Concurrency Control

- Press `Space` to navigate through the slides
- Use `Shift+Space` to go back

Concurrency Control

- In a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the **concurrency of transactions**.
- We have concurrency control protocols to ensure **atomicity**, **isolation** and **serialisability** of concurrent transactions.
- Concurrency control protocols can be broadly divided into two categories:
    - **Lock based protocols**
    - **Time stamp based protocols**

Lock-based Protocols

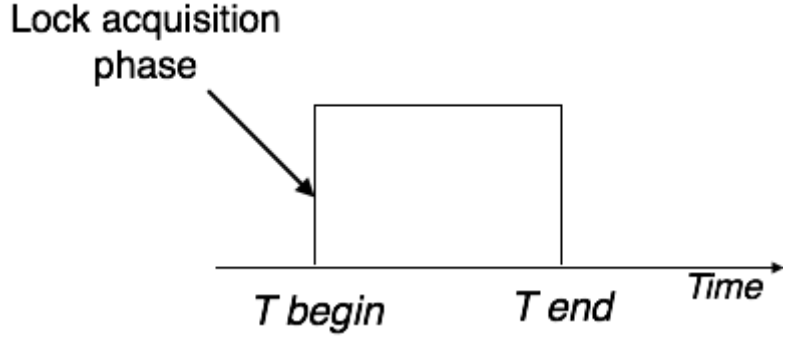
- Database systems equipped with lock-based protocols use a mechanism by which any transaction cannot read or write data until it acquires an appropriate lock on it. Locks are of two kinds:
    - **Binary Locks** – A lock on a data item can be in two states; it is either locked or unlocked.
    - **Shared/exclusive** – This type of locking mechanism differentiates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is an exclusive lock. Allowing more than one transaction to write on the same data item would lead the database into an inconsistent state. Read locks are shared because no data value is being changed.
- Types:
    - **Simplistic**
    - **Pre-claiming**
    - **Two-Phase**
    - **Strict Two-Phase**

Simplistic Lock Protocol

- **Simplistic Lock Protocol**
    - Simplistic lock-based protocols allow transactions to obtain a lock on every object before a 'write' operation is performed.
    - Transactions may unlock the data item after completing the ‘write’ operation.

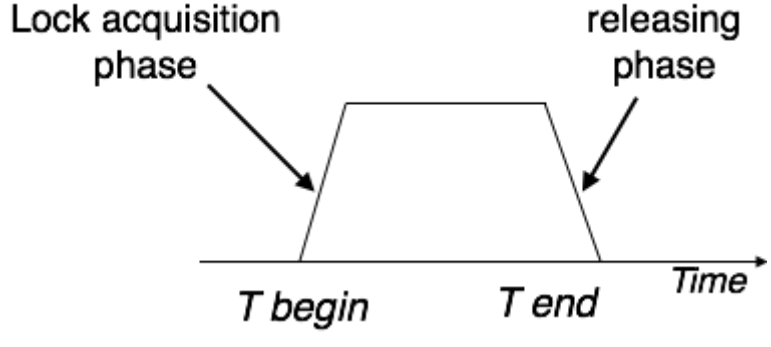
Pre-claiming Lock Protocol

- **Pre-claiming protocols**
  - Evaluate their operations and create a list of data items on which they need locks.
  - Before initiating an execution, the transaction requests the system for all the locks it needs beforehand.
  - If all the locks are granted, the transaction executes and releases all the locks when all its operations are over.
  - If all the locks are not granted, the transaction rolls back and waits until all the locks are granted.



Two-Phase Locking 2PL

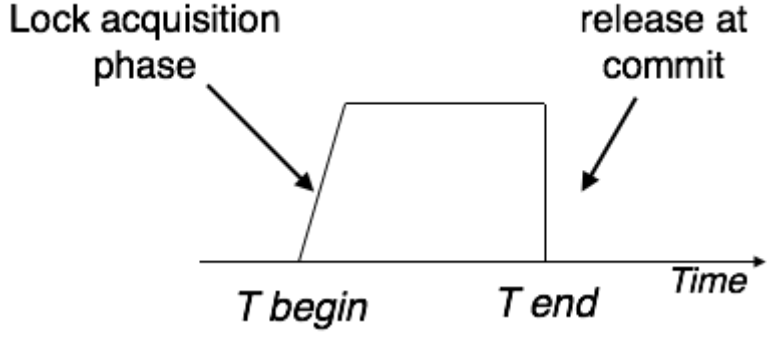
- **Two-Phase Locking 2PL**
  - This locking protocol divides the execution phase of a transaction into three parts.
  - In the first part, when the transaction starts executing, it seeks permission for the locks it requires.
  - The second part is where the transaction acquires all the locks.
  - As soon as the transaction releases its first lock, the third phase starts.
  - In this phase, the transaction cannot demand any new locks; it only releases the acquired locks.



- **Two-phase locking has** two phases:
    - One is **growing**, where all the locks are being acquired by the transaction
    - The second phase is **shrinking**, where the locks held by the transaction are being released
- To claim an exclusive (write) lock, a transaction must first acquire a shared (read) lock and then upgrade it to an exclusive lock.

Strict Two-Phase Locking

- **Strict Two-Phase Locking**
  - The first phase of Strict-2PL is same as 2PL.
  - After acquiring all the locks in the first phase, the transaction continues to execute normally.
  - But in contrast to 2PL, Strict-2PL does not release a lock after using it.
  - Strict-2PL holds all the locks until the commit point and releases all the locks at a time.



- **Strict-2PL does not have cascading abort as 2PL does**

Database Security

- Press `Space` to navigate through the slides
- Use `Shift+Space` to go back

Database Security

- AAA of Security
- Encryption & Disaster Recovery
- General Database Security
- Privileges
    - Granting
    - Revoking
- Views
- SQL Insertion Attacks

AAA of Security

- **Authentication** is the process by which the database verifies that someone is who they claim they are.
- **Authorisation** is the process of establishing if the user (who is already authenticated), is permitted to have access to a resource. Authorisation determines what a user is and is not allowed to do.
- **Access Control** is the process of enforcing the required security for a particular resource in a database.

Encryption 

- **Encryption** is a process which transforms the original information into an unrecognisable form.
    - This new form of the message is entirely different from the original message.
    - Hackers are not able to read the data as senders use an encryption algorithm.
    - Encryption is usually done using key algorithms.
- Data is encrypted to make it safe from stealing.
- Many known companies also encrypt data to keep their trade secret from their competitors.

- Database encryption can generally be defined as a process that uses an algorithm to transform data stored in a database into "cipher text" that is incomprehensible without first being decrypted.
- Database encryption secures the actual data within the database and protects backups.
- Data remains protected even in the event of a data breach.

Disaster Recovery 

- **Disaster recovery (DR)** is used to describe the activities that need to be done to restore the database in the event of a fire, earthquake, vandalism, or other catastrophic events.
- **DR** is the process of resuming normal operations following a disaster by regaining access to data, hardware, software, networking equipment, power and connectivity.
- In case corporate facilities are damaged or destroyed, activities may also extend to logistical considerations like finding alternate work locations, restoring communications, or sourcing anything from desks and computers to transportation for employees.
- **DR response** should follow a disaster recovery plan – a documented process or set of procedures developed specifically to prepare the organisation to recover in the shortest possible time during a period of acute stress.

Database Security

- Database security is about **controlling access to information**
    - Some information should be available freely
    - Other information should only be available to certain people or groups
- Many aspects to consider for security:
    - Physical security
    - OS/Network security
    - Encryption and passwords
    - RDBMS security
- This lecture we will focus mainly on DBMS security

DBMS Security Support

- **RDBMS** can **provide** some security:
    - Each user has an account, username and password
    - These are used to identify a user and control their access to information
- The **RDBMS verifies password and checks a user's permissions** when they try to:
    - Retrieve data
    - Modify data
    - Modify the database structure

Permissions and Privilege

- **SQL** uses privileges to control access to tables and other database objects:
    - `SELECT` privilege
    - `INSERT` privilege
    - `UPDATE` privilege
    - `CREATE` privilege
- For example, in **MySQL** there are actually **30 distinct privileges**
- **The owner (creator) of a database** has all privileges on all objects in the database and can grant these to others |
- **The owner (creator) of an object** has all privileges on that object and can pass them on to others

Privileges in SQL

```
GRANT <privileges>
ON <object>
TO <users>
[WITH GRANT OPTION]
```

`<privileges>` is a list of: `SELECT (<columns>)`, `INSERT (<columns>)`, `DELETE`, `UPDATE (<columns>)` or simply `ALL`

- **`<users>`** is a list of user
- **`<object>`** is the name of a table or view
- **`WITH GRANT OPTIONS`** means that the users can pass their privileges on to others

Privileges Examples

```
GRANT ALL ON Employee
TO Manager
WITH GRANT OPTION;
```

- The user `Manager` can do anything to the `Employee` table and can allow other users to do the same (by using GRANT statements)

```
GRANT SELECT,
UPDATE(Salary)
ON Employee
TO Finance;
```

- The user `Finance` can view the entire `Employee` table, and can change Salary values, but cannot change other values or pass on their privilege

Removing Privileges

- If you want to remove a privilege you have granted you use

```
REVOKE
<privileges>
ON <object>
FROM <users>;
```

- For example:

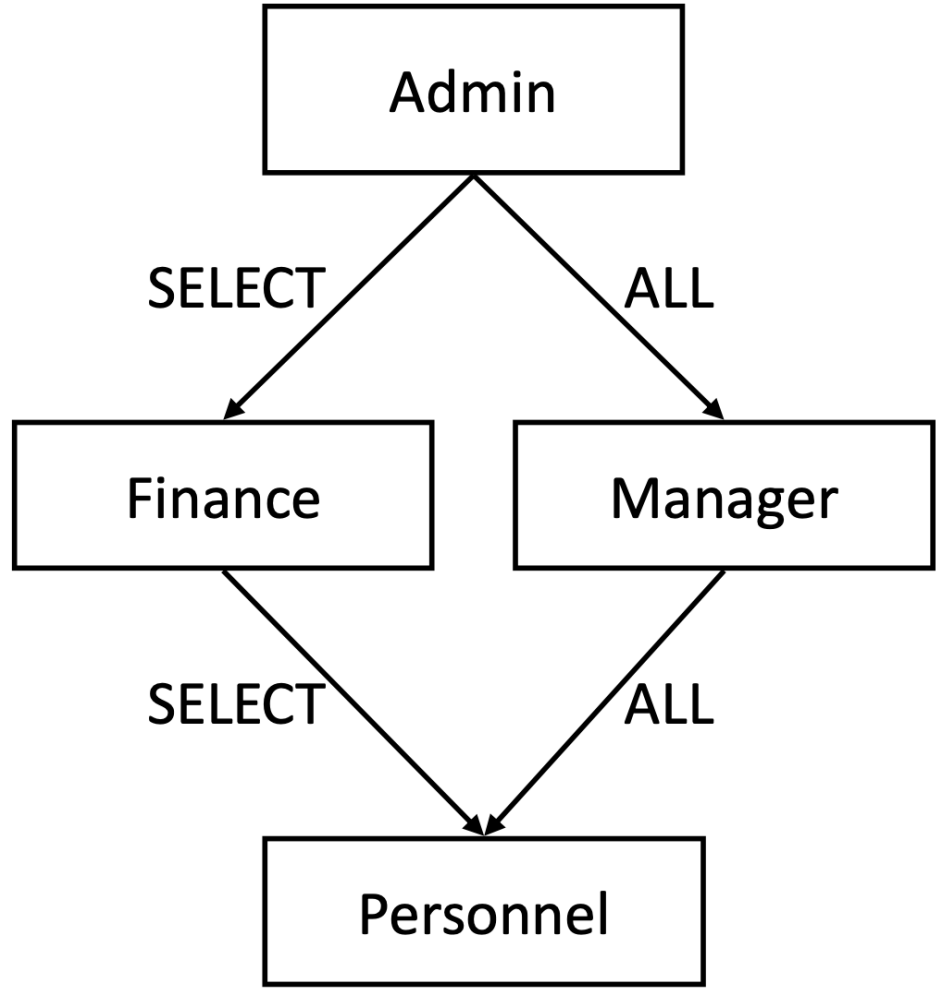
```
REVOKE
UPDATE(Salary)
ON Employee
FROM Finance
```

```
REVOKE ALL
PRIVILEGES, GRANT
OPTION FROM
Manager
```

Removing Privileges

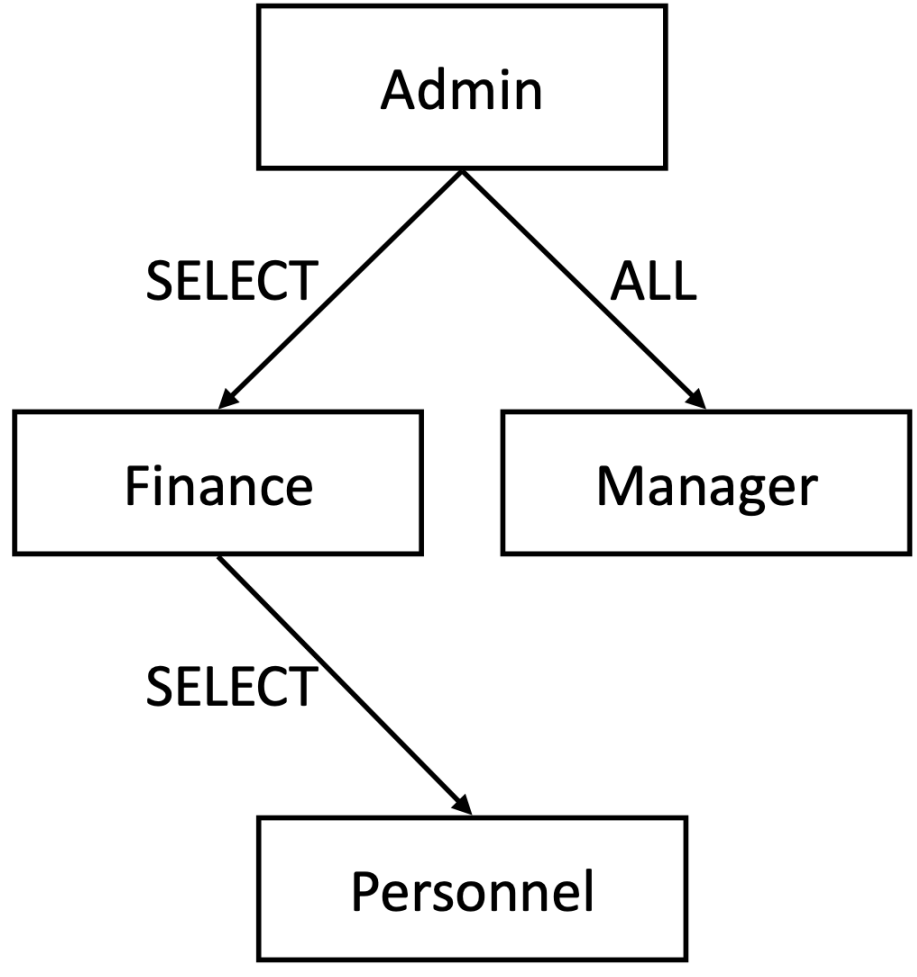
Example:

- `Admin` grants `ALL` privileges to `Manager` and `SELECT` to `Finance` with grant option
- `Manager` grants `ALL` to `Personnel`
- `Finance` grants `SELECT` to `Personnel`


Admin
SELECT
ALL
Finance
Manager
SELECT
ALL
Personnel

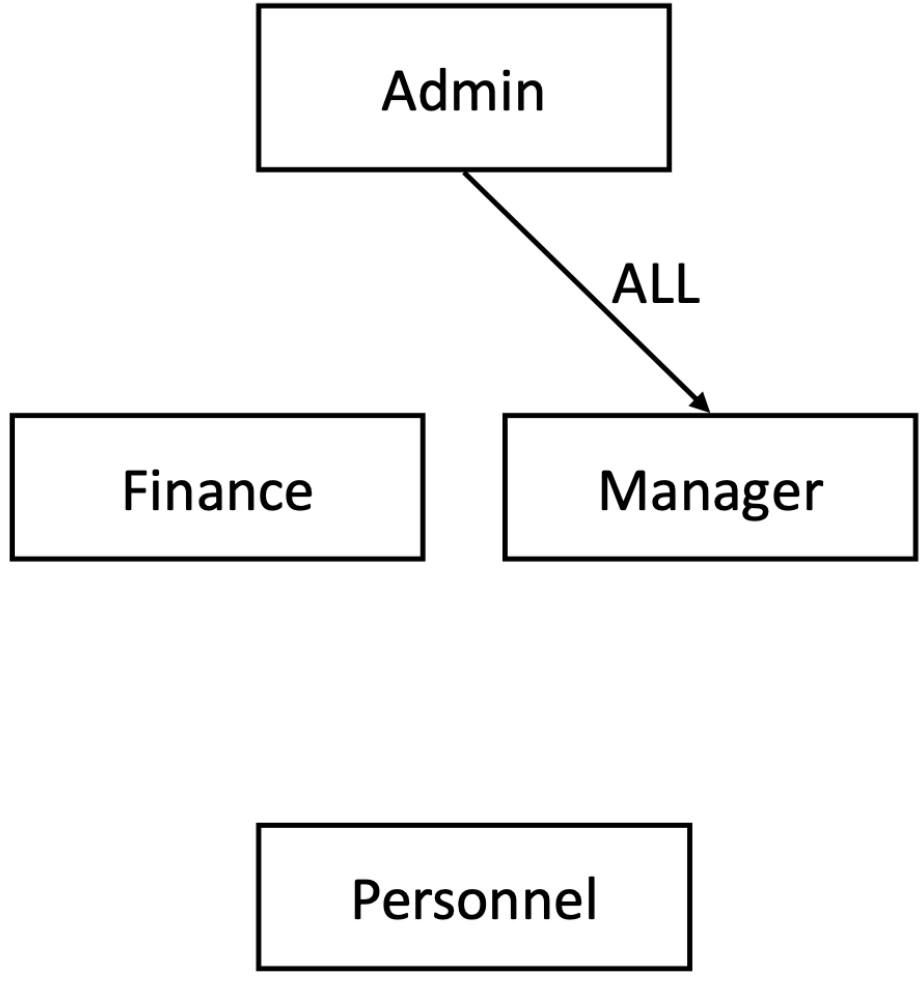
Removing Privileges

- `Manager` revokes `ALL` from `Personnel`
    - `Personnel` still has `SELECT` privileges from `Finance`


Admin
SELECT
ALL
Finance
Manager
SELECT
Personnel

Removing Privileges

- `Manager` revokes `ALL` from `Personnel`
    - `Personnel` still has `SELECT` privileges from `Finance`
- `Admin` revokes `SELECT` from `Finance`
    - `Personnel` also loses `SELECT`


Admin
ALL
Finance
Manager
Personnel

Views

- **Privileges** work at the level of tables:
    - You can restrict access by column
    - You cannot restrict access by row
- **Views**, along with **privileges**, allow for customised access
- **Views provide 'virtual' tables**:
    - A view is the result of a `SELECT` statement which is treated like a table
    - You can `SELECT` from (and sometimes `UPDATE` etc) views just like tables

Advantages of Views

- **Data independence:** a view can present a consistent, unchanging picture of the structure of the database, even if the underlying source tables are changed
- **Currency:** changes to any of the base tables in the defining query are immediately reflected in the view.
- **Improved security:** each user can be given the privilege to access the database only through a small set of views that contain the data appropriate for that user, thus restricting and controlling each user's access to the database.
- **Reduced complexity:** a view can simplify queries, by drawing data from several tables into a single table, thereby transforming multi-table queries into single-table queries.
- **Convenience:** views can provide greater convenience to users as users are presented with only that part of the database that they need to see.
- **Customisation:** views provide a method to customize the appearance of the database, so that the same underlying base tables can be seen by different users in different ways.
- **Data integrity:** if the `WITH CHECK OPTION` clause of the `CREATE VIEW` statement is used, then SQL ensures that no row that fails to satisfy the WHERE clause of the defining query is ever added to any of the underlying base table(s) through the view, thereby ensuring the integrity of the view.

Creating Views

```
CREATE VIEW `<name>`
AS
`<select statement>`;
```

- `<name>` is the name of the new view
- `<select statement>` is a query that returns the rows and columns of the view

Example:

- We want each university tutor to be able to see marks of only those students they actually teach
- We will assume our database is structured with **Student**, **Enrolment**, **Tutors** and **Module** tables

View Example

- Student

sID	sFirst	sLast	sYear

- Enrolment

sID	mCode	eMark	eYearTaken

- Module

mCode	mTitle	mCredits

- Tutors

lID	sID

- Lecturers

lID	lName	lDept

View Example

```
CREATE VIEW TuteeMarks
AS
SELECT sID, sFirst, sLast, mCode, eMark
FROM Student INNER JOIN Enrolment USING(sID)
INNER JOIN Module USING (mCode)
WHERE sIDIN (SELECT sIDFROM Tutors
WHERE lID= CURRENT_USER);

GRANT SELECT ON TuteeMarksTO 'user'@'%';
```

- Note: You should grant for all `Tutors` in **MySQL**, in **Oracle** you can grant to `PUBLIC`. In Oracle `CURRENT_USER` is called `USER`.

Database Integrity

- **Database Security:**
    - Database security makes sure that the user is authorised to access information
    - Beyond security, checks should be made that user mistakes are detected and prevented
- **Database Integrity:**
    - Ensures that authorised users only input consistent data into the database
    - Usually consists of a series of constraints and assertions on data

Database Integrity

- Integrity constraints come in a number of forms:
    - `CREATE DOMAIN` can be used to create custom types with specific values
    - `CREATE ASSERTION` can be used to check manipulation of tables against some test, that must always be true
    - `CHECK` constraints (more widely suppoted) are used to check row-level constraints
- **Oracle** supports `CHECK` constraints
- **MySQL** can emulate them with triggers

Connections to a DBMS

- A major concern with database security should be when your application connects to the **RDBMS**
    - The **user doesn't** connect to the **RDBMS**, the **application does**
    - This often happens with **elevated privileges**
    - If the **application isn't well secured**, it could provide **a conduit for malicious code**

SQL Injection Attacks

- An **SQL Injection attack** is an exploit where a user is able to insert malicious code into an **SQL query**, resulting in an **entirely new query**

SQL Injection Attacks

- It is common for user input to be read and form part of an **SQL query**.
- **For example**, in **PHP**:

```
$query = "SELECT * FROM Products
WHERE pNameLIKE '%". $searchterm. "%'";
```

- If a user is able to pass the application **malicious information**, this information may be combined with regular **SQL queries**
- The resulting query may have a very different effect

SQL Injection Attacks

- An application or website is **vulnerable to an injection attack** if the **programmer hasn't added code to check for special characters** in the input:
    - `'` represents the beginning or end of a string
    - `;` represents the end of a command
    - `/*...*/` represent comments
    - `--` represents a comment for the rest of a line

SQL Injection Attacks

- Imagine a user login webpage that requests a user ID and password.
- These are passed from a form to **PHP** via `$_POST` :
  - `$_POST['id'] = 'Michael'`
  - `$_POST['pass'] = 'password'`
- The ID is later used for a query:

```
SELECT uPass FROM Users WHERE uID = 'Michael';
```

SQL Injection Attacks

- In **PHP** the code for any user might look something like this:

```
$query = "SELECT uPassFROM Users WHERE
uID= '". $_POST['id']. "'";
$result = mysql_query($query);
$row = mysql_fetch_row($result);
$pass = row['uPass'];
```

- The password would then be compared with the other field the user entered

SQL Injection Attacks

- If the user enters *Name* , the command becomes:

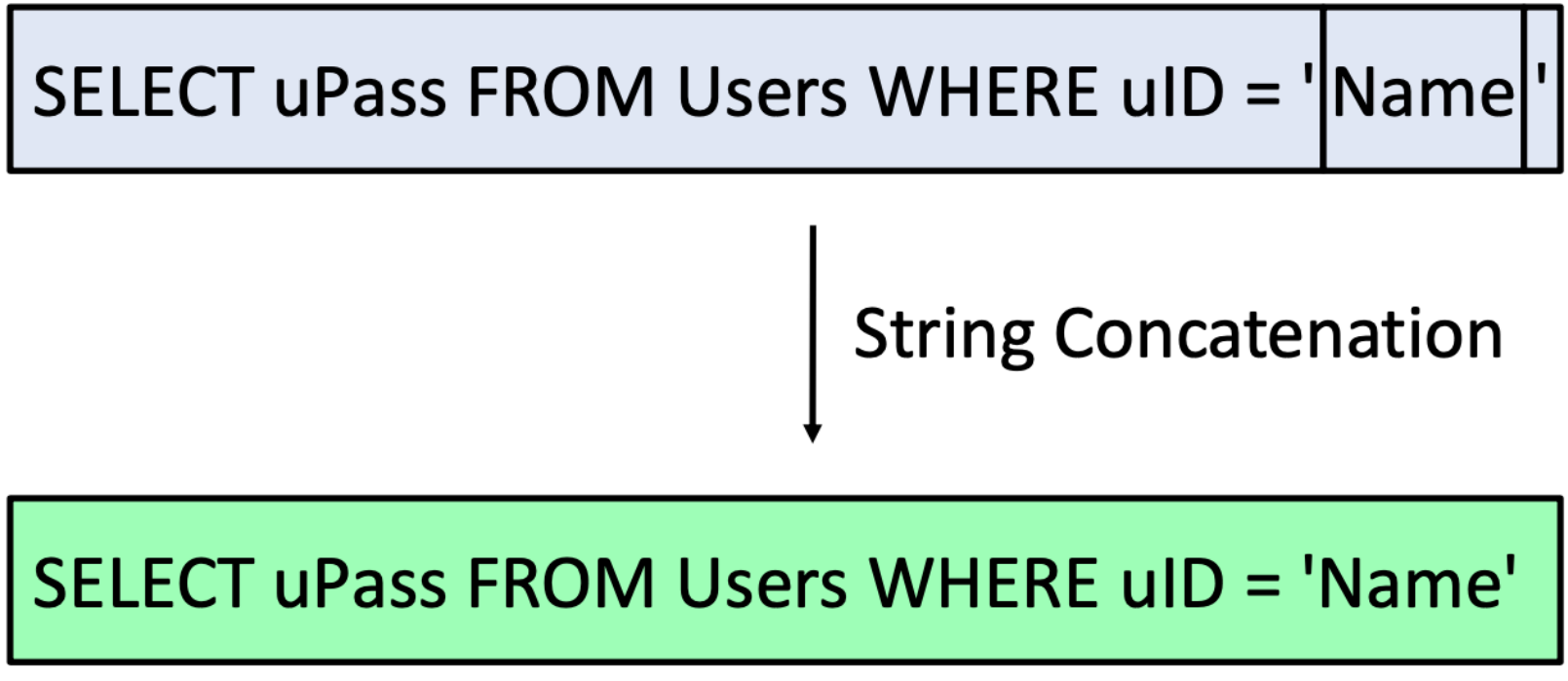
```
SELECT uPass FROM Users
  WHERE uID= 'Name';
```

- But what about if the user entered the following line as their name?:

```
';DROP TABLE Users;--
```

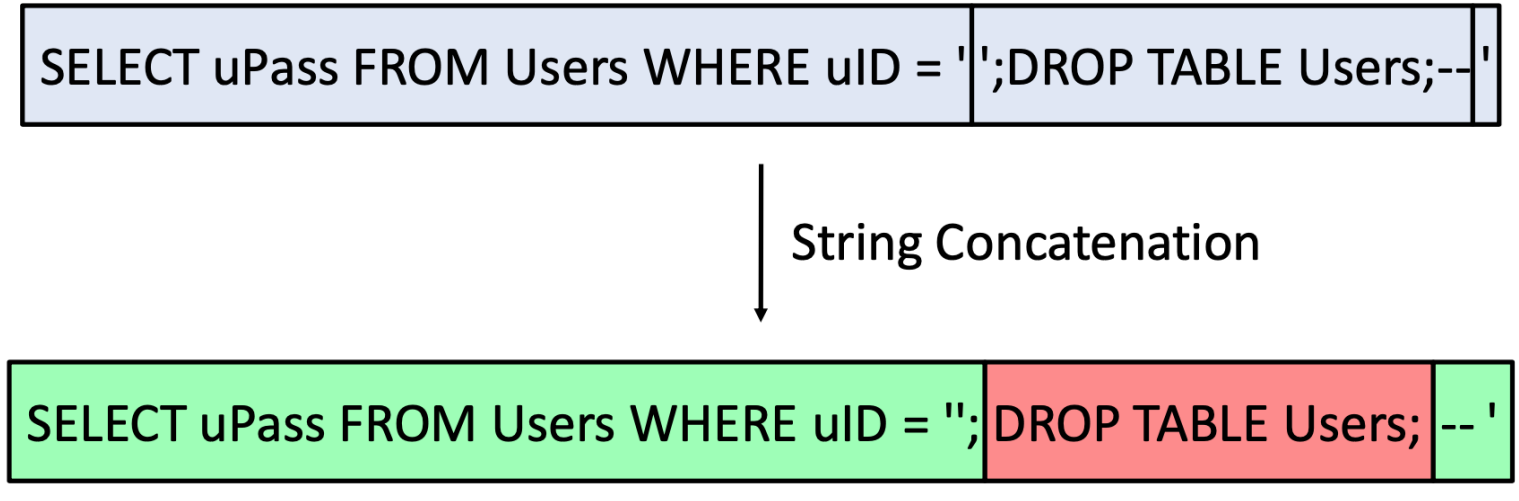
SQL Injection Attacks

- The website programmer intended to execute a single **SQL query**:



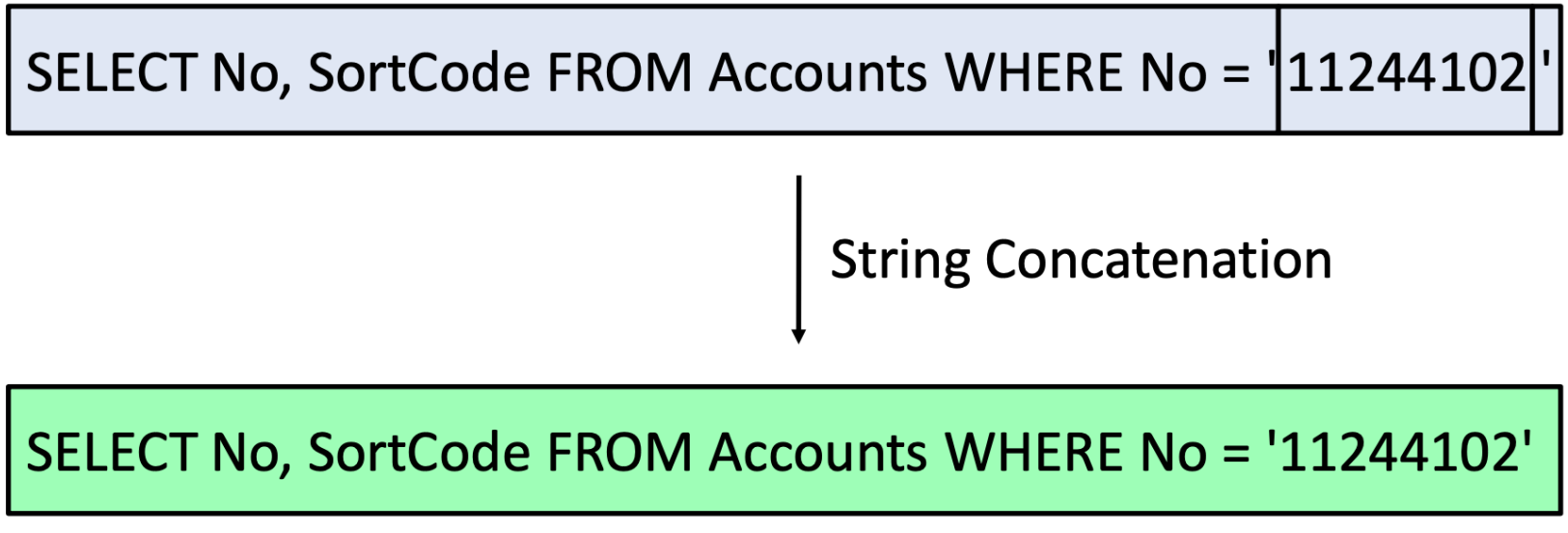
SQL Injection Attacks

- With the malicious code inserted, the meaning of the **SQL** changes into two queries and a comment:



SQL Injection Attacks

- Sometimes the goal isn't sabotage, but information
- Consider an online banking system:



SQL Injection Attacks

- This attack is aimed at listing all accounts at a bank.
- The **SQL** becomes a single, altered query:

SELECT No, SortCode FROM Accounts WHERE No = '	1' OR 'a' = 'a	'

String Concatenation

```
SELECT No, SortCode FROM Accounts WHERE No = '1' OR 'a' = 'a'
```

This is particularly effective with weakly typed languages like PHP

How To Write An SQL Injection Attack

- **Data Protection Act 1998, Section 55(1):**
    - *A person must not knowingly or recklessly, without the consent of the data controller obtain or disclose personal data or the information contained in personal data.*
- Do not do this on a website you do not own
- ***"I was just seeing if it would work"*** is not a valid defence

Defending Against Injection Attacks

- Defending against **SQL injection attacks** is not difficult, but a lot of people still don't
- There are numerous ways you can improve security.
- You should be doing most of these at any time where a user inputs variables that will be used in an **SQL statement**
- In essence, **don't trust that all users** will do what you expect them to do

1. Restrict DBMS Access Privileges

- Assuming an **SQL injection** attack is successful, a user will have access to tables based on the privileges of the account that the application used to connect to the DBMS
- `GRANT` an application or website the minimum possible access to the database
- Do not allow `DROP`, `DELETE`, etc. **unless absolutely necessary**
- Use `Views` to hide as much as possible

2. Encrypt Sensitive Data

- Storing sensitive data inside your database can always lead to problems with security
- If in doubt, encrypt sensitive information so that if any breaches occur, damage is minimal
- Another reason to encrypt data is the majority of commercial security breaches are inside jobs by trusted employees
- **Never** store unencrypted passwords.
- Many shops still do this

3. Validate Input

- Arguably the most important consideration when creating a database or application that handles user input
- Filter any escape characters and check the length of the input against expected sizes
- Checking input length should be standard practice
- This applies to programming in general, as it also avoids buffer overflow attacks

3. Validate Input

- ***Always*** escape special characters.
- All languages that execute **SQL** strings will allow this, in **PHP**:

```
$username = mysql_real_escape_string($input);
$query = "SELECT * FROM Users
WHERE uID= '". $username. "'";
$result = mysql_query($query);
```

- `mysql_real_escape_string()` will escape any special characters, like `'`, with `\`
- **You should do this with any input variables**

4. Check Input Types

- In weakly-typed languages, check that the user is providing you with a type you'd expect
- **For example**, if you expect the ID to be an `int`, make sure it is.
- In **PHP**:

```
if (!is_int($_POST['userid']))
{
// ID is not an integer
}
```

5. Stored Procedures

- Some **RDBMS** allow you to store procedures for use over and over again
- Procedures you might store are `SELECT` s, `INSERTS` s etc, or other procedural code
- This adds another level of abstraction between the user and the tables
- If necessary, a stored procedure can access tables that are restricted to the rest of the application

6. Generic Error Messages

- While it might seem helpful to **output informative error messages**, this actually supplies users with far too much information
- For example, if your **SQL query fails**, do not show the `user mysql_error()`, instead output:
    - ***A system error has occured. We apologise for the inconvenience.***
- You can log the **error** privately for administrative purposes

7. Parameterised Input

- Parameterised input essentially means that **user input** is passed to the database as **parameters**, not as part of the **SQL string**:
    - This makes injection attacks extremely difficult
    - Not all RDBMSs / Languages support this
    - In **PHP**, you need to use **PHP Data Objects (PDO) (http://php.net/manual/en/book.pdo.php)**

PDO 

- Rather than building up a string for your **SQL** and executing it, given a **PDO** mysql connection `$conn`:

```
$stmt = $conn->prepare('SELECT * FROM Users
WHERE uName= :name');
$stmt->bindValue(':name', $_POST['username']);
$stmt->execute();
```

- The statement is pre-compiled during prepare.
- While a malicious parameter may still be passed to the query, it is simply used ather than executed.