Objectives:
1. To introduce the relational algebra.
2. To introduce the SQL select statement
3. To introduce the SQL insert, update, and delete statements

Materials:
1. LIBRARY database and database to be used in lab (CS211) for on-line demos
2. Sample query handout
3. ER diagram for Database query lab - handout
4. Listing of contents of tables for database query lab
5. SQL Syntax handout
6. SQL documentation linked from course web page

I. Introduction

A. We have seen how entities and relationships can be represented by using the relational data model, in which information is stored in tables.

1. Each table has a primary key, which is a set of attributes such that no two rows in the table have the same value.

2. A table may represent either an entity or a relationship.
   a) For an entity, the table includes the entity’s primary key and other attributes.
   b) For a relationship, the table includes the primary keys of the entities being related (called foreign keys) plus any attributes of the relationship itself. The foreign keys, together, constitute the primary key of the table.

B. One of the major strengths of the relational data model is that it supports ad-hoc operations - the ability to access information in the database in a simple way, without having to write a special program to do so. These operations are of two general kinds:

1. Queries - access information without altering it
2. Updates - add, delete, or modify information

C. We now have to consider how to actually perform queries and updates. To do this, we will make use of a query language.
1. Over the years, a number of different query languages have been developed for use with relational databases. They fall into two broad categories:

   a) Formal languages that use mathematical notation, and are most useful for theoretical study.

   b) Commercial languages used in actual systems.

2. Two query languages have emerged as having wide usefulness

   a) The relational algebra is a formal query language.

   b) Structured Query Language (SQL) is a commercial query language that has been standardized by ANSI, which is supported by many relational DBMS products, and which is used by Java’s JDBC (Java Database Connectivity) facility.

3. We will study both relational algebra and SQL as languages for querying a database, and will also look at SQL facilities for updating a database. We will study relational algebra primarily because it provides a simple and clean way to get a handle on what we are doing; we will study SQL because of its usefulness.

D. For our examples, we will utilize a simple database for a very small library, realizing the following ER diagram:

Note that the primary keys of the entity sets are underlined. (assume the library is so small that it has at most one copy of any book - hence callNo suffices as the primary key for Book.) What is the primary key of CheckedOut?

ASK
The foreign keys of the two tables being related - hence the primary key of CheckedOut is borrowerID, callNo. We cannot have two rows in this table relating the same borrower to the same book. (In fact, because any given book can only be checked out to one borrower at a time, we could use just callNo as the primary key in this case.)

**DISTRIBUTE QUERY HANDOUT**

**DEMO:**

```bash
mysql -h jonah.cs.gordon.edu -p
use LIBRARY;
show tables;
```

**II. Querying a Database**

A. Some simple examples

1. One possible query is one that asks for a particular row of some table.

   Example: "What is the book whose call number is RZ12.905?"

   a) Relational algebra formulation:

   ```
   σ  Book
     callNo = 'RZ12.905'
   ```

   the select operator (written as the Greek letter sigma) specifies selecting out the row(s) meeting some criterion.

   b) SQL formulation.

   ```sql
   select *
   from Book
   where callNo = 'RZ12.905';
   ```

   The keyword select is used for all queries, * specifies all columns of the selected row(s), and where specifies the condition the row(s) must meet. Note also that all SQL queries end with a semicolon, that comparison for equality uses = (not == as in Java) and that strings are enclosed in single quotes. It is good form to put each clause on a separate line, indented with respect to the first line - not required by the language, but facilitates reading.
c) Result (in either case) (Demo)

21873 Fire Hydrants I Have Known Dog RZ12.905

2. It is also possible to formulate a similar query that produces several rows as its result.

Example: "What books are written by Dog?"

a) Relational algebra formulation

\[ \sigma \text{ Book} \]
\[ \text{author} = 'Dog' \]

b) SQL formulation:

```sql
select *
from Book
where author = 'Dog';
```

c) Result (in either case) (Demo):

<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Author</th>
<th>ISBN</th>
</tr>
</thead>
<tbody>
<tr>
<td>21873</td>
<td>Fire-hydrants I have known</td>
<td>Dog</td>
<td>RZ12.905</td>
</tr>
<tr>
<td>34938</td>
<td>21 ways to cook a cat</td>
<td>Dog</td>
<td>LM925.04</td>
</tr>
</tbody>
</table>

3. The above queries produce all columns from a one or more rows from the table. Sometimes, we want one or more columns from all rows in a table.

Example: "List the names of all borrowers"

a) Relational algebra formulation:

\[ \Pi \text{ Borrower} \]
\[ \text{lastName, firstName} \]

the project operator (Greek letter pi) specifies projecting out a particular column or columns. (Note that, in general, the result of a relational algebra query is a set of values - three in this case.)

b) SQL formulation:

```sql
select lastName, firstName
from Borrower;
```
the keyword select is still used, but we explicitly list the columns we want instead of using *, and we don't have a where clause.

c) Result (in either case) (Demo):

<table>
<thead>
<tr>
<th>Aardvark</th>
<th>Anthony</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Charlene</td>
</tr>
<tr>
<td>Dog</td>
<td>Donna</td>
</tr>
<tr>
<td>Fox</td>
<td>Frederick</td>
</tr>
<tr>
<td>Gopher</td>
<td>Gertrude</td>
</tr>
<tr>
<td>Zebra</td>
<td>Zelda</td>
</tr>
</tbody>
</table>

4. The operations of selection and projection can be combined in a single query.

Example: "What is the title of the book whose call number is QA76.093?"

a) Relational algebra formulation:

\[
\Pi \sigma_{\text{callNo} = 'QA76.093'} \text{Book} \\
\text{title}
\]

(Note that we select first, then project. Doing the reverse would not work! Because project and select are prefix operators, the right one is done first.)

b) SQL formulation

```sql
select title 
from Book 
where callNo = 'QA76.093';
```

Note that both an explicit column list and a where clause are needed.

c) Result (in either case) (Demo)

| Wenham Zoo Guide |

5. The full power of relational database system comes in when we need to combine information from two or more tables. We will look at a couple of examples now, but will discuss this extensively in a bit.

Example: When is/are the book(s) Charlene Cat has checked out due?"
What tables do we need information from in order to answer this question?

ASK

Borrower - since the name only appears there - and CheckedOut - since the date due appears only there. The two tables are related by the common key borrowerID, which is the primary key of one and a foreign key in the other.

a) Relational Algebra formulation:

$$\Pi_{\text{dateDue} \sigma \text{lastName} = 'Cat' \text{ and } \text{firstName} = 'Charlene'} (\text{Borrower} \times \text{CheckedOut})$$

The \( \times \) operator is called "natural join". It specifies that a new table is to be constructed by taking the Cartesian product of the two tables and then keeping only those rows which agree on their common attribute (called the join attribute): borrowerID. The fact that borrowerID is the join attribute is implicit in the fact that it has the same name in both tables.

b) SQL formulation:

```sql
select dateDue
from Borrower
natural join CheckedOut
where lastName = 'Cat' and firstName = 'Charlene';
```

(Note that SQL uses the word and, not && as in Java.)

c) Result (in either case) (Demo)

```
2001-2-15
```

6. Joins can be used to combine information from more than two tables.

Example (admittedly a bit contrived):

"What borrowers have books checked out whose author has the same last name as they do?"
What tables are required to answer this?

ASK

All three!

a) Relational algebra formulation:

\[ \Pi_{\text{lastName}, \text{firstName}} \sigma_{\text{lastName} = \text{author}} \text{Borrower} \bowtie \text{CheckedOut} \bowtie \text{Book} \]

b) SQL formulation:

```sql
select lastName, firstName
from Borrower
natural join CheckedOut
natural join Book
where lastName = author;
```

c) Result - actually relational algebra defines projection more strictly than SQL, so the result is a bit different in the two cases

(1) Relational algebra

Dog Donna

(2) SQL (Demo)

Dog Donna
Dog Donna

(3) The difference is due to the following: Relational database tables are sets - and a set cannot have two identical elements. The relational algebra operators, being rooted in the algebra of sets, are consistent with this - in particular, part of the definition of \( \Pi \) (projection) is that if any duplicate rows result from the projection, then all but one is eliminated from the result. SQL, by default, does not do this because it is computationally expensive. However, there is variant of the SQL select command that does produce the same result:

```sql
select distinct lastName, firstName
from Borrower
natural join CheckedOut
natural join Book
where lastName = author;
```
B. Review of basic operations

1. Selection - chooses only those rows meeting some criterion
   a) Relational algebra: \(\sigma\)
   b) SQL - where clause in select statement
   c) In effect, this operation squeezes a table vertically

2. Projection - chooses only certain columns.
   a) Relational algebra: \(\Pi\)
   b) SQL - explicit column list (* if we don't want to project)
   c) In effect, this operation squeezes a table horizontally. The relational algebra definition may also result in squeezing out some rows if duplicate rows result from eliminating the column(s) in which they differ. To get this in SQL, we must explicitly specify "distinct".

3. Cartesian product (not used in any of the above examples - we will see an example shortly)
   a) Relational algebra: \(X\)
   b) SQL - listing multiple tables in from clause

4. Natural join
   a) Relational algebra: \(|X|\)
   b) SQL - connecting tables by natural join in from clause
   c) In effect, this operation does a cartesian join, and then selects only those rows in which columns with the same name from different tables have the same value.

5. Another way to look at the correspondence between relational algebra and SQL is this:
   
   \[
   \text{select column names or } * \quad \Pi \quad \text{from } \text{tables} \quad X \quad \text{or } \text{|X|} \quad \text{where } \text{conditions} \quad \sigma
   \]
C. For our next examples, we will need to use a more complex database - the same one that you will use in lab.

1. **HANDOUT** of E-R diagram

2. Note that we have course id's in several tables, but we store them as two or three separate attributes (department, course_number, and possibly section.) The reason for storing these values is separately is that the relational model requires attributes to be atomic, but we sometimes need to use the different components individually (e.g. we use just department and course_number to link a CURRENT_TERM_COURSE to its catalog information stored in COURSE_OFFERED - the section does not appear in the latter.

3. Discuss primary keys

4. **HANDOUT** of sample database tables - note correspondence to E-R diagram, including the fact that one relationship (OFFERING_OF) does not map to a table because it is implicit in presence of foreign key of COURSE_OFFERED in CURRENT_TERM_COURSE.

5. The features we will look at can be used with either relational algebra or SQL, though we will focus our examples on SQL.

D. Additional Query Features

1. Qualified names.

   a) Sometimes, if the same column name occurs in two different tables, it may be necessary to specify from which table you mean for a column to come.

   Example: Suppose we want to print a class schedule for a student with a given ID (say 1111111), giving the course id, days, time and room. We need to join the ENROLLED_IN table with the CURRENT_TERM_COURSES table to get the information we need. However, the following query will not work:

   (switch to CS211 database)

   ```sql
   select department, course_number, section, days, start_time, room
   from ENROLLED_IN natural join CURRENT_TERM_COURSE
   where id = '1111111';
   ```
Why?

ASK

DEMO - note problem with ambiguity of department, course_number, and section since they appear in both tables

b) To formulate this query acceptably, we must use:

```
select ENROLLED_IN.department, 
    ENROLLED_IN.course_number,ENROLLED_IN.section, 
    days, start_time, room 
from ENROLLED_IN natural join CURRENT_TERM_COURSE 
where id = '1111111';
```

(Note that we only need to qualify the otherwise ambiguous columns)

2. Renaming of tables

a) As the last example illustrated, needing to type the full name of a table over and over when qualifying a name can be unpleasant. To avoid this, it is possible to rename a table within a query. This is illustrated by the following variant of the above:

```
select E.department, E.course_number, E.section, 
    days, start_time, room 
from ENROLLED_IN E natural join CURRENT_TERM_COURSE 
where id = '1111111';
```

b) In the above example, renaming the table was a convenience. There are times when it becomes an absolute necessity.

Example: Suppose we had a database that represents the following structure:
The names on the two lines connecting the relationship to the entity denote roles, and are similar in meaning to the use of roles when labelling an association in a UML class diagram.

This might correspond to the following tables:

- Employee(id, salary)
- Supervises(supervisor, supervisee)

(where supervisor and supervisee are employee ids)

Now suppose we want to know what employees make more than their supervisor. This would require joining the Employee table with itself (since we need two different salaries) and can be accomplished like this:

```sql
select E.id
from Employee E, Supervises, Employee B
where E.id = Supervises.supervisee and
    B.id = Supervises.supervisor and
    E.salary > B.salary;
```

Even if we were willing to type out the full table name every time, we couldn't do the query this way because we are using the same table (with the same name) twice, in two different ways.

3. Full Cartesian joins.

   a) The above example also illustrates a second point - although the natural join is often what we need, there are times when we want to join tables in some way other than based on equality of values in columns having the same name.
In the above, we needed to join the first usage of Employee with Supervises based on id = supervisee, while the second usage was joined based on id = supervisor. In neither case are the column names the same.

b) The same issue can arise, even when we don't have to use the same table twice.

Example: in the PROFESSOR table the department attribute is the department to which the professor belongs, while in the TEACHES relationship and in the various course tables it is the department that offers the course. Sometimes, a professor teaches a course in a department other than his/her own. Since a natural join between PROFESSOR and TEACHES would require that the department attribute in both have the same value (since the column has the same name), a natural join involving these two tables would lose data we might not want to lose.

Suppose want the names of all professors who teach a course in a department other than their own. The query must be formulated using a full cartesian join, with join conditions explicitly specified:

1) Relational algebra

\[
\Pi_{P.professor\_name} \sigma_{P.professor\_name = T.professor\_name \land P.department \neq T.department} (\rho \text{PROFESSOR} \times \rho \text{TEACHES})
\]

2) SQL:

```sql
select P.professor_name
from PROFESSOR P, TEACHES T
where P.professor_name = T.professor_name
and P.department <> T.department;
```

3) Result - DEMO. (The large number of listings is because, at the time this data set was prepared, all core courses had numbers with 'CR' as the department code!)

4) DEMO same, but using natural join between P and T (and hence no explicit comparison of names)
4. Union

Recalling that relations are sets, it is natural to ask whether ordinary set operations are applicable. The answer, of course, is yes. One such operation is set union.

Example: Suppose we wanted a listing of all the courses a given student (say the student with id 1111111) either has taken or is taking. The former are recorded in the COURSE_TAKEN table; the latter in the ENROLLED_IN table.

The schemes of the two tables are not identical, of course, because the former records a term, a number of credits, and a grade - which the latter does not need. (Credits is recorded because the number of credits for a course can be changed, but you still get the number of credits in effect when you took it!) Likewise, the latter records a section code, which the former does not need.

However, appropriate projections can make the schemes the same - a requirement for set union to be meaningful.

a) This yields the following relational algebra formulation:

\[
\sigma ( (\Pi_{\text{id} = '1111111'} (\Pi_{\text{id}} (\Pi_{\text{department}} ENROLLED_IN)) \cup (\Pi_{\text{id}} (\Pi_{\text{department}} COURSES_TAKEN)))
\]

b) In mysql, this becomes the following (note that the "where" has to appear inside each select). The parentheses are not required, but do enhance readability.

```sql
(SELECT id, department, course_number
 FROM ENROLLED_IN
 WHERE id = '1111111')
UNION
(SELECT id, department, course_number
 FROM COURSE_TAKEN
 WHERE id = '1111111');
```
5. Difference

Another set operation that is useful is set difference. For example, suppose we had a table listing the requirements for a given major. Then we could find out what courses a given student still needs to take by taking the set difference between the requirements table and the entries for him/her in the COURSES_TAKEN table.

Alas, set difference is not supported in mysql, though many sql implementations do support it.

6. Summarization

One powerful feature of relational databases is the ability to generate summary information easily.

Example: Suppose we wanted to know how many credits a given student (say id '111111') is taken. This information is available by summing up the credits attribute of the join between the ENROLLED_IN and COURSE_OFFERED tables for that student. A SQL query like the following will do this:

```
select sum(credits)
  from ENROLLED_IN natural join COURSE_OFFERED
  where id = '111111';
```

DEMO

7. Grouping

A natural extension of the above is to ask for id and total credits taken for all students.

a) The following query does not work:

```
select id, sum(credits)
  from ENROLLED_IN natural join COURSE_OFFERED;
```

Why?

ASK

We want to sum the credits individually for each student - we don't want the sum for everybody!
b) The following query will work:

```sql
select id, sum(credits)
    from ENROLLED_IN natural join COURSE_OFFERED
    group by id;
DEM
```

c) Note: our first attempt is not only wrong, it is actually rejected by SQL.

DEM

d) We could do a more complicated version of the above, in which we print out the student's name instead of the id:

```sql
select last_name, first_name, sum(credits)
    from STUDENT natural join ENROLLED_IN
    natural join COURSE_OFFERED
    group by last_name, first_name;
DEM
```

8. Outer join

a) Did you notice that there's actually one row that did not show up in the above?

ASK

The row for EMILY ELEPHANT did not show up - since she is not enrolled in any courses, the natural join between STUDENT and ENROLLED_IN produces an empty set.

b) Perhaps what we want in this case is a table with a 0 for total credits for EMILY ELEPHANT. We can get this by using an outer join:

```sql
select last_name, first_name, sum(credits)
    from STUDENT natural left outer join ENROLLED_IN
    natural left outer join COURSE_OFFERED
    group by last_name, first_name;
DEM
```
9. Having

One thing we might want to notice in the above query is what students are taking more than 16 credits. While we could do this by manually scanning the results, its much better to let the DBMS do this for us.

At first glance, we might think the where clause can be used. Unfortunately, this doesn't work because the where clause selects rows before the groups are formed - whereas we are interested in a particular result of the summary after the groups are formed. The solution is a having clause:

```sql
select last_name, first_name, sum(credits)
from STUDENT natural left outer join ENROLLED_IN
natural left outer join COURSE_OFFERED
group by last_name, first_name
having sum(credits) > 16;
```

DEMO

10. Order by

a) We have said that the tables in a relational database are sets. One property of a set is that it has not inherent order. Thus, the order of the rows in response to a query is arbitrary. (In fact, most DBMS's produce the results in the order the rows were inserted into the table - but that's not guaranteed.)

b) If we want to force the DBMS to sort the rows into some order before presenting them to us, we can use an order by clause.

DEMO

```sql
select * from COURSE_OFFERED;

select * from COURSE_OFFERED
order by ctitle;
```

c) This facility should only be used when we need it, because sorting is a time-consuming operation.
E. Summary

**HANDOUT** - SQL Syntax

Go over section on SELECT statements

**III. Updating a Database**

A. The INSERT statement - handout

B. The DELETE statement - handout

C. The UPDATE statement - handout

D. In addition to these statements, SQL also has statements for maintaining the structure of the database. We will not cover them - you can find them in the online documentation for mysql.

DEMO: mysql documentation linked from course web page