Module 6

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Learning Objectives

After successfully completing the module, students will be able to do the following:

1. Describe relational database concepts.
2. Use SQL to create and drop tables and also to retrieve and modify data.
3. Use the JDBC API to access databases and process *ResultSet*s.
4. Use *PreparedStatement*s to execute pre-compiled SQL statements.
5. Use *CallableStatement*s to execute stored SQL procedures.
6. Use the DatabaseMetaData and ResultSetMetaData interfaces.
7. Apply the concepts of TCP, IP, and Internet address.
8. Create servers using server sockets and clients using client sockets.
9. Implement Java networking programs using stream sockets.
10. Develop servers for multiple clients.
11. Send and receive objects on a network.

Module 6 Study Guide and Deliverables

February 20 – February 26

Topics:	• Lecture 6: Batch and Stream • Lecture 7: Java Networking Connectivity
Readings:	• Module 6 online content • Deitel & Deitel, Chapter 24 and Chapter 28 • Instructor's live class slides
Assignments:	No Assignment for this module
Live Classrooms:	Join the live classroom and the facilitator's live office hour session at "Live Classroom/Offices" > "Live Classroom" > Launch Meeting. • **Wednesday, February 21, 6:00 – 8:00 PM ET** • Live Office: Schedule with facilitators as long as there are questions
Course Evaluation:	Course Evaluation opens on **Monday, February 26, at 10:00 AM ET** and closes on **Sunday, March 3, at 11:59 PM ET**.

Please complete the course evaluation. Your feedback is important to MET, as it helps us make improvements to the program and the course for future students.

Module Welcome and Introduction

met_cs622_18_su1_ebraude_mod6 video cannot be displayed here

Lecture 6 – Java Database Connectivity

Relational Database

A database is a repository or storehouse of data. The data in a database contain information about items of interest. A database management system (DBMS) is the collection of software that stores and manages data in a database. Application programs are generally built on top of the DBMS for users to access the database. Most of the present-day DBMSs are relational database management systems or RDBMS. RDBMSs are based on the "relational" data model.

A relational data model is based on the concept of data being stored in relations or tables. In an RDBMS, a "relation" and a "table" are taken to be synonymous. Conceptually, a table is a two-dimensional construct consisting of rows and columns. Each table has a name so that it can be identified. Each column has a name and stores one piece of information. Each row one set of values for all columns in the table. As an example, here is a table for recording information about people.

Person Relational Table

first_name	last_name	birth_date
Bob	Smith	3/27/1974

Jane	Glass	5/19/1999
Vishnu	Santhana	12/13/1950

There is a first_name, last_name, and birth_date column in the table, to store a person's first name, last name, and birth date, respectively. There are three rows in the table. The first row is for Bob Smith, born 3/27/1974. The second is for Jane Glass, born 5/19/1999. The third is for Vishnu Santhana, born 12/13/1950. You can see by looking at this example, each row stores information about one person. Rows of a table are also referred to as tuples or records. Generally, each record in a table contains information about one real or abstract object or thing.

Each table represents a real or abstract concept. For example, the Person table represents the concept of a person. Each row represents one individual person. You can also think of a table as a collection of real or abstract objects. For example, the Person table contains a collection of people. An RDBMS often contains multiple tables, each with its own name.

Every column has a datatype which specifies the type and maximum size of the values in the column. The permissible values are defined by the datatype. Three common datatypes are VARCHAR, DECIMAL, and DATE. VARCHAR allows for a series of characters up to a maximum length. DECIMAL allows for numbers, and whether decimal points are allowed, and how many digits are allowed, is customizable. DATE allows for dates. There are other datatypes, and every database also has its own specific datatypes, but these three are very common.

The relational model also supports relationships between rows. When the same data value is duplicated in two different rows of the same or different tables, those rows are related. Typically, rows in different tables are related, such as between a Person and Address. However, in some cases rows in the same table are related. For example, an Employee can be related to another employee through a "manages" relationship since a manager is also an employee.

In modern times, we are used to working with spreadsheet applications. Each table is akin to a worksheet in a spreadsheet application.

Integrity Constraints

Integrity constraints provide conditions that must at all times be satisfied by the values in the tables. Relational databases support 5 kinds of constraints on tables – primary key, foreign key, not null, unique, and check. These low-level constraints are used to ensure basic data integrity. Violating these constraints would make the data invalid.

Primary key constraint:

A primary key is a column or set of columns that uniquely identifies every row. Primary key values are used as the basis for creating a reference to a table. You can think of a primary key as the "address" of a table row. A primary key constraint enforces the fact that the column(s) are unique and not null. For example, A person_id column in a Person table can be assigned a primary key constraint.

Foreign key constraint:

A foreign key constraint enforces the fact that a reference is valid. Each value specified in a foreign key column must exist in the table being referenced. For example, If an Employee table has a foreign key to an Address table via an address_id column, any value in the address_id column in Employee must exist in Address (i.e. the reference must be valid).

Not null constraint:

A not null constraint requires that every row in the table must have a value for that column. For example, often the last_name column in a Person table is not null so that a last name is required for every person.

Unique constraint:

A unique constraint requires that every value in the table for the column or set of columns must be unique (that is, each value must appear only once). Contrary to popular belief, a unique constraint does not require the column also have a not null constraint. For example, If a person table already has a person_id primary key, but also has a social_security_number field, the social_security_number field can be given a unique constraint to ensure every person's number is unique. This would catch bad data entry, for example.

Check constraint:

A check constraint supports for a Boolean expression that can involve any of the table's columns, to check for cross-column integrity. The other four constraints apply a narrow condition to a column or set of columns. A check constraint applies any Boolean expression to any or all columns in a table.

For example, imagine that an Employee table has an hourly_rate column for hourly employees, and a yearly_salary column for salaried employees. Any particular employee will have a value in only one of these, since they will either be an hourly employee, or a salaried employee. A check constraint can be used to ensure that one of the fields must have a value, but both fields do not simultaneously have a value. None of the other constraints make this possible.

Integrity constraints are continually enforced by the database management system. The database aggressively rejects actions that would violate these constraints.

Relational/Object-Oriented Isomorphism

The relational model and the object-oriented model have several isomorphic concepts. It's useful for programmers to understand how tables and relationships in a relational database are isomorphic with classes and references.

Table/Class Isomorphism

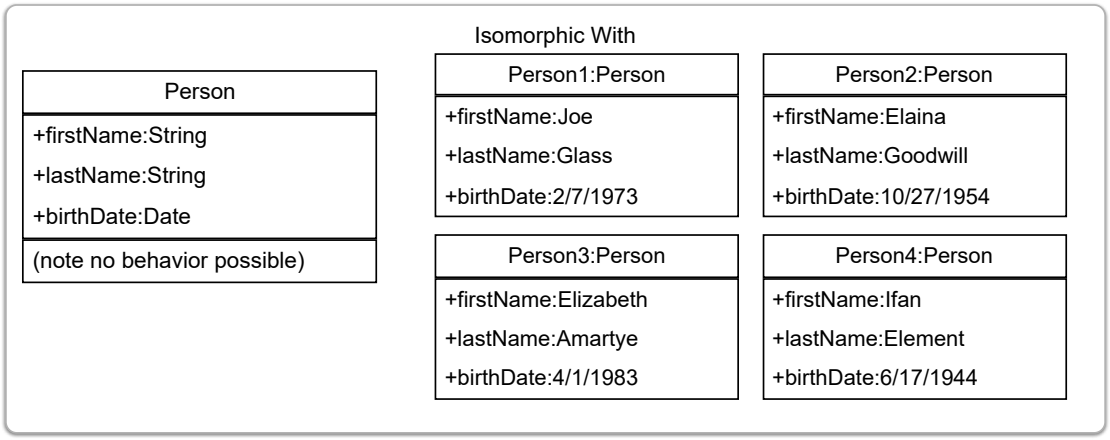
A relational table is isomorphic with a Java class that has no behavior, that is, has no methods. There is no concept of table methods in a relational database table; tables store data. Table columns are isomorphic with public member variables of a class. There is no concept of protected, package, or private data in a relational table, so effectively all columns are "public". Table rows are isomorphic with objects of a class. One row is one object.

Let's look at the example below of a Person table and how it translates to the object-oriented model.

Person Table

first_name	last_name	birth_date
Joe	Glass	2/7/1973

Elaina	Goodwill	10/27/1954
Elizabeth	Amartye	4/1/1983
Ifan	Element	6/17/1944



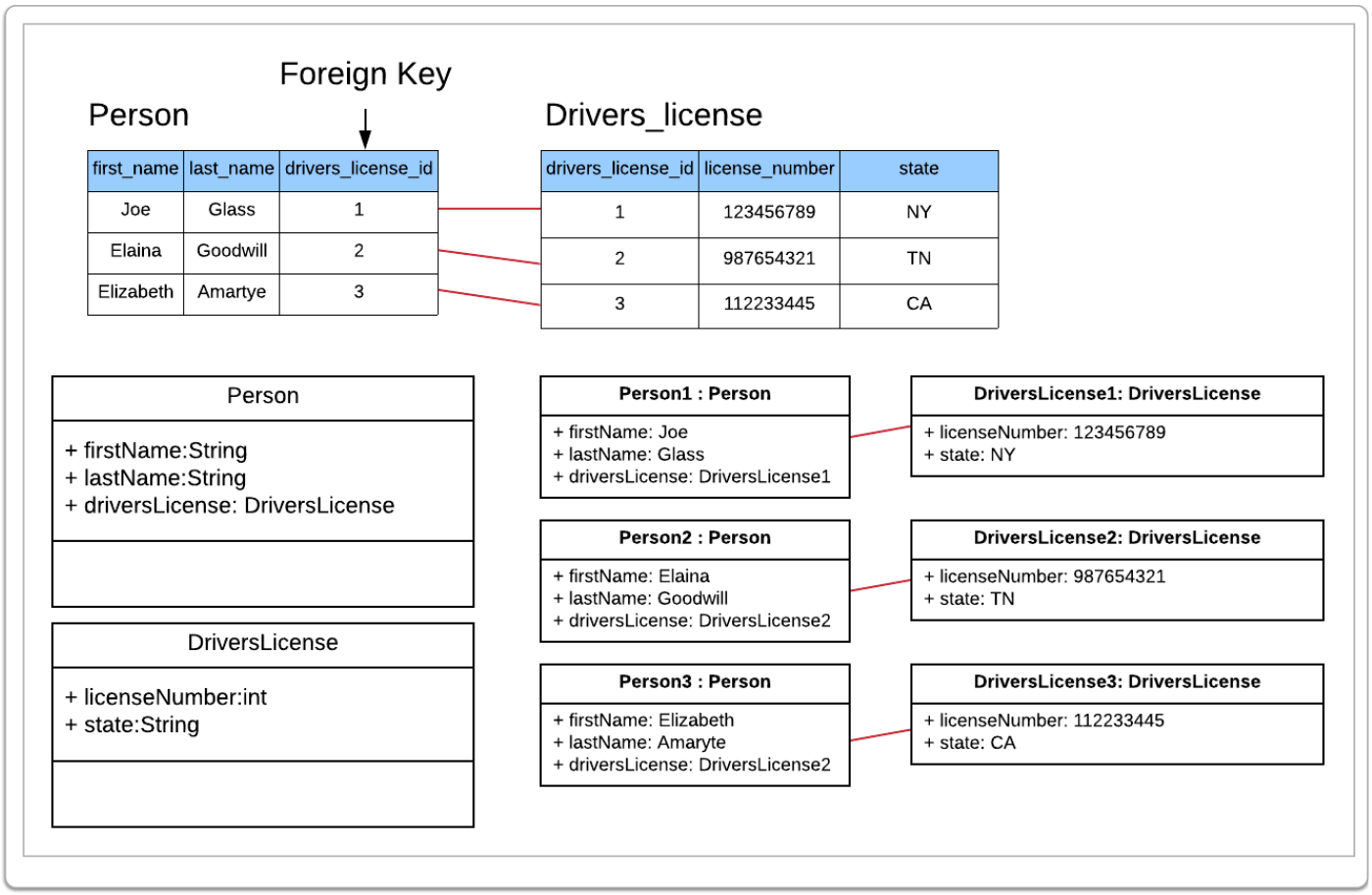
The Person table has three columns, first_name, last_name, and birth_date. An isomorphic Person class would have three public member variables – firstName, lastName, and birthDate – with no methods in the class. The Person table has four rows, and these would be four objects in the object-oriented model, one for each row, as the diagram illustrates.

Relationship/Reference Isomorphism

Relationships in the relational model are created by duplicating data values. For example, if a value in one row the value “5”, and a value in another row has the value “5”, we can consider these rows as related. Relationships are enforced through foreign-key constraints, essentially to ensure that the relationship is valid.

Relationships in the object-oriented model are both created and enforced through references. One object can reference another object.

The example below illustrates that relationships and references are isomorphic.



In the example, the Person table is related to the Drivers_license table through a foreign key on drivers_license_id. This is equivalent to a Person class having a reference to a DriversLicense class in the object-oriented model. Notice that in the relational model, the values that create the relationship are accessible. We can see that driver's license id 1 exists in both tables, for example. We can access and change these values as well. In the object-oriented model, the underlying address that creates the reference is not accessible. We can change which object is being referenced, but cannot see or modify the internal mechanics of the reference.

A Publishing Database

In what follows, we create and use a SQLite database named Module6Lecture.db (more about the SQLite relational database appears later in this module). It is a publishing database. The database contains information on researchers, journals and publications.

- Researchers write up the results of their research in the form of papers that appear in published form in journals. Information stored about a researcher includes his or her ID (unique), first name and last name.
- A journal is represented by its ID (unique), journal title and the publisher's name.
- A given publisher may publish multiple (different) journals. A given journal is published by one and only one publisher. A given researcher can publish (different) papers in different journals. A given researcher never publishes the same paper in more than one journal. A given journal publishes papers by different researchers. A given journal never publishes a given paper more than once.

The database design may include the following relations (tables):

- researcher
  - Attributes: rid, fname, lname
  - Primary key: rid

- journal
  - Attributes: jid, title, publisher
  - Primary key: jid
- publication
  - Attributes: rid, jid, quantity
  - Primary key: rid and jid taken together
  - Foreign key: rid (from Table Researcher), jid (from Table Journal)

The quantity field in the publication table stores the number of papers published by a given researcher in a given journal. It is conceivable that with time, as a researcher publishes new papers in a journal in which he/she had previously published, the quantity field will be incremented. The section on SQL (later in this module) explains how to create these tables from scratch and how to load them with data.

SQLite Overview

We use the SQLite database to illustrate database connectivity in Java throughout this lecture. Before proceeding with the rest of this lecture, you'll want to follow the steps in the Getting Started with SQLite tutorial. That tutorial will get you up and running with SQLite and its associated SQL client, DB Browser for SQLite.

SQLite is the most used, embedded (serverless) relational database in the world. It is open source and free to use. Unlike server-based databases like Oracle and SQL Server, SQLite runs entirely in the application that uses it, and stores all of its durable objects in a single disk file. SQLite can be used across all major platforms, which means the database file can be freely copied and used across devices with difference architectures. SQLite is ideal for applications that would traditionally use files to store data, giving them access to the power of a relational database without the expense and overhead of installing and maintaining a server-based database.

Once you learn to access and use any one modern relational database, you can use the others without much additional effort. All modern relational databases utilize the Structured Query Language (SQL) for data access and manipulation. SQL is highly standardized across databases. Although there are some differences, the significant aspects are the same across databases. In addition, Java supports a standardized API, JDBC, for accessing any database. Connectivity from Java does not differ much between databases. Thus SQLite is an excellent first database for Java developers, because the intricacies of relational databases and connectivity can be learned without the overhead of database installation, yet SQLite is used in serious applications worldwide.

DB Browser for SQLite Overview

It is a best practice to manage your database's structure with a SQL client. Typically, we use a SQL client to first add the tables, indexes, and triggers (if needed), as well as any initial data. Then when our application executes, it will add, modify, and remove data as needed with JDBC, but not modify the structure of the tables and indexes. By separating structure manipulation from data manipulation, we can carefully apply good database design principles, and avoid embedding table structure in our application.

A popular SQL client for SQLite is DB Browser for SQLite. The Getting Started with SQLite tutorial shows you how to install it and create a table with it.

In this lecture, we use DB Browser for SQLite on publishingdb to illustrate several frequently used features of SQL. The following types of SQL statements will be discussed:

- Create and destroy tables: create; drop

- Insert, delete, update tuples: insert; delete; update
- Queries on single tables: select; select distinct; select with aggregate functions; select with the following clauses:
    - where
    - order by
    - group by
    - having
- Queries on multiple tables

SQL

SQL (pronounced "sequel" or "S Q L"), or *structured query language*, is a general language for creating, modifying, and accessing tables in an RDBMS. Different RDBMSs use essentially the same SQL language (with minor variations); in this sense SQL is a universal database language.

Create and Drop

The SQL statement to create a table specifies the table name, attributes (columns or fields), and types of the attributes. The following statement creates a table named "researcher":

```
create table researcher (
    rid int not null,
    fname varchar(20) not null,
    lname varchar(30) not null,
    primary key (rid)
);
```

In the above example, the researcher table has three attributes: "rid" (researcher ID), "fname" (first name), and "lname" (last name), with rid specified as the primary key. The statement also stipulates that none of the attributes can have a null value. The rid field is an integer, while fname and lname are variable-length strings that can be up to 20 and 30 characters long, respectively.

The above create statement (and other SQL statements) can be executed within a SQL client such as DB Browser for SQLite, or from within a Java program (to be discussed later in this module). It is a best practice to manage your database's structure with a SQL client, and run the data commands in your application. This means commands like CREATE, ALTER, and DROP are typically executed in a SQL client, because they are one-time commands, and commands like SELECT, INSERT, UPDATE, and DELETE are typically executed within the application, because they are executed over and over again as the data changes.

The tables journal and publication are created as follows:

```
create table journal (
    jid int not null,
    title varchar(30) not null,
    publisher varchar(20) not null,
    primary key (jid)
);
create table publication (
    rid int not null,
    jid int not null,
    quantity int not null,
    primary key (rid, jid),
    foreign key (rid) references researcher(rid),
```

```
        foreign key (jid) references journal(jid)
    );
```

The publication table is specified as having a (composite) primary key consisting of the fields rid and jid, which are the same as the identically-named fields in tables researcher and journal, respectively. We say that the rid field in publication references the rid field in researcher; similarly, the jid field in publication references the jid field in journal. The two fields rid and jid, therefore, are foreign keys in publication.

When a table is no longer needed, it can be destroyed (dropped permanently) using the drop statement. For instance, the following statement drops the table named researcher:

```
drop table researcher;
```

If the table to be dropped is referenced by other tables, then those other tables must first be dropped. For instance, having created the researcher, journal and publication tables by using the above SQL statements, if we wish to drop all the three tables, publication must be dropped before researcher and journal can be dropped.

Insert

Once a table is created, we can insert data (one or more records) into it by using the insert statement. The following example inserts seven records into the researcher table:

```
insert into researcher
values
(10,'David','Goldberg'),
(70,'James','Smith'),
(30,'David','Kaiser'),
(40,'John','Doe'),
(50,'Greg','Rawlins'),
(20,'Henry','Kimura'),
(60,'John','Smith');
```

Similarly, the journal table is loaded using the SQL statement:

```
insert into journal
values
(501,'Photonics','IEEE'),
(701,'Nature','Macmillan'),
(301,'Power Sources','Elsevier'),
(401,'Physical Review','APS'),
(101,'Quantum Electronics','IEEE'),
(201,'Computer','IEEE'),
(801,'Energy','Elsevier'),
(601,'Information Sciences','Elsevier'),
(901,'Nature Photonics','Macmillan');
```

And four records are stored in table publication by using the following insert statement:

```
insert into publication
values
(10, 501, 1),
(20, 701, 5),
(60, 101, 1),
(20, 401, 2);
```

Queries

We will now use the data in these three tables to illustrate the various SQL queries. We are using the DB Browser for SQLite SQL client to demonstrate the results.

The Select Statement

The (ordinary) select statement retrieves the specified fields (attributes) from all the rows of the specified table. For example, the query

```
select title, publisher
from journal;
```

returns the following information:

```
     title              publisher
1 Photonics            IEEE
2 Nature               Macmillan
3 Power Sources        Elsevier
4 Physical Review      APS
5 Quantum Electronics  IEEE
6 Computer             IEEE
7 Energy               Elsevier
8 Information Sciences Elsevier
9 Nature Photonics     Macmillan
```

These results are reflected in the following screenshot:

Using * in the Select Statement

Instead of one or more attributes, a star (*) can be specified to retrieve all the attributes. The query

```
select * from journal;
```

produces the following output:

```
  jid    title               publisher
1 501  Photonics            IEEE
2 701  Nature               Macmillan
3 301  Power Sources        Elsevier
4 401  Physical Review      APS
5 101  Quantum Electronics  IEEE
6 201  Computer             IEEE
7 801  Energy               Elsevier
8 601  Information Sciences Elsevier
9 901  Nature Photonics     Macmillan
```

These results are reflected in the following screenshot:

Select distinct

The "select distinct" feature selects distinct (non-repeating) values, as in the following example:

```
select distinct publisher
from journal;
```

Output:

```
   publisher
1 IEEE
2 Macmillan
3 Elsevier
4 APS
```

These results are reflected in the following screenshot:

Using the "where" Clause in the Select Statement

The "where" clause specifies a condition that the selected tuples must satisfy. For instance, the following query retrieves only those records that correspond to at least two published papers:

```
select * from publication
where quantity >= 2;
```

Output:

```
  rid  jid  quantity
1 20   701  5
2 20   401  2
```

These results are reflected in the following screenshot:

The Operator "like" and the Percent (%) Character

The operator "like" is used in the where clause to match patterns. The percent (%) character matches zero or more characters. Therefore the following query retrieves all researchers whose first name starts with a 'J':

```
select * from researcher
where fname like 'J%';
```

Output:

```
  rid  fname  lname
1 70   James  Smith
2 40   John   Doe
3 60   John   Smith
```

These results are reflected in the following screenshot:

The underscore (_)

The underscore (_) matches exactly one character. The following query outputs all researchers whose last name has 'a' as the second character from the left:

```
select * from researcher
where lname like '_a%';
```

Output:

```
  rid  fname  lname
1 30   David  Kaiser
2 50   Greg   Rawlins
```

These results are reflected in the following screenshot:

The following statement produces all researchers whose last name has ‘r’ in the second place from the right:

```
select * from researcher
where lname like '%r_';
```

Output:

```
  rid  fname  lname
1 10   David  Goldberg
2 20   Henry  Kimura
```

These results are reflected in the following screenshot:

Using the "order by" Clause in the Select Statement

The output of a query can be produced on sorted order (on one or more fields) by using the "order by" clause. The default order is ascending. The following query produces a list of researchers sorted on ascending order of their lastname:

```
select * from researcher
order by lname;
```

Output:

```
  rid  fname  lname
1 40   John   Doe
2 10   David  Goldberg
3 30   David  Kaiser
4 20   Henry  Kimura
5 50   Greg   Rawlins
6 70   James  Smith
7 60   John   Smith
```

These results are reflected in the following screenshot:

The same result is obtained with the following statement:

```
select * from researcher
order by lname asc;
```

Output:

```
  rid  fname  lname
1 40   John   Doe
2 10   David  Goldberg
3 30   David  Kaiser
4 20   Henry  Kimura
5 50   Greg   Rawlins
6 70   James  Smith
7 60   John   Smith
```

These results are reflected in the following screenshot:

Combining the "where" and "order by" Clauses

The "where" and "order by" clauses can be combined:

```
select * from researcher
where rid > 10
order by lname;
```

The sorting is done on only the records with rid greater than 10, producing the following output:

```
  rid  fname  lname
1 40   John   Doe
2 30   David  Kaiser
3 20   Henry  Kimura
4 50   Greg   Rawlins
5 60   John   Smith
6 70   James  Smith
```

These results are reflected in the following screenshot:

Specifying "desc" in the "order by" Clause

The descending order in sorting can be obtained by specifying "desc" in order by:

```
select * from researcher
order by lname desc;
```

Output:

```
  rid  fname  lname
1 70   James  Smith
2 60   John   Smith
3 50   Greg   Rawlins
4 20   Henry  Kimura
5 30   David  Kaiser
6 10   David  Goldberg
7 40   John   Doe
```

These results are reflected in the following screenshot:

The following statement sorts the records on ascending order of lname, and within lname, on ascending order of fname:

```
select * from researcher
order by lname, fname;
```

Output:

```
  rid  fname  lname
1 40   John   Doe
2 10   David  Goldberg
3 30   David  Kaiser
4 20   Henry  Kimura
5 50   Greg   Rawlins
6 70   James  Smith
7 60   John   Smith
```

These results are reflected in the following screenshot:

The following statement produces sorting on descending order of lname, and within lname, on ascending order of fname (James followed by John):

```
select * from researcher
order by lname desc, fname asc;
```

Output:

```
  rid  fname  lname
1 70   James  Smith
2 60   John   Smith
3 50   Greg   Rawlins
4 20   Henry  Kimura
5 30   David  Kaiser
6 10   David  Goldberg
7 40   John   Doe
```

These results are reflected in the following screenshot:

The following operators can be used in the where clause:

Operators in the where Clause

Operator	Meaning
not	Logical negation
and	Logical conjunction
or	Logical disjunction
=	Equal to
!= (or <>)	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Select with Aggregate Functions

The following five aggregate functions can be used in a select command:

- count
- min
- max
- sum
- avg

Each of these functions produces a single value from a group of values. For example, the statement

```
select count(*), min(quantity), max(quantity),  sum(quantity), avg(quantity)
from publication;
```

produces the following output:

```
  count(*) min(quantity) max(quantity) sum(quantity) avg(quantity)
1 4        1             5             9             2.25
```

These results are reflected in the following screenshot:

where the serial numbers 1 through 5 in the column headings indicate the five items selected for retrieval. The results are self-explanatory (4 is the total count of records in the table, the minimum value of quantity from the four rows is 1, the maximum is 5, and so on).

We could make the output more readable by specifying suitable names to be used as column headings, as follows:

```
select count(*) as totalcount, min(quantity) as minimum, max(quantity) as maximum,
sum(quantity) as grouptotal, avg(quantity) as average from publication;
```

Output:

```
  totalcount minimum maximum grouptotal average
1 4          1       5       9          2.25
```

These results are reflected in the following screenshot:

Group by

The "group by" clause forms groups of rows (records) with the same column (field) value. The query

```
select publisher
from journal
group by publisher;
```

produces the following output:

```
   publisher
1 APS
2 Elsevier
3 IEEE
4 Macmillan
```

These results are reflected in the following screenshot:

This is so because the above query produces four groups, one for each of the four different publishers, and then produces a single row of result from each group. The items specified in the select statement must each be single-valued per group.

As another example, consider

```
select publisher, count(publisher) as count
from journal
group by publisher
order by count;
```

Output:

```
   publisher  count
1 APS         1
```

```
2 Macmillan   2
3 Elsevier    3
4 IEEE        3
```

These results are reflected in the following screenshot:

Having Clause

The having clause can be used with the group by clause to filter out one or more groups. Thus it acts like a condition that a group must satisfy for inclusion in the final result (recall that the where clause conditionally filters out rows). For example, the following example eliminates from the final result groups for which the publisher publishes fewer than two journals:

```
select publisher, count(publisher) as count
from journal
group by publisher
having count(publisher) > 1;
```

Output:

```
   publisher  count
1 Elsevier    3
2 IEEE        3
3 Macmillan   2
```

These results are reflected in the following screenshot:

Multi-table Queries: Join

SQL allows us to combine information from more than one table. This is accomplished through the use of columns (attributes) that are common to multiple tables. For instance, all the researchers in the publication table (indicted by their rid field) must be present in the researcher table. (Note, however, that not all researchers in the researcher table are present in the publication table; only those who published a paper in a journal are.) Thus we can join the researcher table and the publication table on the rid attribute. Similarly, the jid attribute joins the two tables journal and publication.

To find the names (first and last) of all researchers who have journal publications (and the journal titles in which they published), we use the following SQL query:

```
select fname, lname, title
from researcher, journal, publication
where journal.jid = publication.jid and researcher.rid = publication.rid;
```

The output is given by:

```
  fname   lname    title
1 David  Goldberg Photonics
2 Henry  Kimura   Nature
3 Henry  Kimura   Physical Review
4 John   Smith    Quantum Electronics
```

These results are reflected in the following screenshot:

Similarly, the following query retrieves all researchers (first and last names) who have published in an IEEE journal:

```
select fname, lname, title
from publication
join researcher ON researcher.rid = publication.rid
join journal ON journal.jid = publication.jid
where journal.publisher = 'IEEE';
```

Output:

```
  fname   lname    title
1 David  Goldberg Photonics
2 John   Smith    Quantum Electronics
```

These results are reflected in the following screenshot:

Update and Delete

The Update Statement

The update statement allows one or more tuples to be updated, usually conditionally. For example, the following statement causes all instances of ‘Macmillan’ to be changed to ‘Macmillan UK’.

```
update journal
set publisher = 'Macmillan UK'
where publisher = 'Macmillan';
```

A "select * from journal" command would now produce the following:

```
  jid    title               publisher
1 501  Photonics            IEEE
2 701  Nature               Macmillan UK
3 301  Power Sources        Elsevier
4 401  Physical Review      APS
5 101  Quantum Electronics  IEEE
6 201  Computer             IEEE
7 801  Energy               Elsevier
8 601  Information Sciences Elsevier
9 901  Nature Photonics     Macmillan UK
```

These results are reflected in the following screenshot:

The Delete Statement

The delete statement deletes, usually conditionally, one or more tuples. For example, the following statement

```
delete from journal
where title = 'Energy';
```

deletes the row corresponding to the journal Energy. The result can be seen by issuing a "select * from journal" query:

```
  jid    title               publisher
1 501  Photonics            IEEE
2 701  Nature               Macmillan UK
3 301  Power Sources        Elsevier
4 401  Physical Review      APS
5 101  Quantum Electronics  IEEE
```

```
6 201  Computer             IEEE
7 601  Information Sciences Elsevier
8 901  Nature Photonics     Macmillan UK
```

These results are reflected in the following screenshot:

SQL – Summary

Of the SQL statements discussed in this module, create and drop statements are examples of SQL data definition language (DDL), while select, insert, delete, and update are examples of data manipulation language (DML).