Chapter 2
Structured Datasets - Relational Databases

Section 1. What is a Database? What is a DBMS?

A ***database*** is a self-describing, integrated collection of data that models relevant aspects of an enterprise. This collection of data and its description are intended to be stored on persistent storage devices and to be shared among multiple users.

- The database's self-description is known as the ***system catalog***, ***data dictionary***, or ***metadata*** (data about data).

Why are databases of interest? For effective decision-making, individuals or organizations maintain data about circumstances related to decisions that it must make. Through this data the individual or organization (which we shall call the *user*) can build an abstraction (or model) of such circumstances. By processing the abstraction the user tries to learn more about the circumstances to guide them in making a decision.[1]

- Spreadsheets can do the same thing so for keeping data, when would a person choose a database over a spreadsheet, or vice-versa?

  You will have a better appreciation of this later once you know more about databases, of course, but a quick answer could be obtained by asking yourself what are main data elements of interest? Now imagine that you are keeping data relevant to each of these on index cards and consider how frequently you would be performing the following activities:

  a. Would you be frequently be going through the stack of cards and making the same updates to each of these cards, or performing the same calculations using the data on these cards? -- If so, a spreadsheets may be more effective than a database in this case.

  b. Would you commonly be shuffling through the cards, setting aside those cards whose data satisfied a certain condition, or rearranging the cards into groups according to some specified condition? If so, then a database may be more effective than a spreadsheet.

*A **database management system*** (DBMS) is a software system that:

- allows users define a database and to store, use, or modify data in that database,
- controls shared access to a database, and
- provides mechanisms to help ensure the integrity and security of the shared data.

More specifically, among its functions, a DBMS typically has processes to:

a. define the storage structures (in secondary memory) for the data;
b. load the data;
c. accept requests (or *queries*) for data from programs or users,
d. format retrieved data so that it appears in a form appropriate for the person or program that requested it;
e. accept and perform updates to the data; this could include adding new data, deleting data, or changing the values of existing data;
f. perform data backup and recovery;
g. allow concurrent use of the data by several users without having users interfere with one another;
h. provide controlled access to the database.

Among the outcomes of this course are ones to familiarize you with how these functions may be carried out on a widely used DBMS and for you to actually perform them.

---

[1]The notion that data should be kept because it might prove to be useful dates to the origins of writing. Cuneiform goes back to 8000BC in Sumeria where symbols were used to represent trading goods and livestock using wet clay tablets. This led to the establishment of archives (the archives at Ebla dates to 2500BC) and libraries (the most famous of which was the Library at Alexandria, which was destroyed by fire in 48BC).

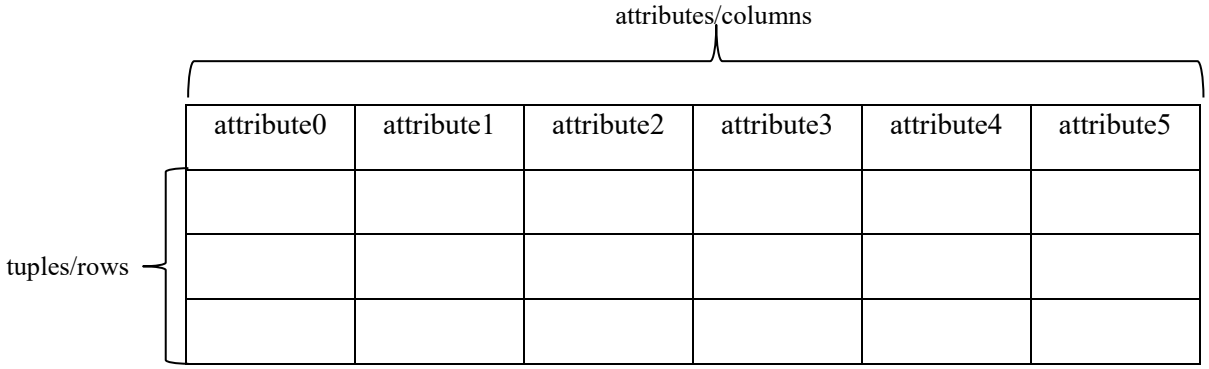
Section 2. Relational Databases

Computerized database systems date back to 1960, when Charles Bachman developed the Integrated Data System using what came to be known as the *network paradigm*, and IBM (not be be outdone by IDS) developed its Information Management System (IMS) based on what was known as the *hierarchical paradigm*.

Since the mid 1980s the dominant paradigm for structuring databases has been the *relational* model.[1] The term "relational" comes from the mathematical concept of a *relation*, which comprises the theoretical foundation on which this approach to database design rests. In practice relations are realized as tables and one will find the terms "table" and "relation" used interchangeably.

The columns of a relation (table) are known as the relation's ***attributes***, while each row of a relation is known as a ***tuple***. The number of rows in a table at any one time is the ***cardinality*** of the table.

- Each attribute must have a name, preferably one that describes what the attribute represents. No two attributes in the same table can have the same name.
- All of the tuples of a relation are expected to be distinct. That is, for any pair of tuples in a relation, there must be at least one attribute where the values of these tuples for that attribute are different.



Example*:* Consider the following relation, where the attribute names (admittedly not descriptive) are given in bold

att0	**att1**	**att2**	**att3**
X	aaa	1	R2D2
Y	bbb	3	C3PO
X	cc	5	OB1
Y	bbb	3	C3PO
W	aaa	2	C3PO

cause the table to be ill-formed as a relation since they are identical

You will notice that the second and fourth tuples in this relation have identical values across all of their attributes. This relation is therefore inappropriate insofar as being a relation in a relational database since there is no way to distinguish the second tuple from the fourth tuple.

[1] Although it wasn't until the mid-80s that the relational database started to become the predominant database model, the idea for the relational model was proposed in a 1970 paper by E.F. Codd "A Relational Model of Data for Large Shared Data Banks" that was published in the *Communications of the ACM*, Volume 13, Number 6, June 1970, pp 377-387. In 1981 Codd was awarded ACM's A.M. Turing Award, the "Nobel Prize of Computing."

Examples Of Simple Relational Databases.

1. **`Employee-Project-AssignedTo`** database (EPA): This database attempts to model a company's employees, the projects the company is working on, and who is working on each project (or what projects each employee is working on).

Employee	
EmpNo	**EmpName**
101	Jones
103	Smith
104	Clark
106	Byron
107	Evans
110	Drew
112	Smith

Project		
ProjNo	**ProjDesc**	**Supervisor**
COMP231	Mobile App	107
COMP278	Web Service	110
COMP353	Database	104
COMP354	OS	110
COMP453	Database	101

AssignedTo	
ProjNo	**EmpNo**
COMP453	101
COMP354	103
COMP353	104
COMP453	104
COMP231	106
COMP278	106
COMP353	106
COMP354	106
COMP231	107
COMP353	107
COMP278	110
COMP354	110
COMP354	112
COMP453	112

2. **`Student-Class-EnrolledIn database (SCE):`** This database attempts to model a university's students, the courses the university offers, and who is enrolled in each course (or what courses each student is taking).

Student			
STID	**Name**	**Major**	**Level**
100	Jones	HIST	JR
150	Parks	DATA	SO
200	Baker	CSCI	JR
250	Glass	HIST	SR
300	Baker	ACCT	SR
350	Russell	CSCI	JR
400	Rye	CSCI	FR
450	Jones	DATA	SR

Class		
Name	**Time**	**Room**
BA200	TR9	SC110
DA210	MWF3	SC213
BF410	MWF8	SC213
CS150	MWF2	EA304
CS250	MWF12	EA304

EnrolledIn	
STID	**ClassName**
100	DA210
150	BA200
200	DA210
200	CS250
300	CS150
400	BA200
400	BF410
400	CS250
450	DA210

3. `Salesperson-Customer-Order database (SCO):` This database attempts to model a sales-based company's salespeople, customers, and the orders those customers have placed.

Salesperson		
Name	**Age**	**Salary**
Abel	63	120000
Baker	38	42000
Jones	22	36000
Murphy	42	50000
Smith	59	118000
Abernathy	22	36000

Customer	
Name	**City**
Abernathy	Miami
Broadway	Charlotte
Office Pro	Charleston
Amalgamated	Charlotte

Order			
Number	**Salesperson**	**Customer**	**Amount**
100	Abernathy	Abernathy	560
200	Jones	Abernathy	2500
300	Abel	Broadway	480
400	Abel	Office Pro	2500
500	Murphy	Amalgamated	6000
600	Jones	Abernathy	7000
700	Jones	Abernathy	2500

4. Database for Managing Rental Property In Multiple Cities (based on the *DreamHome* database used in the book *Database Systems* (6th edition) by Thomas Connolly and Carolyn Begg, Pearson Publishing, 2015)

Branch

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

Staff

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

PropertyForRent

propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
PA14	16 Holhead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93	SG37	B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003

Client

clientNo	fName	lName	telNo	prefType	maxRent	eMail
CR76	John	Kay	0207-774-5632	Flat	425	john.kay@gmail.com
CR56	Aline	Stewart	0141-848-1825	Flat	350	astewart@hotmail.com
CR74	Mike	Ritchie	01475-392178	House	750	mritchie01@yahoo.co.uk
CR62	Mary	Tregear	01224-196720	Flat	600	maryt@hotmail.co.uk

PrivateOwner

ownerNo	fName	lName	address	telNo	eMail	password
CO46	Joe	Keogh	2 Fergus Dr, Aberdeen AB2 7SX	01224-861212	jkeogh@lhh.com	********
CO87	Carol	Farrel	6 Achray St, Glasgow G32 9DX	0141-357-7419	cfarrel@gmail.com	********
CO40	Tina	Murphy	63 Well St, Glasgow G42	0141-943-1728	tinam@hotmail.com	********
CO93	Tony	Shaw	12 Park Pl, Glasgow G4 0QR	0141-225-7025	tony.shaw@ark.com	********

Viewing

clientNo	propertyNo	viewDate	comment
CR56	PA14	24-May-13	too small
CR76	PG4	20-Apr-13	too remote
CR56	PG4	26-May-13	
CR62	PA14	14-May-13	no dining room
CR56	PG36	28-Apr-13	

Registration

clientNo	branchNo	staffNo	dateJoined
CR76	B005	SL41	2-Jan-13
CR56	B003	SG37	11-Apr-12
CR74	B003	SG37	16-Nov-11
CR62	B007	SA9	7-Mar-12

Properties of Relations

1. ***Each relation within a database must have a name.***
   - Two relations in the same database cannot have the same name, although the same relation name can be used in different databases.

   Examples of Relation Names: ***Done in class***

2. ***Each attribute of each relation must have a name***.
   - While within a relation the names used for attributes must be distinct, two different relations in the same database can have one or more attributes with the same names.

   Examples of Relation Attributes: ***Done in class***

3. ***Each attribute must have an associated "type".***
   - The ***type*** defines the structure of the values that will be associated with that attribute. Among the most common data types used in relational databases are[1]:
     + integers, commonly represented as **INTEGER** or **INT**;
     + real numbers -- which may be further classified as decimal, floating point (float), or double precision (double). For our purposes we will use either **DOUBLE**, or a specification of the form **DECIMAL(***size***,** *d***)**, where *size* is the total number of digits and *d* is the number of digits after the decimal point;
     + fixed-length sequences (or strings) of characters, commonly represented as **CHAR(length)**, where length is length of the string. Actual values may be padded on the right with spaces, or right-truncated as necessary for the string to have the given length.
     + variable-length character strings, represented as **VARCHAR(maxLen)**, where *maxLen* is the maximum length that a string can have.;
     + dates, which we shall simply represent as **DATE**.

   - The values of these types must be atomic - that is, when one wants to retrieve a value from, or store a value in, an attribute of this type, the entire value is stored or retrieved. For further clarification, if one had a variable-length string representing a person's name, one cannot retrieve just the person's first name in one operation. Instead one must retrieve the entire name and then use a separate string operation to get the first name.

   - In general when one merely looks at examples of relations one can only *infer* the type of an attribute from the appearance of its values. The actual type of an attribute must be documented somewhere, however.

   Examples of Relation Types: ***Done in class***

4. ***Each relation must have one or more attributes whose combined values are (always) sufficient to distinguish (or identify) each tuple in the relation***. Such a group of attributes is known as the ***primary key*** of the relation.

   - It is usually easiest to work with single-attribute keys. Multi-attribute keys are permitted, however, and are known as ***composite keys*** (or ***compound keys***).

   - It is not uncommon for there to be several attributes or groups of attributes that could serve as a primary key. Each such attribute or attribute group is known as a ***candidate key***.

   Examples of Primary Keys: ***Done in class***

5. Some relations may have an attribute whose values are expected to match those of the primary key from another relation. Such an attribute is known as a ***foreign key***.
   - Note again, foreign keys only reference primary key attributes. A relation may have a foreign key that references its own primary key.

[1] The data types we described here represent some of the data types originally used with relations. As other types of data emerged in popularity, such as images, sounds, video, objects, etc. many vendors of relational databases extended their supported data types to accommodate these.

- Although it is possible to have multi-attributes foreign keys that reference multi-attribute primary keys all of our foreign keys will be single-attribute foreign keys referencing single-attribute primary keys.

Examples of Foreign Keys: ***Done in class***

Section 3. Database Schemas

A database's ***schema*** refers to the organization of the database, including the set of relations, and for each relation the names of the attributes and their types, the relation's primary key, and any foreign keys. For our purposes, we will present the organization of each table in one of two formats:

1. In the first format, we give a complete specification that includes the name of each relation/table and then for each table the name and type of each of its attributes. We also give the attribute(s) constituting the primary key and also list all foreign key attributes and the table and attribute it references.

   The notation we use here is our own. Text in bold must be present, values in italics are ones the database designer provides

```
Table TableName1
      Attribute attribute1
         Type dataType1
      Attribute attribute2
         Type dataType2
      Attribute attribute3
         Type dataType3
      ...
      Attribute attributen
         Type dataTypen

      Primary Key (comma-separated list of attribute names)
      Foreign Key (attributeA) References tableName(attributeName)
      Foreign Key (attributeB) References tableName(attributeName)
      ...
      Foreign Key (attributeK) References tableName(attributeName)

Table TableName2
   similar structure to above
...
Table TableNameM
   similar structure to above
```

   Database Schema Example: Illustration of database schemas based on the example databases given. ***Done in class***

2. Later on we will be working in situations where it is most important to know just the names of relations and their attributes, and each relation's primary key and foreign keys. For this purpose we can use a much more compact notation based on the following notation

```
TableName(att1, att2, att3,…attn)
```

   The ***primary key*** attribute(s) will be ***underlined*** and any ***foreign keys*** will be ***circled or placed in bold-italics***. For foreign keys, we shall assume that the attribute being referenced by the foreign key is apparent from the foreign key's name, although of course this may not always be the case.

   We give no type information in the schema, nor any information as to whether attribute values must be present in a tuple. Full details on each table's structure, including attribute types and a precise accounting of foreign key references, must come from a more detailed specification such as the first one we specified.

Examples: Illustration of database schemas based on the four databases given. ***Done in class***

Example: We give a schema based on approach 2 above for a simple relational database, UNIVERSITY, to represent some data commonly used for colleges or universities. This is a more elaborate and slightly more realistic structure than the Student-Class-Enrolled in database we saw earlier. As such, it involves more relations and more attributes per relation.

DEPARTMENT(DID, DeptName, OfficeLocation, ***ChairID***)

FACULTY(FID, LastName, FirstName, Rank, Tenured, ***DepartmentID***)

STUDENT(STID, LastName, FirstName, Major, Level, GPA, ***AdvisorID***)

COURSE(CID, Title, DeptOffering, CreditHours)

SECTION(SectionID, ***CourseID***, SectionNumber, Semester, Year, ***InstructorID***)

CLASS(***SectID***, ***StudentID***, Grade)

Note that in the relation CLASS the attributes *SectID* and *StudentID* are both components of a compound key, and each is a foreign key as well.

Example: Now let us specify this same schema using the more expansive, and precise, format in our first approach. We designate:

integers with the notation **`INTEGER`**,

decimal values with the notation **`DECIMAL(p,s)`** where p denotes the maximum number of digits for the number and s denotes the number of digits to use to the right of the decimal point.

fixed-length character strings with the notation **`CHAR(n)`**, where n is the fixed size of the character string.

variable length character strings with the notation **`VARCHAR(n)`**, where n represents the maximum length of such a string can attain. For each case we cannot use a symbol n as we've done here, but must provide an actual value. Deciding on such values is an important design decision that must be made when defining a database.

This example will be completed in class.

Section 4. Defining a Database Using the Data Definition Features of SQL

SQL ("structured query language," pronounced S-Q-L by some, and "sequel" by others) is a language developed for relational database systems that has become the de facto standard for working with relational databases. SQL has one set of constructs for data definition, which includes defining tables, and has another set of constructs for data manipulation, which mostly encompasses extracting information from the database, but which also involves inserting, deleting, or modifying the data already in the database (the data manipulation/query features of the language).

Fundamental Data Types in SQL

The main data types available for attributes 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 to the right of 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. There is also a combined **DATETIME** type.

Table Definition in SQL

The `CREATE TABLE` construct is used to specify relations in SQL.

In its simplest form one specifies only the name of the relation/table and the name and type of the table's attributes ***Pay particular attention to the use of parentheses, commas, and the semi-colon***. Misuse of these can result in hard-to-spot errors when you attempt to use them with an actual database management system.

Under this type of declaration SQL will assume that the table has a compound primary key consisting of all of the attributes, and that a value must be provided for each attribute. There are no foreign keys in the table.

```
CREATE TABLE RelationName
( AttributeName1  type,
  AttributeName2  type,
...
  AttributeNamen  type
);
```

Example: Consider the relation **Department** with the table schema given in the example above and the data types as specified in *Lab 1 - Database Creation*. We could specify this table in SQL via the statement

```
CREATE TABLE DEPARTMENT
(
  DID                   CHAR(4),
  DeptName              VARCHAR(30),
  OfficeLocation        VARCHAR(255),
  ChairID               CHAR(8)
);
```

This representation is inaccurate on many levels (the primary key, attributes that do not require a value, failing to identify ChairId as a foreign key), but provides a good starting point for grasping the basic structure of a table specification in SQL)

Attribute and Table Constraints

As we noted in our example above, the basic SQL table specification we gave will most likely be inaccurate in its specification of primary keys, of attributes that do not require a value for a tuple (allowing the tuple to have what is known as a **null** value), and of foreign keys.

The table definition can accommodate these specifications by associating what are known as *constraints* with either an attribute or the table as a whole. Later we shall see how to incorporate even more general constraints into a relational model.

1. ***Attribute constraints***: The following constraints can be applied to an attribute as it is specified:

   a. *NOT NULL constraint*: SQL allows **null** values and for most attributes this is the default constraint for an attribute (meaning you do not have to note it when you specify an attribute). As a result one specifies a constraint of **`NOT NULL`** for an attribute if the attribute should not be allowed to have the value **null** (that is, it must have a value of the associated type).

      Example: In the table declaration for **DEPARTMENT** we can specify that the attribute *DID* not allow **null** values as follows

      ```
      DID   CHAR(4) NOT NULL,
      ```

   b. *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 no default value is specified, a default value of **null** is used.

For example, later when we define the STUDENT table we will specify that GPA be given a default value of 0.000 as follows

```
GPA   DECIMAL(4,3)   DEFAULT 0.000,
```

c. *PRIMARY KEY constraint:* When a relation has a single attribute for its primary key one can designate this when the attribute is specified by giving it the property **PRIMARY KEY**. Note, this automatically associates **NOT NULL** with the attribute as well.

Example: In the table declaration for **DEPARTMENT** we could designate the attribute *DID* as the table's primary key as follows

```
DID   CHAR(4)   PRIMARY KEY,
```

Although this is a viable way for designating single-attribute primary keys, the more common way to do is by means of a table constraint, which we describe below. **You cannot designate a composite primary key for a table by associating the attribute constraint PRIMARY KEY with each of the component attributes.**

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 key constraints and foreign key constraints. Although not required, table constraints can be given a name so that it can be identified later in case it is necessary to drop the constraint and replace it with another. As we suggested above, in general table constraints are more widely used (even where one attribute is involved).

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

```
PRIMARY KEY(PKatt1,...,PKattn)
```
or
```
CONSTRAINT constraint-name  PRIMARY KEY(PKatt1,...,PKattn)
```

b. **Foreign Keys**: With the *foreign key* constraint, one can designate a foreign key, the table and primary key it references.

```
FOREIGN KEY (FKatt) REFERENCES table-name(att)
```
or
```
CONSTRAINT constraint-name
  FOREIGN KEY(FKatt) REFERENCES table-name(att)
```

Important Note: The table being referenced must already been defined before you can use it in a foreign key reference in another relation.

- This may require some planning ahead of time in terms of the order in which tables will be specified in SQL.
- Sometimes there is no way to order table specifications to avoid a foreign key referencing a table before the table has been defined (our University database is an example of this). If this is the case, then we must wait to add any foreign keys after the table being referenced has been created.

Example: Consider the **university** database schema given earlier. We specify the relational model in SQL as follows (here we use table constraints over attribute constraints). The way we split the elements of each command over several lines is our own convention to make it clear what we are defining. SQL does not require this, and indeed we can create the entire database using one line. If you do this, however, you are asking for trouble as inevitably you will make an error somewhere and it will then be near-impossible to figure out where the error occurred.

```
CREATE TABLE DEPARTMENT
(
  DID                   CHAR(4),
  DeptName              VARCHAR(30) NOT NULL,
  OfficeLocation        VARCHAR(255),
```

```
  ChairID              CHAR(8),

  PRIMARY KEY (DID)
);

CREATE TABLE FACULTY
(
  FID              CHAR(8),
  LastName         VARCHAR(20)      NOT NULL,
  FirstName        VARCHAR(20)      NOT NULL,
  Rank             CHAR(4),
  Tenured          CHAR(1),
  DeptID           CHAR(4),

  PRIMARY KEY (FID),
  FOREIGN KEY (DeptID) REFERENCES DEPARTMENT(DID)[1]
);

CREATE TABLE STUDENT
(
  STID             CHAR(8),
  LastName         VARCHAR(20)      NOT NULL,
  FirstName        VARCHAR(20)      NOT NULL,
  Major            CHAR(4),
  Level            CHAR(2),
  GPA              DECIMAL(4,3),
  AdvisorID        CHAR(8),

  PRIMARY KEY (STID),
  FOREIGN KEY (AdvisorID) REFERENCES FACULTY(FID)
);

CREATE TABLE Course
(
  CID              CHAR(8),
  Dept             CHAR(4),
  Title            VARCHAR(50),
  Credits          SMALLINT,

  PRIMARY KEY (CID),
  FOREIGN KEY (Dept) REFERENCES DEPARTMENT(DID)
);

CREATE TABLE Section
(
  SectID           CHAR(6),
  CourseID         CHAR(8)    NOT NULL,
  SectionNumber    CHAR(3)    NOT NULL,
  Semester         CHAR(4)    NOT NULL,
  Year             CHAR(4)    NOT NULL,
  InstructorID     CHAR(8),

  PRIMARY KEY (SectID),
  FOREIGN KEY (CourseID) REFERENCES COURSE(CID),
  FOREIGN KEY (InstructorID) REFERENCES FACULTY(FID)
);
```

[1] For now we are going to leave out constraint names in our table constraints, especially for foreign keys, to make it easier to see the more critical syntax. Later on, however, we will be using named constraints.

```
CREATE TABLE Class
(
  SectID           CHAR(6),
  StudentID        CHAR(8),
  Grade            CHAR(2),

  PRIMARY KEY (SectID, StudentID),

  FOREIGN KEY (SectID) REFERENCES SECTION(SectID),
  FOREIGN KEY (StudentID) REFERENCES STUDENT(STID)
);
```

Note, we were not able to define the foreign key ChairID in the DEPARTMENT relation because the FACULTY table had not yet been created at the time that DEPARTMENT was created. Such constraints can be handled later, however using the

```
ALTER TABLE …ADD CONSTRAINT
```

construct of SQL. In this particular case we would write

```
ALTER TABLE DEPARTMENT
  ADD CONSTRAINT FK_DEPARTMENT_ChairID
    FOREIGN KEY (ChairID) REFERENCES FACULTY(FID)
```

Creating a Database in MySQL Using SQL Commands

Complete Lab 0 - Install MySQL and MySQLWorkbench

Complete Lab 1 – Database Creation Using SQL

Section 4 Working with Data in Tables

Entering Data into a Table

There are two methods that that are commonly used to enter data into a database (referred to as "populating a database")

1. **Entering data via a user interface provided by the DBMS**.

   In MySQL Workbench this can be done using the **Edit Table Data** feature of the SQL Development panel.

   We will examine how to do this via MySQL Workbench in **Laboratory 4 – Working with Table Data**

2. **Entering data via SQL statements.**

   Using SQL one can add tuples to a given table with the **INSERT** statement. There are two forms of the INSERT statement that are germane to us here:

   a. To add a single tuple to a relation by giving the name of the relation that is to receive the tuple and the complete list of values. The values must be presented in the same order as the one in which the corresponding attributes were listed in the CREATE TABLE statement that created the relation. The form of the INSERT statement is

      INSERT INTO *table-name*
      VALUES ($value_1$**,...,**$value_n$**)**

      Notes:
      - When entering character data (either **varchar** or **char**) for a value, the value must be enclosed in single quotes or double quotes (but you cannot use single for one side and double for the other).

- We use the keyword **null** to indicate a NULL value (do not surround **null** with quotes; enter **null**, not **"null"**).

b. To add a single tuple to a relation by directly associating attributes and their value. In this case we use the form

```
INSERT INTO table-name (attribute-name1,...,attribute-namen)
VALUES (value1,...,valuen)
```

Any attributes not specified in this form have their values set to their default value or to **null**.

Examples: We will use the **university** database defined earlier in this section.

1. Add a course DATA210 with the title "Dataset Organization and Management" with 3 credits and associated with the CSCI department.

```
INSERT INTO COURSE
VALUES ('DATA210', 'CSCI', 'Dataset Organization and Management',3)
```

2. Add a 3-credit course DATA220 associated with the CSCI department but with a title as yet to be determined.

```
INSERT INTO COURSE
VALUES ('DATA220', 'CSCI',null,3)
```

or,

```
INSERT INTO COURSE(CID, Dept, CreditHrs)
VALUES ('DATA220', 'CSCI',3)
```

or even,

```
INSERT INTO COURSE(CreditHrs, CID, Dept)
VALUES (3,'DATA220', 'CSCI')
```

If you want to enter values for several tuple, you can just type the INSERT INTO... line once, and then afterwards enter several VALUES lines as shown below, each one ending with a comma except for the last one, where is semi-colon is probably appropriate.

```
INSERT INTO table-name
VALUES
      (value11,...,value1n),
      (value21,...,value2n),
      (valueK1,...,valueKn);
```

Example: Entering multiple tuples into the faculty table with one INSERT INTO statement

```
INSERT INTO department
VALUES
  ('F0000001', 'Manaris', 'Bill','FULL','Y','CSCI'),
  ('F0000002', 'Pothering', 'George','FULL','Y','CSCI'),
  ('F0000003', 'Stalvey', 'RoxAnn','SRIN','N',null),
  ('F0000004', 'Sun', 'Jonathan','ASST','N','CSCI');
```

Modifying the Tuples in a Table

When we talk about modifying tuples in a table me mean one of two actions:

- Removing a tuple from the table – called a **delete**.

- Changing the values of one or more attributes of a tuple – called an **update** of the tuple.

Deleting tuples from a relation: In SQL this is accomplished by the DELETE command. This has only one form

```
DELETE FROM table-name
WHERE condition
```

It will remove from the specified table all tuples whose attribute values satisfy the given condition. WE won't say much about conditions now, but will illustrate some through examples.

Examples:

1. Delete all tuples in the **course** table that are associated with the CSCI department.

```
DELETE FROM course
WHERE Dept='CSCI'
```

2. Delete all 4 credit hour CSCI courses from the **course** table

```
DELETE FROM course
WHERE Dept='CSCI' and CreditHrs=4
```

3. Delete all tuples from the **class** table

```
DELETE FROM class
```

   This one is dangerous!! Inadvertently placing a semi-colon after the table name in an otherwise valid statement could wipe out a whole table as the following example shows:

```
DELETE FROM class;
WHERE SectID = 'A12345' and Grade is null;
```

   This command may intend to drop from section A12345 all students who do not have a grade entered, but the semi-colon after class will cause that line, which is syntactically correct and complete, to be executed. The following line, however, will be regarded as a syntactically incorrect command. Fortunately most DBMSs provide for ways to avoid such potential disasters.

Updating the values of tuples in a table

In SQL changing the values of attributes of tuples in a table is accomplished by the UPDATE command. There is only one form for this command:

```
UPDATE table-name
SET att1 = value1, att2 = value2,...,attp = valuep
WHERE condition
```

Each tuple in the given table who attribute values satisfy the given condition will have the present values of the specified attributes changed to those given. It is possible to designate **null** or **default** as a new value for an attribute.

Example: Change the affiliation of all DATA courses from CSCI to DATA.

```
UPDATE course
SET Dept='DATA'
WHERE Dept='CSCI'
```

More Lab Practice in MySQL/MySQLWorkbench

Complete Lab 2 - Database Backup and Restore
Complete Lab 3 - Database Creation Using MySQLWorkbench
Complete Lab 4 - Managing Table Data