Protection of DBMS

LECTURE 5

DCL. Access Control

IITU, ALMATY

What is Access Control?

- **Access Control** is a security term used to refer to a set of policies for restricting access to information, tools, and physical locations.

- Typically access control falls under the domain of physical access control or information access control.

Information Access Control

Information access control restricts access to data.

Some examples include:

- a user signing into their laptop using a password;
- a user unlocking their smartphone with a thumbprint scan;
- a Gmail user logging into their email account.

In all of these cases, software is used to authenticate and grant authorization to users wishing to access digital information. Both authentication and authorization are integral components of information access control.

Authentication and Authorization

- **Authentication** is the security practice of confirming that someone is who they claim to be, while **authorization** is concerned with the level of access each user is granted.

- When a user signs into their email or online banking account, they use a login and password combination that only they are supposed to know. The software uses this information to authenticate the user.

- Once authenticated, a user can only see the information they are authorized to access. In the case of an online banking account, the user can only see information related to their personal banking account. A fund manager at the bank can log in to the same application and see data on the bank’s financial holdings.

Types of Access Control

Correct configuration of access privileges is a critical component of protecting information. A DBMS should provide a mechanism to ensure that only authorized users can access the database.

DBMSs provide one or both of the following authorization mechanisms:

- Discretionary Access Control (DAC)
- Mandatory Access Control (MAC)

Discretionary Access Control (DAC)

- Each user is given appropriate access rights (or *privileges*) on specific database objects.
- Typically, users obtain certain privileges when they create an object and can pass some or all of these privileges to other users.

- SQL supports only discretionary access control through the GRANT and REVOKE statements.

What is a privilege?

- **Privileges** are the actions that a user is permitted to carry out on a given base table or view.
- Each DBMS allows a different set of privileges.

Possible privileges

The main privileges defined by the ISO standard are:

- SELECT – the privilege to retrieve data from a table;
- INSERT – the privilege to insert new rows into a table;
- UPDATE – the privilege to modify rows of data in a table;
- DELETE – the privilege to delete rows of data from a table;

The privilege may be granted for all columns of a table, or just specific columns.

GRANT

The **GRANT** statement is used to grant privileges on database objects to specific users. The format is:

```
GRANT {privilege_list | ALL PRIVILEGES}
ON ObjectName
TO {AuthorizationList | PUBLIC}
[WITH GRANT OPTION];
```

GRANT example

- **PrivilegeList** consists of one or more of the following privileges separated by commas.
- **ObjectName** can be the name of a base table or a view.
- To allow *vinny* to select and delete data on a table named *member*:

```
GRANT select, delete
ON member
TO vinny;
```

ALL PRIVILEGES

- For convenience, the GRANT statement allows the keyword ALL PRIVILEGES to be used to grant all privileges to a user instead of having to specify the privileges individually.
- The PRIVILEGES key word is optional in PostgreSQL, though it is required by strict SQL.

- To give the user *vinny* super-user access to the *book* table:

```
GRANT ALL PRIVILEGES
ON book
TO vinny;
```

PUBLIC

- It also provides the keyword PUBLIC to allow access to be granted to all present and future authorized users, not just to the users currently known to the DBMS.
- PUBLIC can be thought of as an implicitly defined group that always includes all roles.
- Any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to PUBLIC.

- Example for the *accounts* table:

```
GRANT ALL
ON accounts
TO PUBLIC;
```

WITH GRANT OPTION

- The WITH GRANT OPTION clause allows the user(s) in AuthorizationIdList to pass the privileges they have been given for the named object on to other users. If these users pass a privilege on specifying WITH GRANT OPTION, the users receiving the privilege may in turn grant it to still other users.
- If this keyword is not specified, the receiving user(s) will not be able to pass the privileges on to other users.
- Grant options cannot be granted to PUBLIC.

- Example:

```
GRANT ALL
ON book
TO vinny WITH GRANT OPTION;
```

REVOKE

The **REVOKE** statement is used to take away privileges that were granted with the GRANT statement. The REVOKE can take away all or some of the privileges that were previously granted to a user. The format is:

```
REVOKE [GRANT OPTION FOR] {privilege_list |
ALL PRIVILEGES}
ON ObjectName
FROM {AuthorizationList | PUBLIC}
[RESTRICT | CASCADE];
```

- The RESTRICT and CASCADE qualifiers operate exactly as in the DROP TABLE statement.

REVOKE example

- So, if we wanted to remove the DELETE privilege from *vinny* on the *member* table, we would write:

```
REVOKE delete
ON member
FROM vinny;
```

GRANT/ REVOKE with ROLE

- Add users in the role (group) with:
  GRANT group_role TO role1, ... ;

- Delete users from the role (group) with:
  REVOKE group_role FROM role1, ... ;

Books

- Connolly, Thomas M. Database Systems: A Practical Approach to Design, Implementation, and Management / Thomas M. Connolly, Carolyn E. Begg.- United States of America: Pearson Education

- Garcia-Molina, H. Database system: The Complete Book / Hector Garcia-Molina.- United States of America: Pearson Prentice Hall

- Sharma, N. Database Fundamentals: A book for the community by the community / Neeraj Sharma, Liviu Perniu.- Canada

Protection of DBMS

LECTURE 4

Data Control Language

IITU, ALMATY

Access Control

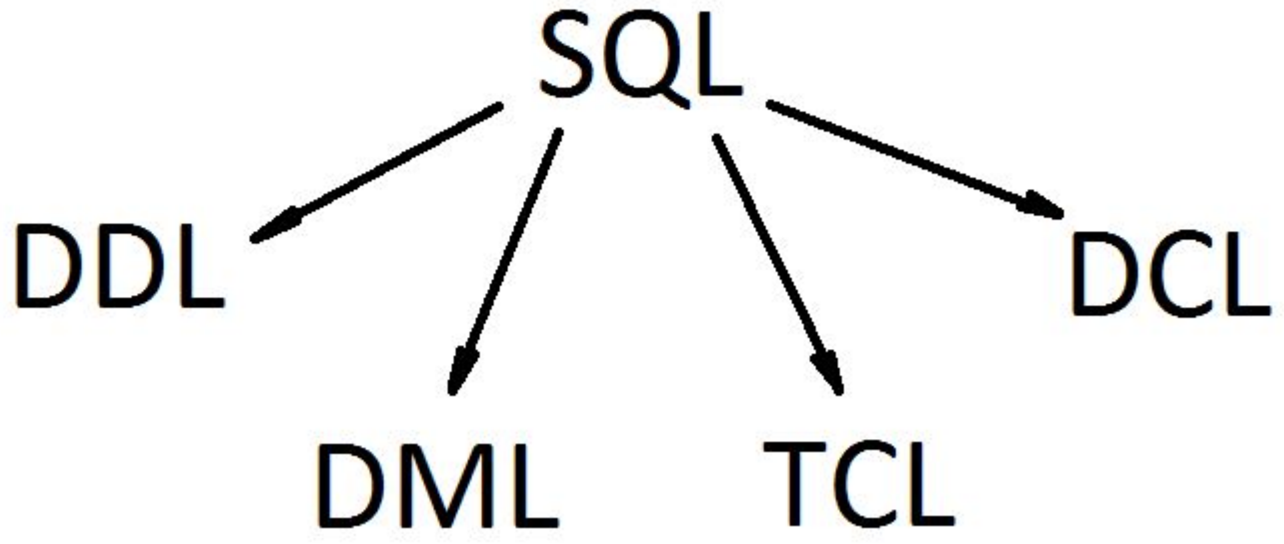
- Thus far, we are the only users that interact with the databases that we create.
- This is not the norm – typically, enterprise databases are used by many users.
- Also, every user of an enterprise database should not have “super-user” privileges.
- That is, some users should only be able to select data while others should be able to select, create, modify and destroy data.

DBA

- A database administrator (DBA) is the “super-user” of a database.

- A DBA can access and modify all data.
- A DBA can create users and grant users different privileges.

SQL Structure



- DDL (Data Definition Language)
- DML (Data Manipulation Language)

DCL statements

- SQL DCL provides the facility to create users, grant privileges to users, and revoke privileges from users.
- We can create users with the CREATE USER or CREATE ROLE statements.
- We can grant privileges to users using the GRANT statement.
- We can revoke privileges from users using the REVOKE statement.

CREATE USER

- To create a user we must tell the DBMS the user’s username and password.
- In PostgreSQL, we may issue the following command:

```
CREATE USER username
WITH PASSWORD ‘password’;
```

- To create a user with username *vinny* and password *123*:

```
CREATE USER vinny WITH PASSWORD ‘123’;
```

CREATE ROLE and CREATE USER

CREATE USER is now an alias for CREATE ROLE. The only difference is that when the command is spelled CREATE USER, LOGIN is assumed by default, whereas NOLOGIN is assumed when the command is spelled CREATE ROLE.

```
CREATE ROLE name LOGIN;
CREATE USER name;
```

CREATE ROLE

- CREATE ROLE defines a new database role
- **Role** is an entity that can own database objects and have database privileges
- **Role** can be considered a "user", a "group", or both depending on how it is used
- You must have CREATEROLE privilege or be a database superuser to use this command

CREATE ROLE

```
CREATE ROLE name [ [ WITH ] option [ ... ] ]

where option can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT connlimit
    | PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, …]
       …
```

CREATE ROLE options

`SUPERUSER | NOSUPERUSER`

These clauses determine whether the new role is a "superuser", who can override all access restrictions within the database. Superuser status is dangerous and should be used only when really needed. You must yourself be a superuser to create a new superuser. If not specified, `NOSUPERUSER` is the default.

`CREATEDB | NOCREATEDB`

These clauses define a role's ability to create databases. If `CREATEDB` is specified, the role being defined will be allowed to create new databases. Specifying `NOCREATEDB` will deny a role the ability to create databases. If not specified, `NOCREATEDB` is the default.

`CREATEROLE | NOCREATEROLE`

These clauses determine whether a role will be permitted to create new roles (that is, execute `CREATE ROLE`). A role with `CREATEROLE` privilege can also alter and drop other roles. If not specified, `NOCREATEROLE` is the default.

`CREATEUSER | NOCREATEUSER`

These clauses are an obsolete, but still accepted, spelling of `SUPERUSER` and `NOSUPERUSER`. Note that they are **not** equivalent to `CREATEROLE` as one might naively expect!

CREATE ROLE options

`INHERIT | NOINHERIT`

These clauses determine whether a role "inherits" the privileges of roles it is a member of. A role with the `INHERIT` attribute can automatically use whatever database privileges have been granted to all roles it is directly or indirectly a member of. If not specified, `INHERIT` is the default.

`LOGIN | NOLOGIN`

These clauses determine whether a role is allowed to log in; that is, whether the role can be given as the initial session authorization name during client connection. A role having the `LOGIN` attribute can be thought of as a user. Roles without this attribute are useful for managing database privileges, but are not users in the usual sense of the word. If not specified, `NOLOGIN` is the default, except when `CREATE ROLE` is invoked through its alternative spelling CREATE USER.

CREATE ROLE options

`CONNECTION LIMIT` ***`connlimit`***
If role can log in, this specifies how many concurrent connections the role can make. -1 (the default) means no limit.

`PASSWORD` `‘`***`password`***`'`
Sets the role's password. (A password is only of use for roles having the `LOGIN` attribute, but you can nonetheless define one for roles without it.) If you do not plan to use password authentication you can omit this option. If no password is specified, the password will be set to null and password authentication will always fail for that user. A null password can optionally be written explicitly as `PASSWORD NULL`.

`VALID UNTIL` `‘`***`timestamp`***`'`
The `VALID UNTIL` clause sets a date and time after which the role's password is no longer valid. If this clause is omitted the password will be valid for all time.

CREATE ROLE options

`IN ROLE` ***`role_name`*** `[, …]`
The `IN ROLE` clause lists one or more existing roles to which the new role will be immediately added as a new member.

`IN GROUP` ***`role_name`*** `[, …]`
`IN GROUP` is an obsolete spelling of `IN ROLE`.

`ROLE` ***`role_name`*** `[, …]`
The `ROLE` clause lists one or more existing roles which are automatically added as members of the new role. (This in effect makes the new role a "group".)

`ADMIN` ***`role_name`*** `[, …]`
The `ADMIN` clause is like `ROLE`, but the named roles are added to the new role `WITH ADMIN OPTION`, giving them the right to grant membership in this role to others.

`USER` ***`role_name`*** `[, …]`
The `USER` clause is an obsolete spelling of the `ROLE` clause.

CREATE ROLE examples

- Create a role with a password:

```
CREATE ROLE davide WITH LOGIN PASSWORD ‘jw8s0F4';
```

or

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

- Create a role with a password that is valid until the end of 2020. After one second has ticked in 2021, the password is no longer valid.

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4’
                                        VALID UNTIL ‘2022-01-01';
```

- Create a role that can create databases and manage roles:

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

DROP ROLE

DROP ROLE removes the specified role(s). To drop a superuser role, you must be a superuser yourself; to drop non-superuser roles, you must have CREATEROLE privilege.

DROP ROLE [IF EXISTS] ***name*** [, …]

- IF EXISTS do not throw an error if the role does not exist. A notice is issued in this case.

Example:

```
DROP ROLE davide;
```

ALTER ROLE

```
ALTER ROLE role_specification [ WITH ] option [ ... ]

where option can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT connlimit
    | PASSWORD 'password'
    | VALID UNTIL ‘timestamp'
   …

ALTER ROLE name RENAME TO new_name
```

Books

- Connolly, Thomas M. Database Systems: A Practical Approach to Design, Implementation, and Management / Thomas M. Connolly, Carolyn E. Begg.- United States of America: Pearson Education

- Garcia-Molina, H. Database system: The Complete Book / Hector Garcia-Molina.- United States of America: Pearson Prentice Hall

- Sharma, N. Database Fundamentals: A book for the community by the community / Neeraj Sharma, Liviu Perniu.- Canada

Protection of DBMS

LECTURE 2

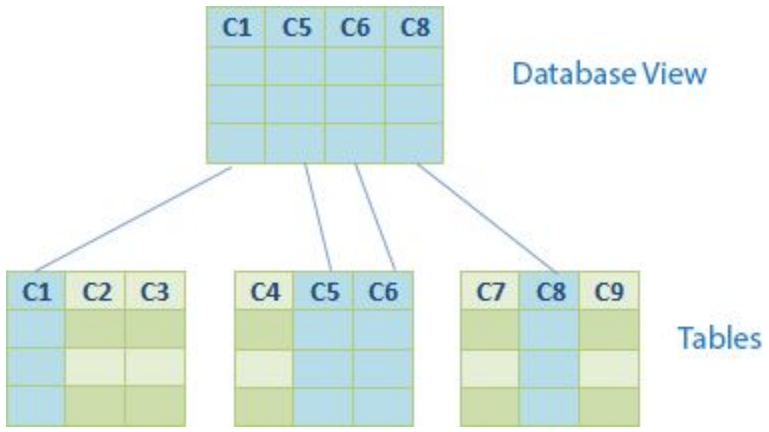
Views

IITU, ALMATY

View

View is a virtual table based on the result-set of an SQL statement.

View contains rows and columns, just like a real table. Fields in a view are fields from one or more real (physical) tables of the database.



View

- Views do not physically exist. Views are **virtual tables**.
- You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.
- A view always shows up-to-date data. The DB engine recreates the data, using the view's SQL statement, every time a user queries a view.
- Views are supported by all main DBMS.

Use of view: case 1

- In some cases, we may not want users to see all information in a table(s).
- Users need to be restricted from accessing this information.

Use of view: case 2

- In other case, a complex set of relational tables does not lend itself to easy use by non-database professionals.

- Consider a clerk at the library performing an audit. This clerk is only interested in the names of each member and the number of books those member have borrowed.
- Should this clerk have to write complex queries involving aggregate functions and joins over multiple tables? Probably not.

Use of views

Views allow users to do the following:

- Restrict access to the data such that a user can only see limited data instead of complete table.
- Structure data in a way that users or classes of users find natural or intuitive.
- Summarize data from various tables, which can be used to generate reports.

CREATE VIEW

- A view is created using the CREATE VIEW SQL command with SELECT on the defining tables.

- Syntax:

```
CREATE VIEW view_name
AS
SELECT …;
```

CREATE OR REPLACE VIEW

- CREATE OR REPLACE VIEW is similar, but if a view of the same name already exists, it is replaced.
- The new query must generate the same columns that were generated by the existing view query (that is, the same column names in the same order and with the same data types), but it may add additional columns to the end of the list.
- The calculations giving rise to the output columns may be completely different.
- CREATE OR REPLACE VIEW is a PostgreSQL extension.

CREATE VIEW example

- To create a view Students_info with only first and last names from the Students table.

```
CREATE VIEW Students_info
AS
SELECT fname, lname
FROM Students;
```

CREATE OR REPLACE VIEW

Syntax:

```
CREATE OR REPLACE VIEW view_name
AS
SELECT …;
```

Example:

```
CREATE OR REPLACE VIEW Students_info
AS
SELECT fname, lname, stud_id
FROM Students;
```

DROP VIEW

- Views can be deleted with the DROP VIEW statement.
- To delete the Students_groups view created on the previous slide.

  DROP VIEW Students_groups;

View with join

- Views may also be built by joining many tables.
- To create a view with last name and group’s name of each student.

```
CREATE VIEW Students_groups
AS
SELECT s.stud_id, s.lname, g.group_id, g.name
FROM Students s, Groups g
WHERE s.group_id = g.group_id;
```

View updating

- Updates to views are not simple. Recall that views are virtual tables – they do not physically exist.

- Any updates to views must be mapped onto the defining tables.
- If an update cannot be mapped, then a view is unupdatable.

View updating

For a view to be updatable, the DBMS must be able to trace any row or column back to its row or column in the source table.

- In general, a view is updatable if it contains a single table and contains a primary key.
- Generally, a view is not updatable if it contains a join operation.
- A view is definitely not updatable if it involves an aggregate function or a subquery.

View updating

- Use UPDATE SQL DML command to update the Students_info view:

```
UPDATE Students_info
SET fname = 'Alan'
WHERE stud_id = 3;
```

- It's identical to operation in the physical table Students.

Migration

Suppose we slightly altered the view to include only students with group_id = 2.

```
CREATE VIEW Group_2
AS
SELECT stud_id, fname, lname, group_id
FROM Students
WHERE group_id = 2;
```

Migration

- One problem with updatable views are the rows that we attempt to insert may violate the selection condition.
- Suppose we tried to update the view to change the student’s group_id to 3.

- Will that student still be part of the view?
- No, that student will not be part of the view. That row will **migrate** from the view.

Using views as physical tables

- Views can be used like any other real tables in DB.
- Also you can build view based on other views.

```
CREATE VIEW Students_infoAS
SELECT fname, lname
FROM Students;
```

- SELECT from the view:

```
SELECT * FROM Students_info;
```

Database Security: Access Control

- The view mechanism provides a powerful and flexible security mechanism by hiding parts of the database from certain users.
- The user is not aware of the existence of any attributes or rows that are missing from the view.
- A view can be defined over several relations with a user being granted the appropriate privilege to use it, but not to use the base relations.
- In this way, using a view is more restrictive than simply having certain privileges granted to a user on the base relation(s).

Summary

- A view is the dynamic result of one or more relational operations operating on the base relations to produce another relation.

- A view is a *virtual relation* that does not actually exist in the database, but is produced upon request by a particular user, at the time of request.

- Views take very little space to store; the database contains only the definition of a view, not a copy of all the data that it presents.

Summary

Views can represent a subset of the data contained in a table.

- A view can limit the degree of exposure of the tables to the outer world: a given user may have permission to query the view, while denied access to the rest of the base table.

- Views can join and simplify multiple tables into a single virtual table. Views can hide the complexity of data.

Books

- Connolly, Thomas M. Database Systems: A Practical Approach to Design, Implementation, and Management / Thomas M. Connolly, Carolyn E. Begg.- United States of America: Pearson Education

- Garcia-Molina, H. Database system: The Complete Book / Hector Garcia-Molina.- United States of America: Pearson Prentice Hall

- Sharma, N. Database Fundamentals: A book for the community by the community / Neeraj Sharma, Liviu Perniu.- Canada

Protection of DBMS

LECTURE 1

Introduction

IITU, ALMATY

Course Information

- Lectures
- Lab works

(individual work, University database)

- Project

(teams of 1-2 students, individual topic)

- Midterm / End of term – Quiz

DBMS Security

- Data is a valuable resource that must be strictly controlled and managed. Corporate data have strategic importance to a company and should be kept secure and confidential.
- DBMS must ensure that the database is secure. The term **security** refers to the protection of the database against unauthorized access, either intentional or accidental.
- Besides the services provided by the DBMS, discussions on database security also includes broader issues related to data protection.

Today’s lecture

- This lecture describes the scope of database security.
- We discuss why organizations must take potential threats to their computer systems seriously.
- We identify the range of threats and their consequences on computer systems.

Database Security

- **Database Security** - mechanisms that protect the database against intentional or accidental threats.

- Security considerations apply not only to the data held in a database: security breaches may affect other parts of the system, which may in turn affect the database.
- Consequently, database security includes hardware, software, people, data.

Database Security

- Effective security requires appropriate controls, which are defined in specific system objectives.
- The need for security has often been ignored in the past but is now increasingly recognized as important.
- The reason for this change is the growing amount of important corporate data stored on computers. Any loss or inaccessibility of this data can be catastrophic.

Risk situations

Database represents an essential corporate resource that should be properly secured using appropriate controls. Database security is considered in the following situations:

- theft and fraud;
- loss of confidentiality (secrecy);
- loss of privacy;
- loss of integrity;
- loss of availability.

These situations represent areas in which organizations should seek to reduce risks. In some situations, these areas are closely related such that an activity that leads to loss in one area may also lead to loss in another.

Theft and fraud

- **Theft and fraud** affect not only the database environment but also the entire organization.
- As it is people who perpetrate such activities, attention should focus on reducing the opportunities for this occurring.
- Theft and fraud do not necessarily alter data, as is the cases of loss of confidentiality or loss of privacy.

Confidentiality and Privacy

- **Confidentiality** refers to the need to maintain secrecy over data, but usually only data which is critical to the organization.
- **Privacy** refers to the need to protect data about individuals.
- Security breaches resulting in loss of confidentiality could, for instance, lead to loss of competitiveness, and loss of privacy could lead to legal action against the organization.

Loss of data integrity

- **Loss of data integrity** results in invalid or corrupted data, which may seriously affect the work in the organization. Many organizations now provide continuous operation, so called 24/7 availability (that is, 24 hours a day, 7 days a week).

Loss of availability

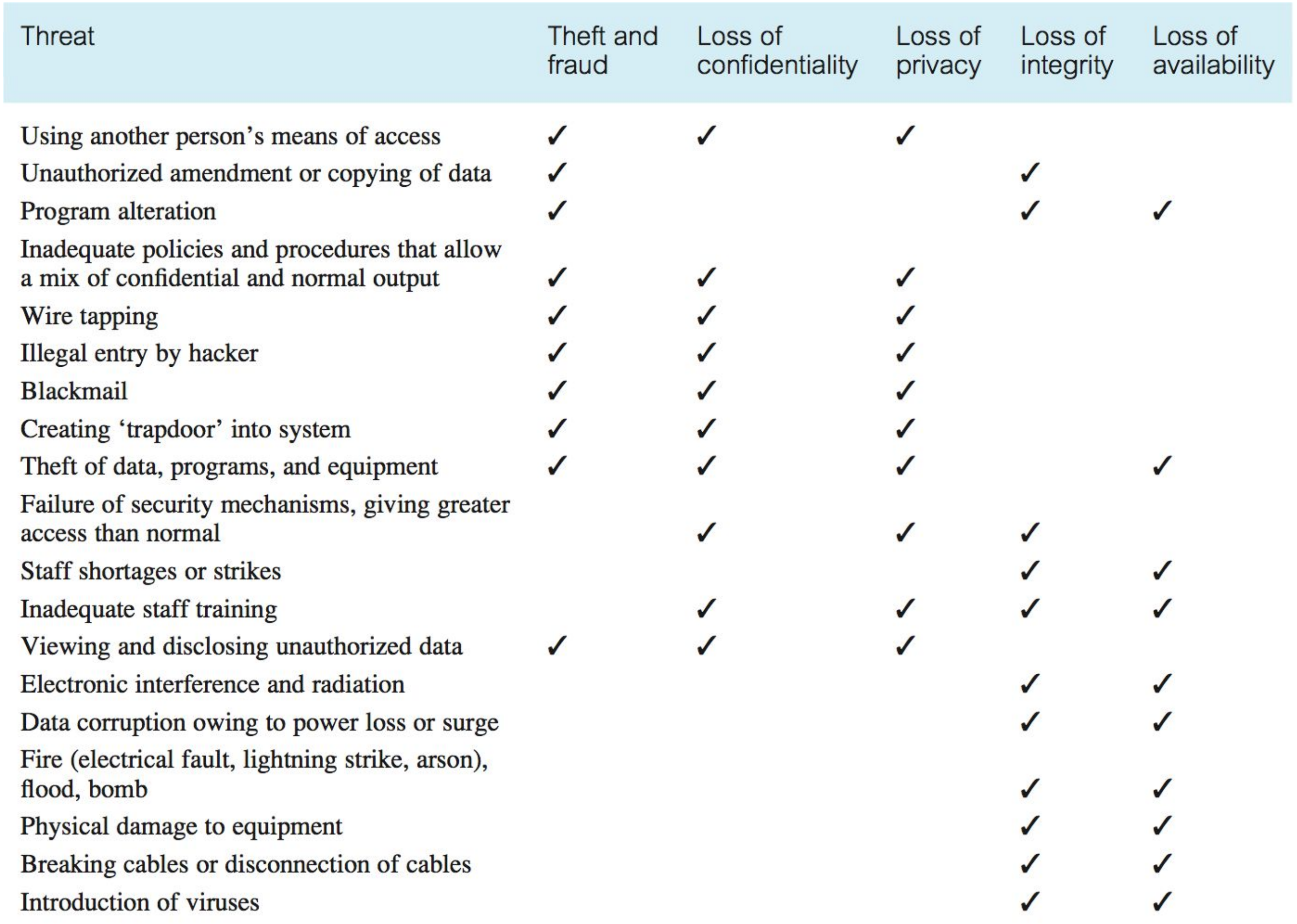
- **Loss of availability** means that the data, or the system, or both cannot be accessed, which can seriously affect the organization’s financial performance.
- In some cases, events that cause a system to be unavailable may also cause data corruption.

Threats

- **Threat** - any situation or event, whether intentional or accidental, that may adversely affect a system and consequently the organization.
- Threat may be caused by a situation or event involving a person, action, or circumstance that is likely to bring harm to the organization.
- The problem facing any organization is to identify all possible threats. Therefore, organizations should invest time and effort in identifying threats.
- Previous slides define areas of loss from intentional or unintentional activities. While some types of threat can be either intentional or unintentional, the impact remains the same.
- Any threat must be viewed as a potential security breach which, if successful, will have a certain impact.

Threats

Examples of various threats with areas on which they may have an impact.



Threat	Theft and fraud	Loss of confidentiality	Loss of privacy	Loss of integrity	Loss of availability
Using another person's means of access	✓	✓	✓		
Unauthorized amendment or copying of data	✓			✓	
Program alteration	✓			✓	✓
Inadequate policies and procedures that allow a mix of confidential and normal output	✓	✓	✓		
Wire tapping	✓	✓	✓		
Illegal entry by hacker	✓	✓	✓		
Blackmail	✓	✓	✓		
Creating 'trapdoor' into system	✓	✓	✓		
Theft of data, programs, and equipment	✓	✓	✓		✓
Failure of security mechanisms, giving greater access than normal		✓	✓	✓	
Staff shortages or strikes				✓	✓
Inadequate staff training		✓	✓	✓	✓
Viewing and disclosing unauthorized data	✓	✓	✓		
Electronic interference and radiation				✓	✓
Data corruption owing to power loss or surge				✓	✓
Fire (electrical fault, lightning strike, arson), flood, bomb				✓	✓
Physical damage to equipment				✓	✓
Breaking cables or disconnection of cables				✓	✓
Introduction of viruses				✓	✓

Threats

- Organization's security depends on the availability of countermeasures and an action plan.
  - For example, if hardware fails and secondary storage becomes damaged, all processing operations must be stopped until the problem is resolved. Recovery depends on the time of the last backup and the recovery time.
- Organization should define types of threats and countermeasures, taking into account costs of their implementation.
  - It is ineffective to waste time, effort, money on potential threats that could result in minor inconvenience. However, rare events should be considered if their impact is significant.

Classifications of Threats

1. by purpose of threat implementation
2. by the origin of a threat
3. by localization of threat source
4. by location of threat source
5. by way of impact on a data storage of information system
6. by the nature of the impact on the information system

1. Classification by purpose of threat implementation

- Violation of the confidentiality of information
  - use of information in the system by persons or processes that have not been identified by the owners
- Violation of information integrity
  - modification or destruction of information to devalue it due to loss of correspondence with the state of the real world
- Total or partial disruption of operation due to failure or improper change in the operation of system components, including their modification or replacement

2. Classification by the origin of a threat

- Natural threats
  - threats caused by the impact on the database of physical processes or spontaneously developing natural phenomena
- Artificial threats
  - threats to information security of DBMS related to human activities

3. Classification by localization of threat source

- Threats, a direct source of which is a human
- Threats, a direct source of which is the usual software and hardware of the IS
- Threats, a direct source of which is unauthorized software and hardware
- Threats, a direct source of which is a habitat.

4. Classification by location of threat source

- Threats, the source of which is located outside the controlled area of the IS’s location.
- Threats, the source of which is located within the controlled area of the IS, including the location of client terminals and servers.

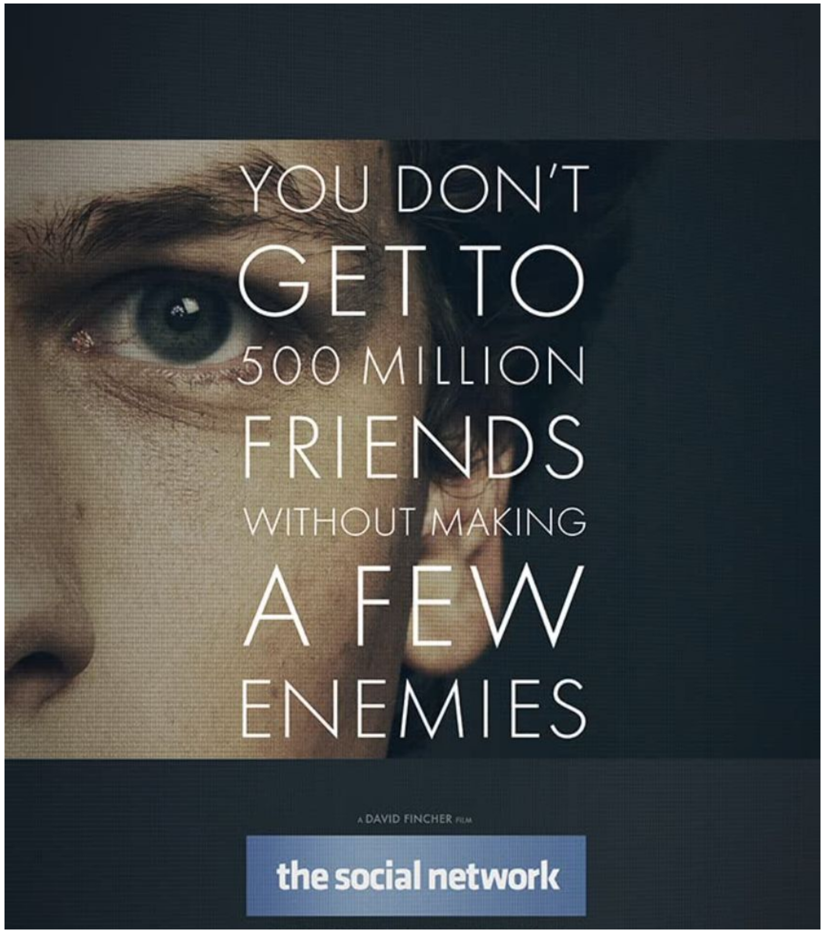
5. Classification by way of impact on a data storage of the IS

- Threat of information security of data stored on external devices.
- Threat of information security of data stored in the RAM of servers and client computers.
- Threat of information security of data displayed on the user's terminal or printer.

6. Classification by the nature of the impact on the IS

- Active impact
  - user actions that go beyond his responsibilities
- Passive impact
  - the user observes values of DBMS parameters and various indirect characteristics in order to obtain confidential information

Example



Movie:

The Social Network (2010)

Hacking Scene:

~ 9 - 12 min

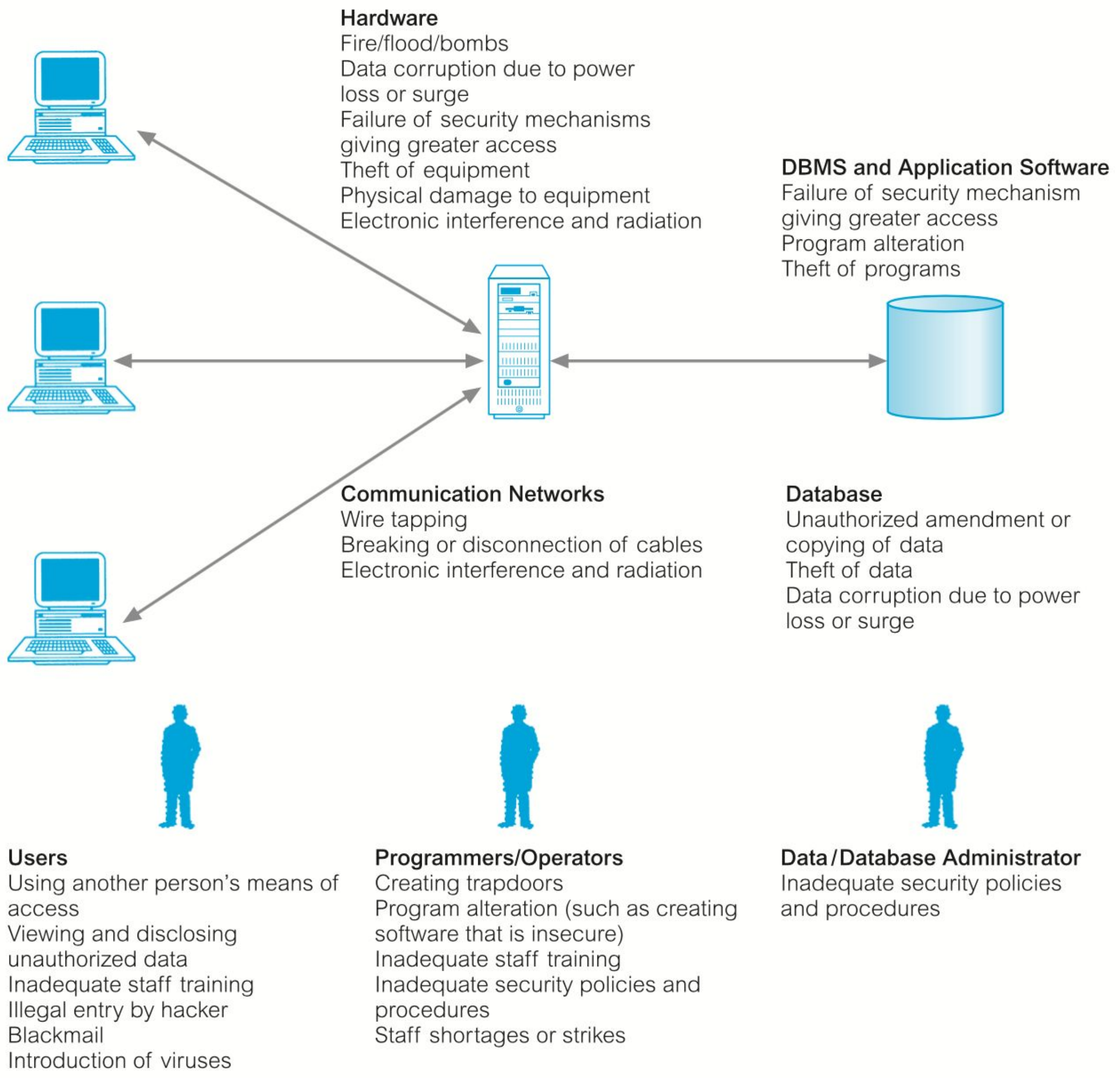
Example: Classification

Classification	Type
by purpose of threat implementation	Mark Zuckerberg used face books, where people provided their photos, that means that he violated the **confidentiality of information**
by the origin of a threat	We can classify that threat as an **artificial**, because Mark did all the work himself
by localization of threat source	In this case, the only threat was caused by a **human**(direct source – Mark)
by location of threat source	Mark had access to all sources and information systems as a student of Harvard, so it can be classified as **a threat, the source of which is located within the controlled area of the information system**
by way of impact on a data storage of the information system	Mark used the information that was stored on servers, so it's classified as **a threat of information security of data stored in the RAM of servers**
by the nature of the impact on the information system	It was shown that Mark did **active impact** (the user actions that go beyond his responsibilities)

Books

- Connolly, Thomas M. Database Systems: A Practical Approach to Design, Implementation, and Management / Thomas M. Connolly, Carolyn E. Begg.- United States of America: Pearson Education
- Garcia-Molina, H. Database system: The Complete Book / Hector Garcia-Molina.- United States of America: Pearson Prentice Hall
- Sharma, N. Database Fundamentals: A book for the community by the community / Neeraj Sharma, Liviu Perniu.- Canada

Summary of potential threats


Hardware
Fire/flood/bombs
Data corruption due to power loss or surge
Failure of security mechanisms giving greater access
Theft of equipment
Physical damage to equipment
Electronic interference and radiation
DBMS and Application Software
Failure of security mechanism giving greater access
Program alteration
Theft of programs
Communication Networks
Wire tapping
Breaking or disconnection of cables
Electronic interference and radiation
Database
Unauthorized amendment or copying of data
Theft of data
Data corruption due to power loss or surge
Users
Using another person's means of access
Viewing and disclosing unauthorized data
Inadequate staff training
Illegal entry by hacker
Blackmail
Introduction of viruses
Programmers/Operators
Creating trapdoors
Program alteration (such as creating software that is insecure)
Inadequate staff training
Inadequate security policies and procedures
Staff shortages or strikes
Data/Database Administrator
Inadequate security policies and procedures

Summary

- Database security aims to minimize losses caused by unacceptable events in a cost-effective way without constraining users.

- Computer-based crime has increased significantly recently and will continue to grow in the coming years.

Protection of DBMS

LECTURE 10

Backup and Recovery

IITU, ALMATY

Content

- Backup
- Recovery (Restore)

Database Backup

- **Backup** is the process of periodically taking a copy of the database to offline storage.
- A DBMS should provide backup facilities to assist with the recovery of a database following failure.
- It is always advisable to make backup copies of the database at regular intervals and to ensure that the copies are in a secure location.
- In the event of a failure that renders the database unusable, the backup copy is used to restore the database to the latest possible consistent state.

Database Backup

A DBMS should provide the following facilities to assist with recovery:

- a backup mechanism, which makes backup copies of the database;
- a recovery manager, which allows the system to restore the database to a consistent state following a failure.

Database Backup

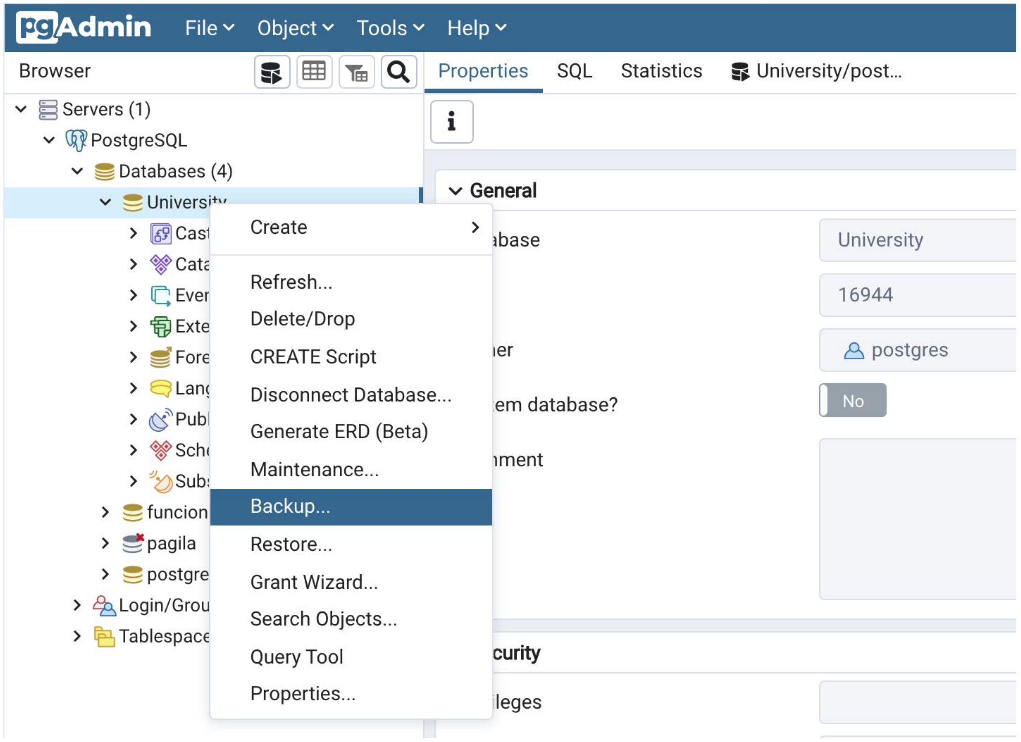
- PostgreSQL provides pg_dump and pg_dumpall tools to backup databases.
- pg_dump - extract a database into a script file or other archive file.
- It makes consistent backups even if the database is being used concurrently. pg_dump does not block other users accessing the database.
- pg_dump only dumps a single database. To backup global objects that are common to all databases in a cluster, such as roles, use pg_dumpall.

Backup with pgAdmin

- The pgAdmin provides an intuitive user interface that allows you to backup a database using pg_dump tool.
- For example to backup the University database, you can follow the steps.

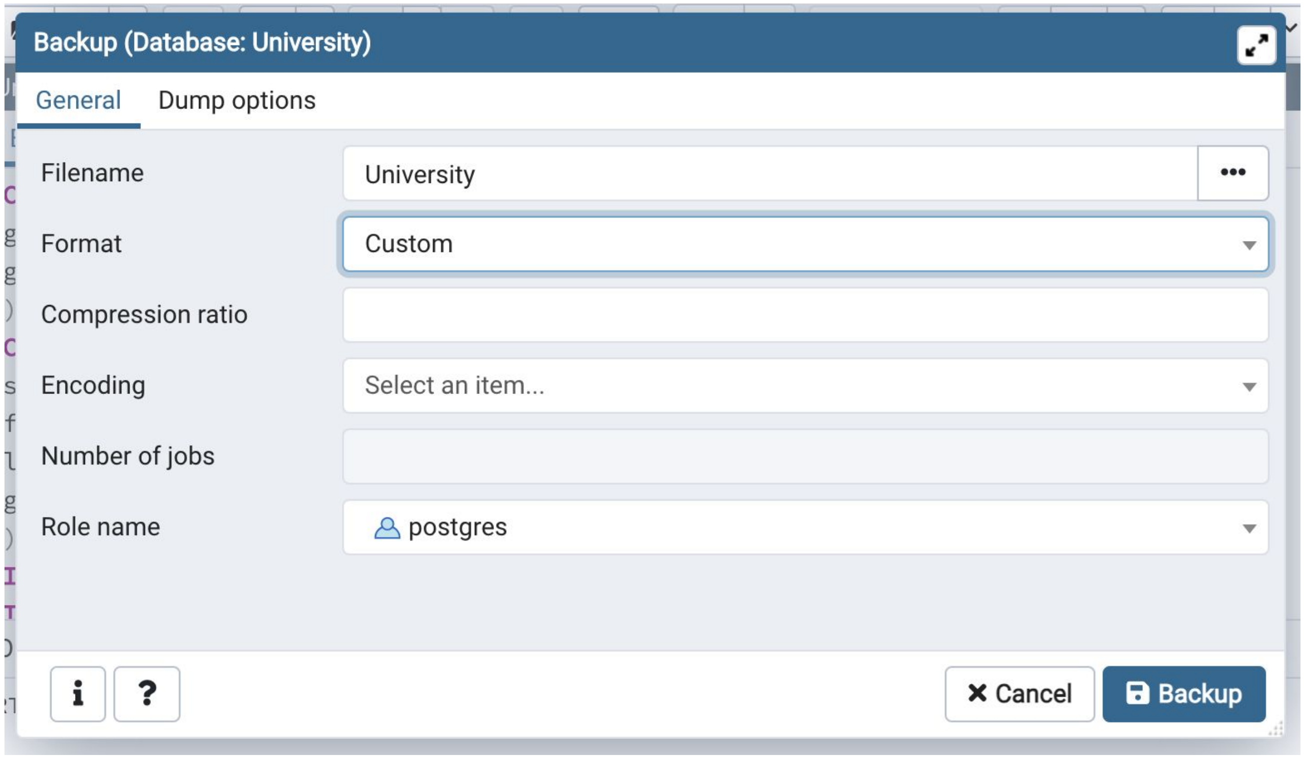
Backup with pgAdmin

- First, right mouse click on the University database, and choose the Backup menu item.



Backup with pgAdmin

- Second, enter the output file name and choose the file format.



Backup (Database: University)

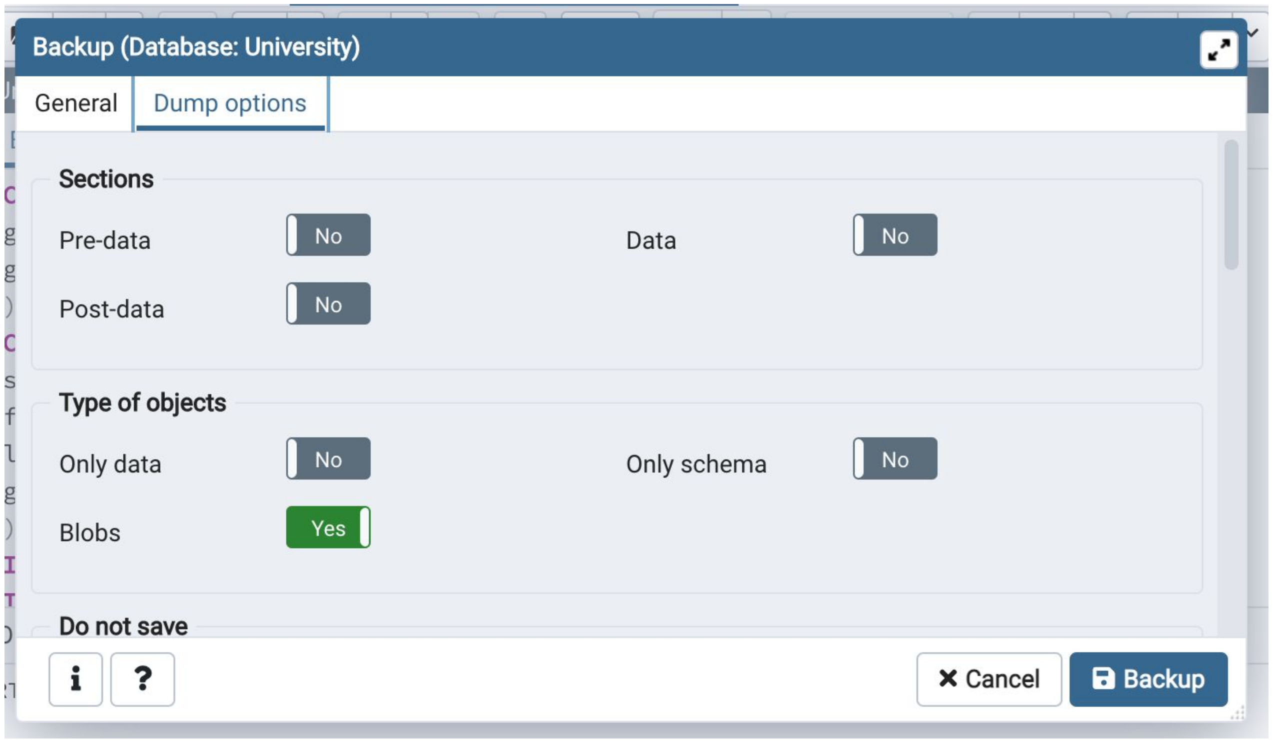
Filename	University
Format	Custom
Compression ratio	
Encoding	Select an item...
Number of jobs	
Role name	postgres

Backup formats

- **Plain.** Output a plain-text SQL script file.
- **Custom.** Output a custom-format archive suitable for input into pg_restore. Together with the directory output format, this is the most flexible output format in that it allows manual selection and reordering of archived items during restore. This format is also compressed by default.
- **Directory.** Output a directory-format archive suitable for input into pg_restore. This will create a directory with one file for each table, plus a so-called Table of Contents file describing the dumped objects in a machine-readable format that pg_restore can read. This format is compressed by default.
- **Tar.** Output a tar-format archive suitable for input into pg_restore. The tar format is compatible with the directory format: extracting a tar-format archive produces a valid directory-format archive. However, the tar format does not support compression. Also, when using tar format the relative order of table data items cannot be changed during restore.

Backup with pgAdmin

- pgAdmin backup tool provides various dump options as follows:



Backup (Database: University)

General | Dump options

Sections

Pre-data	No	Data	No
Post-data	No		

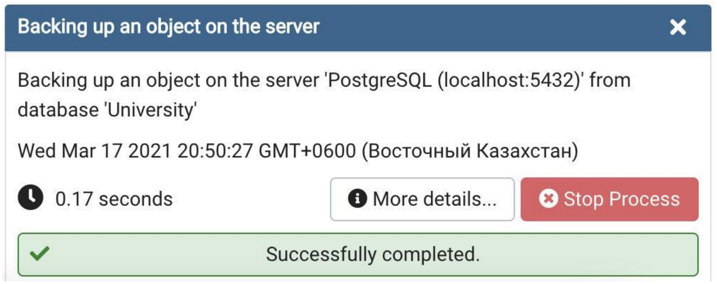
Type of objects

Only data	No	Only schema	No
Blobs	Yes		

Do not save

Backup with pgAdmin

- Third, click Backup button to start performing a backup.
- pgAdmin provides detailed information of the backup process.



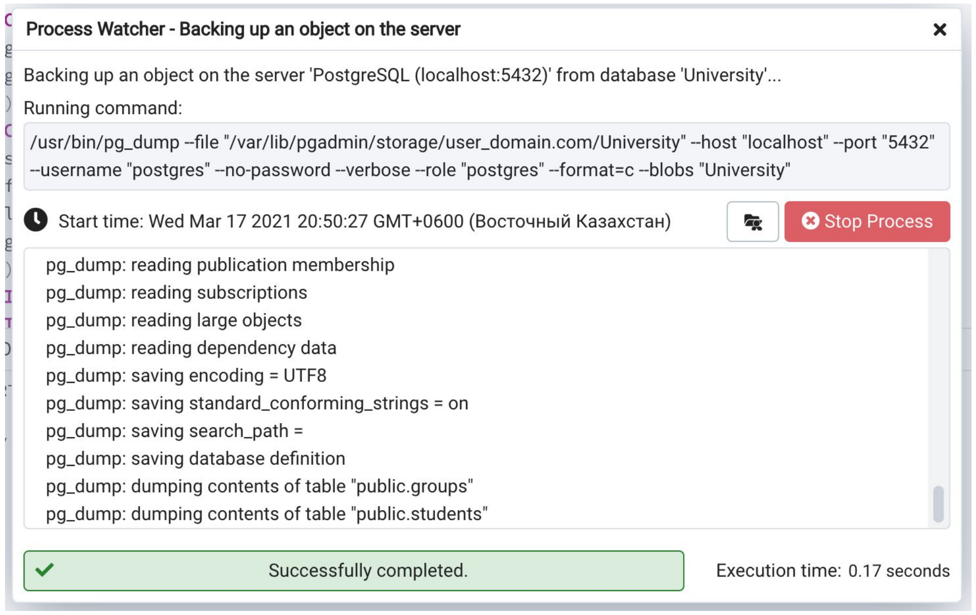
Backing up an object on the server

Backing up an object on the server 'PostgreSQL (localhost:5432)' from database 'University'

Wed Mar 17 2021 20:50:27 GMT+0600 (Восточный Казахстан)

0.17 seconds

Successfully completed.



Process Watcher - Backing up an object on the server

Backing up an object on the server 'PostgreSQL (localhost:5432)' from database 'University'...

Running command:

```
/usr/bin/pg_dump --file "/var/lib/pgadmin/storage/user_domain.com/University" --host "localhost" --port "5432"
--username "postgres" --no-password --verbose --role "postgres" --format=c --blobs "University"
```

Start time: Wed Mar 17 2021 20:50:27 GMT+0600 (Восточный Казахстан)

```
pg_dump: reading publication membership
pg_dump: reading subscriptions
pg_dump: reading large objects
pg_dump: reading dependency data
pg_dump: saving encoding = UTF8
pg_dump: saving standard_conforming_strings = on
pg_dump: saving search_path =
pg_dump: saving database definition
pg_dump: dumping contents of table "public.groups"
pg_dump: dumping contents of table "public.students"
```

Successfully completed.

Execution time: 0.17 seconds

Recovery

- Dumps can be output in script or archive file formats.
- Script dumps are plain-text files containing the SQL commands required to reconstruct the database to the state it was in at the time it was saved. To restore from such a script, feed it to psql.
- Script files can be used to reconstruct the database even on other machines and other architectures; with some modifications, even on other SQL database products.
- The alternative archive file formats must be used with pg_restore to rebuild the database. They allow to be selective about what is restored.

Recovery

In PostgreSQL, you can restore a database in two ways:

- Using psql to restore plain SQL script file generated by pg_dump and pg_dumpall tools.
- Using pg_restore to restore tar file and directory format created by the pg_dump tool.

Recovery with pgAdmin

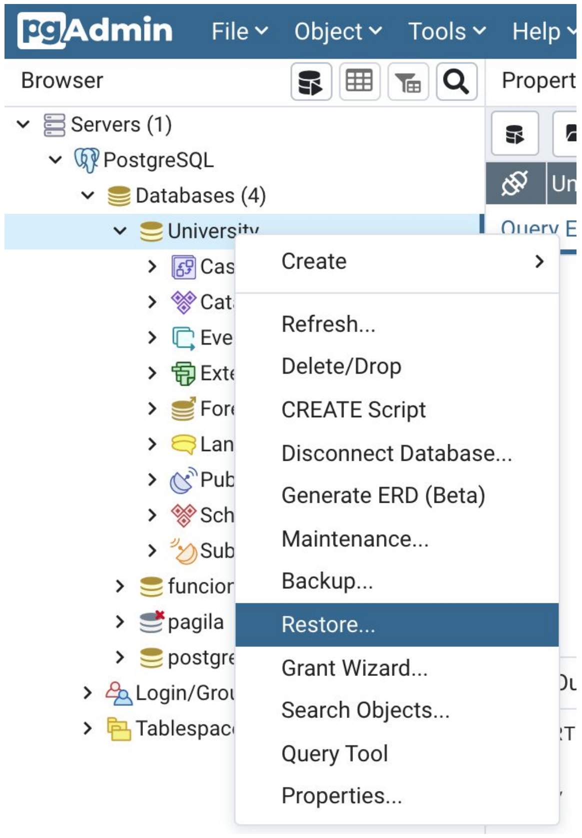
- If you want to run the recovery via an intuitive user interface instead of the command line, you can use the pgAdmin restore tool.
- The following example demonstrates how to restore the University database.

Recovery with pgAdmin

- First, drop the existing University database.
- Second, create an empty University database.

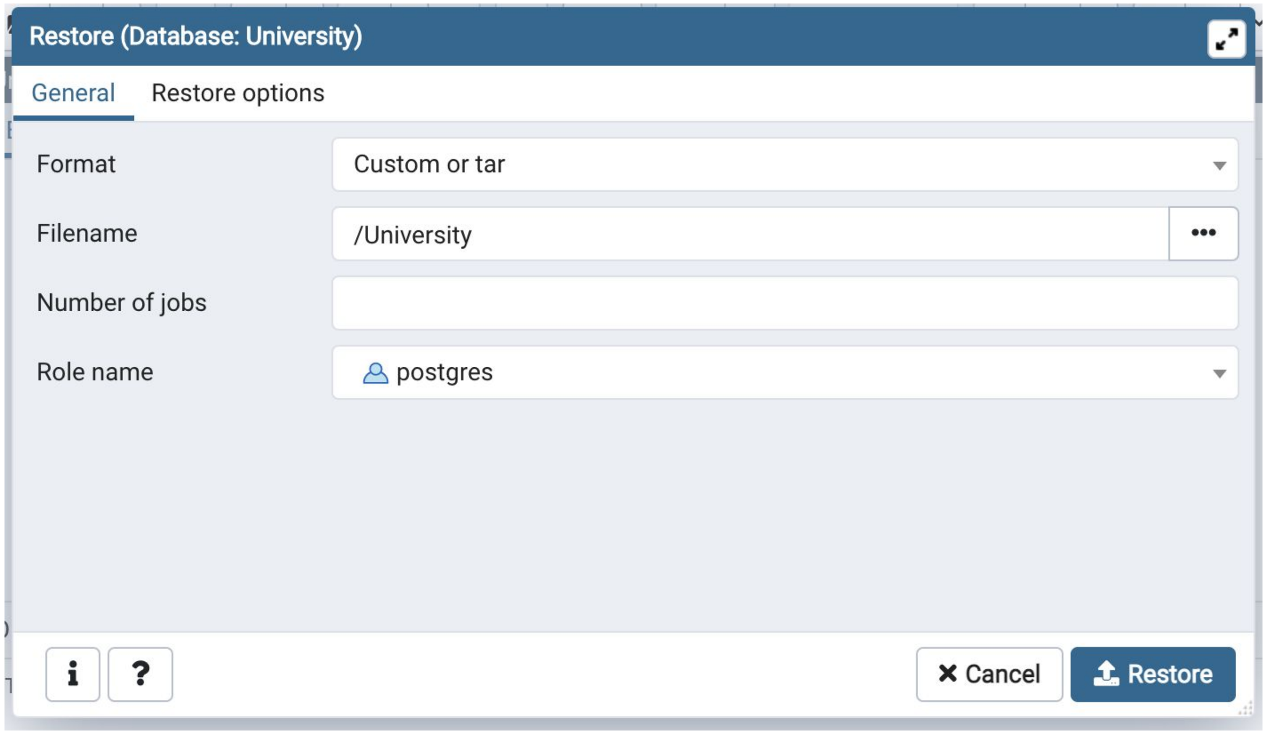
Recovery with pgAdmin

- Third, choose the University database, right mouse click and choose the Restore menu item.



Recovery with pgAdmin

- Fourth, choose appropriate options such as backed up file, role, restore options, and click the Restore button to start restoring the database.



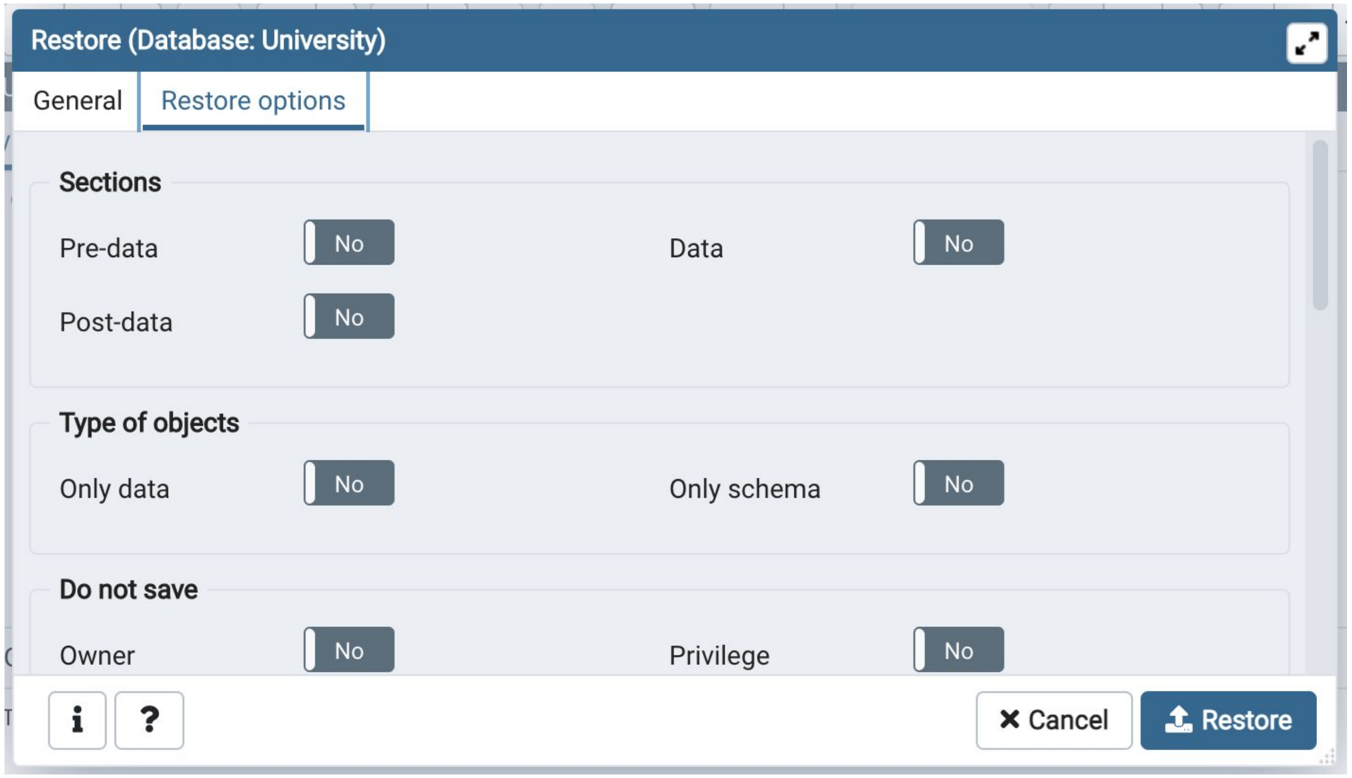
Restore (Database: University)

General | Restore options

Format	Custom or tar
Filename	/University
Number of jobs	
Role name	postgres

Recovery with pgAdmin

- Possible restore options:



Restore (Database: University)

General | Restore options

Sections

Pre-data	No	Data	No
Post-data	No		

Type of objects

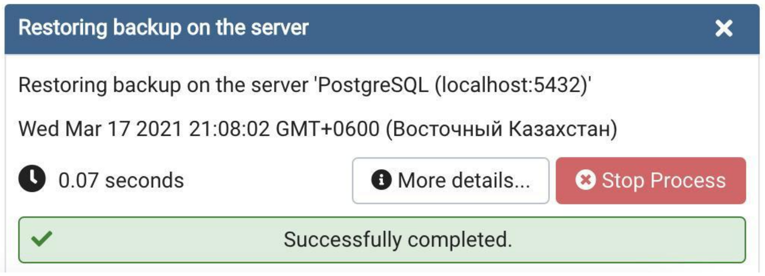
Only data	No	Only schema	No

Do not save

Owner	No	Privilege	No

Recovery with pgAdmin

- pgAdmin displays detailed information.

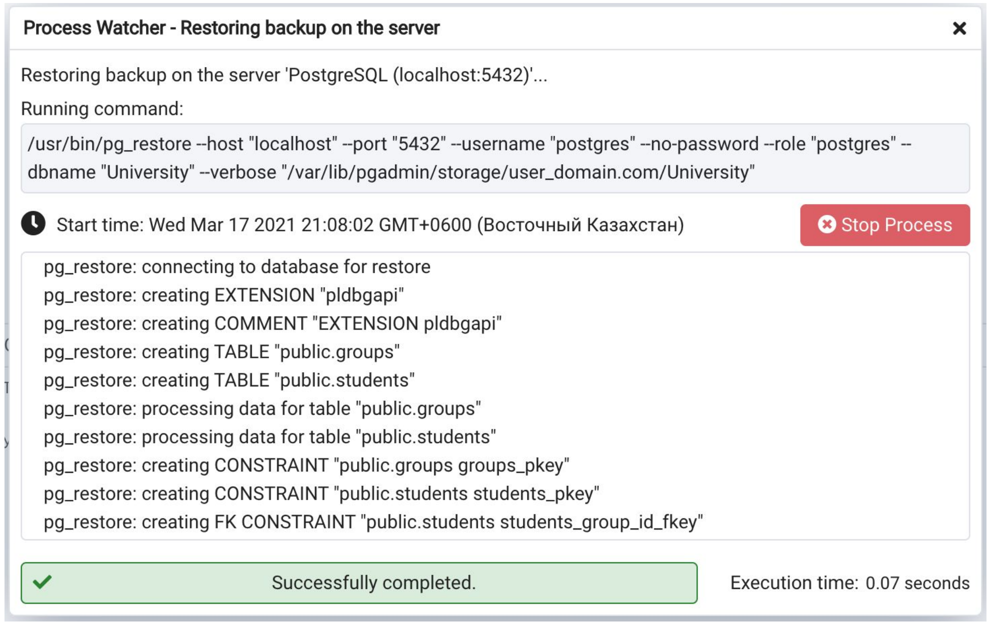


Restoring backup on the server 'PostgreSQL (localhost:5432)'

Wed Mar 17 2021 21:08:02 GMT+0600 (Восточный Казахстан)

0.07 seconds

Successfully completed.



Restoring backup on the server 'PostgreSQL (localhost:5432)'...

Running command:

```
/usr/bin/pg_restore --host "localhost" --port "5432" --username "postgres" --no-password --role "postgres" --
dbname "University" --verbose "/var/lib/pgadmin/storage/user_domain.com/University"
```

Start time: Wed Mar 17 2021 21:08:02 GMT+0600 (Восточный Казахстан)

```
pg_restore: connecting to database for restore
pg_restore: creating EXTENSION "pldbgapi"
pg_restore: creating COMMENT "EXTENSION pldbgapi"
pg_restore: creating TABLE "public.groups"
pg_restore: creating TABLE "public.students"
pg_restore: processing data for table "public.groups"
pg_restore: processing data for table "public.students"
pg_restore: creating CONSTRAINT "public.groups groups_pkey"
pg_restore: creating CONSTRAINT "public.students students_pkey"
pg_restore: creating FK CONSTRAINT "public.students students_group_id_fkey"
```

Successfully completed.

Execution time: 0.07 seconds

Copy DB within the same server

- While the previous method copies a database from a server to another, here is another one to copy database within the same server (for testing purposes, for example).
- PostgreSQL makes it easy to do it via the CREATE DATABASE statement as follows:

```
CREATE DATABASE targetdb
WITH TEMPLATE sourcedb;
```

- This statement copies the sourcedb to the targetdb.

Copy DB within the same server

- For example, to copy the University database to the University_copy database, you use the following statement:

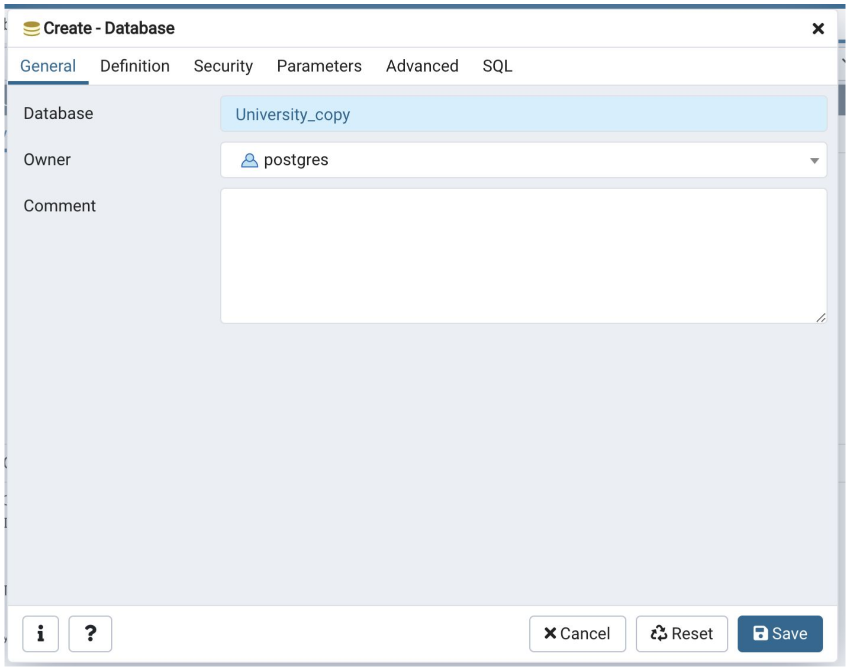
```
CREATE DATABASE University_copy
WITH TEMPLATE University;
```

Copy DB with pgAdmin

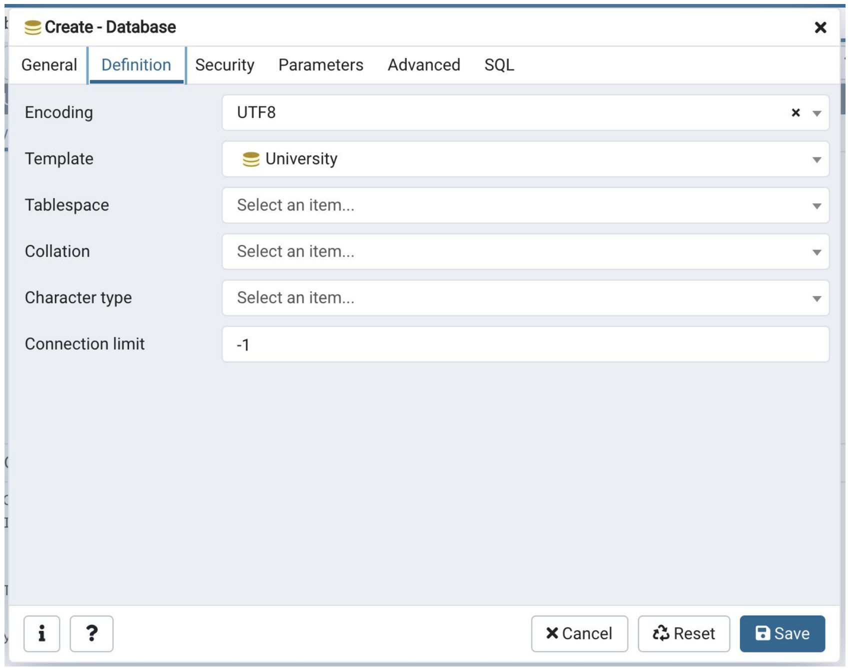
- Firstly, to create a new database:

  Databases -> Create -> Database

- To create a copy of the database, write the Database name (General tab) and Template (Definition tab).



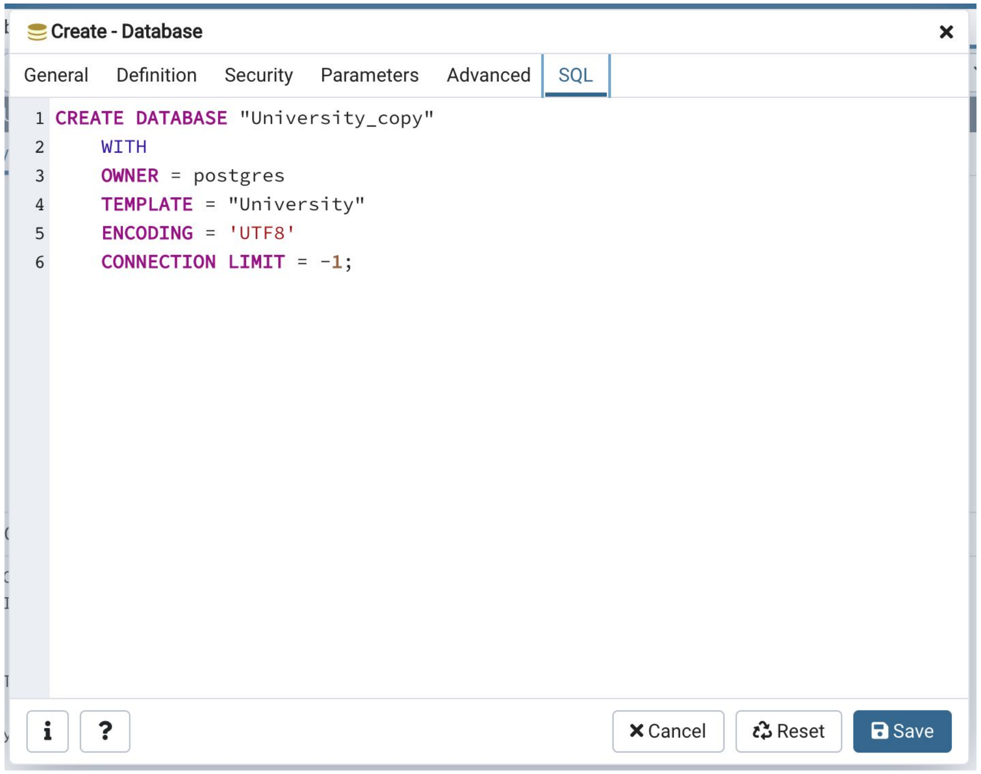
Database	University_copy
Owner	postgres
Comment	



Encoding	UTF8
Template	University
Tablespace	Select an item...
Collation	Select an item...
Character type	Select an item...
Connection limit	-1

Copy DB with pgAdmin

- Resulting SQL script (SQL tab):



```
CREATE DATABASE "University_copy"
    WITH
    OWNER = postgres
    TEMPLATE = "University"
    ENCODING = 'UTF8'
    CONNECTION LIMIT = -1;
```

Books

- Connolly, Thomas M. Database Systems: A Practical Approach to Design, Implementation, and Management / Thomas M. Connolly, Carolyn E. Begg.- United States of America: Pearson Education

- Garcia-Molina, H. Database system: The Complete Book / Hector Garcia-Molina.- United States of America: Pearson Prentice Hall

- Sharma, N. Database Fundamentals: A book for the community by the community / Neeraj Sharma, Liviu Perniu.- Canada

- www.postgresql.org

Protection of DBMS

LECTURE 11

SQL Injections

IITU, ALMATY

Today’s lecture

- Embedded SQL
- SQL injections

Embedded SQL

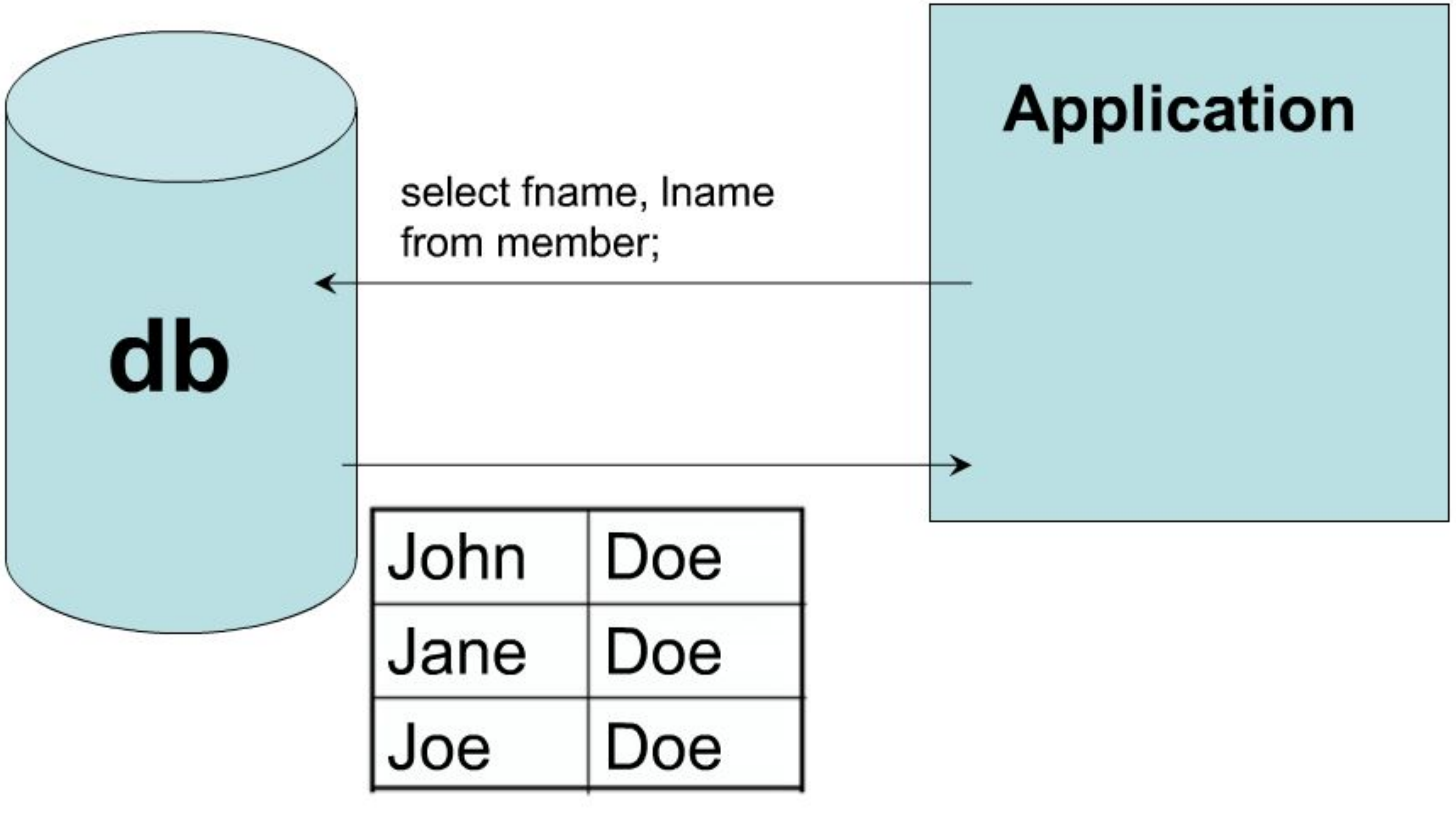
- **Embedded SQL** is a method of combining the computing power of a programming language and the database manipulation capabilities of SQL.

- Embedded SQL statements are SQL statements written inline with the program source code of the host language.

Embedded SQL

- We can write SQL statements in code written in a programming languages like Java, C++, and etc.
- The SQL statements are transferred to the database that executes the query.
- The results of the query are returned to the application.

Embedded SQL



John	Doe
Jane	Doe
Joe	Doe

Embedded SQL

- Formally, the process of placing an SQL statement within an application to query a database is known as **embedding** a SQL.

- The language in which we embed SQL is known as the **host** language.

- Most modern computer languages, including Java, C++, PHP, and Python, may serve as a host language to SQL.

Embedded SQL

- Embedded SQL comes in two flavors: **static** and **dynamic.**

- We are familiar with writing static SQL statements. These SQL statements are complete queries that can run inside a database. For example:

```
SELECT * FROM Students;
```

Embedded SQL

- Dynamic embedded SQL statements allow to place the value of program variables in queries.

- Suppose we have a variable with name *studentname* and value ‘John’. We can write a query to select all data from the Students table where the *fname* is the value of *studentname* :

```
string SQLQuery="SELECT * FROM Students
WHERE fname=' "+studentname+" ‘ “;
```

- Note, when the previous command is executed, the string SQLQuery becomes

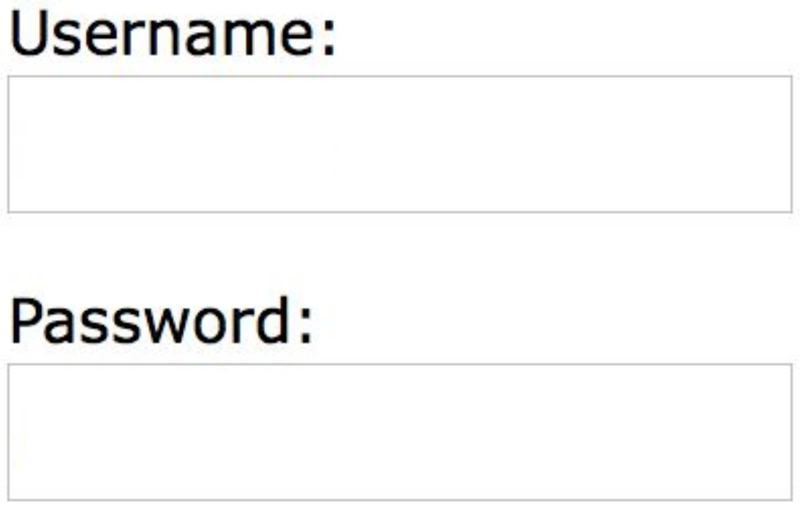
```
SELECT * FROM Students WHERE fname = 'John'
```

SQL Injection

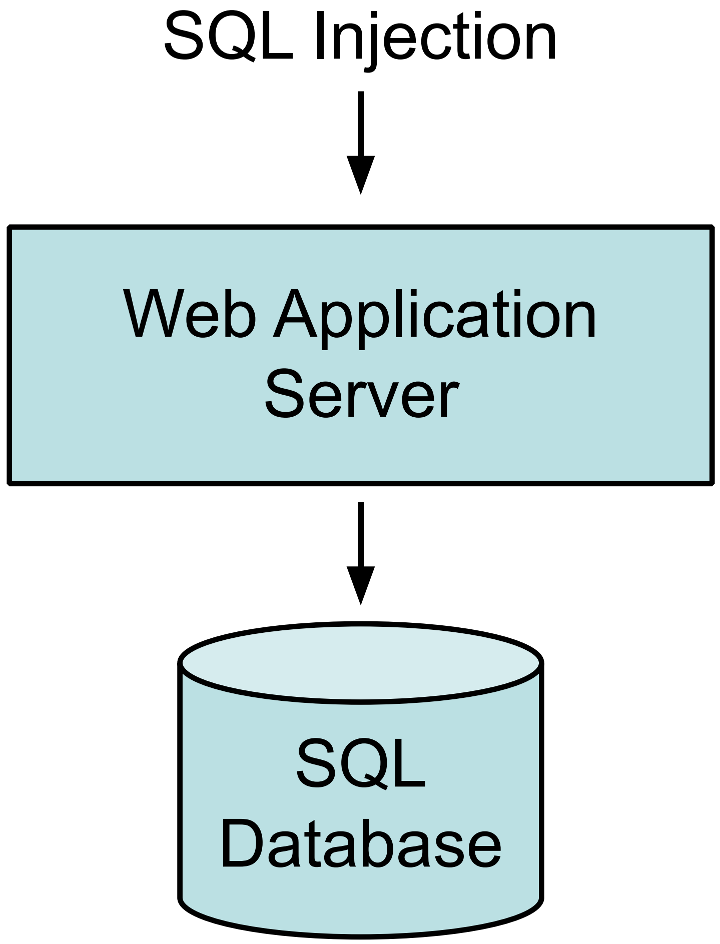
- **SQL injection** is a code injection technique that might destroy your database.
- **SQL injection** is one of the most common web hacking techniques.
- **SQL injection** is the placement of malicious code in SQL statements, via web page input.

SQL in Web Pages

- SQL injection usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you an SQL statement that you will **unknowingly** run on your database.



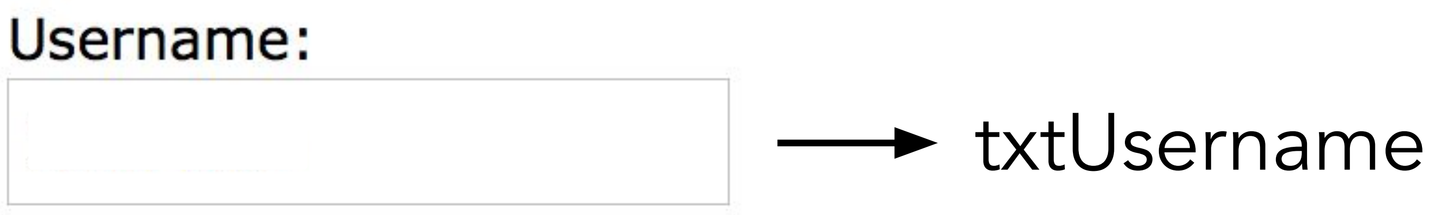
Anatomy of an SQL attack



SQL in Web Pages

- The following example creates a SELECT statement by adding a variable (txtUserId) to a select string. The variable is fetched from user input:

```
sql = "SELECT * FROM Users WHERE Username =
‘ " + txtUsername + ” ’ ";
```

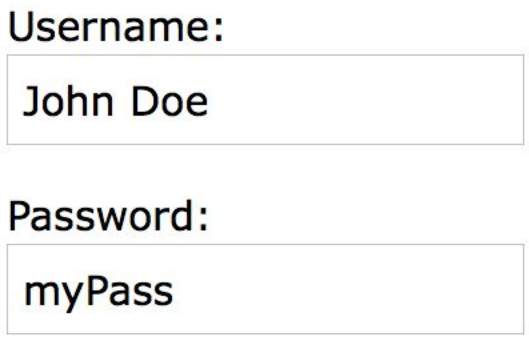


SQL in Web Pages

- A similar query is generally used from the web application in order to authenticate a user.
- If the query returns a value, it means that inside the database a user with that set of credentials exists, then the user is allowed to login to the system, otherwise access is denied.
- The values of the input fields are generally obtained from the user through a web form.

SQL in Web Pages

- Here is an example of a user login on a web site:



Username:

John Doe

Password:

myPass

```
sql = "SELECT * FROM Users WHERE Name = ' " +
            uName + " ' AND Pass = ' " + uPass + " ' ”
```

- Result:

```
SELECT * FROM Users
WHERE Name ='John Doe' AND Pass ='myPass'
```

SQL injection examples

SQL Injection Based on:

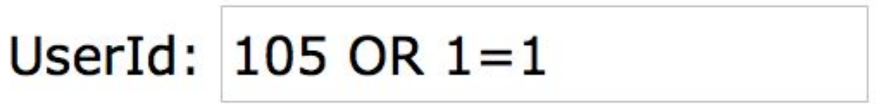
- 1=1 is Always True
- ' ' = ' ' is Always True
- Batched SQL Statements

1=1 is Always True

- The original purpose of the code is to create an SQL statement to select a user with a given user id:

  SELECT * FROM Users WHERE UserId = …

- If there is nothing to prevent a user from entering "wrong" input, the user can enter some input like this:



UserId: 105 OR 1=1

- Then, the SQL statement will look like this:

  SELECT * FROM Users WHERE UserId = 105 OR 1=1

1=1 is Always True

- The following SQL is valid and will return ALL rows from the "Users" table, since **1=1 is always TRUE**.

```
SELECT *
FROM Users
WHERE UserId = 105 OR 1=1
```

- In this way the system has authenticated the user without knowing the username and password.In some systems the first row of a user table would be an administrator user. This may be the profile returned in some cases.

1=1 is Always True

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1
```

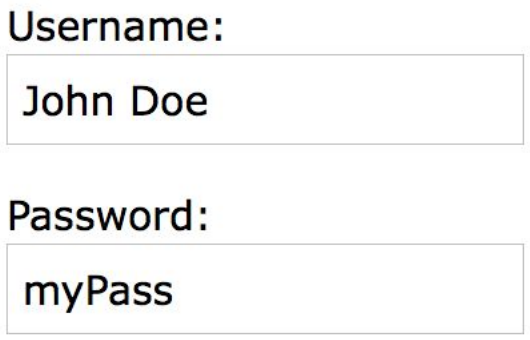
- The SQL statement above is much the same as this:

```
SELECT UserId, Name, Password
FROM Users
WHERE UserId = 105 OR 1=1
```

- A hacker might get access to all the user names and passwords in a database, by simply inserting ”105 OR 1=1” into the input field.

' ' = ' ' is Always True

- Here is an example of a user login on a web site:


Username:

John Doe

Password:

myPass

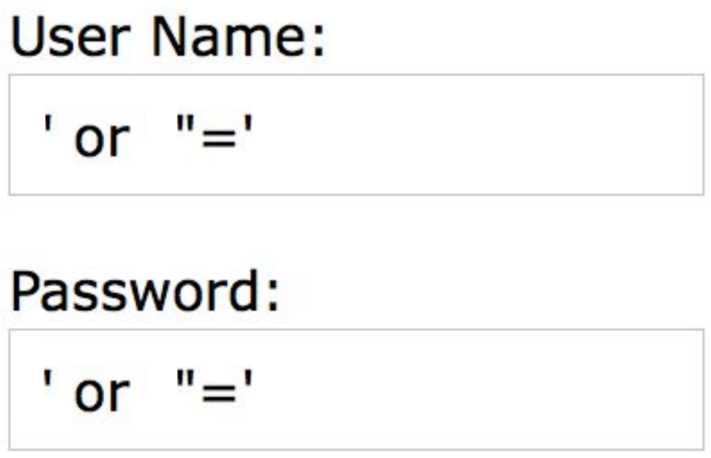
```
sql = "SELECT * FROM Users WHERE Name = ' " +
          uName + " ' AND Pass = ' " + uPass + " ' ”
```

- Result:

```
SELECT * FROM Users
WHERE Name ='John Doe' AND Pass ='myPass'
```

' ' = ' ' is Always True

- A hacker might get access to user names and passwords in a database by simply inserting ' OR ' ' = ' into the user name or password text box:



User Name:

' or ''='

Password:

' or ''='

' ' = ' ' is Always True

- The code at the server will create a valid SQL statement like this:

```
SELECT *
FROM Users
WHERE Name = ' ' OR ' ' = ' '
        AND Pass = ' ' OR ' ' = ' '
```

- The SQL above is valid and will return all rows from the "Users" table, since ' ' = ' ' is always TRUE.

Batched SQL Statements

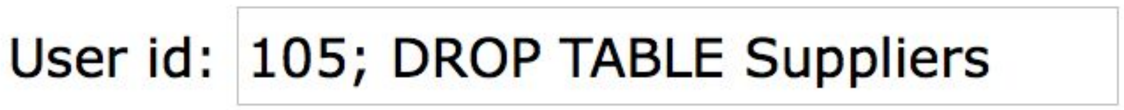
- DBMSs support batched SQL statement.
- A batch of SQL statements is a group of two or more SQL statements, separated by semicolons.
- The SQL statement below will return all rows from the "Users" table, then delete the "Suppliers" table:

```
SELECT * FROM Users; DROP TABLE Suppliers
```

Batched SQL Statements

```
sql = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

- The following input:



User id: 105; DROP TABLE Suppliers

- The valid SQL statement would look like this:

```
SELECT * FROM Users
WHERE UserId = 105;
DROP TABLE Suppliers;
```

Reaction

You've just detected a SQL injection attack.

Your actions:

1. As quickly as possible disable access and prevent the attacker from doing anything else. Their next injected SQL statement could be a DROP TABLE. Do as much as is needed to stop it right away - don't worry about fixing the hole yet.
2. Once things are disabled, start patching up the holes. Check your logs carefully to see if this was an isolated event, or if the hole had been used before.
3. Learn why this happened in the first place.

Detection

- If someone were to start a SQL injection attack against your site right now, would you even know?
- Fortunately, SQL injection attacks almost always generate some SQL errors as the attacker tries to work around your SQL.
- This is the number one way to detect an attack while it is happening.

Detection

- In addition to pure SQL errors, permission errors often occur as well, as the attacker tries to do something not allowed by the current database user.
- Remember to never treat a strange error as an uninteresting isolated event, or assume that it is probably one of your developers. Follow up on everything.

Detection

- Sometimes, when the attacker is very good, no SQL errors are generated, and the problems have to be detected in other ways.

- One way is to scan for common SQL injection items. In most cases, attacker access to your database is fairly limited without knowing the names of your tables, columns, functions, and views, so one thing to look for is references to system tables and system views.

Prevention

Preventing SQL injection is mostly a matter of following some standard software development practices:

- Never assume any database input is safe
- Be proactive in looking for problems
- Use the least privileges possible
- Keep your software up to date
- Teach people about SQL injection
- More than one set of eyes

Conclusion

- If you take a user input through a webpage and insert it into a SQL database, there is a chance that you have left yourself wide open for a security issue known as the **SQL Injection**.
- Injection usually occurs when you ask a user for input, like their name and instead of a name they give you a SQL statement that you will unknowingly run on your database. Never trust user provided data, process this data only after validation.
- A successful SQL injection can read, modify sensitive data from the database, and can also delete data from a database. It also enables the hacker to perform administrative operations on the database such as dropping databases.

Books

- Connolly, Thomas M. Database Systems: A Practical Approach to Design, Implementation, and Management / Thomas M. Connolly, Carolyn E. Begg.- United States of America: Pearson Education

- Garcia-Molina, H. Database system: The Complete Book / Hector Garcia-Molina.- United States of America: Pearson Prentice Hall

- Sharma, N. Database Fundamentals: A book for the community by the community / Neeraj Sharma, Liviu Perniu.- Canada

- www.postgresql.org

Protection of DBMS

LECTURE 9

Database Activity Monitoring

IITU, ALMATY

Statistics Collector

- PostgreSQL's **statistics collector** is a subsystem that supports collection and reporting of information about server activity.
- Presently, the collector can count accesses to tables. It also tracks the total number of rows in each table, and and analyze actions for each table.
- PostgreSQL also supports reporting dynamic information about exactly what is going on in the system right now, such as the exact command currently being executed by other server processes, and which other connections exist in the system.

Viewing Statistics

- When using the statistics to monitor collected data, it is important to realize that the information does not update instantaneously.
- Each individual server process transmits new statistical counts to the collector just before going idle; so a query or transaction still in progress does not affect the displayed totals.
- Also, the collector itself emits a new report at most once per 500 milliseconds.

pg_stat_activity

- pg_stat_activity belongs to **Dynamic Statistics Views**.
- One row per server process, showing information related to the current activity of that process, such as state and current query.

pg_stat_activity

Column	Description
datid	OID of the database
datname	Name of the database
pid	Process ID
usesysid	OID of the user
usename	Name of the user
application_name	Name of the application that is connected
backend_start	Time when this process was started, i.e., when the client connected to the server (datatype - timestamp with time zone)
xact_start	Time when this process' current transaction was started, or null if no transaction is active. If the current query is the first of its transaction, this column is equal to the *query_start* column.
query_start	Time when the currently active query was started, or if *state* is not *active*, when the last query was started

pg_stat_activity (cont)

Column	Description
state_change	Time when the *state* was last changed
state	Current overall state. Possible values are: • active: The backend is executing a query. • idle: The backend is waiting for a new client command. • idle in transaction: The backend is in a transaction, but is not currently executing a query. • idle in transaction (aborted): This state is similar to idle in transaction, except one of the statements in the transaction caused an error.
query	Text of the most recent query. If *state* is *active* this field shows the currently executing query. In all other states, it shows the last query that was executed.
…	…

pg_stat_database

- pg_stat_database belongs to **Collected Statistics Views**.
- One row per database, showing database-wide statistics.

pg_stat_database

Column	Description
datid	OID of a database
datname	Name of this database
xact_commit	Number of transactions in this database that have been committed
xact_rollback	Number of transactions in this database that have been rolled back
tup_returned	Number of rows returned by queries in this database
tup_inserted	Number of rows inserted by queries in this database
tup_updated	Number of rows updated by queries in this database
tup_deleted	Number of rows deleted by queries in this database
…	…

pg_stat_all_tables

- pg_stat_all_tables belongs to **Collected Statistics Views**.
- One row for each table in the current database, showing statistics about accesses to that specific table.

pg_stat_all_tables

Column	Description
relid	OID of a table
relname	Name of this table
n_tup_ins	Number of rows inserted
n_tup_upd	Number of rows updated
n_tup_del	Number of rows deleted
…	…

pg_stat_statements

- The **pg_stat_statements** module provides a means for tracking execution statistics of all SQL statements executed by a server.
- The statistics gathered by the module are made available via a view named pg_stat_statements.
- This view contains one row for each distinct database ID, user ID and query ID.

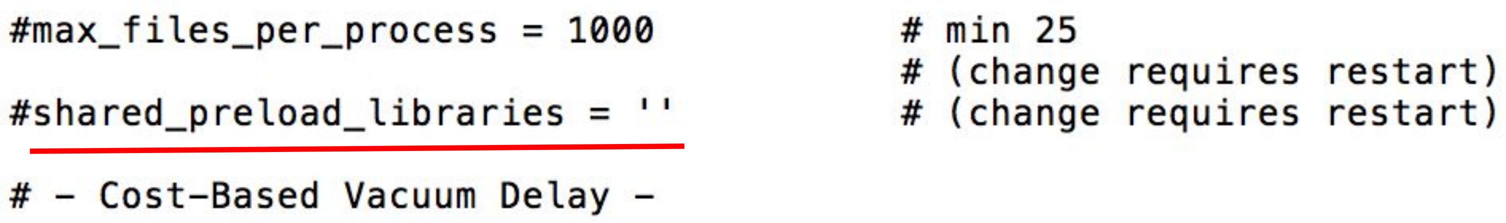
pg_stat_statements

- The module must be loaded by adding **pg_stat_statements** to **shared_preload_libraries** in **postgresql.conf**, because it requires additional shared memory. This means that a server restart is needed to add or remove the module.
- When pg_stat_statements is loaded, it tracks statistics across all databases of the server. To access and manipulate these statistics, the module provides a view, **pg_stat_statements**. These are not available globally but can be enabled for a specific database with **CREATE EXTENSION pg_stat_statements**.

pg_stat_statements: step 1

postgresql.conf :

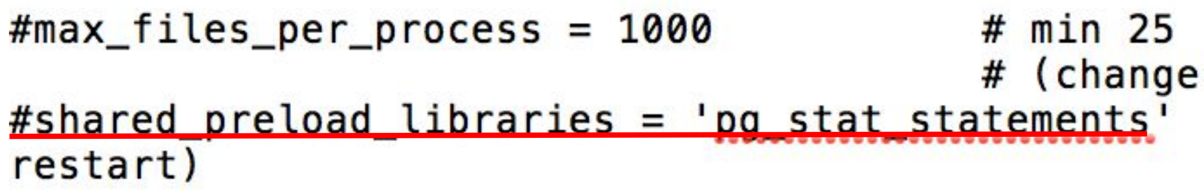
BEFORE



```
#max_files_per_process = 1000           # min 25
                                        # (change requires restart)
#shared_preload_libraries = ''          # (change requires restart)

# - Cost-Based Vacuum Delay -
```

AFTER



```
#max_files_per_process = 1000           # min 25
                                        # (change
#shared_preload_libraries = 'pg_stat_statements'
restart)
```

pg_stat_statements: step 2

- Then you need to restart the database server. After that in the database, run the following statement:

```
CREATE EXTENSION pg_stat_statements;
```

- After that, in the database where you run this statement, the view pg_stat_statements will appear:

```
SELECT *
FROM pg_stat_statements
```

pg_stat_statements

Column	Description
userid	OID of user who executed the statement (references to pg_authid.oid)
dbid	OID of database in which the statement was executed (references to pg_database.oid)
queryid	Internal hash code, computed from the statement's parse tree
query	Text of a representative statement
total_time	Total time spent in the statement, in milliseconds
rows	Total number of rows retrieved or affected by the statement
…	…

- For security reasons, non-superusers are not allowed to see the SQL text or queryid of queries executed by other users.

pg_stat_get_activity()

- **pg_stat_get_activity(integer)** returns a record of information with the specified PID, or one record for each active backend in the system if NULL is specified.
- The fields returned are a subset of those in the pg_stat_activity view.

Server Signaling Functions

- The question now is this: once you have found bad queries, how can you actually get rid of them?
- PostgreSQL provides two functions to take care of these things: **pg_cancel_backend** and **pg_terminate_backend**.

pg_cancel_backend()

- The **pg_cancel_backend** function will terminate the query but will leave the connection in place.

pg_cancel_backend(pid)

- The function returns true if successful and false otherwise.
- The process ID of an active backend can be found from the pid column of the pg_stat_activity view.

pg_terminate_backend()

- The **pg_terminate_backend** function is a bit more radical and will kill the entire database connection along with the query.

```
pg_terminate_backend(pid)
```

- The function returns true if successful and false otherwise.
- The process ID of an active backend can be found from the pid column of the pg_stat_activity view.

Books

- Connolly, Thomas M. Database Systems: A Practical Approach to Design, Implementation, and Management / Thomas M. Connolly, Carolyn E. Begg.- United States of America: Pearson Education

- Garcia-Molina, H. Database system: The Complete Book / Hector Garcia-Molina.- United States of America: Pearson Prentice Hall

- Sharma, N. Database Fundamentals: A book for the community by the community / Neeraj Sharma, Liviu Perniu.- Canada

- www.postgresql.org

Protection of DBMS

LECTURE 8

System Catalogs and Information Functions

IITU, ALMATY

Systems Catalogs

- The **system catalogs** are the place where a RDMS stores schema metadata, such as information about tables and columns, and internal bookkeeping information. PostgreSQL's system catalogs are regular tables.
- Normally, one should not change the system catalogs by hand, there are normally SQL commands to do that.
- For example, CREATE DATABASE inserts a row into the pg_database catalog — and actually creates the database on disk.

Systems Catalogs

- Syntax:

```
SELECT attribute_name / *
FROM catalog_name
[WHERE ….];
```

pg_roles

- The view pg_roles provides access to information about database roles:

Name	*Description*
rolname	*Role name*
rolsuper	*Role has superuser privileges*
rolcreaterole	*Role can create more roles*
rolcreatedb	*Role can create databases*
rolcanlogin	*Role can log in*
rolvaliduntil	*Password expiry time (only used for password authentication); null if no expiration*
…	…

List of users

- To show all users:

```
SELECT rolname
FROM pg_roles;
```

pg_authid / pg_roles

- The catalog **pg_authid** contains information about database authorization identifiers (roles).
- Since this catalog contains passwords, it must not be publicly readable. **pg_roles** is a publicly readable view on pg_authid that blanks out the password field.

Column Name	Description
oid	Row identifier (hidden attribute; must be explicitly selected)
rolname	Role name
…	…

pg_auth_members

- The catalog **pg_auth_members** shows the membership relations between roles.

Column Name	Description
roleid	ID of a role that has a member
member	ID of a role that is a member of roleid
grantor	ID of the role that granted this membership
admin_option	True if member can grant membership in roleid to others

role_table_grants / table_privileges

- The view **role_table_grants** identifies all privileges granted on tables or views where the grantor or grantee is a currently enabled role.
- Also information can be found under **table_privileges**.
- The only effective difference between this view and table_privileges is that this view omits tables that have been made accessible to the current user by way of a grant to PUBLIC.

Column Name	Description
grantor	Name of the role that granted the privilege
grantee	Name of the role that the privilege was granted to
table_name	Name of the table
privilege_type	Type of the privilege (SELECT, INSERT, UPDATE, DELETE, etc.)
…	…

role_table_grants example

- By default, the information schema is not in the schema search path, so you need to access all objects in it through qualified names:

```
SELECT *
FROM
information_schema.role_table_grants;
```

pg_database

- The catalog pg_database stores information about the available databases. Databases are created with the CREATE DATABASE command.
- Unlike most system catalogs, pg_database is shared across all databases of a cluster: there is only one copy of pg_database per cluster, not one per database.

Attribute name	Description
oid	Row identifier (hidden attribute; must be explicitly selected)
datname	Database name
datdba	Owner of the database, usually the user who created it (reference to pg_authid.oid)
datistemplate	If true, then this database can be cloned by any user with CREATEDB privileges; if false, then only superusers or the owner of the database can clone it.
datconnlimit	Sets maximum number of concurrent connections that can be made to this database. -1 means no limit.
…	…

pg_class

- The catalog pg_class catalogs tables and most everything else that has columns or is otherwise similar to a table. This includes views, materialized views and etc.

Attribute name	Description
oid	Row identifier (hidden attribute; must be explicitly selected)
relname	Name of the table, view, etc.
relowner	Owner of the relation (reference to pg_authid.oid)
relkind	r = ordinary table, v = view, m = materialized view
relnatts	Number of user columns in the relation (system columns not counted). There must be this many corresponding entries in pg_attribute.
relhaspkey	True if the table has (or once had) a primary key
relhastriggers	True if table has (or once had) triggers; see pg_trigger catalog
…	…

pg_attribute

- The catalog pg_attribute stores information about table columns. There will be exactly one pg_attribute row for every column in every table in the database (There will also be attribute entries for all objects that have pg_class entries).

Attribute name	Description
attrelid	The table this column belongs to (reference to pg_class.oid)
attname	The column name
atttypid	The data type of this column (reference to pg_type.oid)
attnum	The number of the column. Ordinary columns are numbered from 1 up.
attnotnull	This represents a not-null constraint.
atthasdef	This column has a default value, in which case there will be a corresponding entry in the pg_attrdef catalog that actually defines the value.
…	…

pg_constraint

- The catalog pg_constraint stores check, primary key, unique, foreign key. Not-null constraints are represented in the pg_attribute catalog, not here.

Attribute name	Description
oid	Row identifier (hidden attribute; must be explicitly selected)
conname	Constraint name (not necessarily unique!)
contype	c = check constraint, f = foreign key constraint, p = primary key constraint, u = unique constraint, t = constraint trigger.
conrelid	The table this constraint is on (reference to pg_class.oid)
confrelid	If a foreign key, the referenced table; else 0 (reference to pg_class.oid)
conkey	If a table constraint (including foreign keys, but not constraint triggers), list of the constrained columns (reference to pg_attribute.attnum)
confkey	If a foreign key, list of the referenced columns (reference to pg_attribute.attnum)
consrc	If a check constraint, a human-readable representation of the expression
…	…

System Information Functions

- Following slides show functions that extract session and system information.
- In addition to the following functions, there are a number of functions related to the statistics system that also provide system information.

System Information Functions

- Syntax:

```
SELECT function_name();
```

- Note. Some functions (current_catalog, current_role, current_user, user) have special syntactic status in SQL: they must be called without trailing parentheses.

```
SELECT function_name;
```

Current Database

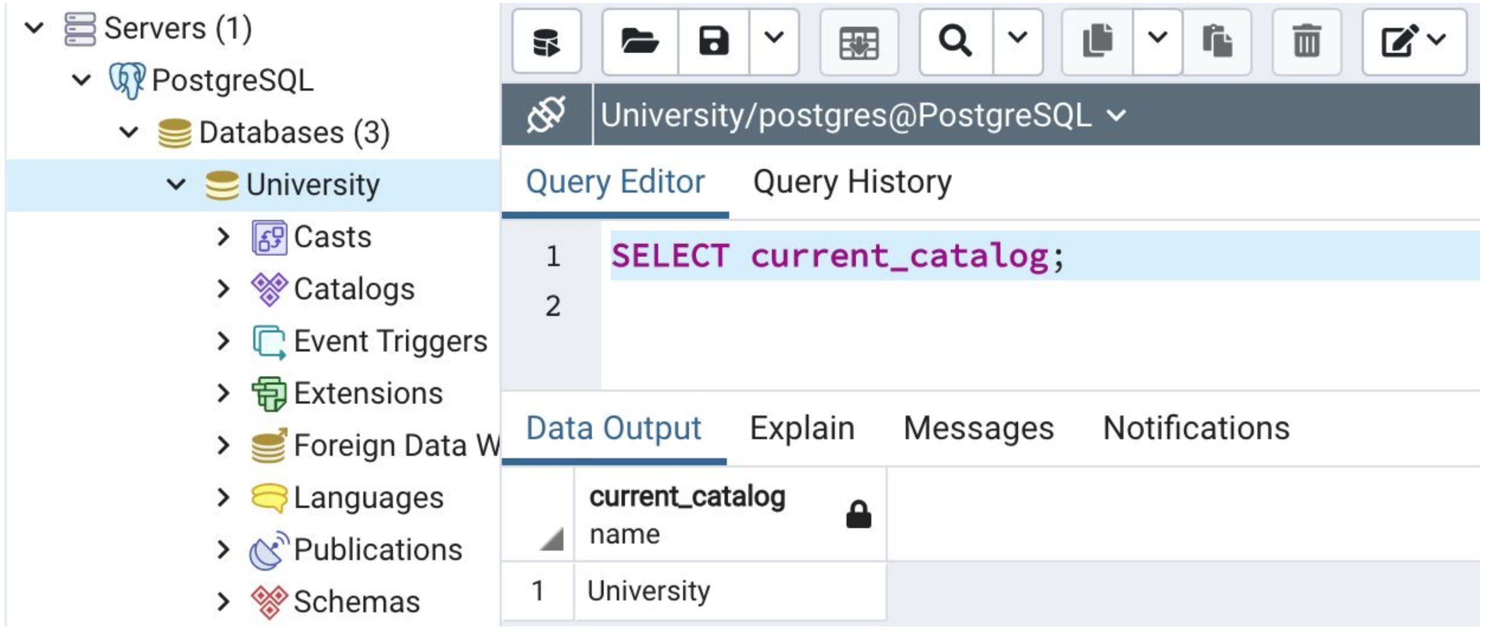
- Following functions return a name of the current database:

Name	Description
current_catalog	name of current database (called "catalog" in the SQL standard)
current_database()	name of current database

```
SELECT current_catalog;
SELECT current_database();
```

Current Database

SELECT current_catalog;



```
SELECT current_catalog;
```

```
  current_catalog
  name
1 University
```

Current User

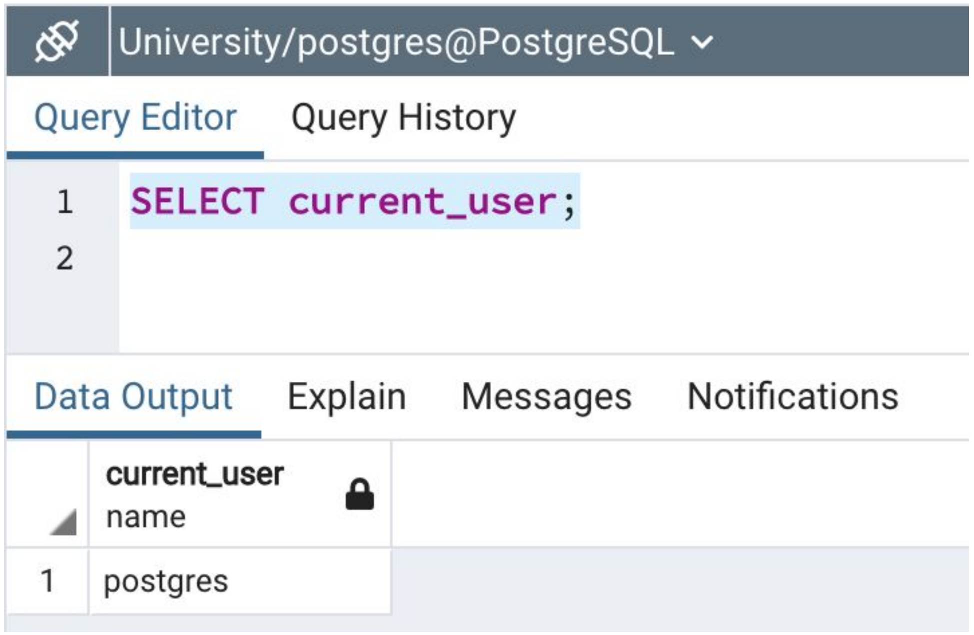
- Following functions return a name of the current user:

Name	Description
current_user	user name of current execution context
user	equivalent to current_user
current_role	equivalent to current_user
getpgusername()	equivalent to current_user

```
SELECT current_user;
SELECT user;
SELECT current_role;
SELECT getpgusername();
```

Current User

SELECT current_user;



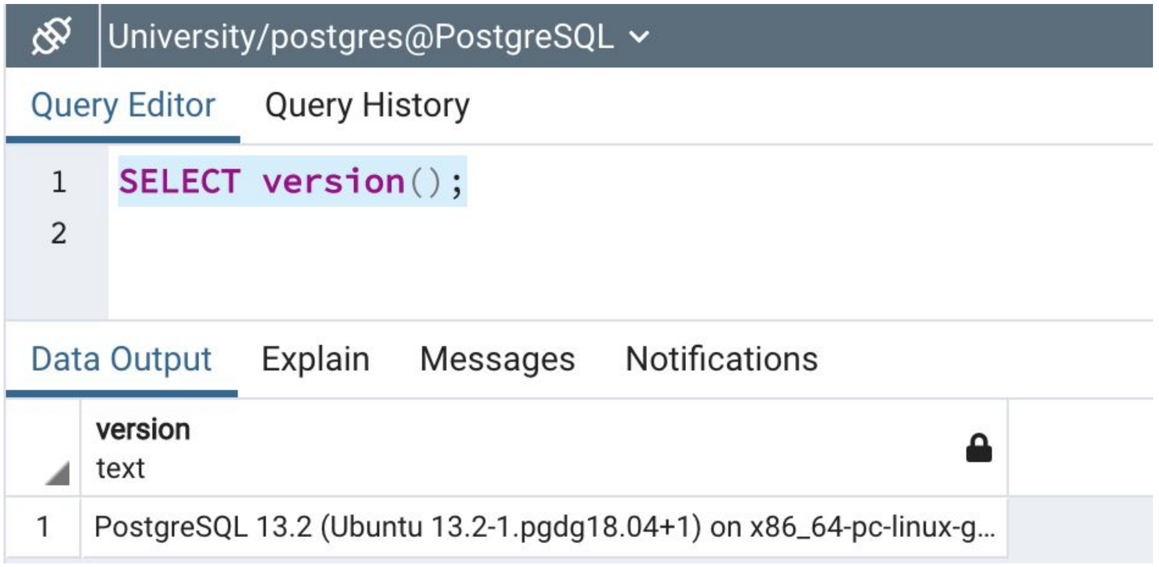
```
SELECT current_user;
```

```
  current_user
  name
1 postgres
```

Current Version

- **version()** function shows PostgreSQL version information:

SELECT version();



University/postgres@PostgreSQL

Query Editor | Query History

```
SELECT version();
```

Data Output | Explain | Messages | Notifications

```
   version
   text
1  PostgreSQL 13.2 (Ubuntu 13.2-1.pgdg18.04+1) on x86_64-pc-linux-g...
```

Access Privilege Inquiry Functions

Name	Description
has_any_column_privilege (user, table, privilege)	does user have privilege for any column of table
has_any_column_privilege (table, privilege)	does current user have privilege for any column of table
has_column_privilege (user, table, column, privilege)	does user have privilege for column
has_column_privilege (table, column, privilege)	does current user have privilege for column
has_table_privilege (user, table, privilege)	does user have privilege for table
has_table_privilege (table, privilege)	does current user have privilege for table

has_table_privilege

Function **has_table_privilege** checks whether a user can access a table in a particular way:

```
has_table_privilege(user, table, privilege)
```

- The **user** can be specified by name, by OID, public.
- The **table** can be specified by name or by OID.
- The desired access **privilege** type is specified by a text string (SELECT, INSERT, UPDATE, DELETE, etc).

```
SELECT has_table_privilege
('students', 'schedule', 'SELECT');
```

has_table_privilege

Function **has_table_privilege** checks whether a user can access a table in a particular way:

```
has_table_privilege(table, privilege)
```

- If the user argument is omitted current_user is assumed.

```
SELECT has_table_privilege
('schedule', 'SELECT');
```

has_table_privilege

- Optionally, WITH GRANT OPTION can be added to a privilege type to test whether the privilege is held with grant option.
- Also, multiple privilege types can be listed separated by commas, in which case the result will be true if any of the listed privileges is held.

```
SELECT has_table_privilege
('students', 'schedule',
'INSERT, SELECT WITH GRANT OPTION');
```

Books

- Connolly, Thomas M. Database Systems: A Practical Approach to Design, Implementation, and Management / Thomas M. Connolly, Carolyn E. Begg.- United States of America: Pearson Education

- Garcia-Molina, H. Database system: The Complete Book / Hector Garcia-Molina.- United States of America: Pearson Prentice Hall

- Sharma, N. Database Fundamentals: A book for the community by the community / Neeraj Sharma, Liviu Perniu.- Canada

- www.postgresql.org

Protection of DBMS

Control Structures

IITU, ALMATY

PL/pgSQL

- **PL/pgSQ**L (**Procedural Language/PostgreSQL**) is a procedural programming language supported by the PostgreSQL.
- PL/pgSQL, as a fully featured programming language, allows much more procedural control than SQL, including the ability to use loops and other control structures.
- **Control structures** are probably the most useful (and important) part of PL/pgSQL. With PL/pgSQL's control structures, you can manipulate PostgreSQL data in a very flexible and powerful way.

Conditionals

IF and CASE statements let you execute alternative commands based on certain conditions. PL/pgSQL has three forms of IF:

- IF ... THEN ... END IF
- IF ... THEN ... ELSE ... END IF
- IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF

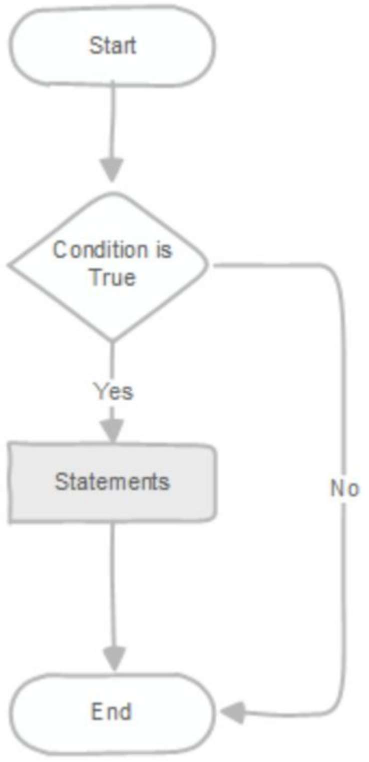
and two forms of CASE:

- CASE ... WHEN ... THEN ... ELSE ... END CASE
- CASE WHEN ... THEN ... ELSE ... END CASE

IF-THEN

- IF-THEN statements are the simplest form of IF. The statements between THEN and END IF will be executed if the condition is true. Otherwise, they are skipped.

```
IF boolean-expression THEN
    statements
END IF;
```



IF-THEN

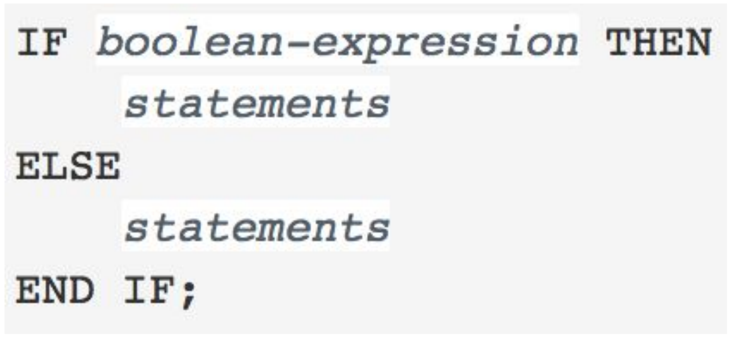
```
IF boolean-expression THEN
    statements
END IF;
```

- Example:

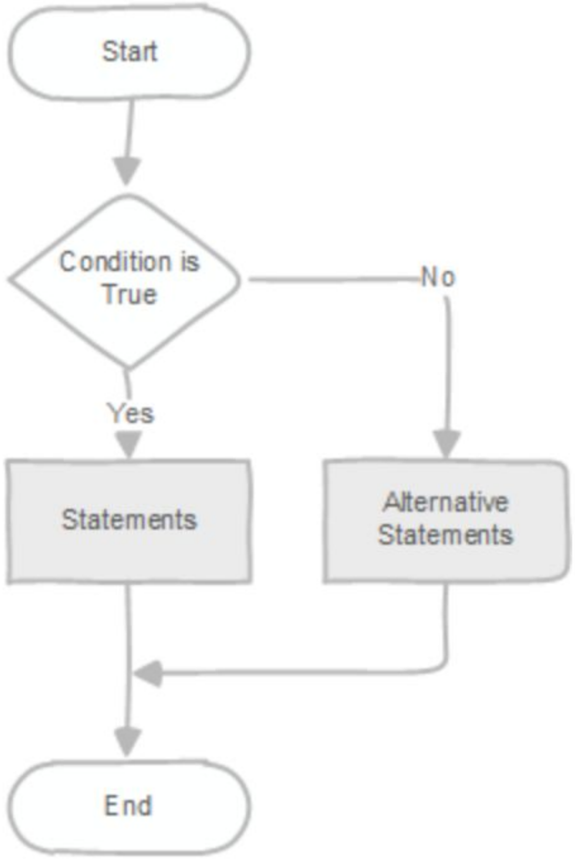
```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

IF-THEN-ELSE

- IF-THEN-ELSE statements add to IF-THEN by letting you specify an alternative set of statements that should be executed if the condition is not true.



```
IF boolean-expression THEN
    statements
ELSE
    statements
END IF;
```



IF-THEN-ELSE

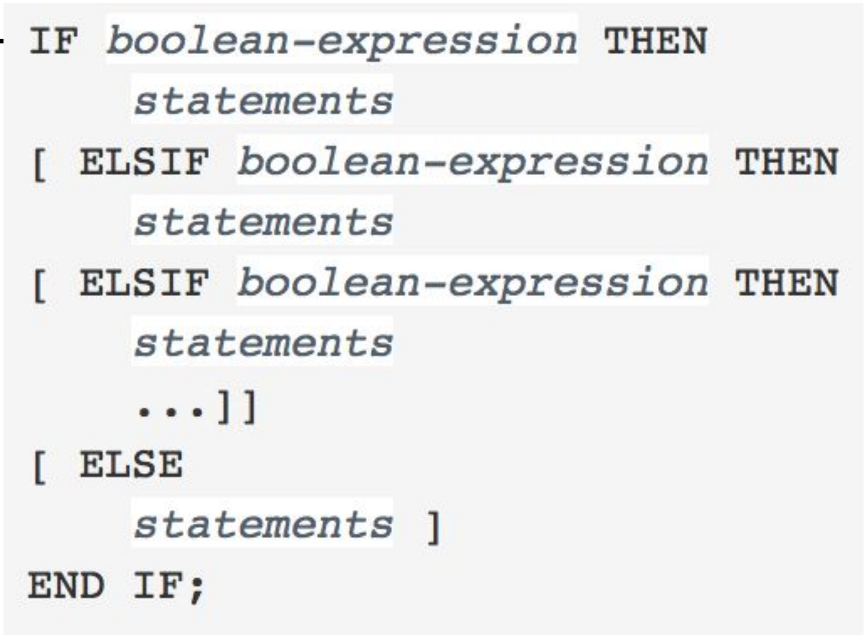
```
IF boolean-expression THEN
    statements
ELSE
    statements
END IF;
```

- Example:

```
IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

IF-THEN-ELSIF

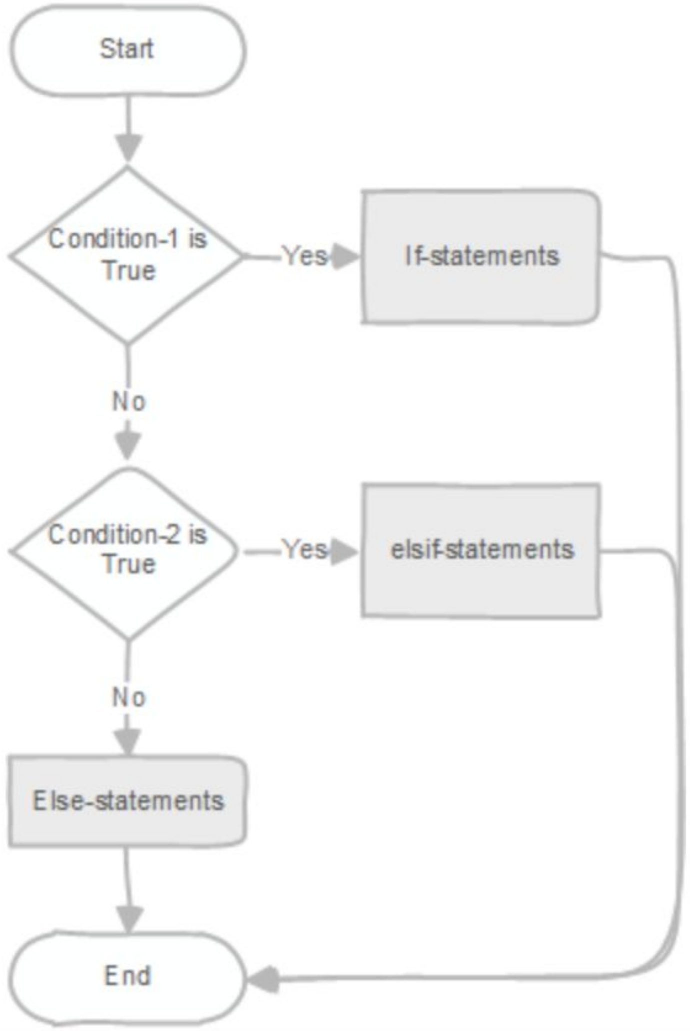
- Sometimes there are more than just two alternatives. IF-THEN-ELSIF provides a convenient method of checking several alternatives in turn. The IF conditions are tested successively until the first one that is true is found. Then the associated statement(s) are executed, after which control passes to the next statement after END IF. If none of the IF conditions is true, then the ELSE block (if any) is executed.
- The key word EL



```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;
```

IF-THEN-ELSIF

```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;
```



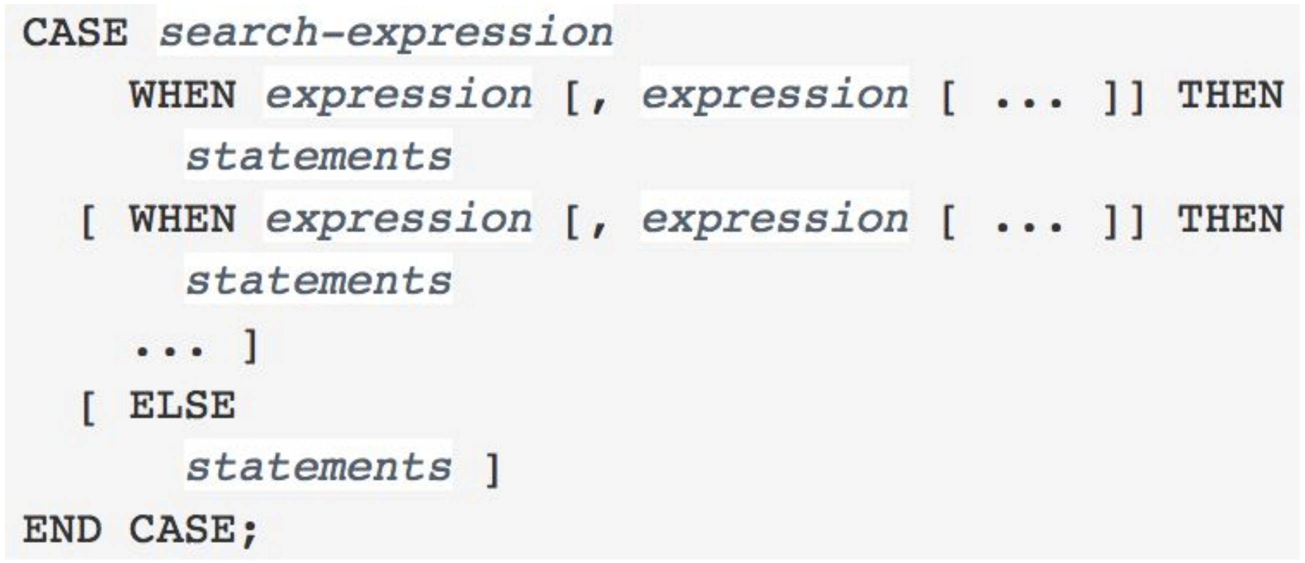
IF-THEN-ELSIF

- Example:

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- hmm, the only other possibility is that number is null
    result := 'NULL';
END IF;
```

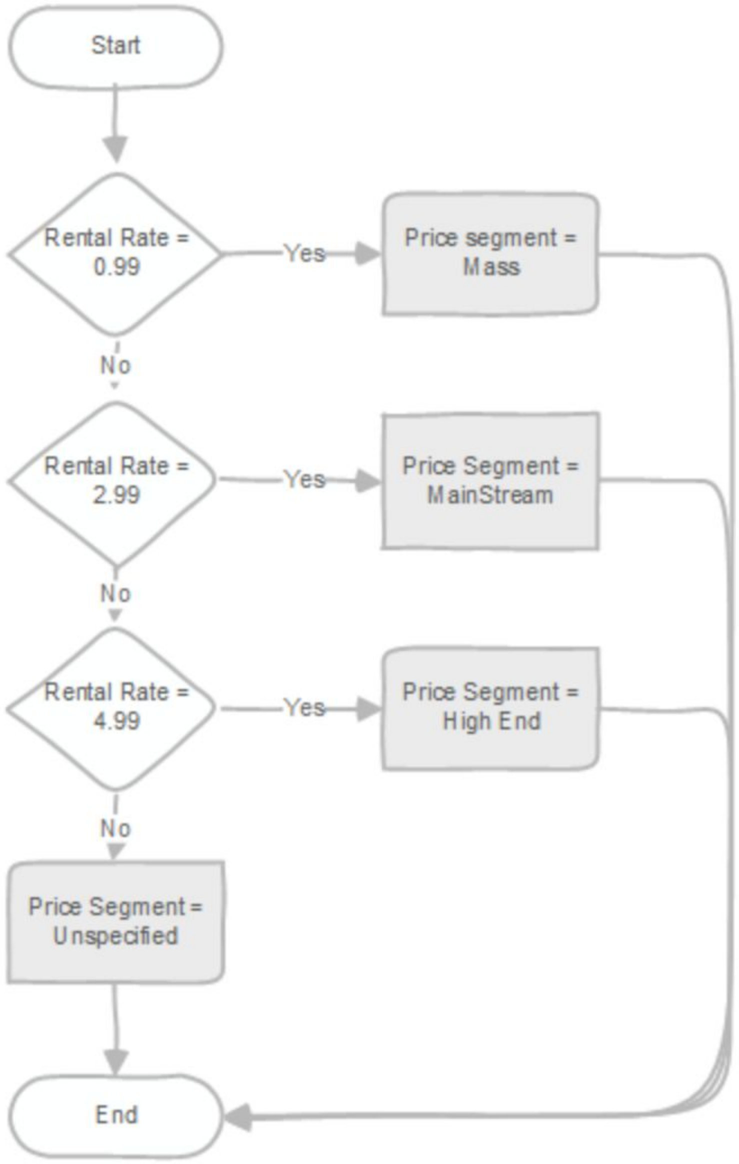
Simple CASE

- The simple form of CASE provides conditional execution based on equality of operands. The *search-expression* is evaluated (once) and successively compared to each *expression* in the WHEN clauses. If a match is found, then the corresponding *statements* are executed, and then control passes to the next statement after END CASE.



```
CASE search-expression
    WHEN expression [, expression [ ... ]] THEN
      statements
  [ WHEN expression [, expression [ ... ]] THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

Simple CASE


Start
Rental Rate = 0.99
Yes
Price segment = Mass
No
Rental Rate = 2.99
Yes
Price Segment = MainStream
No
Rental Rate = 4.99
Yes
Price Segment = High End
No
Price Segment = Unspecified
End

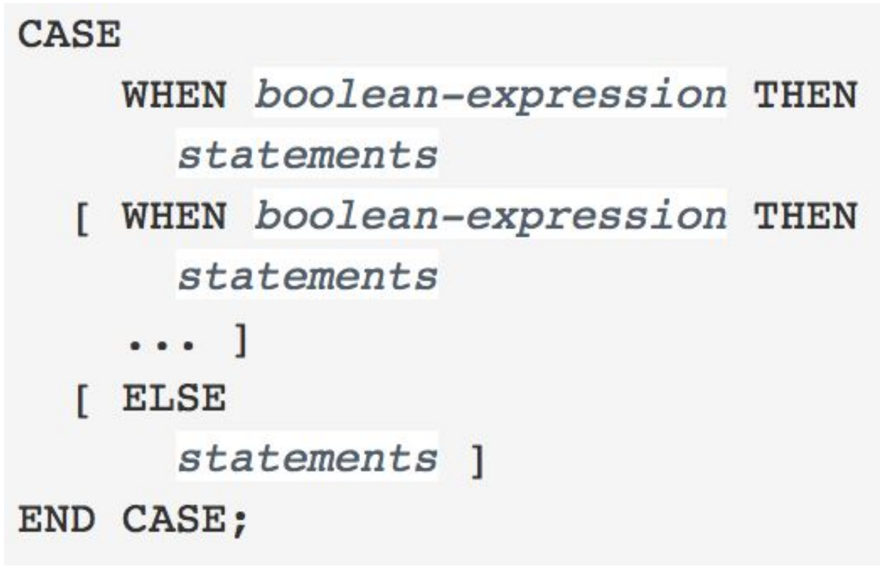
Simple CASE

- Example:

```
CASE x
    WHEN 1, 2 THEN
        msg := 'one or two';
    ELSE
        msg := 'other value than one or two';
END CASE;
```

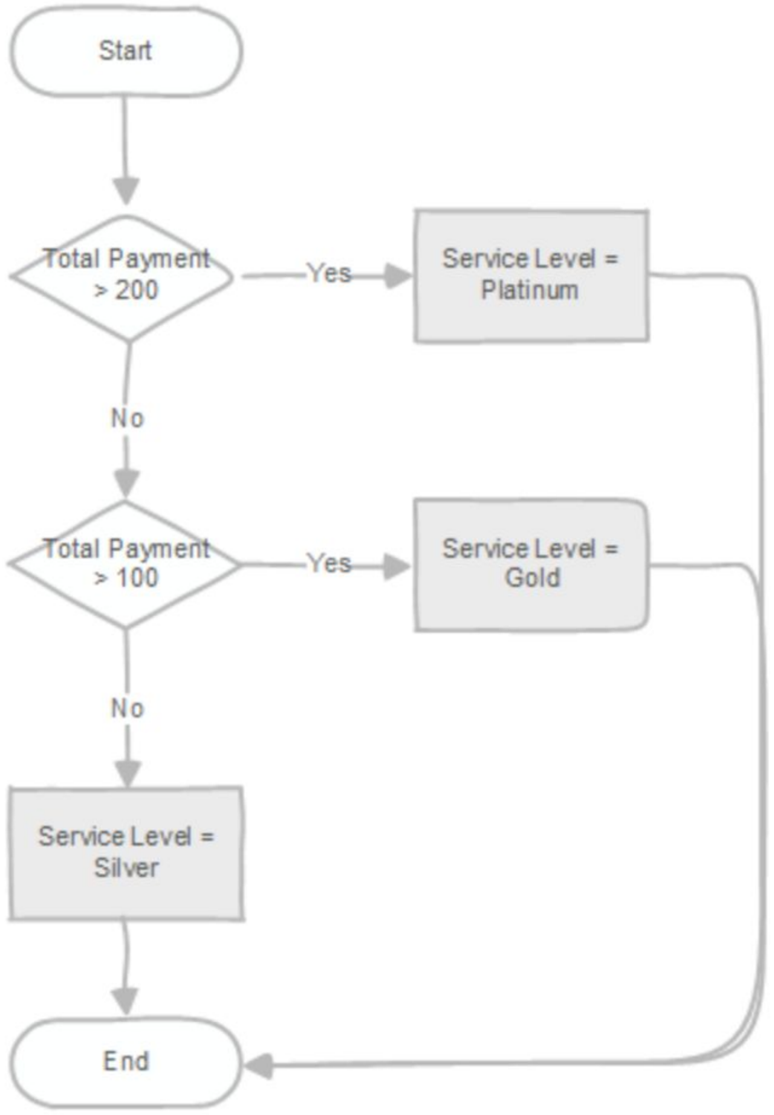
Searched CASE

- The searched form of CASE provides conditional execution based on truth of Boolean expressions. Each WHEN clause's *boolean-expression* is evaluated in turn, until one is found that yields true. Then the corresponding *statements* are executed, and then control passes to the next statement after END CASE.



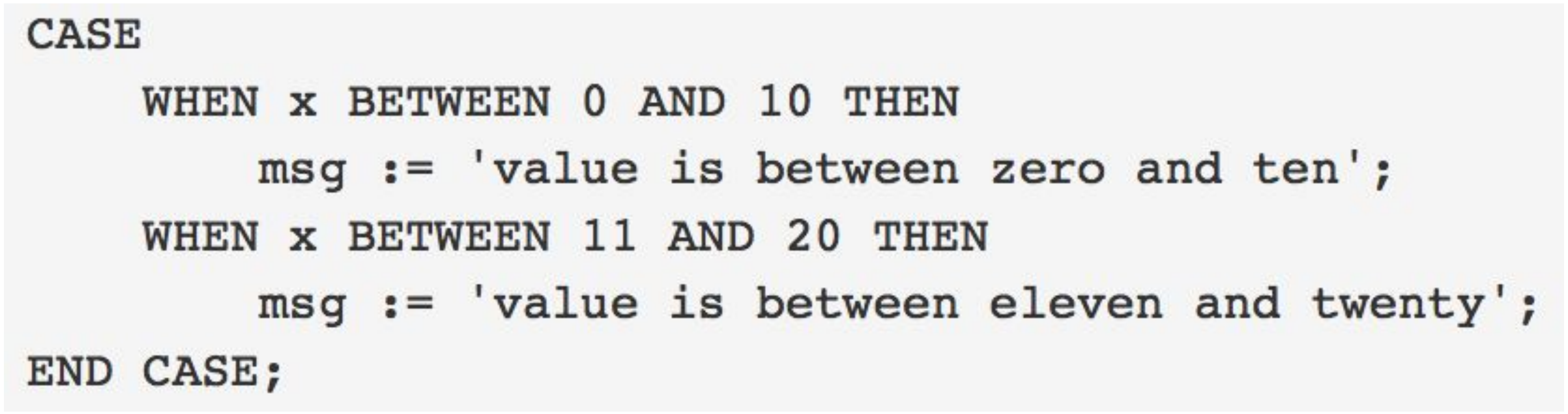
```
CASE
    WHEN boolean-expression THEN
      statements
  [ WHEN boolean-expression THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

Searched CASE



Searched CASE

- Example:



```
CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'value is between zero and ten';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'value is between eleven and twenty';
END CASE;
```

Loops

PostgreSQL provides three loop statements:

- LOOP
- WHILE loop
- FOR loop

LOOP

- Sometimes, you need to execute a block of statements repeatedly until a condition becomes true. To do this, you use the PL/pgSQL **LOOP** statement.
- Syntax:

```
<<label>>
LOOP
   Statements;
   EXIT [<<label>>] WHEN condition;
END LOOP;
```

LOOP

- The **LOOP** statement (unconditional loop) executes the statements until the condition in the EXIT statement evaluates to true.
- Note that the condition specified after the WHEN keyword in the EXIT statement is a Boolean expression that evaluates to true or false.
- Loop statements can be nested. A LOOP statement is placed inside another LOOP statement is known as a nested loop. In this case, you need to the loop label to specify explicitly which loop you want to terminate in the EXIT statement.

Examples

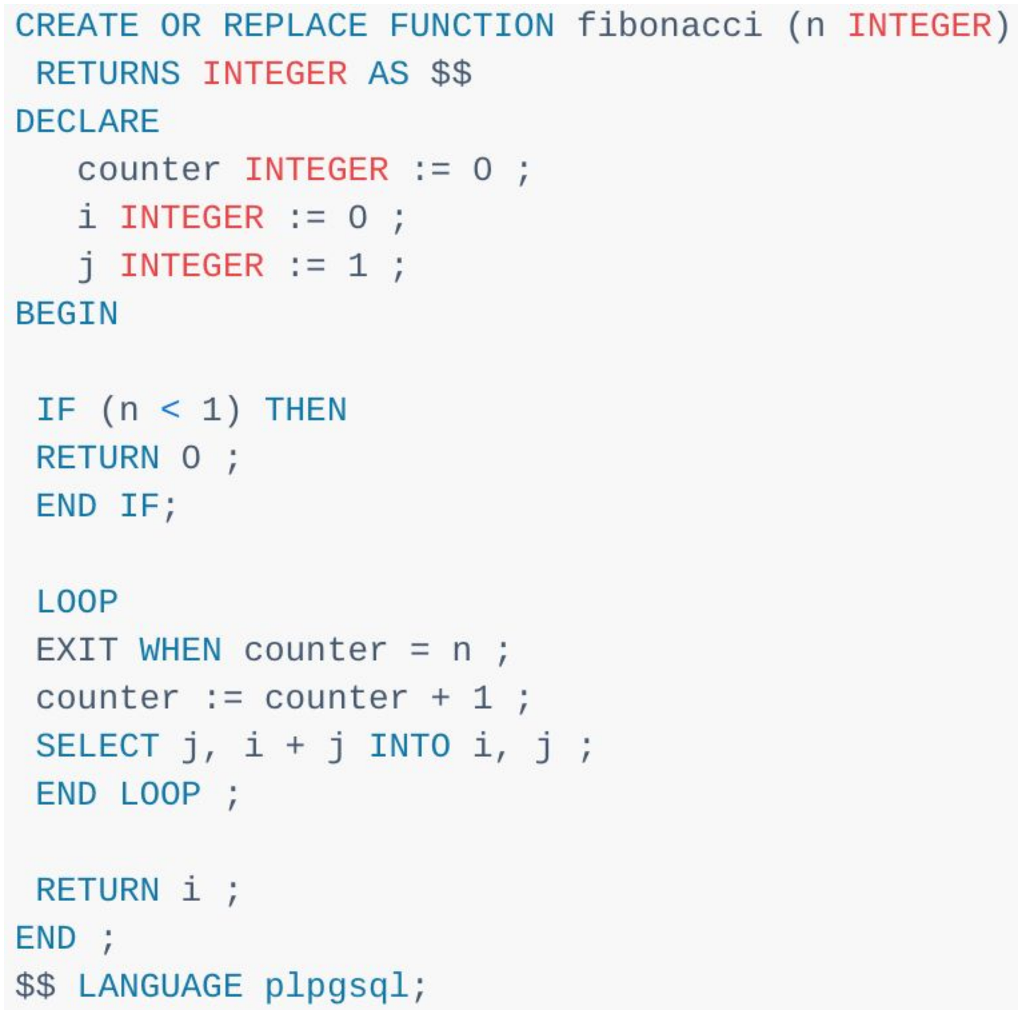
```
LOOP
    -- some computations
    IF count > 0 THEN
        EXIT;  -- exit loop
    END IF;
END LOOP;

LOOP
    -- some computations
    EXIT WHEN count > 0;  -- same result as previous example
END LOOP;

<<ablock>>
BEGIN
    -- some computations
    IF stocks > 100000 THEN
        EXIT ablock;  -- causes exit from the BEGIN block
    END IF;
    -- computations here will be skipped when stocks > 100000
END;
```

Example (Fibonacci sequence)

- In this example, we will use the LOOP statement to develop a function that returns the nth Fibonacci sequence number.



```
CREATE OR REPLACE FUNCTION fibonacci (n INTEGER)
 RETURNS INTEGER AS $$
DECLARE
   counter INTEGER := 0 ;
   i INTEGER := 0 ;
   j INTEGER := 1 ;
BEGIN

 IF (n < 1) THEN
 RETURN 0 ;
 END IF;

 LOOP
 EXIT WHEN counter = n ;
 counter := counter + 1 ;
 SELECT j, i + j INTO i, j ;
 END LOOP ;

 RETURN i ;
END ;
$$ LANGUAGE plpgsql;
```

Example (Fibonacci sequence)

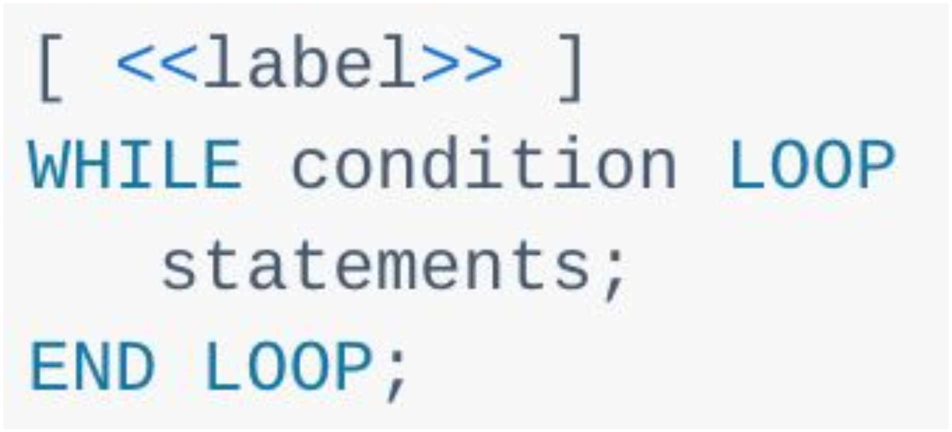
- The Fibonacci function accepts an integer and returns the nth Fibonacci number.
- By definition, Fibonacci numbers are the sequence of integers starting with 0 and 1, and each subsequent number is the product the previous two numbers, for example, 1, 1, 2 (1+1), 3 (2+1), 5 (3 +2), 8 (5+3), …
- In the declaration section, the counter variable is initialized to zero (0). Inside the loop, when counter equals n, the loop exits. The statement:

```
SELECT j, i + j INTO i, j ;
```

swaps i and j at the same time without using a temporary variable.

WHILE loop

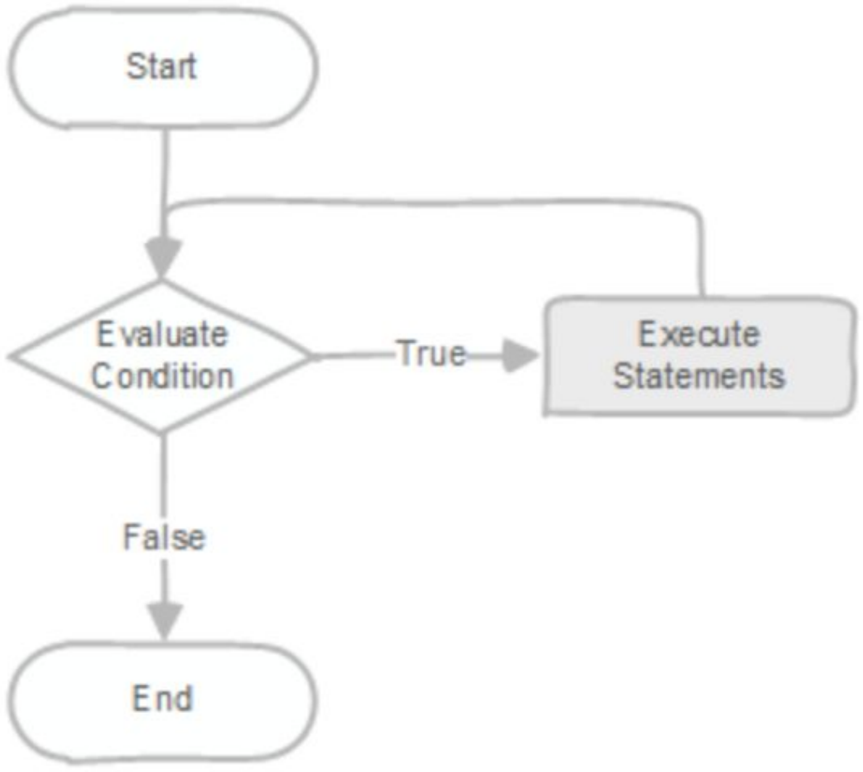
- The WHILE loop statement executes a block of statements until a condition evaluates to false.
- In the WHILE loop statement, PostgreSQL evaluates the condition before executing the block of statements. If the condition is true, the block of statements is executed until it is evaluated to false.
- Syntax:



```
[ <<label>> ]
WHILE condition LOOP
   statements;
END LOOP;
```

WHILE loop

- In the WHILE loop statement, PostgreSQL evaluates the condition before executing the block of statements. If the condition is true, the block of statements is executed until it is evaluated to false.
- The following flowchart illustrates the WHILE loop statement.



Example (Fibonacci sequence)

- We can use the WHILE loop statement to rewrite the Fibonacci function in the first example as follows:

```
CREATE OR REPLACE FUNCTION fibonacci (n INTEGER)
 RETURNS INTEGER AS $$
DECLARE
   counter INTEGER := 0 ;
   i INTEGER := 0 ;
   j INTEGER := 1 ;
BEGIN

 IF (n < 1) THEN
 RETURN 0 ;
 END IF;

 WHILE counter <= n LOOP
 counter := counter + 1 ;
 SELECT j, i + j INTO i, j ;
 END LOOP ;

 RETURN i ;
END ;
```

FOR loop for looping through a range of integers

- The following illustrates the syntax of the FOR loop statement that loops through a range of integers:

```
[ <<label>> ]
FOR loop_counter IN [ REVERSE ] from.. to [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

FOR loop for looping through a range of integers

- First, PostgreSQL creates an integer variable loop_counter that exists only inside the loop. By default, the loop counter is added after each iteration, If you use the REVERSE keyword, PostgreSQL will subtract the loop counter.
- Second, the from and to are expressions that specify the lower and upper bound of the range. PostgreSQL evaluates those expressions before entering the loop.
- Third, the expression following the BY clause specifies the iteration step. If you omit this, the default step is 1. PostgreSQL also evaluates this expression once on loop entry.

FOR loop for looping through a range of integers

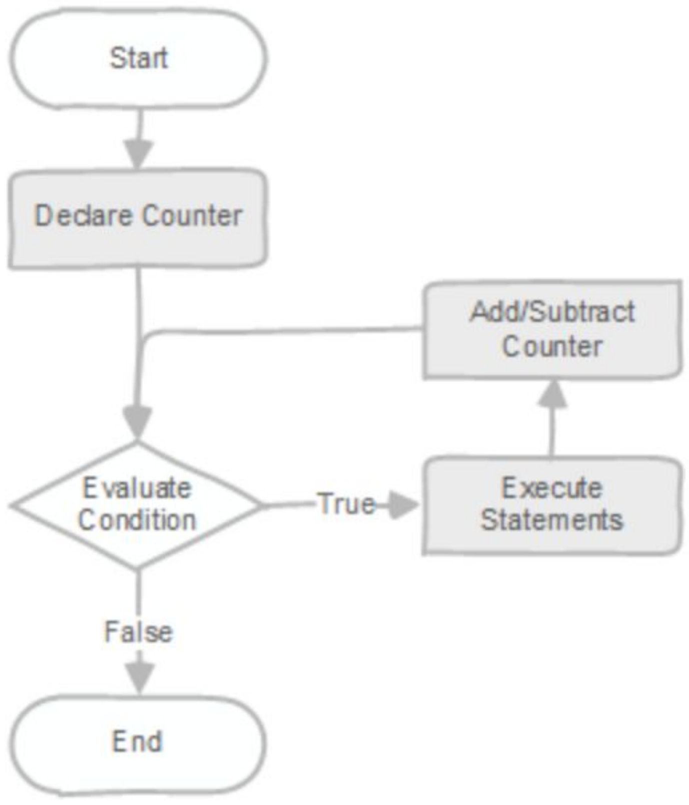
```
FOR i IN 1..10 LOOP
    -- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- i will take on the values 10,8,6,4,2 within the loop
END LOOP;
```

FOR loop for looping through a range of integers

- The following flowchart illustrates the FOR loop statement:



Example

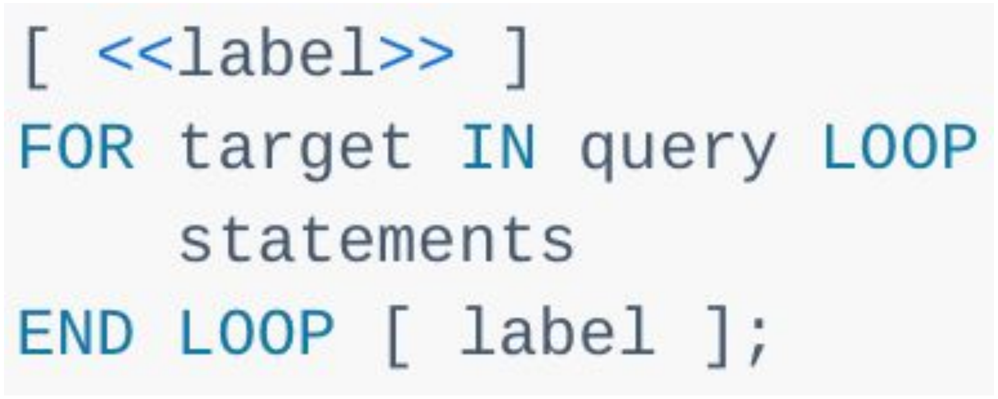
- Loop through 1 to 5 and print out a message in each iteration. The counter takes 1, 2, 3, 4, 5. In each loop iteration, PostgreSQL adds 1 to the counter.

```
DO $$
BEGIN
   FOR counter IN 1..5 LOOP
 RAISE NOTICE 'Counter: %', counter;
   END LOOP;
END; $$
```

```
NOTICE:  Counter: 1
NOTICE:  Counter: 2
NOTICE:  Counter: 3
NOTICE:  Counter: 4
NOTICE:  Counter: 5
```

FOR loop for looping through a query result

- You can use the FOR loop statement to loop through a query result. The syntax is as below:



```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

FOR loop for looping through a query result

- The following function accepts an integer which specifies the number of rows to query.
- The FOR loop statement loops through rows returned from the query and print out the film title.

```
CREATE OR REPLACE FUNCTION for_loop_through_query(
   n INTEGER DEFAULT 10
)
RETURNS VOID AS $$
DECLARE
    rec RECORD;
BEGIN
    FOR rec IN SELECT title
        FROM film
        ORDER BY title
        LIMIT n
    LOOP
 RAISE NOTICE '%', rec.title;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT for_loop_through_query(5);
```

```
NOTICE:  Academy Dinosaur
NOTICE:  Ace Goldfinger
NOTICE:  Adaptation Holes
NOTICE:  Affair Prejudice
NOTICE:  African Egg
```

Books

- Connolly, Thomas M. Database Systems: A Practical Approach to Design, Implementation, and Management / Thomas M. Connolly, Carolyn E. Begg.- United States of America: Pearson Education

- Garcia-Molina, H. Database system: The Complete Book / Hector Garcia-Molina.- United States of America: Pearson Prentice Hall

- Sharma, N. Database Fundamentals: A book for the community by the community / Neeraj Sharma, Liviu Perniu.- Canada

- www.postgresql.org