Security and Authorization

- Access to large databases is generally selective:
  - Distinct users have distinct privileges.
  - The process of defining and granting these privileges is called authorization.
  - Authorization is generally a **positive** action, designed to grant specific users specific privileges.

- Large databases must also be protected from those trying to obtain information which they are not intended to have.
  - Intruders may attempt to gain access to the system from the outside.
  - “Insiders” may attempt to bypass the authorization mechanism and look at information which they are not supposed to have.
  - Authorized users (e.g., the general public) may attempt to extract unintended information from statistical databases via techniques such as trackers.
  - Measures taken to control such access fall under the general heading of security, which is generally a **negative** or **preventive** measure.

Authorization

Generally speaking, there are two flavors of authorization:

- **Discretionary:** Individuals are given certain access privileges on data objects, as well as to propagate (*grant*) such privileges.

- **Mandatory:** Each data object has a certain fixed classification, as does each user. Only users with an appropriate classification may access a given data object.

Discretionary Access Control

An *authority* is a specification that a certain user has, or group of users have, the right to perform a given action on the database.

- The action of assigning authority is called *granting.*
- The action of relinquishing authority is called *revocation*.

The basic rules are the following:

- A user U has privilege P if and only if *some* other user V with the authority to grant that privilege has in fact granted it to U.

- Only a user U with privilege P *and* the authority to grant P to others may in fact grant that privilege to a second user V.

- A user U may revoke a privilege P from user V if and only if U had earlier granted that privilege to V.

- The database administrator (DBA) grants initial privileges; to avoid a chicken-and-egg problem.

- Authorization and SQL

The general syntax for assignment of a privilege is as follows:

GRANT <list of privileges>
ON <list of database objects>
TO <list of users>
[WITH GRANT OPTION] ;

The legal privileges are:
- SELECT, INSERT, UPDATE, DELETE;
- USAGE, REFERENCES (not discussed here).

Examples:

The following gives users Smith and Jones have the right to issue read-only (*i.e.,* Select) queries on the tables Employee and Department.

GRANT SELECT
ON Employee, Department
TO Smith, Jones

The following gives user Smith not only the select privilege on this table, but also the right to pass this privilege along to other users.

GRANT SELECT
ON Employee, Department
TO Smith
WITH GRANT OPTION

The next statement gives Smith both select and update privileges on the Employee table.

- Note that UPDATE has a specific semantics here; namely “change entries.” It does not imply the right to insert new tuples or to delete existing ones.

```
GRANT SELECT, UPDATE
ON Employee
TO Smith;
```

The following statement grants all three forms of modification:

```
GRANT SELECT, UPDATE, INSERT, DELETE
ON Employee
TO Smith;
```

In principle, it is possible to grant modification privileges without view privileges, but this would be rare.

```
GRANT INSERT
ON Employee
TO Smith;
```

To grant privileges on only part of a relation or relations, one must first create a view:

```
CREATE VIEW POOR_NAMES_ONLY AS
    SELECT Lname, Minit, Fname
    FROM Employee
    WHERE (Salary < 20000);

    GRANT SELECT
    ON POOR_NAMES_ONLY
    TO Smith;
```

It is even possible to grant privileges which are valid only at certain times.

```
CREATE VIEW POOR_NAMES_9_TO_5 AS
    SELECT Lname, Minit, Fname
    FROM Employee
    WHERE (Salary < 20000)
        AND CURRENT_TIME >= '09:00:00'
        AND CURRENT_TIME <= '17:00:00';

    GRANT SELECT
    ON POOR_NAMES_9_TO_5
    TO Smith;
```

The complement of GRANT is REVOKE. The general syntax is as follows:

```
REVOKE [GRANT OPTION FOR]
    <list of privileges>
    ON <list of database objects>
    FROM <list of users>
    RESTRICT | CASCADE;
```

Examples:

The following revokes the privilege of Smith to execute select operations on the relation employee, and also also revokes (in cascading fashion) any such privileges which Smith (alone) granted:

```
REVOKE SELECT
    ON Employee
    FROM Smith
    CASCADE;
```

The following is similar, except that it fails to do anything if it would be required that the privilege be revoked from some other user in cascading fashion.

```
REVOKE SELECT
    ON Employee
    FROM Smith
    RESTRICT;
```

It is possible for more than one user to grant the same privilege to another.

Example: Suppose that both Washington and Lincoln issue the following identical grant commands, which they have the authority to execute:

```
GRANT SELECT
    ON Employee
    TO Smith;
```

Now suppose that Washington issues the following command:

```
REVOKE SELECT
    ON Employee
    FROM Smith
    RESTRICT;
```

In this case, although the command “succeeds,” Smith retains the priviliges because it was also granted by Lincoln. On the other hand, if Lincoln subsequently issues the same command, Smith will lose the privilege.

However, suppose that the situation is as follows:

First, Washington grants the right to Lincoln:

```
GRANT SELECT
    ON Employee
    TO Lincoln
    WITH GRANT OPTION;
```

Now Lincoln passes this right on to Smith:

```
GRANT SELECT
    ON Employee
    TO Smith;
```

If Washington now issues the following command, Smith as well as Lincoln will lose the associated privileges.

```
REVOKE SELECT
    ON Employee
    FROM Lincoln
    CASCADE;
```

However, if CASCADE is replaced by RESTRICT, the directive will fail and both Smith and Lincoln will retain the privilege. (It is not clear how one is informed of this failure, since SQL does not have a standard status-return mechanism.)

Authorization in PostgreSQL:

- Privileges may be granted to any other user, but these privileges are useful only if that user is allowed to connect to the database on which the privileges were granted.

- If a user is allowed to connect to a database, then that user **always** has the privilege of creating new relations and using them.

- A user is always the owner of a relation created from that user account, regardless of the ownership of the actual database.

- Thus, if access is to be granted at all to a database, then the privilege of creating and owning new relations by those with access is irrevocable, even by the system administrator.

- If you allow a user to connect to your database, then that user will be able to create and control relations within your database. You may not even be able to read them. The creator must grant privileges to you!

- This is not good.

But...

- This applies only to access directly via PostgreSQL.
- More limited access may be achieved via applications written using ODBC or PHP.

Mandatory Access Control

Mandatory access control is applied in situations in which users may be assigned *security classes.* Assumptions and notation:

- The security classes form a total order; *e.g.,* top secret > secret > confidential > unclassified.

- Each user is assigned a security class. Write Clearance(U) to denote the security class of user U (called the *clearance* of U).

- Each database object is also assigned a value from this set of security classes. Classification(P) denotes the security class associated with database object P.

The following rules are then enforced:

1. User U has read access to object P iff
   Clearance(U) $\geq$ Classification(P).

(This is called the *simple security property.*)

2. User U has update privileges on object P iff
   Clearance(U) = Classification(P).

(This is called the *star property.*)

The first property is intuitive. The second seems strange and requires elaboration.

Analysis of the Star Property:

The intent of the star property is to prevent information from being passed down from a higher classification to a lower one.

Question: The textbook stipulates

$$\text{Clearance}(U) \leq \text{Classification}(P)$$

Is this not more flexible?

Answer: Yes, in a way, but then user U could write information which U would not subsequently be allowed to read!

Question: Is this requirement realistic in practice?

Answer: Probably not without some modification.

- It should be possible to trust people with higher classifications not to carelessly write this information into documents with lower classification.

Authority of the Database Administrator

- The *database administrator* (DBA) is the database equivalent of a system administrator.

Typically, the DBA has sole authority in the following areas of authorization:

- Create new accounts, and delete existing ones.
- Define security levels of accounts.
- Assign initial authorities.

Some of these responsibilities may be delegated in the management of a very large system, but only in very controlled ways.

Security

Key security issues:

- Prevent attacks from outside intruders. The issues here are similar to those for operating systems.

- Prevent unauthorized access from insiders. A key technique here is the maintenance of detailed transaction logs.

- Use care not to grant privileges unintentionally. This problem is particularly relevant in the context of statistical databases.

Security for Statistical Databases

It is common to grant “summary” access to large databases, without permitting detailed access.

Example query for company database:

> Provide the average salary of all employees in the research department.

- The idea here is to provide information about the general state of things, without revealing detailed, confidential information about individuals.

Some databases, particularly those maintained by government agencies, are explicitly stated to be maintained for purposes of summary information only, with details about individuals held “strictly confidential.”

Question: Can we maintain such privacy, and if so, how?

A basic problem is that of using so-called *individual trackers,* which are queries designed to identify a unique individual.

The following is a simple example, from
From D. E. Denning and P. J. Denning, *Data Security,* ACM Computing Surveys, Vol. 11, No. 3, 1979, pp. 227-249.

Suppose that we have a medical database which contains allows only statistical queries.

Query: How many patients have these characteristics?

Male
Age 45-50
Married
Two children
Harvard law degree
Bank vice president

Answer: 1

Suppose the questioner knows that Jones has these characteristics. Now the following query is posed.

Query: How many patients have these characteristics?

Male
Age 45-50
Married
Two children
Harvard law degree
Bank vice president
Took drugs for depression

The answer will be either 1 or 0, and will then tell us whether or not Jones took drugs for depression.

So, if the querier knows enough about Jones to construct the first query, further information may be obtained easily.
A candidate solution to this problem is a so-called *minimum query set control*. The idea is as follows:

Assume that the database contains n records. Let k be a relatively large positive integer which is less than n.

- Strategy: Prohibit queries for which there are fewer than k or more than n-k records in the query set.

Problem: Even with such controls, security may be comprised.

Example: This example uses the Company database of the text, and the specific instance shown in Figure 5.6 (7.6 in the 3rd edition).

Query: Find the salary of Joyce English.

The query

```
SELECT Salary
FROM    Employee
WHERE  (Lname = “English”)
            AND (Fname = “Joyce”);
```

is not allowed, since only statistical queries are permitted.

Suppose it is known that Joyce is the only female who works on the ProductY project. The “statistical” query P shown below delivers the correct answer.

```
P:  SELECT AVG(Salary)
    FROM    Employee, Works_On, Project
    WHERE  (SSN = ESSN) AND
           (PNO = PNumber) AND
           (PName = 'ProductY') AND
           (Sex = “F”);
```

However, it is not allowed, since it returns only one record.

- Note that n=8; there are 8 employees in the database.

- Suppose that k is set to 2 for this example. (It would be much larger in a real example.) Thus, any query must return aggregate data for at least two records, and no more than 6 records.

Start by building the following so-called *general tracker*, which we will call T0.

```
T0:  SELECT Count(*), AVG(Salary)
     FROM    Employee, Department
     WHERE  (DNO = Dnumber) AND
            (Dname = “Administration”);
```

It returns a count of 3. Since 3 >2, this query retrieves enough tuples to satisfy the above condition.

The complementary query T1 returns a count of 5.

```
T1:  SELECT Count(*), AVG(Salary)
     FROM    Employee, Department
     WHERE  (DNO = Dnumber) AND
                (NOT (Dname = “Administration”));
```

Thus, we know that the database consists of eight Employee tuples.

To proceed, we first need to know which of the two sets Joyce English is in. The query Q0, defined as

```
Q0: SELECT Count(*), AVG(Salary)
     FROM    Employee, Department
     WHERE  (DNO = Dnumber) AND
                ((Dname = “Administration”) OR
                (SSN IN
          (SELECT E.SSN
           FROM Employee E, Works_On, Project
           WHERE  (E.SSN = ESSN) AND
                     (PNO = PNumber) AND
                     (PName = 'ProductY') AND
                     (Sex = “F”)));
```

yields a count of four tuples, one more tuple than T0. Thus, we know that Joyce English is a member of the result of T1, and not of T0.

At this point, it is easy to compute the salary of Joyce English from the results of Q0 and T0. Just take the differences of the total salaries in each case.

If Q0 had returned a count of only three tuples, then it would be necessary to obtain the result of the following query Q1, and then use that result and the result of T1 to obtain the salary of Joyce English.

```
Q1: SELECT Count(*), AVG(Salary)
      FROM    Employee, Department
      WHERE  (DNO = Dnumber) AND
                ((NOT (Dname = "Administration"))
                OR
                (SSN IN
          (SELECT E.SSN
           FROM Employee E, Works_On, Project
           WHERE  (E.SSN = ESSN) AND
                    (PNO = PNumber) AND
                    (PName = 'ProductY') AND
                    (Sex = “F”)));
```

In either case, it is easy to find the exact salary of Joyce English from statistical queries over large sets alone.

Here is the general idea.

- Let T be the tracker formula, which divides the database into two large sets. Let T^c denote the complementary set. (These are T0 and T1 in the example.)

- Let P be the query which identifies the individual U uniquely.

First determine which of $\{T,T^c\}$ includes the individual, by issuing queries which measure the result size. Let $T^\wedge \in \{T,T^c\}$ denote the query which does not include the individual to be traced. Then

$(T^\wedge \vee P)$ delivers information on U aggregated with T.

$(T^\wedge \vee \neg P)$ delivers information on T without U.

From these two, information on U alone may easily be extracted.

How to deter tracking queries:

- **Database partitioning**: Partition the database into groups. Only queries whose record sets consists of the union of entire groups are allowed. Queries on subsets of groups are not allowed.

- **Noise:** Introduce “noise” into the result of a query, so that numerical answers are not exact. This must be done carefully, so that the noise cannot be filtered out by massaging a large number of queries.

- **Random samples:** Instead of presenting a database with all individuals, include only a random sample. This technique is useful for very large statistical-only databases, such as census databases.