td>CHARLESTON</td>
<td>160</td>
</tr>
<tr>
<td>5</td>
<td>eee</td>
<td>1000</td>
<td>JONES</td>
<td>N CHARLESTON</td>
<td>140</td>
</tr>
<tr>
<td>5</td>
<td>eee</td>
<td>8156</td>
<td>GJP</td>
<td>MT. PLEASANT</td>
<td>190</td>
</tr>
</tbody>
</table>

We can now see several problems with this scheme:

1. **(Update Anomalies)** A relation is prone to an update anomaly when changing an attribute value for a tuple in the relation may require that other tuples in the relation be updated also.

   In the relation above, if a Supplier changes Name or Location, or if a Part Description is changed, then all affected tuples in the relation (or database if such redundancy is present in other relations) would have to be changed — leading to the possibility that the name or location for a supplier, or the description for a part in one tuple may be changed, but not another. Consequently we may not have a unique name or location for each supplier or a unique name for each part, when we feel intuitively that we should.

   **Example:** Suppose in the above relation that supplier JONES changes location to SAVANNAH and renames itself SMITH, then the following update anomalies are possible:

<table>
<thead>
<tr>
<th>Part#</th>
<th>PartDesc</th>
<th>Supplier#</th>
<th>SupplierName</th>
<th>SupplierLocation</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>aaa</td>
<td>1000</td>
<td>SMITH</td>
<td>SAVANNAH</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>aaa</td>
<td>1500</td>
<td>ABC</td>
<td>CHARLESTON</td>
<td>28</td>
</tr>
<tr>
<td>1</td>
<td>aaa</td>
<td>2050</td>
<td>XYZ</td>
<td>COLUMBIA</td>
<td>22</td>
</tr>
<tr>
<td>1</td>
<td>aaa</td>
<td>1900</td>
<td>P&amp;H</td>
<td>ATLANTA</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>bbb</td>
<td>3100</td>
<td>ALLEN</td>
<td>ATLANTA</td>
<td>520</td>
</tr>
<tr>
<td>2</td>
<td>bbb</td>
<td>1000</td>
<td>SMITH</td>
<td>NORTH CHARLESTON</td>
<td>500</td>
</tr>
<tr>
<td>2</td>
<td>bbb</td>
<td>2050</td>
<td>XYZ</td>
<td>COLUMBIA</td>
<td>590</td>
</tr>
<tr>
<td>3</td>
<td>ccc</td>
<td>2050</td>
<td>XYZ</td>
<td>COLUMBIA</td>
<td>1000</td>
</tr>
<tr>
<td>4</td>
<td>ddd</td>
<td>1000</td>
<td>JONES'</td>
<td>N CHARLESTON</td>
<td>80</td>
</tr>
</tbody>
</table>
2. **(Insertion Anomalies)** An insertion anomaly arises when we may not be able to represent information about an entity instance or relationship instance without including information about some other entity or relationship instance that does not exist.

In the example relation scheme we have been discussing, we cannot enter any information about a supplier into our database unless the supplier supplies at least one part. Thus, if we had information about a potential supplier, say WAM from SALZBURG (whom we assign supplier number 1756), we cannot enter this information until we have been supplied with one of their parts.

- We might try putting null values for Part#, PartDesc, and Price, but then, when we enter an item for the supplier will we remember to delete the tuple with the null values?

- Since Part# and Supplier# together form a key for the relation, it might be impossible to look up tuples via this key if there are null items in the key field Part#.

3. **(Deletion Anomalies)** A deletion anomaly occurs when deleting a tuple from a relation to reflect the disappearance of an instance of an entity or a relationship may cause us to lose information about an instance of an entity or relationship that we do not wish to lose.

In our example relation scheme, if we want to delete the last tuple of the relation shown to reflect that supplier 8156 no longer supplies part 5, we will lose all information about that supplier.

In the database/relation scheme used in the above example, the problems cited can be eliminated by using the following database scheme instead:

<table>
<thead>
<tr>
<th>Part#</th>
<th>PartDesc</th>
<th>Supplier#</th>
<th>SupplierName</th>
<th>SupplierLocation</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>3100</td>
<td>ALLEN</td>
<td>ATLANTA</td>
<td>90</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1900</td>
<td>P&amp;H</td>
<td>ATLANTA</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1500</td>
<td>ABC</td>
<td>CHARLESTON</td>
<td>160</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>SMITH</td>
<td>SAVANNAH</td>
<td>140</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8156</td>
<td>GJP</td>
<td>MT. PLEASANT</td>
<td>190</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There is a disadvantage to the above decomposition, however. To find the suppliers of part 1, for example, we must re-relate the data in the PART and SUPPLIER relations via the PARTSSUPPLIED relation using a potentially time-expensive operation known as a join. With the single relation scheme we could avoid this operation. Nevertheless, the advantages of the multi-relation scheme generally outweigh those of the single relation scheme.

**Normalization, Functional Dependency, and Normal Forms**

**Normalization** is a process used in the design of a relational database to provide a systematic way of ensuring a minimal amount of data redundancy, while at the same time avoiding the various types of anomalies discussed in the previous section. Normalization theory is based on the notion of normal forms, where a relation is said to be in a particular normal form if it satisfies a certain set of conditions. In this section we shall discuss the following types of normal forms: First Normal Form
First Normal Form

A relation scheme is said to be in First Normal Form (or 1NF) if the values in the domain of each attribute are atomic; that is they cannot be decomposed into component values. A database scheme is in First Normal Form if all its relation schemes are in 1NF. A relation scheme which is not in 1NF is said to be unnormalized.

Example: Consider the following the relation scheme

\[
\text{STUDENT(} \text{STID, StName, Major, Credits, Status)}
\]

where we assume a student can have more than one major. If we use an array of major descriptions to represent these values, and display the values vertically, an instance of the relation may look as follows.

<table>
<thead>
<tr>
<th>STID</th>
<th>StName</th>
<th>Major</th>
<th>Credits</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1001</td>
<td>Smith, Tom</td>
<td>History</td>
<td>90</td>
<td>SR</td>
</tr>
<tr>
<td>S1003</td>
<td>Jones, Mary</td>
<td>Math</td>
<td>95</td>
<td>SR</td>
</tr>
<tr>
<td>S1006</td>
<td>Lee, Susan</td>
<td>CompSci Math</td>
<td>16</td>
<td>FR</td>
</tr>
<tr>
<td>S1010</td>
<td>Burns, Edward</td>
<td>Art English</td>
<td>63</td>
<td>JR</td>
</tr>
<tr>
<td>S1060</td>
<td>Jones, Mary</td>
<td>CompSci</td>
<td>40</td>
<td>SO</td>
</tr>
</tbody>
</table>

A relation scheme that is unnormalized can be changed to one that is in first normal form by replacing attributes whose domain admits values that are non-atomic with one or more other attributes, depending on the nature of the components of the non-atomic values.

a. If the domain values are multiple values of the same base type then a relation scheme that is in 1NF can be derived from the original scheme by replacing an attribute with an associated given domain by one that has as its domain the base type.

- A relation such as the one above relation can be changed to one which is in first normal form by taking each tuple t which has a nonatomic attribute value, splitting this value up into its components, and replacing t with new tuples which agree with t in all other attribute values but having a different component value in place of the nonatomic value. In doing so, it will be necessary to include the attribute in the key for the relation.

Example: Consider unnormalized relation scheme given in the previous example.

\[
\text{STUDENT(} \text{STID, StName, Major, Credits, Status)}
\]

Following the above approach for placing STUDENT in 1NF would yield the relation scheme

\[
\text{STUDENT(} \text{STID, StName, Major, Credits, Status)}
\]

And the following sample relation.
Alternatively, one can split the given relation into two relations. The first relation is the same as the unnormalized relation except the attribute which caused it to be unnormalized is removed and is placed in new relation together with the attributes that comprised the primary key of the unnormalized relation. All attributes of the new relation will comprise the primary key for this relation. The tuples for this relation will be determined in a manner similar to that of the first option above.

**Example:** Consider once again the unnormalized relation scheme given in the previous example.

\[ \text{STUDENT}(\text{STID}, \text{StName, Major, Credits, Status}) \]

Following the above approach for placing STUDENT in 1NF would yield the relation scheme

\[ \text{STUDENT} (\text{STID, StName, Credits, Status}) \]
\[ \text{STUDENTMAJOR} (\text{STID, Major}) \]

And the following sample relations.

<table>
<thead>
<tr>
<th>STID</th>
<th>StName</th>
<th>Major</th>
<th>Credits</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1001</td>
<td>Smith, Tom</td>
<td>History</td>
<td>90</td>
<td>SR</td>
</tr>
<tr>
<td>S1003</td>
<td>Jones, Mary</td>
<td>Math</td>
<td>95</td>
<td>SR</td>
</tr>
<tr>
<td>S1006</td>
<td>Lee, Susan</td>
<td>CompSci</td>
<td>16</td>
<td>FR</td>
</tr>
<tr>
<td>S1006</td>
<td>Lee, Susan</td>
<td>Math</td>
<td>16</td>
<td>FR</td>
</tr>
<tr>
<td>S1010</td>
<td>Burns, Edward</td>
<td>Art</td>
<td>63</td>
<td>JR</td>
</tr>
<tr>
<td>S1010</td>
<td>Burns, Edward</td>
<td>English</td>
<td>63</td>
<td>JR</td>
</tr>
<tr>
<td>S1060</td>
<td>Jones, Mary</td>
<td>CompSci</td>
<td>40</td>
<td>SO</td>
</tr>
</tbody>
</table>

b. If the components of the domain values are fields whose values are atomic (such as in a record), then one can replaces the given attribute with several attributes -- one for each component.

- A relation such as this one can be changed to one which is in first normal form by taking each tuple t which has a nonatomic attribute value, splitting this value up into its
components, placing each component value in a new attribute field, and replacing t with a
tuple that agrees with t in all other attribute values but having a several atomic values in
place of the nonatomic value

**Example:** Consider the relation scheme STUDENT given as normalized under the second
approach above

\[ \text{STUDENT} (\text{STID}, \text{StName}, \text{Credits}, \text{Status}) \]

but where the domain of \text{StName} is a compound type such as

\[
\text{StNameDom} = \text{record}
\begin{align*}
\text{LastName} & : \text{string}; \\
\text{FirstName} & : \text{string};
\end{align*}
\text{end};
\]

Under this scheme, our sample relation may appear as follows:

<table>
<thead>
<tr>
<th>STID</th>
<th>StName</th>
<th>Credits</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1001</td>
<td>Smith Tom</td>
<td>90</td>
<td>SR</td>
</tr>
<tr>
<td>S1003</td>
<td>Jones Mary</td>
<td>95</td>
<td>SR</td>
</tr>
<tr>
<td>S1006</td>
<td>Lee Susan</td>
<td>16</td>
<td>FR</td>
</tr>
<tr>
<td>S1010</td>
<td>Burns Edward</td>
<td>63</td>
<td>JR</td>
</tr>
<tr>
<td>S1060</td>
<td>Jones Mary</td>
<td>40</td>
<td>SO</td>
</tr>
</tbody>
</table>

Using the method described above for converting to 1NF, our revised relation scheme would
become something like

\[ \text{STUDENT} (\text{STID}, \text{LastName}, \text{FirstName}, \text{Credits}, \text{Status}) \]

and our sample relation would be converted to one such as

<table>
<thead>
<tr>
<th>STID</th>
<th>LastName</th>
<th>FirstName</th>
<th>Credits</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1001</td>
<td>Smith</td>
<td>Tom</td>
<td>90</td>
<td>SR</td>
</tr>
<tr>
<td>S1003</td>
<td>Jones</td>
<td>Mary</td>
<td>95</td>
<td>SR</td>
</tr>
<tr>
<td>S1006</td>
<td>Lee</td>
<td>Susan</td>
<td>16</td>
<td>FR</td>
</tr>
<tr>
<td>S1010</td>
<td>Burns</td>
<td>Edward</td>
<td>63</td>
<td>JR</td>
</tr>
<tr>
<td>S1060</td>
<td>Jones</td>
<td>Mary</td>
<td>40</td>
<td>SO</td>
</tr>
</tbody>
</table>

All relational models are assumed to use tables in first normal form; later, however, when we study
the object-relational model we shall no longer require this.
**Functional Dependency**

The remaining normal forms we discuss are all aimed at removing some of the conditions that lead to the anomalies we discussed earlier. As we shall see, these conditions inevitably revolve around dependencies among attributes within a single table. In order to describe these conditions satisfactorily, however, we must introduce the more formal notion of a "functional dependency."

Given attributes $A_1, ..., A_n$ and $B$ of a relation $R$, we say that $A_1, ..., A_n$ *functionally determine* $B$ (or that $B$ is *functionally dependent on* $A_1, ..., A_n$) if whenever tuples $t_1$ and $t_2$ agree on the values of $A_1, ..., A_n$ they must also agree on their value of $B$. We write $A_1, ..., A_n \rightarrow B$. If the same set of attributes functionally determine more than one attribute, say $B_1, B_2$ we write $A_1, ..., A_n \rightarrow B_1, B_2$.

**Example:** Consider the relation scheme

```
STUDENT(STID, StName, Major, Credits, Status, SSN)
```

with the following constraints:

- Every student has a unique STID value and SSN value.
- Every student has at most one major.
- Names are not unique.
- Credits refers to credit hours completed.
- Status refers to the student's year in school - freshman (FR), sophomore (SO), etc.

Based on these assumptions we could identify (or assign) the following functional dependencies in this relation scheme:

- $STID \rightarrow StName, Major, Credits, Status, SSN$
- $SSN \rightarrow STID, StName, Major, Credits, Status$
- $Credits \rightarrow Status$

It is important to note that a functional dependency is a property of the semantics of a relation schema $R$, and is not a property of a particular instance of $R$. In particular, we use our understanding of the associations we want to hold between attributes of $R$ to specify the functional dependencies that must exist for all its instances. A functional dependency cannot be deduced from instances of $R$, but must be defined explicitly by someone who knows the semantics of the attributes of $R$. On the other hand it only requires a single counterexample to disprove a functional dependency.

Given attributes $A_1, ..., A_n$ and $B$ of a relation $R$, we say that $A_1, ..., A_n$ *fully functionally determine* $B$ if $A_1, ..., A_n$ functionally determine $B$ and $B$ is not dependent on a subset of $A_1, ..., A_n$.

**Example:** Consider the following relation scheme

```
CLASS(Course#, STID, StName, FACID, Sched, Room, Grade)
```

where

- **Course#** includes the department, course, and section.
- Only one faculty member teaches a given section of a course (that is, no team teaching).
- **Sched** gives the meeting times and days (as one string).
• Room gives the building and room (again as one string).

We can identify the following dependencies:

\[
\begin{align*}
\text{Course#}, \ STID & \rightarrow \text{StName, FACID, Sched, Room, Grade} \\
\text{Course#} & \rightarrow \text{FACID, Sched, Room} \\
\text{STID} & \rightarrow \text{StName}
\end{align*}
\]

Here StName, FACID, Sched, and Room are functionally dependent on Course#, STID but are not fully functionally dependent on it. Grade is fully functionally dependent on Course#, STID.

A superkey for a relation R is a set of attributes for R that functionally determine all of the other attributes of R. A candidate key of a relation is a set of attributes that fully functionally determine all of the other attributes of R. An attribute A of a relation R is a prime attribute if A is a part of a candidate key for R; otherwise A is a non-prime attribute.

**Second Normal Form (2NF)**

A relation is in second normal form (2NF) if it is in 1NF and every non-prime attribute is fully functionally dependent on every candidate key of R.

- The only time one needs to be concerned about a relation not being in 2NF is when there are composite candidate keys.

A relation which is not in 2NF is prone to the anomalies shown in the motivating example for this discussion.

A relation which is not in 2NF can be transformed into a set of relations which are in 2NF without losing information from the original relation by identifying each non-full functional dependency and forming projections to remove the attributes that depend on the determinants of each of the non-full dependencies. The determinant and their dependent attributes are placed in a separate relation.

**Transitive Dependence and Third Normal Form**

Although Second Normal Form can eliminate transaction anomalies that result from attributes not being fully functionally dependent on the key of a relation, 2NF by itself will not guarantee that transaction anomalies will not take place. Consider, for example, the following relation schema

\[
\text{TEACHES}(\text{Course, Prof, Room, RoomCap})
\]

where the domain of the attribute Course is the courses offered by a department in a particular semester, the domain of the attribute Prof is the faculty members of the department, the domain of Room is the rooms assigned to that department for teaching its courses, and the domain of RoomCap is an integer indicating the seating capacity of the room. We assume that only one professor teaches a given course (although a professor may teach several courses) and that a given course always meets in the same room (whose capacity does not change), but the same room may be used for more than one course.

From this narrative we can deduce the following dependencies

\[
\begin{align*}
\text{Course} & \quad \text{Professor, Room,}
\end{align*}
\]
From this we can deduce that Course is a key for TEACHES, and is in fact the only candidate key. The relation is therefore automatically in 2NF. This relation however is still prone to insertion, deletion, and update anomalies of the type we saw earlier. To see this, suppose the following table represents an instance of this relation:

<table>
<thead>
<tr>
<th>Course</th>
<th>Prof</th>
<th>Room</th>
<th>RoomCap</th>
</tr>
</thead>
<tbody>
<tr>
<td>353</td>
<td>Smith</td>
<td>A532</td>
<td>45</td>
</tr>
<tr>
<td>351</td>
<td>Smith</td>
<td>C320</td>
<td>100</td>
</tr>
<tr>
<td>355</td>
<td>Clark</td>
<td>H940</td>
<td>400</td>
</tr>
<tr>
<td>456</td>
<td>Turner</td>
<td>B278</td>
<td>50</td>
</tr>
<tr>
<td>459</td>
<td>Jamieson</td>
<td>D110</td>
<td>50</td>
</tr>
<tr>
<td>480</td>
<td>Clark</td>
<td>A532</td>
<td>45</td>
</tr>
</tbody>
</table>

Note:

- We cannot insert the information that A560 has capacity 50 until a course is assigned to that room.
- If we delete the only course scheduled for a given room, say course 355 in the above instance, we lose information about the capacity of that room.
- If the capacity of a room changes, say the capacity of room A532 is decreased to 40, then the database will be susceptible to update anomalies.

The reason why 2NF could not prevent these anomalies from possibly arising is that 2NF only addresses functional dependencies between non-key attributes and candidate keys. If we examine the dependencies which exist in the TEACHES scheme, namely

**Course → Prof, Room, RoomCap**

and

**Room → RoomCap**

we see that the latter dependency does not involve a candidate key of TEACHES. If we can eliminate such dependencies from the relations in our database scheme then we might be able to eliminate transaction anomalies from the database. It seems that what we need is a general way to classify a relation scheme so that functional dependencies "away from a key" are not permitted.

One can show that for a relation with attributes X, Y, and Z, if X → Y, and Y → Z, then X → Z - a phenomenon known as transitive dependency. In the above example, in addition to the (full) functional dependency of the (nonkey) attribute RoomCap on Courses (the relation scheme's key), there is also a transitive dependency of RoomCap on Course through Room.

A relation scheme is in *third normal form* (3NF) if it is in 2NF and no non-prime attribute that is transitively dependent on a candidate key.

- "The non-prime attribute depends on a key, the whole key, and nothing but the key."
A relation which is 2NF but not in 3NF can be transformed into an equivalent set of 3NF relations by finding any non-prime attributes which are transitively dependent on a key and placing them and their determinants in a new relation.

**Example:** Given the relation scheme **TEACHES** above, we can decompose it into the following pair of 3NF relation schemes:

- **COURSE(Course#, Prof, Room#)**
- **ROOM(Room#, RoomCap)**

**Boyce-Codd Normal Form**

The specification of non-prime attributes in the definition of Third Normal Form introduces a weakness that may still admit update, insertion, and deletion anomalies in a relation where that is in 3NF.

**Example:** Consider the following relation scheme

- **FACULTY(FacName, Dept, Office, Rank, YearHired)**

Here we assume that no two faculty members in the same department have the same name, though there may be faculty with the same name as long as they are from different departments. Also, each faculty member is assigned only one office, though faculty in the same department may share an office. A department normally has several offices for its faculty and no two departments will share an office. Expressing these constraints as functional dependencies we have

- FacName, Dept → Office, Rank, YearHired
- Office → Dept

Thus, \{FacName, Dept\} can serve as a key for the FACULTY relation. Note also, however that \{FacName, Office\} can also serve as a candidate key.

We now indicate how the dependency Office → Dept in our relation, can cause some transaction problems for this scheme. Consider the following instance of this relation:

<table>
<thead>
<tr>
<th>FacName</th>
<th>Dept</th>
<th>Office</th>
<th>Rank</th>
<th>YearHired</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>Math</td>
<td>101BA</td>
<td>Professor</td>
<td>1981</td>
</tr>
<tr>
<td>Jones</td>
<td>CompSci</td>
<td>203BB</td>
<td>Associate</td>
<td>1976</td>
</tr>
<tr>
<td>Smith</td>
<td>Physics</td>
<td>414BC</td>
<td>Instructor</td>
<td>1997</td>
</tr>
<tr>
<td>Edwards</td>
<td>English</td>
<td>101BD</td>
<td>Assistant</td>
<td>2000</td>
</tr>
<tr>
<td>Gordon</td>
<td>Physics</td>
<td>414BC</td>
<td>Instructor</td>
<td>1987</td>
</tr>
</tbody>
</table>

and observe that the following tuple modification anomalies are possible, even though the relation scheme given is in 3NF:

- In deleting the tuple with key value (Jones, CompSci) we lose the information that office 203BB belongs to CompSci.
- If Physics acquires office 424BC, we cannot insert this information into our relation until the office is assigned to a faculty member.
The problem here is that \texttt{Dept} is not a non-prime attribute, which means that the partial dependency on the candidate key \{\texttt{FacName, Office}\}, which would otherwise cause \texttt{FACULTY} to not be even in 2NF (much less 3nF) does not come into play:

The following definition strengthens the criteria a relation must satisfy in order to be sure that it is not susceptible to the type of insertion, deletion, or update anomalies we have just seen:

A relation is in \textit{Boyce-Codd Normal Form} (BCNF) if every determinant is a candidate key.

Suppose we are given a relation \( R \) that is not in BCNF. Then there must be a determinant \( X \) which is not a candidate key. Let us suppose that \( Y \) is a candidate key for \( R = R(A_1,\ldots,A_n) \) also and that \( X \rightarrow Z \). Here, of course, \( X, Y \) and \( Z \) may be sets of attributes taken from \{\( A_1,\ldots,A_n \)\}. If neither of the relations

\[
\begin{align*}
R1(X,Z) \\
R2(Y,\{A_1,\ldots,A_n\} - Z)
\end{align*}
\]

have no determinants that are not candidate keys, then \( R1 \) and \( R2 \) are both in BCNF; otherwise we can repeat the process with the non-BCNF relation(s), ultimately producing a set of relations, all of which are in BCNF.

**Example:** Using relation

\[
\textsc{FACULTY}(\texttt{FacName, Dept, Office, Rank, DateHired})
\]

considered earlier we could have made the association \( X \leftrightarrow \texttt{Office} \) and \( Y \leftrightarrow \texttt{FacName} + \texttt{Office,} Z\leftrightarrow \texttt{Dept} \). This would have yielded the decomposition

\[
\begin{align*}
R1(\texttt{Office,Dept}) \\
R2(\texttt{FacName,Office,Rank,DateHired})
\end{align*}
\]

both of which are BCNF.

While a decomposition into relations in BCNF may appear to be desirable, in some circumstances BCNF may be too strong a condition to impose on a decomposition in that in reaching BCNF may cause some functional dependencies to be lost from the original relation.

**Example:** In the decomposition of \textsc{FACULTY} into \( R1 \) and \( R2 \) above, we lose the dependency \( \texttt{FacName,Dept} \rightarrow \texttt{Office,Rank,DateHired} \) as a derivation from the structure of \( R1 \) and \( R2 \) alone. The practical consequence of this is that it may be possible to insert a tuple (Harris, 203BB, Assistant, 1990) into the \( R2 \) relation even though this insertion should be disallowed since Harris is actually a member of the Physics Department, while office 203BB is a computer science office. This could not happen, however, without independently checking the \( R1 \) relation.

Given the non-dependency preserving potential of BCNF decompositions, 3NF is seen as a condition that has almost the benefits of BCNF as far as eliminating anomalies is concerned, but which preserves functional dependencies. A database schema, all of whose relations are in 3NF is usually the preferred objective.

It has been conjectured by Ullman (\textit{Principles of Database Systems} (2nd Ed.). Computer Science Press, 1982. pp. 237- 238) that the functional dependencies which cause 3NF to be violated in a relation are in a sense irrelevant in that they tell us nothing about the “real world” that is of use to the
database designer. For example, in the **FACULTY** relation which we have been using in this discussion, the functional dependency **Office** \(\rightarrow\) **Dept** does in fact tell us how a department is structured in terms of the offices it has been assigned, but this information is not really useful in terms of the apparent intent of the **FACULTY** relation, which is not to identify a department by its offices, but to record the offices which have been assigned to faculty members.

**Lossless Decomposition of Relations**

Besides preserving functional dependencies (even at the sake of allowing anomalies), a decomposition of a relation into normal forms should be capable of reproducing the relation exactly by “rejoining” all the relations in the decomposition. Before defining precisely what this means, however, we need to specify more precisely what we mean by a decomposition and rejoining.

Let R be a relation. A **decomposition** of R into n relations is a set of n relations \(\{R_1,\ldots,R_n\}\) such that:

- a. For every relation \(R_i\) the attributes of \(R_i\) are a subset of the attributes of R
- b. The union of the attributes of the \(R_i\)s and the attributes of R are the same.

Let R be a relation and let \(A=\{A_1,\ldots,A_n\}\) be a subset of the attributes of R (all of the \(A_i\)’s are distinct). The **projection of R on K** is the relation T with attributes A such that for every tuple \(t=(a_1,\ldots,a_n)\in T\) there is a tuple \(r\in R\) where t and r agree on all the values \(a_i\) (i=1,...,n). We denote this by \(\pi_A(R)\).

Let R and S be relations and suppose that they have an attributes \(A=\{A_1,\ldots,A_n\}\) in common. The **natural join** of R and S, denoted \(R \times S\) is defined to be the relation whose attributes are the union of the attributes of R and S and

\[
t \in R \times S \iff \exists r \in R \text{ and } \exists s \in S \text{ and } \pi_R(t)=r \text{ and } \pi_S(t)=s \text{ and } \pi_{A_i}(r)=\pi_{A_i}(s) \text{ for } i=1,...,n.
\]

Let R be a relation with an associated set of functional dependencies FD. A decomposition of R, \(\{R_1,\ldots,R_n\}\) is said to be **lossless** (or **nonloss**, or **lossless join**) if for any possible future content of \(R = R_1 \times R_2 \times \ldots \times R_n\)

Not all decompositions are nonloss. An example of such a decomposition (known as a **lossy** decomposition) is the following:

The relation **CLASS(STID, Course#, Grade)** is intended to record the grades taken by students in a given course, say

<table>
<thead>
<tr>
<th>STID</th>
<th>Course</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1001</td>
<td>Art103A</td>
<td>B+</td>
</tr>
<tr>
<td>S1001</td>
<td>Mat120B</td>
<td>A</td>
</tr>
<tr>
<td>S1003</td>
<td>Comp100A</td>
<td>C</td>
</tr>
<tr>
<td>S1003</td>
<td>Art103A</td>
<td>C+</td>
</tr>
<tr>
<td>S1010</td>
<td>Art103A</td>
<td>B</td>
</tr>
</tbody>
</table>

Now decompose **CLASS** into the relations **CLASS1(STID,Course#)** and **CLASS2(STID,Grade)**, yielding the following relations
This decomposition is a lossy decomposition since in forming \( \text{CLASS1} \times \text{CLASS2} \) we introduce tuples such as \((\text{S1001}, \text{Art103A}, \text{C+})\) that were not part of the original relation. This represents a “loss” not because we don’t get back all the rows we had before, but because we get back rows that were not present originally. For example in the above example when we rejoin \( \text{CLASS1} \) and \( \text{CLASS2} \) we get

<table>
<thead>
<tr>
<th>STID</th>
<th>Course</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1001</td>
<td>Art103A</td>
<td>B+</td>
</tr>
<tr>
<td>S1001</td>
<td>Art103A</td>
<td>A</td>
</tr>
<tr>
<td>S1001</td>
<td>Mat120B</td>
<td>B+</td>
</tr>
<tr>
<td>S1001</td>
<td>Mat120B</td>
<td>A</td>
</tr>
<tr>
<td>S1003</td>
<td>Comp100A</td>
<td>C</td>
</tr>
<tr>
<td>S1003</td>
<td>Comp100A</td>
<td>C+</td>
</tr>
<tr>
<td>S1010</td>
<td>Art103A</td>
<td>B</td>
</tr>
</tbody>
</table>

which has more tuples than the original relation \( \text{CLASS} \). Here, for example, we lose the information that student S1001 did not receive a grade of C+ in Art103A. Another way of looking at the “loss” in this decomposition is that we lost the ability to tell what table content we started with.

One can show that every relation scheme \( R \) having a set of functional dependencies \( F \) has a lossless decomposition into relations in 3NF that preserves all of the dependencies in \( F \), as well as a lossless (though not necessarily dependency preserving) decomposition into relations in BCNF.

**Determining a Minimum Set of 3NF Tables**

A widely regarded design objective for a relational database schema is to decompose the relational schema into a set of tables that is at least 3NF but which preserves all prescribed functional dependencies, is lossless, and requires a minimal number of tables. There is a well-known algorithm by Philip Bernstein\(^7\) that can do just this. Bernsteins’ algorithm and an illustrative example are discussed in your textbook on pages 113-117. We will not pursue this further here, however.

Bernstein’s algorithm was designed for a time when high-level data modeling was not widely used (its publication year coincides with that of Chen’s paper on the entity-relationship modeling). In fact a well-thought out high level data model should almost certainly yield relational design that is

---

\(^7\) Bernstein, P. “Synthesizing 3NF Relations from Functional Dependencies,” ACM Transactions on Database Systems 1,4(1976), pp. 272-298