CS352 Lecture: Integrity and Security Constraints revised 9/8/06

Materials:

1. Handout of SQL statements for creating example library database, showing entity and referential integrity constraints. (Students already have)
2. Handout of these statements modified to incorporate additional domain integrity constraints

I. Introduction

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

B. 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:

1. Data entry errors
2. System crashes
3. Anomalies due to concurrent and/or distributed processing

C. Data security 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.

D. These are actually two distinct topics, and each could be treated entirely separately. However, the book considers them together (under the heading "Advanced SQL" in ch. 4 and "Application Design" in ch. 8); therefore we will consider them together as well.

II. Integrity

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. 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. We have already seen that 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 wanted 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.

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.

C. 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 occuring 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. Such situations most often arise because of the way we transform E-R diagram relationships into tables

a. If the relationship is many to many, we must create a separate table for it, and even if it is one to one or one to many, we can do so. In this case, each row in the table representing the relationship will contain the primary keys of the entities being related.

b. If the relationship is not many to many, we can represent it by having one of the attributes of one of the tables be a foreign key referring to the primary key of the other table.

3. The requirement that a matching row occur in another table for each value occurring in a certain column (or set of columns) in a particular table is called REFERENTIAL INTEGRITY.

4. 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.

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

      ii. 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.

      i. 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.

      ii. 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.) [Note that this constraint represents something not directly representable in an ER diagram]

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

   c. Examples:

      i. Note references clauses in checked_out and reserve_book in the handout.

      ii. 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 borrower_id_type not null references borrower,
    call_number call_number_type not null,
    copy_number smallint not null,
    date_due date,
    foreign key (call_number, copy_number) references book
)
```

      iii. 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 ssn_type not null primary key,
    last_name name_type not null,
    first_name name_type not null,
    salary integer,
    supervisor_ssn ssn_type,
    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 there references clause were written (first_name, last_name), then we could only add Emily Elephant as an employee if Elephant Emily were a borrower!)

5. 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 borrowr. Thus, a change to borrower could cause a violation of a constraint for an existing row in checked_out

      i. 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.

      ii. 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.

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 delete from the borrower table, then any rows referencing that row in the fine table are also to be deleted.

iii. 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.

D. 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, SQL 92 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 a 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!
```

   i. 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))

   ii. It is possible to explicitly cast a value from one type to another.

      (a) The book discusses the SQL 92 syntax for type casting

      (b) DB2s 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))
```

   iii. 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:

   i. 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.

   ii. 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 SQL 92 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.)

   i. 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') )
)
```

   ii. 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.

E. An additional features applies to most of the types of constraints we have discussed thus far: A constraint may be given a name by preceeding 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.

F. 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.

G. 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 is notified. 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 is 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".

H. 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.

I. 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:
      i. Primary key constraints
      ii. Unique constraints
   b. Facilities to help preserve referential integrity: foreign key constraints
   c. Facilities to help preserve domain integrity:
      i. Not null constraints
      ii. User-defined domains
      iii. Check constraints
   d. Facilities to enforce more complex requirements:
      i. Assertions
      ii. 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.

J. We turn now to the topic of security - a distinct topic, but a clearly related one since all the data integrity measures in the world cannot protect an insecure system from malicious damage.

III. Security

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

B. System security is 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.

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)

   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 71 - 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.) A privilege may be granted to an individual, or to a group; an individual has a privilege if granted to him/her or to a group to which he/she belongs.

   a. 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 password.

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

      ii. 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.

   b. 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. 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

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 privileges that apply to objects at different levels.

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 Adminstrator Privilege. This is the highest level of authority, and implicitly includes all other privileges.

i. One place where this authority is needed is to actually create or drop individual databases within an instance.

ii. In DB2, this 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.)

b. SYSCTRL - the authority to manage system resources such as disk space. This authority 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. "A database administrator has all privileges against all objects in the database and may grant these privileges to others" (SQL Reference Manual Vol II page 570)

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 updateable view on the table)

d. DELETE - the privilege to delete rows from a table (by using a delete statement on the table or a deleteable 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.

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 authority

1. For each object, the DBMS maintains a record of what authorities 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 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?

i. A holder of SYSADM or DBADM on the instance or database.

ii. The owner (creator) of the object.

iii. The holder of CONTROL privilege on that object.

iv. 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 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!

IV. Views

A. We have introduced the notion of views as a very useful tool in the design of relational databases.

B. One way that view can be used is to give very fine-grained access control to a table.

   1. Sometimes, a given individual must be given access to only a portion of a table - e.g.

      a. 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.

      b. A given user may be allowed to see only certain rows. Again, in the case of a personnel database we may wish to let a department manager look only at the records of employees in the department he manages.

   2. These needs can be addressed by defining an appropriate VIEW. SQL allows the creator of a view to grant access to the view in the same way that access can be granted to a table. In particular:

      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 tables. (I.e. the DBMS checks two authorizations 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. 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 part of a key or 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 does not satisfy 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.

   i. 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?

   ii. What if we insert a row into a view involving a union? Into which table does it go?

   iii. 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?

   iv. What if we update a column that is the basis of a join between tables? Which table do we change?

   v. 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!)

V. Additional Issues

A. Another important tool for maintaining security - particularly in distributed systems - is ENCRYPTION.

1. Any time information is transmitted over a communication link from one place to another, it is vulnerable to being intercepted and even altered by unauthorized users.

2. Systems which can be accessed from remote sites are also potentially vulnerable to penetration by unauthorized users, who may be able to then obtain unauthorized access to read or motify the database by masquerading as a legitimate user.

3. Encryption can be used to reduce the risk of both types of unauthorized access.

   a. Sensitive information being transmitted over a network should certainly be encrypted in some suitable way.

   b. Stored data may also be encrypted to prevent unauthorized users from understanding it if they do obtain illicit access to it.

   c. Since encryption is discussed in other places in the curriculum (e.g. operating systems, networks) we will not discuss it further here.

B. One important class of security problems arises in conjunction with STATISTICAL DATABASES.

1. For a variety of reasons, it may be desirable to allow wide access to certain statistical functions (e.g. averages) of data for broad classes of people, while forbidding acces to specific data for individuals.

   Example: The US Census bureau publishes data on median household income for individual census tracts across the country. However, data on the income for a specific individual household is protected by law and cannot be released.

   If the number of people living in a given census tract is too small, however, the average for that tract is not published. (E.g. if a person lived in a tract with only two households, he could easily obtain his neighbor's income given the published information plus knowledge of his own.)

2. When such data is made available, care needs to be taken to make it impossible for some one to use a series of legitimate queries to obtain a piece of information that should remain confidential.

   Example: Suppose it were possible to query a database to obtain the average gpa for any desired list of 20 or more students at Gordon. One might then obtain the average gpa for all CS majors, or all students from Maine, or whatever. Given that each such group must include at least 20 students, it would seem hard to learn anything about the gpa of a specific individual, which should certainly remain confidential.

   However, a penetrator who wanted to obtain the gpa of a given individual might do so in two queries - by asking first for a group of 20 students that includes himself but does NOT include the target invidual, and then for the same group of 20 students except that it does not include himself but DOES include the target individual. From the difference of these two results, plus a knowledge of his own gpa, the malicious individual can calculate the gpa of the target person.

3. A strategy for dealing with problems like this is to impose rules restricting the degree of overlap between different queries by the same user. Even so, there are some subtleties involved, which will be explored in a bonus problem on the homework.