CPS352 Lecture - SQL

last revised January 17, 2017

Objectives:

1. To provide background on the SQL language
2. To review/expand upon basic SQL DML operations (select, insert, update, delete, commit, rollback), with added coverage of subqueries, joins, recursive queries
3. To introduce selected SQL DDL operations (create table, view)

Materials:

1. Demonstration databases: library, sample, genesis, security
2. Ability to connect to database / project operations; file with queries
3. Projectable of architecture of db2 system at Gordon
4. DB2 Manual on Blackboard site
5. Birchall Book accessible from Blackboard site
6. Projectable of example syntax diagram from SQL Reference (connect)
7. SQL Syntax handout from CPS221 for review
8. Handout of SQL versions of queries used in RA lecture
9. Handout showing commands used to create library example database, showing entity and referential integrity constraints.
10. Handout of these statements modified to incorporate additional domain integrity constraints
11. Projectable of SQL data types - book §3.2.1, §4.5.1
12. Security database from CPS221 - creation code to project and executable version

I. Introduction

A. As you know, although there are quite a number of commercially-available relational query language, one language has come to be especially important: Structured Query Language. (SQL-pronounced Seequel or S Q L)

1. You worked with some basic features of SQL in CPS221.

2. This lecture will serve to provide a bit of background, and also to introduce many key features of SQL that you have not seen yet.

B. SQL was originally developed for use with IBM's System R - the earliest research implementation of the relational model. In its original form, it was known as SEQUEL (Structured English Query Language). Since then, it has been adopted by many commercial vendors, and has become an ANSI standard - the only query language to be thus standardized - and has undergone a number of revisions - each of which is considerably more complex than its predecessor.

1. The first ANSI standard was 1986. This was revised in 1989 to yield what is now known as SQL 89 - which many commercial products implemented. (The SQL 89 standard is 120 pp.,)

2. The first major revision to the standard standard was SQL 92 (also known as SQL 2). (This standard is 579 pp!) The standard defines three levels of conformance, plus a transitional level between entry and intermediate:

   a) Entry level

   b) Intermediate level

   c) Full level

3. The next major revision was SQL:1999 (also known as SQL 3).

   a) This standard is in multiple parts, totaling well over 1000 pages.

   b) It incorporates many extensions to more easily support multimedia and object-orientation, so we will discuss it more fully later in the course.

   c) There are those who argue that the extensions have resulted in a model that is no longer truly relational - for example, there is a paper I found while researching this topic on the web entitled “Great News, The Relational Data Model is Dead!”, which is basically about SQL 3.

4. The standard has since been revised in 2003, 2006, 2008, and 2011. It currently consists of 14 parts.

5. The definitive versions of the standards must be purchased from ANSI; however, freely available draft versions of some exist. For example, when I initially prepared this lecture, I downloaded a draft of SQL2008 for which one part (alone) has over 1000 pages!

6. As it turns out, database software from different vendors typically supports slightly different *dialects* of SQL.

7. Actually, although SQL is based on the relational data model, vendors of database systems based on other models have included a facility for accessing their database using SQL.

C. SQL is both a data definition language (DDL) and a data manipulation language (DML). We will classify statements this way, but the language itself does not draw a distinction between the two types of statement in terms of how they are used. DDL and DML statements can be freely intermixed with one another.

D. SQL can be used in three ways:

1. Interactively.

2. Embedded in an application program, to allow the program to access and or modify the database. Actually, this latter form has two variants:

   a) SQL statements may be embedded in the application program, and processed by a suitably modified compiler or by a pre compiler. (This is called static SQL)

   b) SQL statements may be generated as character strings and processed at run time (This is called dynamic SQL). (You have had some exposure to this using JDBC)

3. Modules consisting of SQL statements can be stored with the tables in the database, to be invoked under various circumstances. (The OO idea of combining state and behavior!)

4. We will focus on interactive SQL for now - static SQL will come later in the course (when you do your project)

   a) Static SQL can use any of the capabilities of interactive SQL, plus there are some statements that are only needed in embedded SQL

   b) We will not deal with dynamic SQL at all in this course - you had some exposure to it by way of JDBC in CS221.

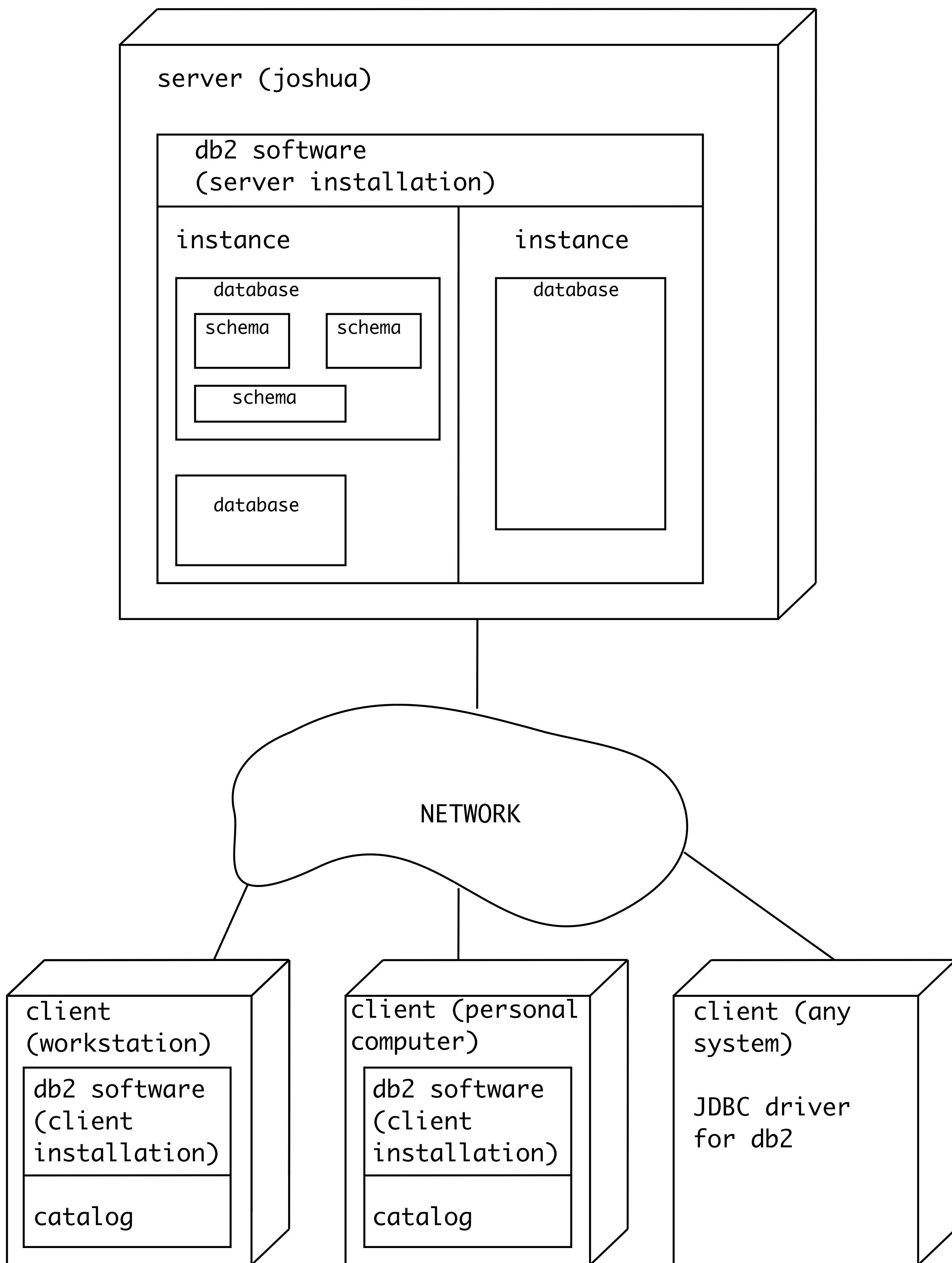
   c) We will not deal with SQL modules in this course - that's an advanced topic beyond the scope of CS352.

E. The version of SQL we will be studying is that implemented as part of IBM's DB2 product.

1. This version of SQL implements most (but not all) of the SQL 92 standard, plus many parts of the SQL 99 standard.

2. It is possible to install DB2 on a single computer, but more often it is installed on multiple computers, with the databases residing on a server computer and being accessed from client computers.

3. The following diagram shows the architecture of the way we have installed DB2 here at Gordon. The diagram uses few terms that are used in the IBM documentation in a way that is somewhat differently from the way we used them in our theoretical discussion - specifically, the terms "instance", "database" and "schema".



PROJECT

4. There are three types of software installations we are using:

    (1) The server version of the db2 software is installed on our departmental server machine - joshua. This is also where all the database data physically resides.

    (2) Client versions of the db2 software are installed on our 12 workstations, and can also be installed on your own computers. Client computers can access a database on the server interactively, or can run application programs that access a database on the server. They do <u>not</u> contain any of the actual database data. They do, however, contain a catalog that records information about databases they can connect to (in this case, databases on joshua; but a client's catalog could actually contain references to databases on many different servers.)

    (3) It is also possible for any system that has the db2 JDBC driver installed to access the database via JDBC. The JDBC driver is written in Java, and hence runs on any system that runs Java - it does not have to be running db2. For a JDBC connection, full information about the server must be provided when the connection is made.

b) The software on the server supports any number of <u>instances.</u> Each instance is a totally separate entity, and has no connection to any other instance besides residing on the same system. For this course, we will be using an instance called `db2inst1`. (This is the default name when doing a server installation - I could have called it "aardvark" if I had wanted to!)

c) Each instance contains any number of individual <u>databases.</u> For example, the db2inst1 instance currently contains 8 databases, including the database I used for examples in the first lecture, the sample database you are using for homework, a database you will use for the design project, and an example

library database I will be using later in this lecture. Each database has its own name in the catalog of the instance - e.g. `intro, sample, design, library`.

d) As we will note in conjunction with the homework, each database contains any number of individual schema.

(1) The major objects in the database (e.g. tables, views) have a two part name of the form SCHEMA.NAME. By default, the schema name is the username of the person who created the object.

EXAMPLE:

If I connect to a database using the username "bjork", and then create a table called "foo", the full name of the table will be bjork.foo.

If, however, I connect to the database using the username "aardvark", and create a table called "foo", the full name of the table will be aardvark.foo.

(2) Two objects that belong to different schema may have the same name.

EXAMPLE: I could create two different tables called "foo" under two different schema names, as described above.

(3) When you reference objects in the database, you can always use their full, two part name. If you only specify an object name, but not a schema name, then a default schema name.

(a) Normally, the default schema name is the name by which you connected to the database.

(b) However, you can use the SET SCHEMA command of SQL to specify a different default schema, as you have been doing for the homework.

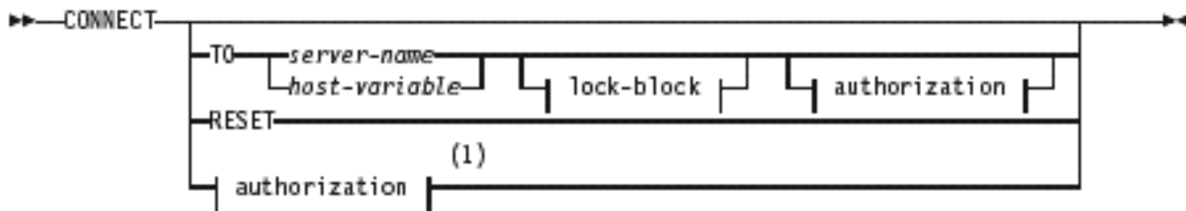
(4) DB2 comes with an extensive set of documentation (10’s of 1000’s of pages!).

SHOW MANUAL ON BLACKBOARD SITE

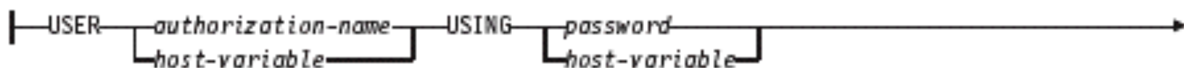
(a)One thing you will find in the Volume 2 of the SQL reference manual is complete syntax diagrams for each SQL statement.

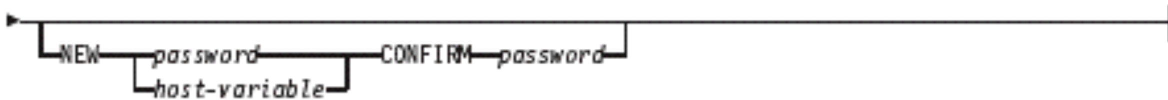
EXAMPLE: The syntax diagram for the connect statement, used to initially establish a connection to a database:

PROJECT



authorization:





lock-block:

e) The notation used in syntax diagrams is discussed on pages vii-x of the manual - looking this over before working through the diagrams in the manual is a good idea!

   (Note: These are the page numbers appearing at the foot of the page. When reading the manual using Adobe Acrobat Reader, it will give its notion of physical page number, based on numbering the very first physical page in the document “1” (i.e. not recognizing the separate numbering for the introductory material.)

f) The select statement alone is the subject of a full chapter in the manual (chapter6 of volume 1 - queries) which is 78 pages long! This is because its syntax is broken into portions (subselect, fullselect, and select statement) - some of which can appear in other statements as well!

F. Also accessible on the Blackboard site is a book by Graeme Birchall called DB2 UDB Cookbook - which is a SQL book specifically based on version 9.7 of the DB2 implementation of SQL. You will find this very helpful as you do homework throughout the course.

   SHOW

G. In addition to DB2, we also have an open-source product known as mysql available. The latter is, in some respects, nicer to use than DB2: it has a nicer interactive interface, and its syntax includes an explicit natural join operator. However, because it does not support transactions, we will not be using it extensively in this course.

II. **Review of Basic DML Operations for Querying the Database**

A. Probably the most fundamental DML concept in SQL is that of a query.

1. The most frequently used SQL statement is the SELECT statement: a statement intended to get information out of the database. A SELECT statement is basically a query, possibly with some additional components.

2. Queries can also be embedded in certain other statements, as we shall see later.

3. You were introduced to queries in CPS221, and will get lots of practice with various sorts of queries in the homework, and both the text and Birchall books cover them extensively; therefore, I will not spend extensive time on them here.

   HANDOUT - SQL Syntax handout from CPS221 - Review

   HANDOUT: SQL versions of RA Queries discussed in RA lecture (go over quickly with class)

4. However, I do want to spend some time on three feature that are powerful, but potentially tricky.

B. Subqueries

1. One important capability of SQL is the possibility of embedding a query as a *subquery* of another query.

2. Consider the following: “List the names and salaries of all employees earning more than the average salary for all employees”

   a) One way to do this would be to issue a select to get the average, and then issue a second select to get the individuals.

b) SQL allows this to be done in one query by using a subquery

```
select last_name, first_name, salary
     from employee
     where salary >
          (select avg(salary) from employee);
```

Note that this could also be formulated in relational algebra, though it would be a bit messy!

c) Projectable version - connect to library

3. It is also possible to use a subquery whose result is a set, rather than a single value.

   EXAMPLE: “List the names of all borrowers whose last name is the same as that of the author of a book”

```
select last_name
       from borrower
       where last_name in (select author from book);
```

   a) The subquery (select author from book) forms a set - a list of all the authors.

   b) The “in” predicate occurring in the where clause then checks to see if the borrower’s last name is in this set.

   c) This one would be hard to formulate in relational algebra. (You could do so using a theta join - but that’s really a rather inefficient way to actually go at computing it.)

   d) Projectable version - connect to library

4. We can also use quantified predicates with sets created by a subquery. “Print the name and salary of any employee earning more than all the employees in department E11” (Using employee in sample)

```
select firstnme, midinit, lastname, salary
       from employee
       where salary > all
            (select salary from employee
                 where workdept = 'E11');
```

Projectable version - connect to sample/set schema db2inst1

5. “Print the name and salary of any employee earning more than some employee in department A00”

```
select firstnme, midinit, lastname, salary
       from employee
       where salary > any
            (select salary from employee
                 where workdept = 'A00');
```

Projectable version

C. Various kinds of Joins

1. In our study of relational algebra, we learned about a variety of join operations:

a) The full cartesian join - represented by X - e.g.

A X B

means each row of A is paired with each row of B

How big will the resulting table be?

ASK

This results in a table whose number of rows is (number of rows in A) * (number of rows in B).

b) The natural join - represented by |X| - e.g.

A |X| B

means each row of A is paired with those rows of B which have the same values for all the columns they have in common - e.g. if both A and B have a `call_number` column, then rows of A are paired only with rows of B having the same value in the `call_number` column.

How big will the resulting table be?

ASK

We can't know for sure. It could be empty (if no row in A matches a row in B). It could be the same size as the cartesian join (if there are no columns with the same names, or all the rows have the exact same value in the relevant column).

What if the join column(s) include the primary key for one of the tables? (Say A)

ASK

In this case, the result will be no bigger than number of rows in the other table (e.g. B). The reason for this is that each row in B will join with at most one row in A.

c) The theta join - represented by X_{Θ} - which is like the cartesian join, except that some test is applied and only joined rows passing that test are kept.

d) The outer join - which comes in three varieties (left, right, and full), written as ⊐|X|, |X|⊏, and ⊐|X|⊏. - e.g.

A ⊐|X| B

means that we do a natural join between A and B, but then any row of A that does not join with some row of B is paired with a "manufactured" row of B (all attributes null) so no rows of A are lost.

2. SQL has forms that are equivalent to each of these

a) The cartesian join is specified by simply listing the names of the tables to be joined, separated by commas - e.g.

```
SELECT something FROM A, B
```

is equivalent to A X B

and

```
SELECT something FROM A, B, C, D
```

is equivalent to A X B X C X D

b) The natural join is specified by using the phrase natural join - e.g.

```
SELECT something FROM A natural join B
```

is similar to A |X| B

(1) However, it differs in one important respect. Any idea?

ASK

|X| is defined in relational algebra as keeping only one copy of common columns (i.e. there is an implicit project). In some implementations of SQL, one might have to specify this explicitly in the select list

(2) Unfortunately, not all dialects of SQL support natural join at all. In particular, DB2 does not. One achieves the effect of natural join by either using the WHERE clause or the next form of join we will consider (which is equivalent to the theta join)

(a) Example: Consider the tables

```
book(call_number, title, author)
checked_out(borrower_id, call_number, date_due)
```

The relational algebra book |X| checked_out could be expressed in a SQL dialect that supports natural join by:

```
SELECT book.call_number, title,
          borrower_id, date_due
     FROM book natural join checked_out;
```

But in a dialect like DB2 that does not support natural join, one could use

```
SELECT book.call_number, title,
          borrower_id, date_due
     FROM book, checked_out
     WHERE book.call_number =
          checked_out.call_number;
```

Projectable version - connect to library

(3) Note the need for the use of qualified names where both tables have a column name in common. Typing is often reduced by using correlation names - e.g.

```
SELECT b.call_number, title,
            borrower_id, date_due
       FROM book b, checked_out c
            ON b.call_number = c.call_number;
```

c) The theta join is expressed by using the phrase join .. on - e.g.

```
SELECT author
     FROM book JOIN borrower ON author = name
```

is equivalent to the relational algebra

book X borrower
Θ author = name

(1) One important use of join .. on is to produce the equivalent of a natural join in implementations that lack the natural join keyword:

Example: book |X| checked could also be written

```
SELECT b.call_number, title,
            borrower_id, date_due
       FROM book b JOIN checked_out c
            ON b.call_number = c.call_number
```

(Important: For natural join, this is preferable to using where, since on is part of the join operation, selection is done after the join).

(2) It is important to remember, though, that join on need not involve a natural join - the join condition can be any join condition.

d) The outer join is specified by using the phrase left outer, right outer, or full outer before the word join.

(1) Example: include the borrower id's of all the borrowers, even if they don't have a book out:

```
SELECT b.call_number, title,
            borrower_id, date_due
       FROM book b LEFT OUTER JOIN checked_out c
            ON b.call_number = c.call_number
```

(2) The condition in the join is often equivalent to a natural join - though it need not be In fact, SQL implementations that do support the phrase natural join typically also allow outer joins to be combined with it (yielding a meaning similar to relational algebra operations) - e.g.

A ⟕ B

would be written ... A natural left outer join B

(3) By way of contrast, joins that are not outer joins are sometimes called inner joins, and the word inner can be explicitly used, though it is not required

e.g. `A join B on` ... could be written `A inner join B on` ...

D. Recursive Queries

1. An interesting kind of problem arises when it is necessary to perform a recursive query in SQL.

a) For example, consider a table listing a person's name and the name of his/her parent - e.g.

```
person(name, parent)
```

For simplicity we will use just first names, and each row will record just one parent (so a given person might appear in the table twice - one row for each parent).

(1) It is easy to print the names of all the children of a given parent:

```
select name
    from person
    where parent = ____
```

(2) A more complicated case arises if we want all of a given parent's grandchildren. In this case, we need to join the table with itself, and make use of the rename operation:

```
select p.name
   from person p join person q
     on p.parent = q.name
   where q.parent = ___
```

We could, of course, use a similar approach to list someone's great-grandchildren (in which case the table is joined with itself twice)

b) Now consider the following more challenging problem: print the names of all of a given person’s descendants, regardless of

how many generations are involved. This is, of course, a recursive query, based on the following recursive definition.

A is a descendant of B if B is A's parent, or A's parent is a descendant of B

2. At one point, it was impossible to formulate a query like this in SQL, because SQL does not provide for recursion. (Instead, one would need to embed a SQL query in a host language that did support recursion, or use some other query language like Datalog).

3. Current versions of SQL provide for handling case like this by using a union between a base table and a recursive table to create a temporary table using a with clause.

   For example, here is a SQL formulation of this query

```
with descendant(name) as
  (    select name
            from person
            where parent = _____
   union all
       select person.name
            from person, descendant
            where person.parent = descendant.name
  )
select *
       from descendant;
```

   Two things are going on here

   a) The with clause is used to create a <u>temporary table</u>. A temporary table is one that exists just for the duration of the query. Thus, in the with clause, we have to specify the table's name, its scheme, and an "as" clause that defines its content. (In this case, it is a table containing all the descendants of a given person, and has just a single column, though it could have multiple columns if needed). A "select *" is used to print this table out. [Note: use of join .. on in this case is forbidden in DB2 SQL - hence the use of where]

b) The content of the table is defined by using a "union all" between the base case and the recursive case of the definition

c) Demonstration (in database genesis under schema bjork)

```
select * from person;

select name
    from person
    where parent = 'adam';

select p.name
     from person p join person q
       on p.parent = q.name
     where q.parent = 'lamech';

with descendant(name) as
  (  select name
          from person
          where parent = 'adam'
   union all
     select person.name
          from person, descendant
          where person.parent = descendant.name
  )
select *
     from descendant;
```

(Note: "on" cannot be used because we're doing a union, not a join)

III. DML Statements for Modifying the Database

A. In CPS221, we looked at a few DML statements for modifying the database. We will review them here - also looking at an additional variant of insert we didn't discuss then plus two new statements: commit and rollback.

B. **Start up db2 with the +c option before doing any of the following (will explain why later)** Connect to library

C. INSERT - Three forms

1. Simplest form: insert explicit values into all columns

```
insert into borrower values ('98765','raccoon',
'ralph');
```

```
select * from borrower;
```

(Note that values are matched with columns positionally - first value goes with first column etc.)

2. It is possible to explicitly specify column names if one is unsure of actual order of columns

```
insert into borrower (first_name,last_name, borrower_id)
       values ('ursula', 'unicorn', '87654');
```

```
select * from borrower;
```

This form of insert can also be used if one does not have values for all the columns (and the column allows null values)

```
insert into book (call_number, title)
       values ('ABC', 'Author is unknown');
```

However, this fails because first_name was declared not null

```
insert into borrower (last_name, borrower_id)
       values ('xerus', '55555');
```

3. Finally, it is possible to embed a select into an insert to copy information.

   Example: suppose we want to make all of our employees eligible to be borrowers if they are not currently such

```
insert into borrower
  select right(ssn, 4), last_name, first_name
      from employee
      where not (last_name, first_name) in
        (select last_name, first_name from borrower);
```

   (Note: this is a pretty poor way to generate borrower id's - but it illustrates the point!)

D. UPDATE

1. General form: update table set (column = value) where condition

2. Example: Give all employees supervised by aardvark a 10% raise:

```
select * from employee;
```

```
update employee
       set salary = salary * 1.1
       where supervisor_ssn =
            (select ssn
                 from employee
                 where last_name = 'aardvark');
```

```
select * from employee;
```

E. DELETE

1. General form: delete from table where condition

2. Example:

   Delete the borrower entry for raccoon:

```
delete from borrower where last_name = 'raccoon';
```

```
select * from borrower;
```

3. What would happen if we did a delete without specifying a condition? (no where clause)

   ASK

   ```
   delete from employee;
   ```

   ```
   select * from employee;
   ```

F. COMMIT and ROLLBACK

1. It appears that - at this point - we have mangled the database. But we really haven't.

   Issue the following commands:

   ```
   rollback;
   ```

   ```
   select * from employee;
   select * from borrower;
   ```

2. As we noted at the start of the course, a very important concept in SQL is the notion of a TRANSACTION

   a) We will discuss this concept more later in the course. For now, we will think of a transaction as a unit of work such that either all the operations in it must succeed or all must fail.

   b) A transaction is normally terminated by entering one of the following statements:

   ```
   commit
   ```

   or

   ```
   rollback
   ```

   c) The former causes all changes to the database made during the transaction to become permanent; the latter undoes all of them.

   (Note: until a transaction is committed or rolled back, its effects will NOT be visible to other users of the database)

d) The system starts an initial transaction when the connection to the database is first made; and starts a new transaction when one is either committed or rolled back. If execution terminates for any reason (user specified or crash) with some transaction still in process, it is automatically rolled back.

DEMO:

Terminate session
Start new session (with +c flag)
Insert a new row into borrowers
Show it's there with select *
Terminate session without committing
Start a new session
Do select * to show it's no longer there

e) Suppose someone does two hours' worth of data, then terminates their session without committing. What happens?

ASK

Because this is generally not a good thing, most interactive command processors include an automatic commit mode, in which each command typed by the user is automatically committed - which is generally appropriate for interactive input.

In db2 SQL, this is the default mode of operation for the command line processor. For the last couple of demonstrations, I had to disable it. That's what the +c on the command line did:

`db2` - start DB2 with automatic commit enable
`db2 +c` - start DB2 with automatic commit disabled

IV. Basic DDL Statements

A. The set of DDL statements available in SQL is rich, and we will only introduce them briefly here. Three that are commonly used are:

`create ...` - to create a new object (e.g. a schema, table, or view - among others.)

`alter ...` - to modify an existing object (e.g. a table or view)

`drop ...` - to delete an existing object (e.g. a table or view)

(Note the difference between these and insert, update, and delete. These affect the <u>structure</u> of the tables of the database, while the statements we looked at earlier affect the <u>data</u> actually stored in a table - eg. contrast delete from *sometable* - which deletes all the rows from a table (leaving behind an empty table), and drop *sometable* - which drops the entire table (including its data but also its scheme)

B. For now, the only statement we will talk about is `create table.` (We will talk about another -`create view` - later.)

Go over create table statements used to create library sample database. (HANDOUT)

1. Note specification of primary key and foreign key constraints - we will discuss these and other constraints more shortl.y.

2. Each column in the table must have an appropriate data type. Sections 3.2.1 and 4.5.1 of the text list some of the more important data types available (not an exhaustive set).

   PROJECT

   Note the syntax for specifying dates, times, and datetimes when inserting or comparing values. Note, too, that the syntax used for input is not he same as the way SQL displays the values by default!

C. We will not discuss the syntax of alter or drop now. They are documented in the *SQL Reference Manual*.

V. **Integrity Constraints**

A. When designing a database, it is possible to specify various CONSTRAINTS that data in the database must satisfy. As we shall see, SQL provides a number of mechanisms that allow these constraints to be incorporated in the system's metadata so that they can be enforced by the DBMS. We will look at them in the following order, which is slightly different from that in the book:

1. Entity integrity constraints

2. Referential integrity constraints

3. Domain integrity constraints

4. Use of assertions and triggers to specify more general constraints.

B. Most of these are specified as part of the statement that creates a table, as we shall see.

C. Recall that an entity is a member of an entity SET, and therefore must be unique among all the elements of that set. This translates into the notion of a "key" that we discussed earlier. ENTITY INTEGRITY constraints are concerned with ensuring that each row in a table is distinct from all other rows in the same table in the necessary way(s).

1. The process of designing a relational database should result in each table having a primary key. This should be incorporated into the declaration for the table by means of a PRIMARY KEY constraint. This has the following characteristics:

    a) No two rows in the table will be allowed to have the same value(s) in the specified column(s).

b) If the primary key is a single column, the constraint can be specified as part of the declaration of the column.

Example: the various tables in the handout.

c) If the primary key is composite (consists of more than one column, then the constraint must be expressed as a table constraint.

Example: Suppose (as is more often the case) that books have both a call number and a copy number, and the two columns together constitute the primary key. Then we could define the book table as follows:

```
create table book (
     call_number call_number_type not null,
     copy_number smallint not null,
     title char(30) not null,
     author char(20),
     primary key (call_number, copy_number)
)
```

(Note the presence of the comma after the declaration of the author attribute, which signals that either a new column or a table constaint is coming. In the absence of the comma, the primary key constraint would be taken as applying to the author column - but would be syntactically invalid since columns are named explicitly.)

d) A given table can only have one primary key, of course.

2. A Related sort of constraint is the UNIQUE constraint.

   a) Like the PRIMARY KEY constraint, the UNIQUE constraint specifies that the same value or values cannot appear in two different rows in the table in a particular column or set of columns.

   b) Like the PRIMARY KEY constraint, the UNIQUE constraint can appear either as a column constraint or a table constraint - and in the latter case can specify any number of columns to be treated as a unit.

      Example: I chose to require that each borrower have a unique name, and likewise each employee. That may not be a good idea in general - but in a small library with only 4 borrowers it works! (Actually, I what I wanted to do is to illustrate the constraint!)

      Note: It is a peculiarity of this example that the primary key constraints all apply to only single columns, and so can be expressed as column constraints, while the unique constraints happen to have to be table constraints. This is not generally the case, of course.

   c) Unlike the PRIMARY KEY constraint, a table can have any number of UNIQUE constraints defined for it.

   d) The UNIQUE constraint is typically applied to any candidate key(s) not chosen as the primary key.

D. Thus far, the constraints we have described all pertain to data within a single table, and can be enforced by looking at that table alone. The next sort of constraint we need to consider pertains to REFERENTIAL INTEGRITY.

1. It is frequently the case that the logic of a system demands that an entry cannot logically occur in one table without a related entry occurring in another table:

   Example: A checked_out row for a book should not occur unless corresponding entries exist in the book table and the borrower table.

2. The requirement that a matching row occur in another table for each value occurring in a certain column (or set of c

3. Referential integrity constraints are expressed in SQL by using a FOREIGN KEY constraint, which is specified by the use of the reserved words FOREIGN KEY and/or REFERENCES.

   a) Again, can be either a column constraint or a table constraint.

      (1) If the former, a references clause is used as part of the column definition, and applies to that column only.

      (2) If the latter, FOREIGN KEY is separated off by commas from other column definitions, and is followed by a parenthesized list of the columns constrained.

   b) A foreign key constraint always has a references clause.

      (1) May be just the name of another table - in which case the value in the column(s) being constrained must occur in the primary key column(s) of the referenced table.

      (2) May explicitly list the column(s) in the referenced table where the value is to be found - necessary if the foreign key is not the primary key of the referenced table. (Columns can be explicitly listed even if they are the primary key - no harm done.)

(3) If the foreign key constraint is expressible as a column constraint, the word references is all that is needed.

c) Examples:

(1) Note references clauses in checked_out and reserve_book in the handout.

(2) Suppose we stored both a call number and a copy number for a book, and, as a result, it had a composite primary key - declared as in an example above. Then our declaration for checked_out would have to look like:

```
create table checked_out (
        borrower_id char(5) not null
             references borrower,
        call_number char(10) not null,
        copy_number smallint not null,
        date_due date,
        foreign key (call_number, copy_number)
             references book
)
```

(3) Suppose we want to require that no one can be added to the employee table unless already in the borrower table (perhaps because employees automatically have borrower privileges). Since last_name, first_name is NOT the primary key of borrower, this would have to be written as follows:

```
create table employee (
        ssn char(11) not null primary key,
        last_name char(20) not null,
        first_name char(20) not null,
        salary integer,
        supervisor_ssn char(11),
        foreign key (last_name, first_name)
          references borrower(last_name, first_name)
)
```

(The fact that the columns happen to have the same names in both tables is not essential. Moreover, the order of the columns is important. If the references clause were written (first_name, last_name), then we could only add Emily Elephant as an employee if Elephant Emily were a borrower!)

4. Support for referential integrity adds to new issues that need to be addressed.

   a) Note that changing that data in a REFERENCED table may cause an error because of a constraint in a REFERRING table.

      Example: checked_out has a foreign key constraint that references the borrower_id column of borrower. Thus, a change to borrower could cause a violation of a constraint for an existing row in checked_out

      (1) To cope with this possibility, the DBMS keeps a record of which columns are referenced by constraints in other tables, and checks updates of such a column (as well as deletions of an entire row) in the referenced table to be sure doing so does not cause a constraint violation in the referring table.

      (2) Sometimes, it makes sense to allow a change in the referenced table and handle the potential constraint violation by also changing the referring table.

         Example: Suppose we have a weak entity like fine. The definition for a fine table might look like this:

         ```
         create table fine (
                borrower_id char(10) references borrower,
                 ...
         ```

         As it stands, we cannot delete a borrower who has unpaid fines.

However, it might make sense to provide that if a borrower is deleted from the borrower table, then any fine(s) that the borrower owes are also automatically deleted.

In this case, the foreign key constraint in the fine table could have a CASCADE clause:

```
create table fine (
       borrower_id char(10) references borrower
           on delete cascade,
        ...
```

which specifies that if a row is deleted from the borrower table, then any rows referencing that row in the fine table are also to be deleted.

(3) The book discusses the possibility of a similar option for update, where the values in the referencing table are automatically updated to reflect changes to the referenced table. DB2 does not support this, however.

E. Domain Integrity Constraints constrain the values that can be stored in a particular column of a table. These are called domain integrity constraints because they constrain the domain of values from which a given attribute can be drawn.

1. The SQL standard incorporates a number of facilities for doing so that we will consider. In SQL, these are enforced whenever a new row is inserted into a table, or an existing row is updated. Some of these constraints are specified in the CREATE TABLE statement that creates a given table; others are specified by explicitly creating new domains.

2. The most commonly-used domain integrity constraint is the NOT NULL constraint, which prohibits storing a null value into a given column.

A `not null` constraint is specified as part of the declaration of a column.

Example: Handout - note columns that are and are not declared not null and discuss reasoning for each

Note: Any attribute that is part of a primary key or is declared to be a candidate key MUST be declared not null - otherwise the primary key / unique constraint is rejected by the DBMS.

3. As discussed in the book, the SQL standard also has a CREATE DOMAIN statement that allows the user to create a named domain which can then be used to declare columns in tables. While DB2 does not support this, it does have a similar facility called CREATE DISTINCT TYPE.

   a) Example:

      HANDOUT - library database creation modified to use named domains

   b) An advantage of using this mechanism is that data types are defined in terms of their SEMANTICS (meaning) - not just their physical representation.

      Example: Both a telephone number (without area code) and an old-style Gordon student id are 7-digit numbers - however, it wouldn't make sense to formulate a join like

      ```
      student join borrower on student.id =
      borrower.phone!
      ```

      (1) A distinct domain (type) is not compatible with a different domain - even if they have the same internal representation (e.g. INTEGER or CHAR(x))

(2) It is possible to explicitly cast a value from one type to another.

(a) The book discusses the SQL standard syntax for type casting

(b) DB2 has a cast syntax that is slightly different - but the idea is the same.

(c) In addition, whenever a distinct type is created, DB2 also automatically creates functions for converting between the distinct type and its source type.

Example: the statement

```
create distinct type borrower_id_type as char(5)
```

Creates the functions:

```
     char(-- a column of type borrower_id_type)
```

and

```
     borrower-id-type(-- any expression of type
                         char(5))
```

(3) A downside of this is additional cumbersomeness when using constants. (Bottom of handout.)

c) This type of constraint differs from all the others we will discuss in that it can be checked as a matter of the SYNTAX of a query - thus, if we are using static SQL, it can be checked at compile time.

4. The standard SQL data types, because they are based on physical representations for data, sometimes do not adequately restrict the values that can appear in a given column.

a) Examples:

   (1) A letter grade in a course could be stored in a field declared CHAR(2). In fact, though, only a very small number of one or two character strings are valid grades.

   (2) The first character in a call number has to be a letter.

b) To deal with situations like this, it is possible to specify a CHECK clause that tests a value about to be stored into a field. A check clause can be specified as part of a table definition. (In standard SQL it can also be specified in a domain declaration, which is the preferred place since it then applies to every column defined in terms of that domain - however, db2 does not support this.)

   (1) Example: (Assume student_id_type and course_id_type already defined)

```
create table enrolled_in (
       student_id student_id_type,
       course_id course_id_type,
       grade char(2) check
            (grade is null or
             grade in ('A', 'A-', 'B+', 'B', 'B-',
                 'C+', 'C', 'C-', 'D+', 'D', 'D-',
                 'F') )
)
```

   (2) Example: Definition of book in handout

   Note need to convert call_number from a user-defined domain to an ordinary string before applying left.

c) When a table definition includes a check constraint, any data being stored into the column(s) in question is checked to be sure it satisfies the constraint whenever an insert or update is done.

F. An additional features applies to most of the types of constraints we have discussed thus far: A constraint may be given a name by preceding the constraint specification with CONSTRAINT constraint-name

example - the last example above

```
... constraint employee_borrower
        foreign key (last_name, first_name)
        references borrower(last_name, first_name)
```

1. The constraint name is included by SQL in any error message reporting that the constraint has been violated. This is especially useful when using embedded SQL, since a program can now determine which constraint was violated when an operation fails.

2. If you don't specify a name for a constraint, SQL creates a default name.

G. To specify more complex integrity rules, it is possible to store general ASSERTIONS in the database - representing invariants that are to be enforced whenever data in the database is modified.

1. The book gives an example of an assertion.

2. DB2 does not support assertions, so we won't discuss this further.

H. In the period between SQL92 and SQL99, many SQL implementations added "triggers" - procedures to be executed whenever a specific event occurs. Although these were included in the SQL99 standard, many DBMS's use a syntax that is somewhat different from the standard because the implementation added the facility before the standard was written.

1. A trigger is a statement that the DBMS is to execute whenever a certain kind of modification to the database occurs.

2. The book noted some examples. Here's another one:

   Faculty at Gordon are allowed to request that books be ordered for the library. When the book comes in, the requesting faculty member might be notified. (This is not the way this is currently done at Gordon ...) This could be handled by a trigger on the book table, which adds a row to a table listing people to be notified - e.g.

```
create trigger book_arrived after insert on book
       -- appropriate action
            - syntax implementation-specific
```

3. Key features of a trigger definition:

   a) A trigger name - because a trigger is an object in the database that can be subsequently altered or dropped.

   b) One of the words "before" or "after" to specify whether the triggered action is done before or after the operation in question. (The "before" option allows a trigger to prevent an action from occurring if it fails.)

   c) A clause specifying the table whose modification will cause the triggered action to occur - one of "insert on xxx", "update on xxx", or "delete on xxx".

I. Some concepts we will consider later in the course are also closely related to the idea of data integrity.

   1. The concept of an ATOMIC TRANSACTION is really an integrity concept. A properly-coded transaction preserves the integrity of the data by ensuring that data that is consistent before the transaction starts does not become inconsistent as a result of the transaction.

2. Crash control measures that we will explore later help to ensure that data integrity is not damaged due to hardware or software failure.

3. Concurrency control measures that we will explore later help to to protect the integrity of the data against anomalies arising from simultaneous updates.

J. To summarize this section:

1. Data integrity is concerned with ensuring the ACCURACY of the data. In particular, the concern is with protecting the data from ACCIDENTAL inaccuracy, due to causes like:

   a) Data entry errors

   b) System crashes

   c) Anomalies due to concurrent and/or distributed processing

2. SQL incorporates a number of mechanisms to permit the DBMS to help preserve data integrity:

   a) Facilities to help preserve entity integrity:

      (1) Primary key constraints

      (2) Unique constraints

   b) Facilities to help preserve referential integrity: foreign key constraints

   c) Facilities to help preserve domain integrity:

      (1) Not null constraints

(2) User-defined domains

(3) Check constraints

d) Facilities to enforce more complex requirements:

(1) Assertions

(2) Triggers

e) I would argue that, as a matter of good design practice, EVERY DATABASE DESIGN should incorporate appropriate entity integrity, and referential integrity constraints - which also implies the need to use the not null domain integrity constraints. Other constraints are perhaps less critical in databases where a high degree of support for maintaining data integrity is not essential - but become important in "industrial strength" applications.

(Of course, the design project you are doing will make excellent use of facilities to ensure integrity!)

VI. SQL Security Facilities

A. Persons responsible for databases needed to be concerned with preserving both the integrity and security of the data.

1. As we have seen, data integrity is concerned with ensuring the ACCURACY of the data. In particular, the concern is with protecting the data from ACCIDENTAL inaccuracy.

2. Data security, on the other hand, is concerned with ensuring only AUTHORIZED ACCESS to the data. In particular, we don't want unauthorized persons to be able to read sensitive data, and we don't want malicious persons to be able to damage the data by unauthorized insertions, deletions, or updates. In particular, the concern is with protecting the data from DELIBERATE improper access or alteration..

B. System security is. of course, a HUGE topic - one that could easily be the subject of multiple courses at the undergraduate or graduate level, as well as being an ongoing focus of reserarch. Here, though, we want to limit our consideration to SQL facilities that allow controlling access to the content of the database.

1. We assume, as a starting point that the DBMS is running in an environment and under an operating system that provides a basic foundation for security, including:

   a) Appropriate physical security

   b) Trustworthy system administrators and users (human security)

   c) User authentication (minimally login passwords - but may be more than this)

   d) Protection for files (operating system security)

   e) Network security

2. The DBMS builds on this by allowing access to information in a single file (or collection of files) - the database - based on user identity - i.e. making it possible to limit a given user to accessing a subset of the entire database. (A finer-grained level of access control than the all-or-nothing sort of file access typically provided by an operating system.)

3. We assume that the rest of the system has ensured that the database can only be accessed through the DBMS, and that a person who accesses the database is who he/she claims to be. (These are far from trivial considerations - especially in a networked environment - but they are the focus of other courses.) For this course, we focus on SQL mechanisms that allow the DBA to control what a person may do with the data, once properly authenticated.

C. There are three key concepts involved in understanding the security mechanisms of SQL. (We will use the terminology used in IBM's DB2 documentation - other systems may use slightly different names for the same concepts.)

1. "An authorization ID is a character string that is obtained by the database manager when a connection is established between the database manager and ... a process." (SQL Reference Manual Vol I page 67 - note that "database manager" here means the DBMS - not a human manager!)

a) DB2 recognizes both individual authorization IDs and group authorization IDs. (A given individual may belong to one or more groups.) An authority may be granted to an individual, or to a group; an individual has an authority if granted to him/her or to a group to which he/she belongs.

b) On our systems, DB2 uses the login mechanism of Linux - thus an authorization ID is typically a Linux user name or group name, and the user authenticates himself/herself through knowing the associated Linux password.

(1) Note that this is done on the SERVER - and so uses the user names and passwords on the server.

(2) It is also possible to set up a database in which authentication is done on the CLIENT (using its password database). This means that the server trusts the client's claim that a certain user is who the client says he/she is - which avoids the need for the user to have an account on the server, but creates a new possibility for penetration of the system.

c) It is also possible, in principle, for a DBMS to have its own authentication mechanism with authorization id's that are specific to the DBMS. (For example, mySQL uses this approach). We will not pursue this further.

2. An object is a protectable entity, such as

   a) The database instance (db2 meaning of this term)

   b) A specific database

   c) A schema within a database

   d) A table or view within a schema

   e) A specific column within a table or view

   f) These objects form a hierarchy - i.e. to access a table, a user must first have rights to access the database in which it is contained.

3. An authority (also called privilege) is the right to perform a certain operation on an object. There are different kinds of authorities that apply to objects at different levels.

   (Note: The DB2 documentation calls these authorities, but the SQL standard uses the term privileges. I'll use the SQL standard term)

D. SQL privileges for various kinds of objects.

1. The privilege names are standard SQL names - not specific to DB2

2. We will not attempt to exhaustively cover all privileges - just selected important ones. In particular, we will only consider privileges related to the logical and view level of database access. There are a number of privileges that pertain to physical level operations (allocation of space, creation of indices etc) that we will not discuss now - but may refer to later.

3. Privileges that apply to the entire instance

   a) SYSADM - the System Administrator Privilege. This is the highest level of authority.

      (1) One place where this privilege is needed is to actually create or drop individual databases within an instance.

      (2) In DB2, SYSADM authority is granted to any user who is logged in to the Linux account that actually owns the database instance (e.g. db2inst1 in our case.) This is often different from the root account for administering Linux itself..

      (3) A person with SYSADM privilege implicitly has all privileges with regard to all objects except those specifically related to system security, for which SECADM is needed. (So in this regard it is much like root on Unix, but only for the database).

   b) SYSCTRL - the authority to manage system resources such as disk space. This privilege is like SYSADM in some ways, except that it does not include any authority to actually see or alter data - only space.

c) SYSMAINT - the authority to perform maintenance tasks such as backup. It does not include any authority to actually see or alter data.

4. Privileges that apply to a specific database

   a) DBADM - adminstrative privileges within a particular database - similar to SYSADM, but only with regard to a specific database.

   b) CONNECT - the privilege to connect to the database. (Obviously, no other access to the database is possible without this.)

   c) IMPLICIT_SCHEMA - the ability to create an implicit schema within the database (with the same name as the authorization id of the user creating it.) Only the database administrator can create a schema with any other name.

   d) CREATETAB - the privilege to create tables within the database.

5. Privileges that apply to a schema within a database

   a) CREATEIN - the privilege to create objects within the schema. CREATETAB on the database is also needed to create tables.)

   b) ALTERIN - the privilege to alter objects within the schema

   c) DROPIN - the privilege to drop objects within the schema

6. Privileges that apply to a specific table or view

   a) SELECT - the privilege to see the content of a table or view (by using a select statement), and to create views based on a table.

b) INSERT - the privilege to insert rows in a table (by using an insert statement on the table or an insertable view on the table)

c) UPDATE - the privilege to update rows in a table (by using an update statement on the table or an updatable view on the table)

d) DELETE - the privilege to delete rows from a table (by using a delete statement on the table or a deletable view on the table)

e) ALTER - the privilege to add columns to a table or to add or drop constraints on a table.

f) REFERENCES - the privilege to create a table that includes a references constraint that refers to this table. (This is needed because otherwise someone could prevent a deletion of a row from the table by creating another table with a foreign key constraint and storing a value into a row that prevents deletion of the matching row from the table he is trying to interfere with.)

g) CONTROL - we will discuss this shortly

7. Privileges that apply to a specific column within a table or view.

a) The UPDATE privilege may be granted only on specific columns within a table, rather than on the entire table.

b) The REFERENCES privilege may be granted only on specific columns within a table, rather than on the entire table.

c) There is no form of the SELECT privilege that allows seeing only specific columns within a table - this is not needed, because a view can be used for this purpose. (We will discuss views shortly)

d) There is obviously no column-specific form of INSERT or DELETE, since these operations inevitably affect an entire row.

8. Note that there are no privileges that apply to a specific row within a table - such privileges can be achieved through views.

E. Granting of privileges

1. For each object, the DBMS maintains a record of what privileges are granted to specific authorization id's. With the exception of SYSADM (which is determined by one's login), these privileges are stored in the system catalog.

2. For each object, the DBMS also maintains a record of who created the object, who is therefore the owner of the object and receives all appropriate privileges relative to that object.

   Example: To create a table, once must have the CREATETAB privilege within the database and the CREATEIN privilege within the schema. Once one creates a table, he/she automatically has SELECT, INSERT, UPDATE, DELETE, ALTER, CONTROL (and several other privileges we haven't mentioned) on the table.

   The same is true with views, except that CREATETAB is not needed, and ALTER is not applicable.

3. A privilege can be granted to an authorization ID that does not already have it by the SQL GRANT statement

   a) This has the following general form:

```
GRANT privilege ON object TO recipient
```

   Example: to give the user "aardvark" authority to look at the data in table "foo" the following statement would be used:

```
grant select on foo to user aardvark
```

b) So who has the right to grant privileges on an object?

(1) A holder of SYSADM or DBADM on the instance or database.

(2) The owner (creator) of the object.

(3) The holder of CONTROL privilege on that object.

(4) The grant statement for table-level privileges includes a `with grant option` clause which allows the recipient of a particular privilege on a particular object to grant the same privilege to others.

c) Of course, there are often objects which every person (who has the ability to connect to the database) may be entitled to access. Explicitly granting privileges to each user is painful to say the least, and breaks down if a user is added after privileges are granted. It also makes the storage of privilege information in the system tables unwieldy, to say the least.

For this reason, the grant statement allows privileges to be granted to a group or to everyone (public), as well as to an individual user.

Example: Let anyone who can connect to the database see the content of table foo:

```
grant select on foo to public
```

This translates into a single entry in the system catalog that grants select access to all users.

F. Revoking of Authority

1. The granter of a privilege may withdraw it by using the SQL REVOKE statement. Note, though, that if the same person has been granted a given privilege by two grantors, then if one revokes the privilege it will still remain in force.

   (This forms the basis of a security loophole in some SQL systems. If user A grants access to some object to user B with grant option, then user B can conspire with user C to retain that access even if user A chooses to withdraw it, as follows: User B grants access to user C with grant option, then user C grants access back to user B. The resulting loop in the grants prevents some implementations from revoking the grant if A now tries to revoke the access from B.)

2. DB2 handles this somewhat differently - to revoke a privilege, one needs to have SYSADM or DBADM authority, or CONTROL authority on the object in question - i.e. the original grantor of the privilege is not an issue. However, it is still possible that if A grants some privilege to B, who then grants it to C (both with grant option), and A revokes the privilege from B, C can give it back to B!

VII. Views

A. One important facility provided by relational DBMS systems is the notion of a VIEW.

1. A view is created by a statement of the form:

```
create view viewname as query
```

(where the query is typically a SELECT statement)

2. A view can be used just like a table in queries - but instead of the data being stored as a table, it is created "on the fly" when the query is run

B. One important use for a view is to give very fine-grained access control to a table. Sometimes, a given individual must be allowed to see to only a portion of a table - e.g.

1. A given user may be allowed to only see certain columns. For example, in a personnel database most users would be prohibited from seeing the salary column.

a) Recall that there is no "SELECT" privilege at the column level.

b) Instead, a view can be used.

Example: The `account_owners` view created in the examples we did for the introductory lecture only showed the names of account owners - not their balances.

```
create view account_owners as
          select owner, address
               from accounts;
```

2. A given user may be allowed to see only certain rows.

Example: the own_accounts view in the examples we did for the introductory lecture only one to see his/her own accounts.

```
create view own_accounts as
            select *
                 from accounts
                 where upper(owner) = user;
```

Implementing this is more complicated - the query used in defining the view must have a "where" clause that grants access to a row based on correlation between some value(s) in the row and the identify of the person accessing the view.

Example: security database from CPS221 - Project code for creating view; demo when logged in as bjork, aardvark. (Set schema to registrar in each case)

3. To use a view to limit access to certain columns or rows of a table.

   a) The creator of a view must have suitable access to the underlying table (e.g. SELECT authority.)

   b) Others may be given access to data through the view, even though they don't have direct access to the underlying table(s).

      (1) This is done by giving SELECT authority on the VIEW.

      (2) Now, the DBMS checks two SELECT privileges when a view is accessed: the authority of the accessor to access the view, and the authority of the creator of the view to access the underlying tables.

C. Another important use for a view is to provide simple access to the results of a complex query.

1. Example: in our library schema, we might want to make it easy to produce a report listing the titles of all books checked out. This could be done by

```
create view books_out as
        select title from book join checked_out on
             book.call_number = checked_out.call_number
```

A desk attendant could see a list of books checked out by doing the simple query `select * from books_out`, instead the more complex query used in defining it.

2. DEMO (with AUTOCOMMIT DISABLED)

   a) Create a view called books_out that lists the titles of all books that are checked out - as above

   b) Do select * from it.

   c) Note that creating a view does not store new data in the database. Rather, a reference to the view is handled by "running" the defining query. Any changes in the underlying tables will therefore be reflected automatically the next time the view is accessed.

   DEMO: drop a checkout, then repeat select * from view.

3. Note that the action of altering the database scheme with a DDL statement like create view is also under transaction control!

   DEMO: rollback and the attempt a select * from the view.

D. Views can be used to give selective access for inserting, deleting, or updating data as well as reading data.

1. That is, an insert, update, or delete operation can be performed on a view, and will result in changes to the underlying table -

provided the person doing the operation is authorized to perform the operation on the view, and the creator of the view is authorized to perform the operation on the base table(s).

2. Modifying the database through views does, however, present some interesting problems:

   a) If a new row is inserted into a view, what happens to columns that are not part of the view?

      Answer: they are given the value NULL - which may not be desirable. (And the operation will fail totally if any of these columns is declared NOT NULL.)

   b) There is the problem of DISAPPEARING ROWS. What if a user inserts or updates a row through a view in such a way that the row doesn't (or no longer) satisfies the conditions for being included in the view?

      Answer: the row is still in the table, but the one who put it there can't see it!

      To prevent this, it is possible to specify WITH CHECK OPTION when defining a view that is intended to be modifiable. In this case, any row that is created or changed as a result of an insert or update is checked to ensure that it still satisfies the view conditions; if it does not, the operation is not allowed.

   c) If the view involves a join or union, inserting into or deleting from or even updating the view becomes problematical.

      (1) What if we insert into a view involving a join? Do we add rows to both tables? What if one table has a row that could participate but the other does not?

(2) What if we insert a row into a view involving a union? Into which table does it go?

(3) What if we delete a row from a view involving a join? Do we delete the corresponding rows from both tables? Or just one? Which one?

(4) What if we update a column that is the basis of a join between tables? Which table do we change?

(5) Issues like these lead to most DBMS implementations either severely restricting or forbidding insertions, deletion, and even updating a view involving a join or union.

For example, DB2 defines the notion of an "insertable view", a "deletable view" and an "updatable view", with precise conditions the view must satisfy in order to allow these operations. (See the manual for the details, which get quite technical!)