Database Security

G51DBS Database Systems

Jason Atkin

jaa@cs.nott.ac.uk

This Lecture

- General Database Security
- Privileges
  - Granting
  - Revoking
- Views
- SQL Injection Attacks
- Further Reading
  - The Manga Guide to Databases, Chapter 5
  - Database Systems, Chapter 20

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
  - DBMS security
- This lecture will focus on DBMS security
  - The other elements are not really G51DBS concerns

DBMS Security Support

- DBMSs 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 DBMS 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. E.g.
  - SELECT privilege *
  - INSERT privilege *
  - UPDATE privilege *
  - CREATE privilege

  ** - these can apply to specific columns*
- 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

- Will use PostgreSQL (not MySQL) – more 'standard'

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

- **WITH GRANT OPTION** means that the users can pass their privileges on to others
- "If WITH GRANT OPTION is specified, the recipient of the privilege may in turn grant it to others..." http://www.postgresql.org/docs/8.2/static/sql-grant.html

- **<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

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 or grant option you have granted you use:

```
REVOKE
 [GRANT OPTION FOR]
 <privileges>
 ON <object>
 FROM <users>
[CASCADE | RESTRICT];
```

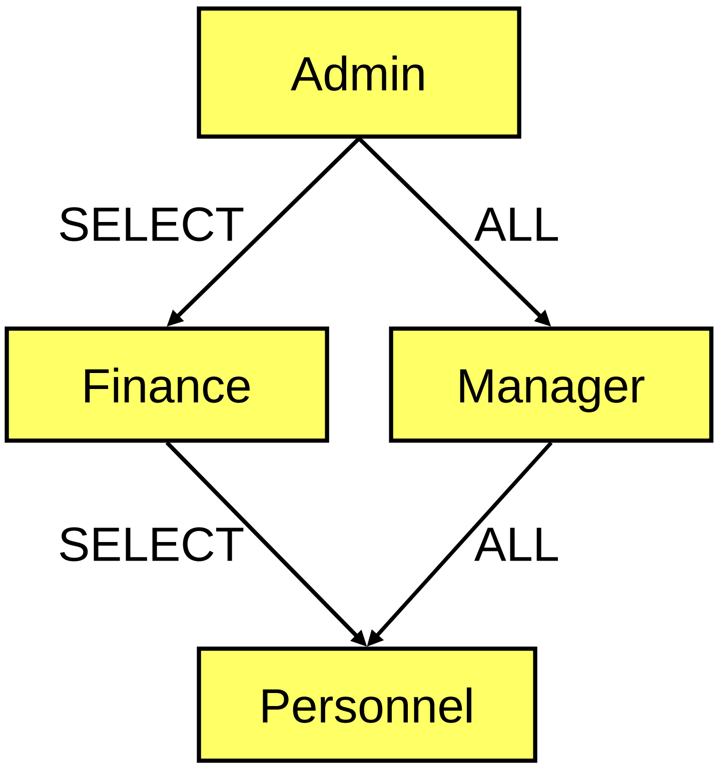
- For example:

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

REVOKE
 GRANT OPTION FOR
 ALL PRIVILEGES
 FROM Manager
```

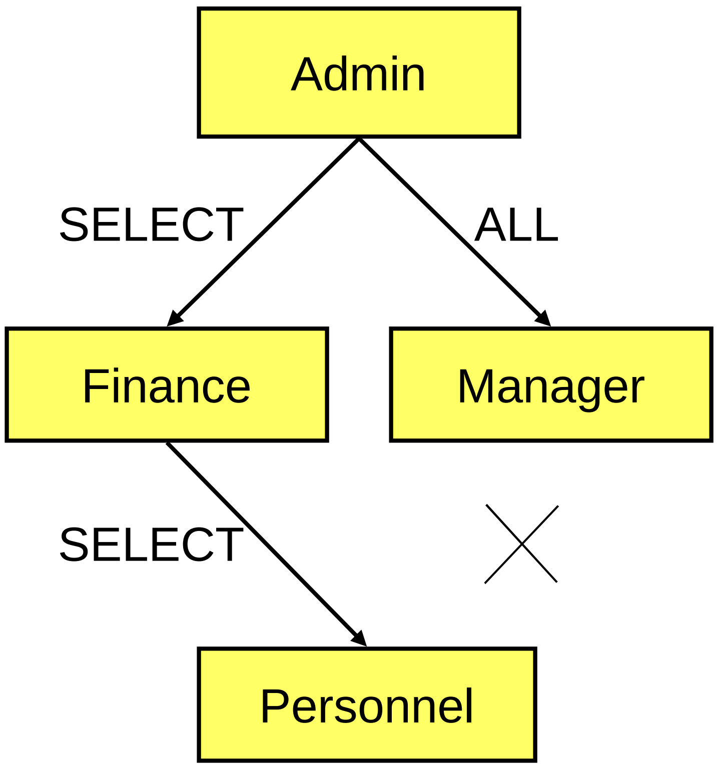
Removing Privileges

- Example
  - ‘Admin’ grants **ALL** privileges to ‘Manager’, and **SELECT** to ‘Finance’ with grant option
  - ‘Manager’ grants **ALL** to Personnel
  - ‘Finance’ grants **SELECT** to Personnel



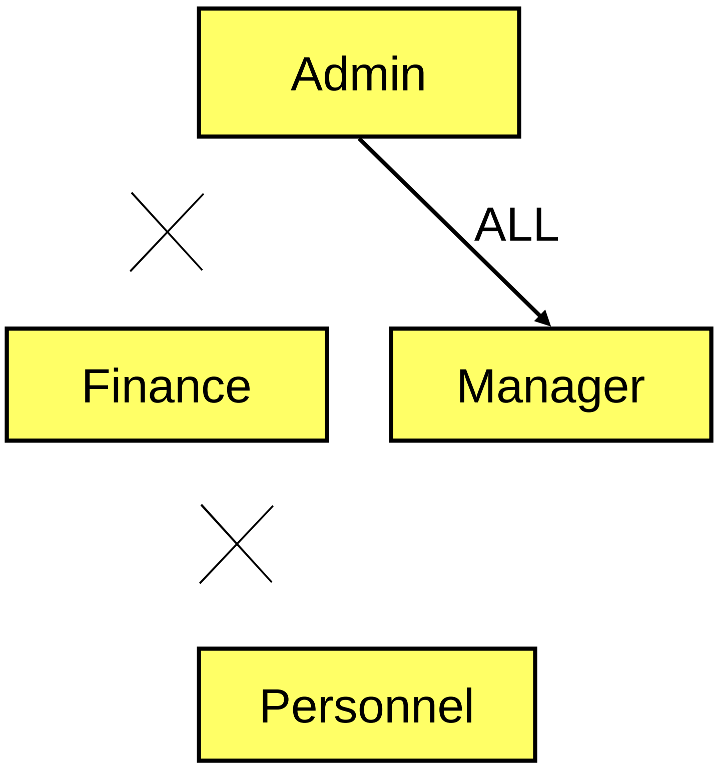
Removing Privileges

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



Removing Privileges

- Manager’ revokes ALL from ‘Personnel’
  - ‘Personnel’ still has SELECT privileges from ‘Finance’
- ‘Admin’ revokes SELECT from ‘Finance’ (with cascade)
  - Personnel also loses SELECT, since it only had it from ‘Finance’



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
  - i.e. Views allow you to limit access to only certain rows or columns
- 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

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 similar to those seen in previous lectures

View Example : the tables

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 sID IN (SELECT sID FROM Tutors
          WHERE lID = CURRENT_USER);
```

Grant permissions to access this to every tutor:

```
GRANT SELECT ON TuteeMarks TO ‘tutorname'@'%';
```

Note: CURRENT_USER is the current mysql user. This is called USER in Oracle. % means 0 or more characters. Only ‘need’ to quote *user_name* or *host_name* if it contains control characters. In Oracle you can grant to PUBLIC. MYSQL does not allow wildcard usernames.

Live example... (coursework 2 table)

- CREATE SQL SECURITY INVOKER VIEW DList AS SELECT description FROM Description;
- REVOKE SELECT ON DList FROM jaa;
- GRANT SELECT ON DList to jaa;
- SELECT * FROM DList;
- REVOKE SELECT ON DList FROM jaa;
- DROP VIEW DList;
- SELECT * FROM DList;

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:
  - **CHECK** can be used in a column definition, e.g.:
    - `gender CHAR NOT NULL CHECK( gender IN (‘M’,‘F’) )`
  - **DOMAIN**s can be used to create custom types, e.g.:
    - `CREATE DOMAIN gendertype AS CHAR DEFAULT ‘F’ CHECK (VALUE IN (‘M’,‘F’) );`
  - **ASSERTION**s can be used to validate checks across multiple tables/columns, assert something must always be true
- Note: check the syntax for your database. Oracle supports CHECK constraints. MySQL seems to accept the syntax and often ignore it. MySQL can emulate these with triggers – see CREATE TRIGGER …

Connections to a DBMS

- A major concern with database security should be when your application connects to the DBMS
  - The user doesn't connect to the DBMS, 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!

WARNING!!!

- 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 try this on a website you do not own**
- *"I was just seeing if it would work"* is not a valid defence

Usual web page / PHP interaction

Page is displayed with an HTML Form

User enters data in the fields and presses ‘submit’

HTML code extracts the fields and ‘post’s them to a web page (as ‘get’ or ‘post’)

PHP code in destination page:
Extracts posted values
Connects to database
Builds a query using the values
Sends query to MySQL (etc)
Retrieves response
Creates an HMTL web page

e.g. see Lecture 14, http://www.cs.nott.ac.uk/~jaa/dbs/htmlform.php
or http://www.cs.nott.ac.uk/~jaa/dbs/exercise4/dbdemo.php

SQL Injection Attacks : The Cause

- If a page sends a search string in a parameter called ‘searchfor’, PHP code could do the following:

```
$search = $_POST[‘searchfor']
$query = "SELECT * FROM Products WHERE
    pName LIKE '%" . $search . "%'";
```

- In many cases the posted value came from something which the user typed into a web page
- If user can pass malicious information, this information may be combined with regular SQL queries
- The resulting query may have a very different effect

SQL Injection Attacks : The Mistake

- 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 : Example

- e.g. User login webpage, requests user ID & password
- Passed from form to PHP, as ‘id’ and ‘pass’
- ID later used for a query, which should take the form:

```
SELECT uPass FROM Users WHERE uID = ‘Jason';
```

- Example PHP code using the variable ‘id’:

```
$query = "SELECT uPass FROM Users WHERE uID
  = '" . $_POST['id'] . "'";

$result = mysql_query($query);
$row = mysql_fetch_row($result);
$pass = row['uPass']; (Compare vs input)
```

- Password then compared with the provided password field

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

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

  as their **name**?

SQL Injection Attacks

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

SELECT uPass FROM Users WHERE uID = ' | Name | '

String Concatenation

SELECT uPass FROM Users WHERE uID = 'Name'

SQL Injection Attacks

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

SELECT uPass FROM Users WHERE uID = ' | ';DROP TABLE Users;-- | '

String Concatenation

SELECT uPass FROM Users WHERE uID = ''; | DROP TABLE Users; | -- '

SQL Injection Attacks : mysql_query()

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

```
SELECT uPass FROM Users WHERE uID = ' | ';DROP TABLE Users;-- | '
```

String Concatenation

```
SELECT uPass FROM Users WHERE uID = ''; | DROP TABLE Users; | -- '
```

Note: this one shouldn't actually be a problem with mysql_query() : "**mysql_query()** sends a unique query (multiple queries are not supported) to the currently active database on the server that's associated with the specified link_identifier."

SQL Injection Attacks

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

SELECT No, SortCode FROM Accounts WHERE No = '	11244102	'

String Concatenation

```
SELECT No, SortCode FROM Accounts WHERE No = '11244102'
```

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'
```

Particularly effective with weakly typed languages like PHP

Defending Against Injection Attacks

- Defending against SQL injection attacks is not difficult, but many people still don't
- There are many ways to improve security
- You should be doing most of these all of the time when a user inputs values that will be used in an SQL statement
- **Summary: 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 that the majority of commercial security breaches are '*inside jobs*' by trusted employees
- **Never** store unencrypted passwords, although many shops still do this ☹

3a. Validate Input

- Always validate the input values
- Arguably the most important consideration when creating a database or application that handles user input is to validate it
- 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

3b. Validate Input

- ***Always*** escape special characters. All languages that execute SQL strings should allow this, e.g.:

```
$username = mysql_real_escape_string($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 all 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 DBMSs allow you to store procedures for use over and over again
- Procedures you might store are SELECTs, INSERTSs 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. Use 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 occurred. We apologise for the inconvenience.

- You can **log** the error **privately** for administrative purposes

7a. Use 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 DBMSs / Languages support this
- In PHP, you need to use PHP Data Objects (PDO) to provide parameterisation
- Reference: http://php.net/manual/en/book.pdo.php

7b. Use PDO prepare()

- PHP Data Objects provides a Prepare() function:

```
$conn = new PDO(
'mysql:host=localhost;dbname=test',$user, $pass );
$stmt = $conn->prepare(
      'SELECT * FROM Users WHERE uName = :name‘ );
$stmt->bindValue( ':name', $_POST['username'] );
$stmt->execute();
```

- Rather than building up a string for your SQL using the posted variable…
- The statement is pre-compiled during prepare
- A malicious parameter may still be passed to the query, but it is simply used as the variable, not as part of the statement

Next Lecture

- Transactions
  - ACID Properties
  - COMMIT and ROLLBACK
- Recovery
  - System and Media Failures
- Concurrency

Transactions and Recovery

G51DBS Database Systems

Jason Atkin

jaa@cs.nott.ac.uk

This Lecture

- Transactions
  - ACID Properties
  - COMMIT and ROLLBACK
- Recovery
  - System and Media Failures
- Concurrency
- Further reading
  - The Manga Guide to Databases, Chapter 5
  - Database Systems, Chapter 22

Transactions

- A transaction is an action, or a series of actions, carried out by a single user or an application program, which reads or updates the contents of a database.
- All database access by users is thought of in terms of transactions

Transactions

- A transaction is a ‘logical unit of work’ on a database
  - Each transaction does something on the database
  - No part of it alone achieves anything useful or of interest
- Transactions are the unit of recovery, consistency and integrity
- ACID properties
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Atomicity

- Transactions are atomic
  - Conceptually do not have component parts
  - In reality a transaction may include numerous read, write and other operations
- Transactions **cannot** be executed partially
  - Either performed entirely, or not at all
  - It should not be **detectable** that they interleave with another transaction

Consistency

- Transactions take the database from one **consistent** state to another
- Consistency isn't guaranteed part-way through a transaction
  - Because of atomicity, this won't be a problem
- Enforced by the DBMS, and application programmers also have some responsibility

Isolation

- All transactions execute independently of one another
- The effects of a transaction are invisible to other transactions until it has been completed
- Enforced by the scheduler

Durability

- Once a transaction has completed, its changes are made permanent
- If the database system crashes, completed transactions must remain complete
- Enforced by the recovery manager

Transaction Example

- Transfer £50 from bank account A to account B

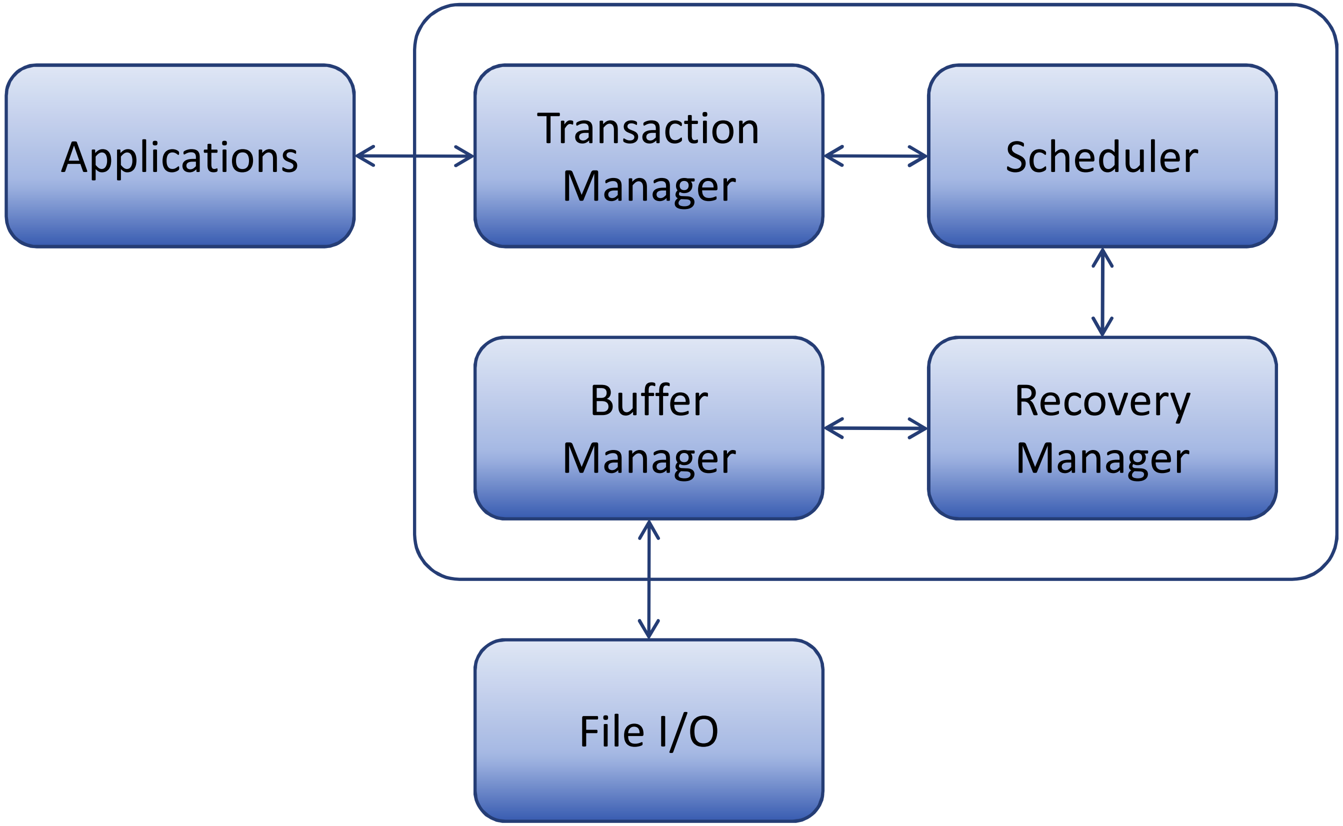
$$\left.\begin{array}{l} \text{Read}(A) \\ A = A - 50 \\ \text{Write}(A) \\ \text{Read}(B) \\ B = B + 50 \\ \text{Write}(B) \end{array}\right\}\text{Transaction}$$

- **Atomicity** – Shouldn’t take money from A without giving it to B
- **Consistency** – Money isn’t lost or gained overall
- **Isolation** – Other queries shouldn’t see A or B change until completion
- **Durability** – The money does not return to A, even after a system crash

Transaction Subsystem

- The transaction subsystem enforces the ACID properties
  - Schedules the operations of all transactions
  - Uses COMMIT and ROLLBACK to ensure atomicity
  - Locks and/or timestamps are used to ensure consistency and isolation (next lectures)
  - A log is kept to ensure durability

Transaction Subsystem



Database Systems, Connolly & Begg, p574

COMMIT and ROLLBACK

- COMMIT is used to signal the successful end of a transaction
  - Any changes that have been made to the database should be made permanent
  - These changes are now available to other transactions
- ROLLBACK is used to signal the unsuccessful end of a transaction
  - Any changes that have been made to the database should be undone
  - It is now as if the transaction never happened, it can now be reattempted if necessary

Recovery

- Transactions must be durable, but some failures will be unavoidable
  - System crashes
  - Power failures
  - Disk crashes
  - User mistakes
  - Sabotage
  - etc
- Prevention is better than a cure
  - Reliable OS
  - Security
  - UPS and surge protectors
  - RAID arrays
- Can't protect against everything, system recovery will be necessary

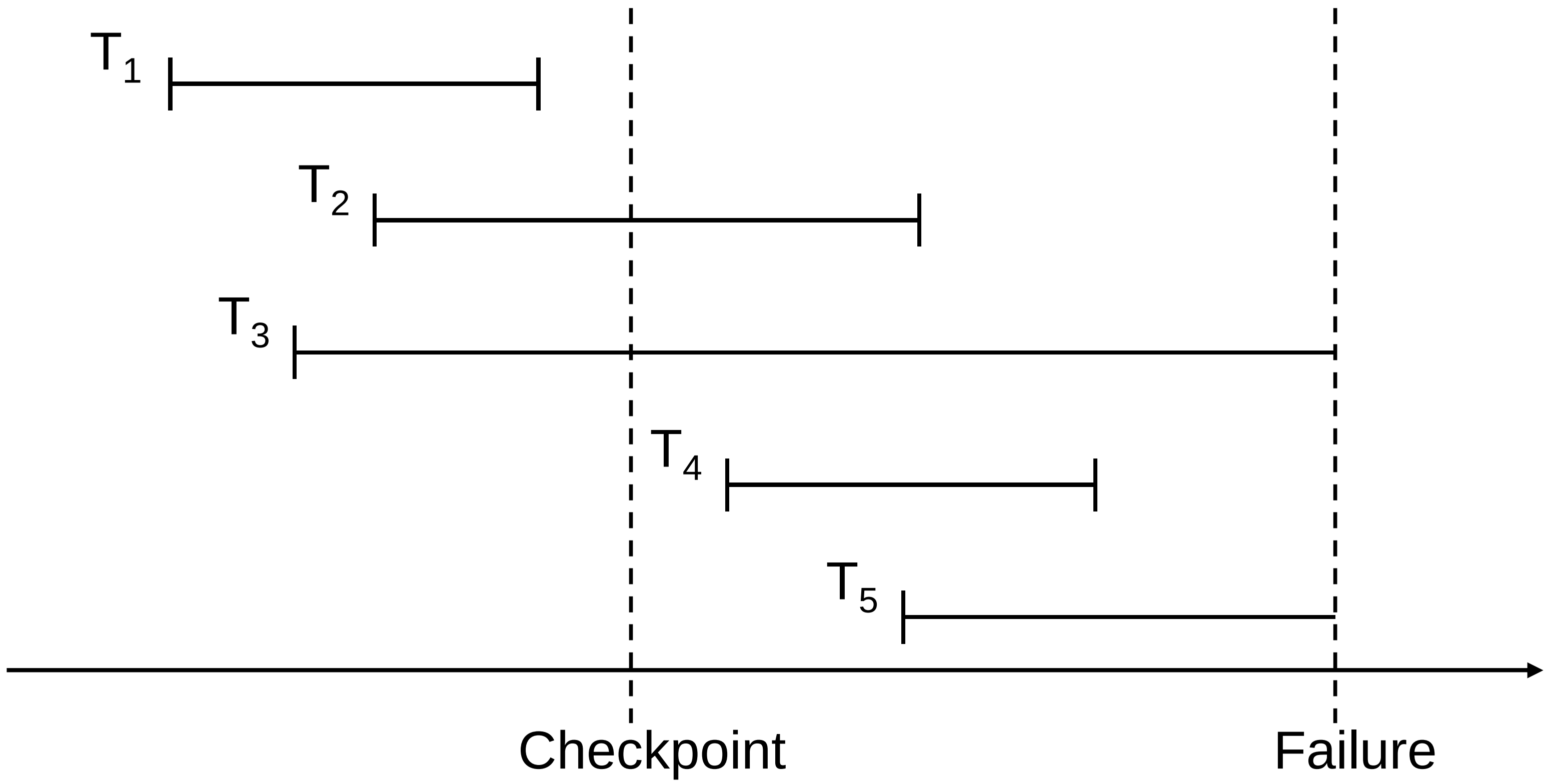
The Transaction Log

- The transaction log records details of all transactions
  - Any changes the transaction makes to the database
  - How to undo these changes
  - When transactions complete and how
- **The log is stored on disk, not in memory**
  - If the system crashes, the log is preserved
- Write ahead log rule
  - The entry in the log must be made before COMMIT processing can complete

System Failures

- A system failure affects all running transactions
  - Software crash
  - Power failure
- The physical media (disks) are not damaged
- Things in memory are lost, things on the disk are kept
- At various times a DBMS takes a checkpoint
  - All transactions are written to disk
  - A record is made (on disk) of all transactions that are currently running
- Until that time, changes could be in memory but not on disk

Transaction Timeline



System Recovery

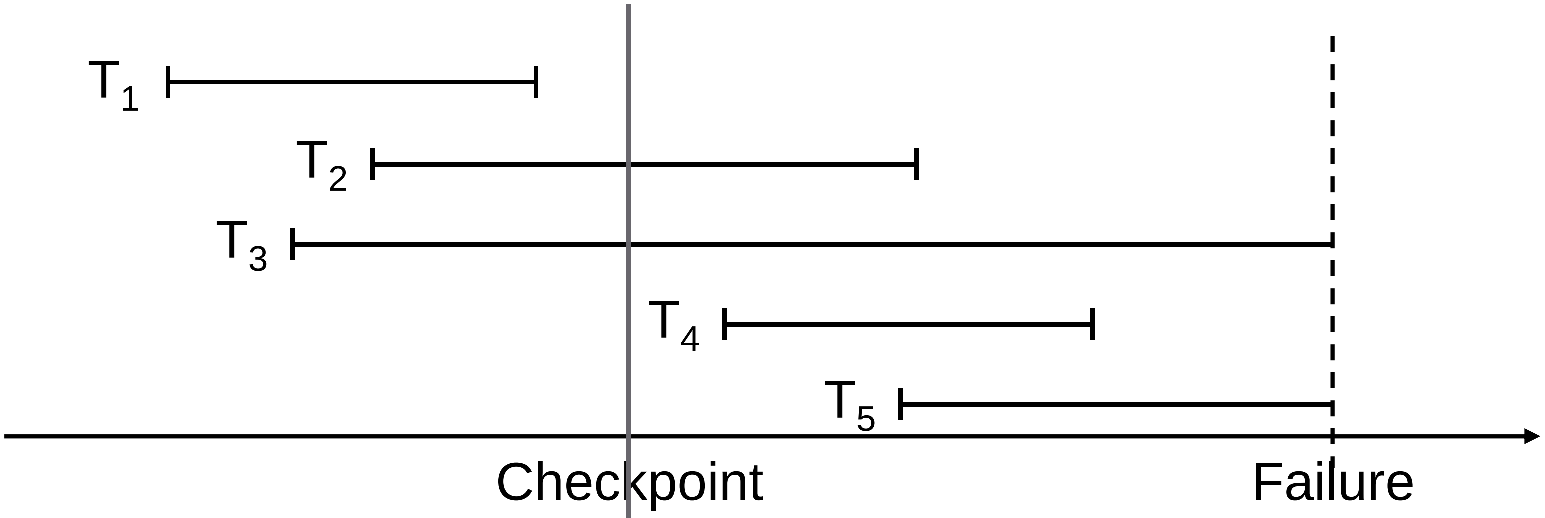
- Any transaction that was running at the time of failure needs to be undone and possibly restarted
- Any transactions that committed since the last checkpoint need to be redone

- Transactions of type T_1 (completed before check point) need no recovery
- Transactions of type T_3 or T_5 (uncompleted) need to be undone
- Transactions of type T_2 or T_4 (completed SINCE check point) need to be redone

Transaction Recovery

- Create two lists of transactions: UNDO and REDO
  - UNDO – all transactions running at the last checkpoint
  - REDO – empty
- For every entry in the log since the last checkpoint, until the failure:
  1. If a BEGIN TRANSACTION entry is found for T:
     - Add T to UNDO
  2. If a COMMIT entry is found for T:
     - Remove T From UNDO
     - Add T to REDO (it had finished)

Transaction Recovery



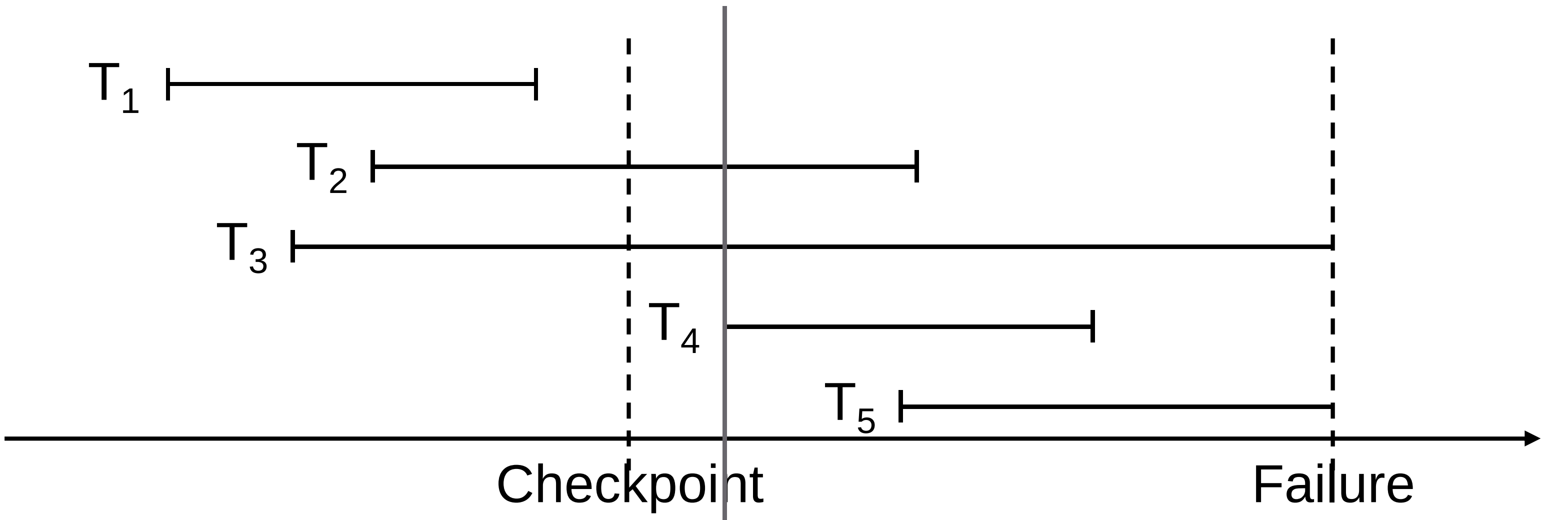
UNDO: T_2, T_3

REDO:

Last Checkpoint

Active transactions: T_2, T_3

Transaction Recovery



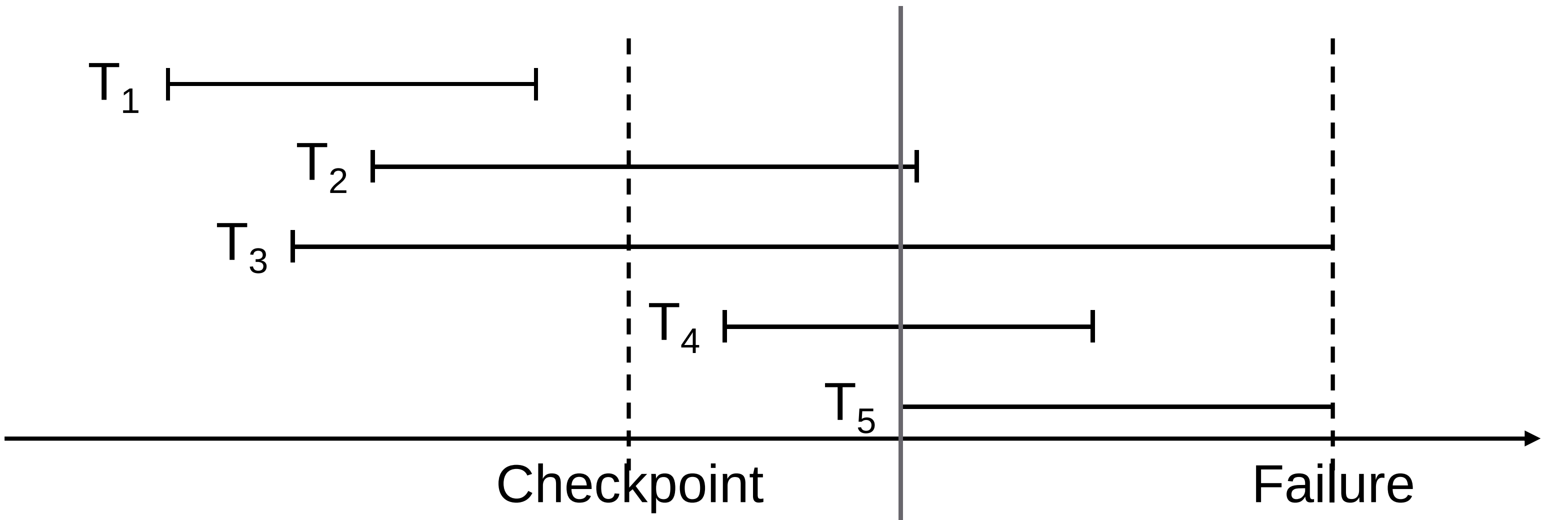
UNDO: T_2, T_3, T_4

REDO:

T4 Begins

Add T_4 to UNDO

Transaction Recovery



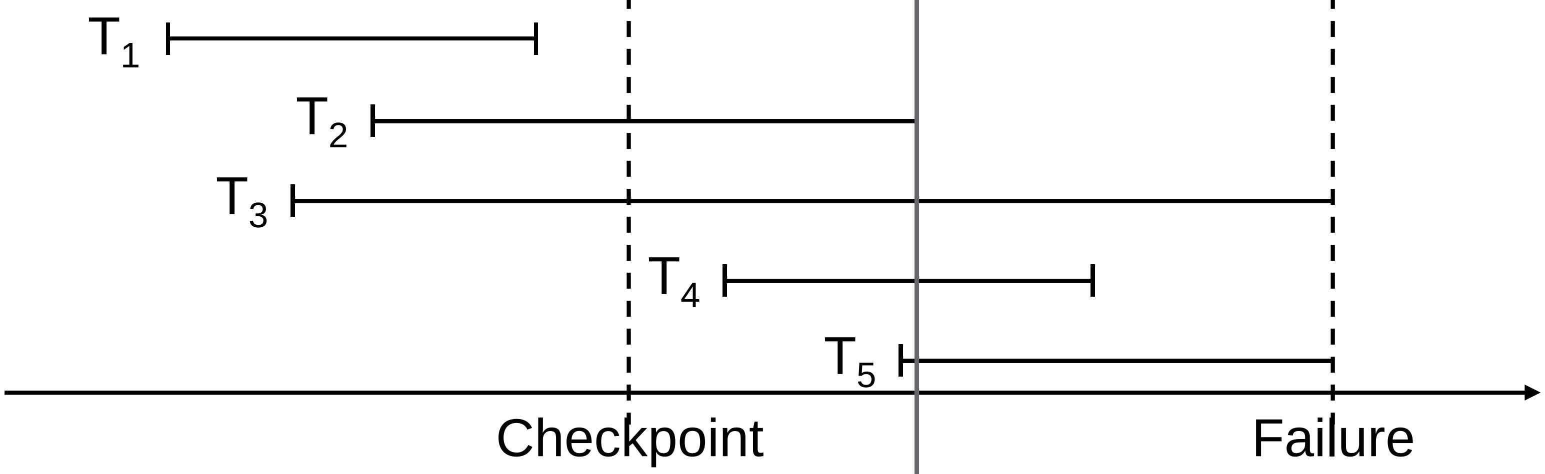
UNDO: T_2, T_3, T_4, T_5

REDO:

T_5 begins

Add T_5 to UNDO

Transaction Recovery



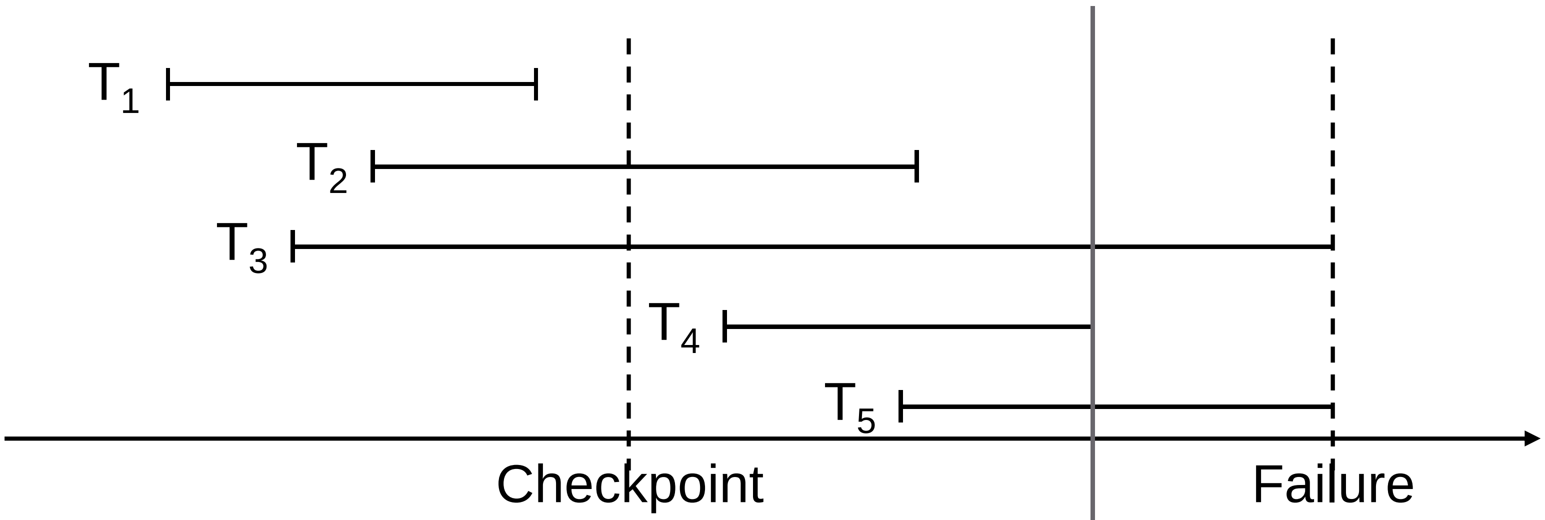
UNDO: T_3, T_4, T_5

REDO: T_2

T_2 Commits

Move T_2 to REDO

Transaction Recovery



UNDO: T_3, T_5

REDO: T_2, T_4

T_4 Commits

Move T_4 to REDO

Forwards and Backwards

- Backwards recovery - ROLLBACK
  - We need to undo some transactions
  - Working backwards through the log we undo every operation by any transaction on the UNDO list
  - This returns the database to a consistent state – although with some uncompleted transactions (those on the redo list)
- Forwards recovery - ROLLFORWARD
  - Some transactions need to be redone
  - Working forwards through the log we redo any operation by a transaction on the REDO list
  - This brings the database up to date

Media Failures

- System failures are not too severe
  - Only information since the last checkpoint is affected
  - This can be recovered from the transaction log
- Media failures (e.g. Disk failure) are more serious
  - The stored data is damaged
  - The transaction log itself may be damaged

Backups

- Backups are necessary to recover from media failure
  - The transaction log and entire database is written to secondary storage
  - Very time consuming, often requires downtime
- Backup frequency
  - Frequent enough that little information is lost
  - Not so frequent as to cause problems
  - Every night is a common compromise

Recovery from Media Failure

1. Restore the database from the last backup
2. Use the transaction log to redo any changes made since the last backup

- If the transaction log is damaged you can't do step 2
  - Store the log on a separate physical device to the database
  - This reduces the risk of losing both together

Transactions in MySQL

- Most DBMSs support transactions
- In MySQL in school only the InnoDB engine supports transactions
- There are other engines which support this, that are not installed, like Falcon
- On the school servers, autocommit is set so that every command is instantly committed
- This is very slow and inefficient
- And does not make it easy to undo changes
- You can turn autocommit off with

```
SET autocommit = 0 | 1;
```

Managing Transactions

- In MySQL, a transaction is executed in the following way:

```
BEGIN | START TRANSACTION;
INSERT INTO table VALUES (...);
SELECT col1, col2 FROM table;
UPDATE table SET col1 = col2 + 3;
DROP TABLE table;
COMMIT | ROLLBACK;
```

(| *optional*)

Managing Transactions

- In PHP, you can send off these commands with mysql_query:

```
mysql_query(‘BEGIN’);
mysql_query(‘...’);
if (some test)
{
   mysql_query(‘COMMIT’);
}
else
{
   mysql_query(‘ROLLBACK’);
}
```

Managing Transactions

- In general, this approach is far superior to autocommit. Remember, however:
  - If your transaction locks a table, all other transactions will have to wait
    - So COMMIT as soon as possible
  - MyISAM and most engines ignore commands like ROLLBACK. So use InnoDB if you need transaction support
  - Subqueries are good when using autocommit to avoid outdated information

Concurrency

- Large databases are used by many people
  - Many transactions are to be run on the database
  - It is helpful to run these simultaneously
  - Still need to preserve isolation
- If we don't allow for concurrency then transactions are run sequentially
  - Have a queue of transactions
  - Easy to preserve atomicity and isolation
  - Long transactions (e.g. backups) will delay others

Concurrency Problems

- In order to run two or more concurrent transactions, their operations must be interleaved
- Each transaction gets a share of the computing time
- This can lead to several problems
  - Lost updates
  - Uncommitted updates
  - Incorrect updates
- **All arise when isolation is broken**
  - i.e. we want a way to avoid this

Lost Update

T1	T2
Read(X)	
X = X - 5	
	Read(X)
	X = X + 5
Write(X)	
	Write(X)
COMMIT	
	COMMIT

- T1 and T2 both read X, both modify it, then both write it out
  - The net effect of both transactions should be no change to X
  - Only T2's change is seen however

Uncommitted Update

T1	T2
Read(X)	
X = X - 5	
Write(X)	
	Read(X)
ROLLBACK	
	X = X + 5
	Write(X)
	COMMIT

- T2 sees the change to X made by T1, but T1 is then rolled back
  - The change made by T1 is rolled back
    - i.e. the change would not actually be made
    - X goes back to what it was prior to the transaction
    - But T2 already used it
  - It should be as if that change never happened

Inconsistent Analysis

T1	T2
Read(X)	
X = X - 5	
Write(X)	
	Read(X)
	Read(Y)
	Sum = X + Y
Read(Y)	
Y = Y + 5	
Write(Y)	

- T1 doesn't change the sum of X and Y, but T2 records a change
  - T1 consists of two parts - take 5 from X then add 5 to Y
  - T2 sees the effect of the first change, but not the second

Resolutions for these problems

- We need a way to stop multiple transactions reading/updating the same data if it will cause a problem
- We need to know which things can be done concurrently (e.g. read + read) and which things cannot be done (e.g. read + write, write + write)
- We need a way to either stop something from changing (Locking, lecture 17), or to know it was changed and 'roll back' to restart if we hit a problem (Timestamping, lecture 18)
  - Each approach has advantages and disadvantages

This Lecture in Exams

Define a transaction in the context of database management

Explain how a DBMS uses a transaction log to recover from a system failure using ROLLBACK and ROLLFORWARD

Explain the difference between a system failure and a media failure

Next Lecture

- Concurrency
  - Locks and Resources
  - Deadlock
- Serialisability
  - Schedules of transactions
  - Serial and serialisable schedules
- Further reading
  - The Manga Guide to Databases, Chapter 5
  - Database Systems, Chapter 22