

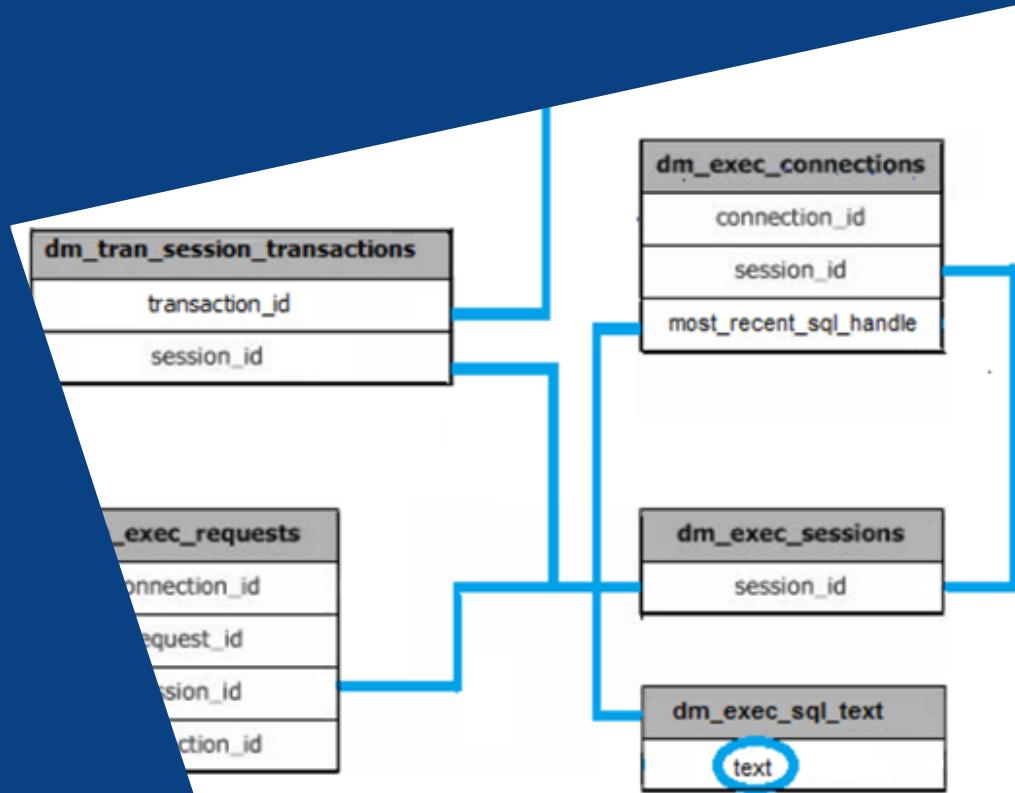


THE UNIVERSITY OF
MELBOURNE

Transactions and Concurrency

Database Systems & Information Modelling
INFO90002

Week 7 – Database transactions
Dr Tanya Linden
Dr Simon D'Alfonso
David Eccles





This Lecture Objectives

Why we need user-defined transactions

Properties of transactions

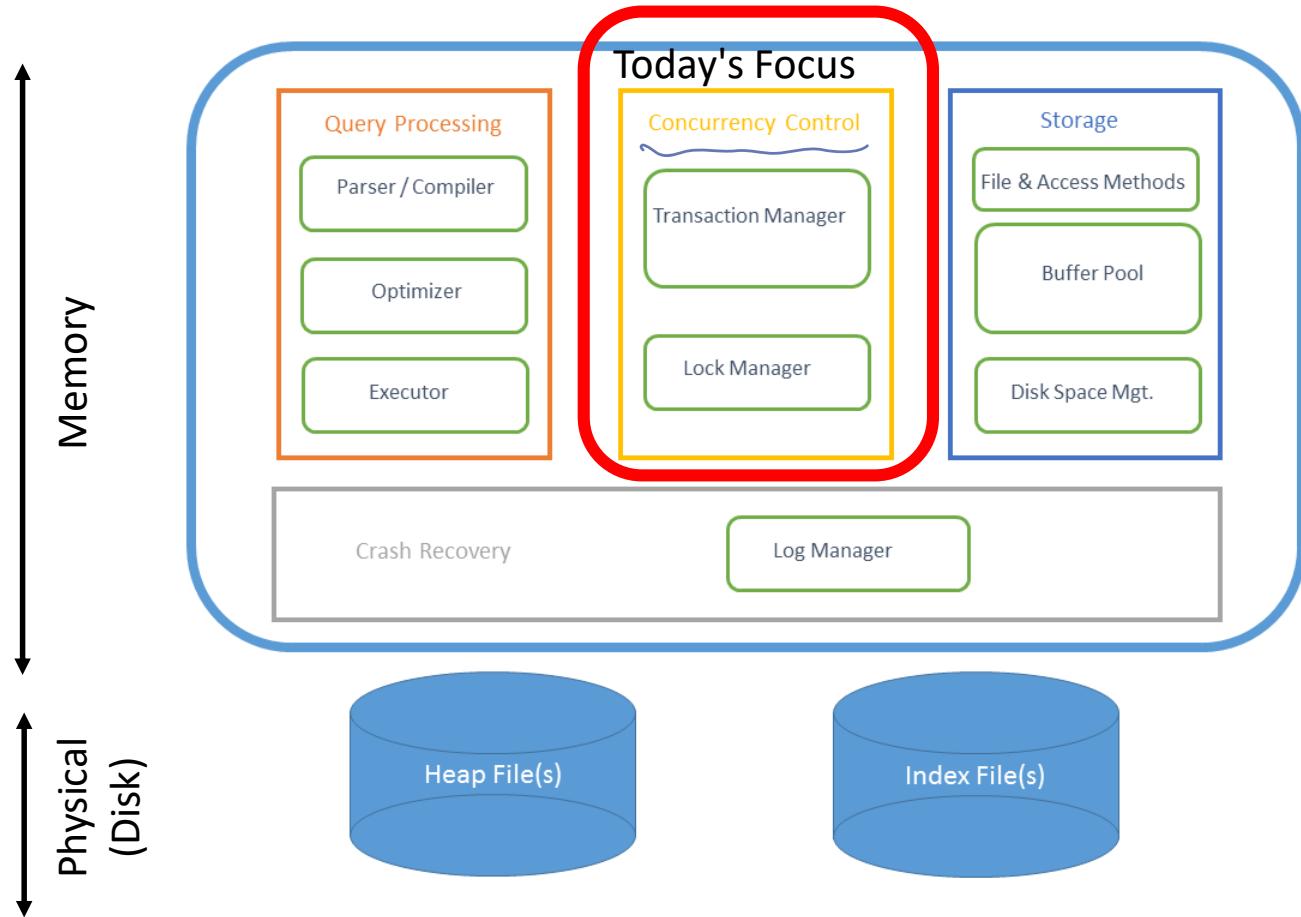
How to use transactions

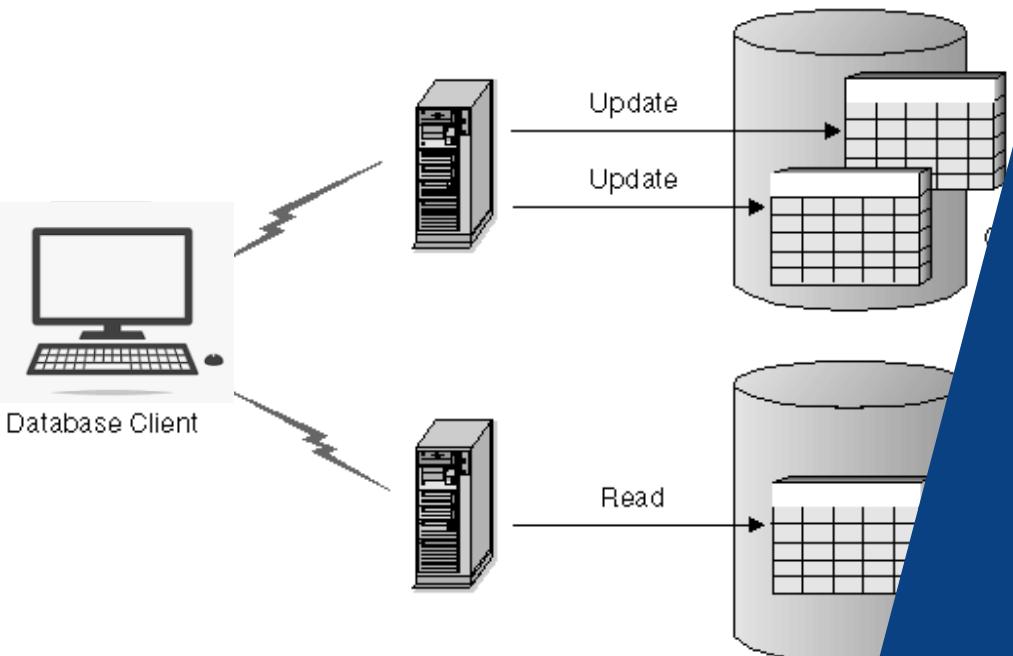
Concurrent access to data

Locking and deadlocking

Transaction's role in database recovery

Components of a DBMS





Transactions

Why we need database transactions



Business Transactions

Example “business transactions”:

- Placing order: Insert one row in *Order* table, then several in *OrderItem* table
- Funds transfer: Check amount < balance. If so, subtract amount from one row in *BankAccount* table, then add amount to another row. Record details in *Transactions* table
- Sending monthly statements: For all rows in *Customer* table, generate and send out monthly statements

Each requires several distinct database operations ...

- Example: move money from savings account to credit card account
 - Accept inputs from user (via ATM, internet banking or mobile app)
 - Select balance from savings account
 - Is there enough money to withdraw? If so:
 - Update savings account balance = balance – withdrawal
 - Update credit card balance = balance + withdrawal
 - If no errors encountered, end successfully



What is a (database) Transaction?

A logical unit of work that must either be entirely completed or aborted (indivisible, atomic)
DML statements are already atomic in MySQL*, SQL Server*

RDBMS also allows for *user-defined* transactions

These are a sequence of DML statements, such as

- a series of UPDATE statements to change values
- a series of INSERT statements to add rows to tables
- DELETE statements to remove rows

(usually don't delete row, but to active →
inactive)

Transactions will be treated as atomic

A successful transaction changes the database from one consistent state to another

- All data integrity constraints are satisfied

Ex:	account1	account2	total
	100	800	900
	if transfer money		unsistent

* Database specific, e.g. Oracle DML are not atomic transactions by default



Why do we need Transactions?

Transactions solve TWO problems:

1. users need the ability to define a unit of work
2. concurrent access to data by >1 user or program



A transaction should not include tasks that are not relevant to each other.

For example, *updating quantity in stock* and *updating prices for products to go on sale* would not be in the same transaction.

Problem 1: Unit of work

data manipulation statement



data definition language



- Single DML or DDL command (implicit transaction)

- Example: update 700 records, but database crashes after 200 records processed
- Restart server: you will find no changes to any records

~~• Changes are “all or none”~~

- Multiple statements (user-defined transaction)

- START TRANSACTION; (or, ‘BEGIN’)

- SQL statement;

- SQL statement;

- SQL statement;

- ...

- COMMIT; (commits the whole transaction, i.e. saves changes permanently)
 - Or ROLLBACK (to undo everything)

roll back every change.
↓
there is checkpoint, so you don't need to roll back every data.



SQL keywords: **BEGIN; START TRANSACTION; COMMIT, ROLLBACK**



Business transactions as units of work

Each transaction consists of several SQL statements, embedded within a larger application program

Transaction needs to be an indivisible unit of work

- “**Indivisible**” means that either the whole job gets done, or none gets done:
 - if an error occurs, we don't leave the database with the job half done, in an inconsistent state

In the case of an error:

- Any SQL statements already completed must be reversed
- Show an error message to the user
- When ready, the user can try the transaction again
- This is briefly annoying – but inconsistent data is disastrous



Transaction Properties (ACID)

Atomicity 原子性

- A transaction is treated as a single, indivisible, logical unit of work. All operations in a transaction must be completed; if not, then the transaction is aborted *If's all or none.*

Consistency

- Constraints that hold before a transaction must also hold after it. A transaction must not violate any integrity constraints or rules that govern the database.
- (multiple users accessing the same data see the same value)

Isolation

- Multiple transactions can occur concurrently without leading to the inconsistency of the database state. The intermediate states of transactions are invisible to other transactions until they are committed. The data required by an executing transaction cannot be accessed by any other transaction until the first transaction finishes. (This ensures Consistency)

Durability

- When a transaction is complete, the changes made to the database are permanent, even if the system fails. The updates are permanent and are stored in non-volatile memory, such as a hard disk or solid-state drive.



Exercise 1

A designer brands retailer has the following tables in their database: Product, ProductStockLevel, Action. Note there are other tables which are not relevant to this question. The Action table keeps track of all sales and returns.

Product					ProductStockLevel	
ProdID	Brand	Description	Retail Price	PurchasePrice	ProdID	QtyInStock
G43546	Gucci	Leather mid-heel pump	1080.00	450.00	G43546	12

Action						
ActionID	ActionDateTime	ActionType	ProdID	ProdQty	ProdCost	
1008	21/01/2021 10:24	Purchase	G43546	2	2160.00	
1026	23/01/2021 13:28	Return	G43546	-1	1080.00	

What is the problem with the following transaction?

START TRANSACTION;

```
INSERT INTO Product VALUES (G43547, "Gucci", "BAMBOO 1947 MINI TOP HANDLE BAG", 5840, 4300);  
UPDATE Product SET RetailPrice = RetailPrice * 0.95;
```

COMMIT;

↓ This should be two transaction. Shouldn't be



down together.

Exercise 2

A designer brands retailer has the following tables in their database: Product, ProductStockLevel, Action. Note there are other tables which are not relevant to this question. The Action table keeps track of all sales and returns.

Product					ProductStockLevel	
ProdID	Brand	Description	Retail Price	PurchasePrice	ProdID	QtyInStock
G43546	Gucci	Leather mid-heel pump	1080.00	450.00	G43546	12

Action						
ActionID	ActionDateTime	ActionType	ProdID	ProdQty	ProdCost	
1008	21/01/2021 10:24	Purchase	G43546	2	2160.00	
1026	23/01/2021 13:28	Return	G43546	-1	1080.00	

Create an SQL transaction that records a sale of the product G43546 with the quantity being 2.

Exercise 2 - solution

Product					ProductStockLevel	
ProdID	Brand	Description	Retail Price	PurchasePrice	ProdID	QtyInStock
G43546	Gucci	Leather mid-heel pump	1080.00	450.00	G43546	12

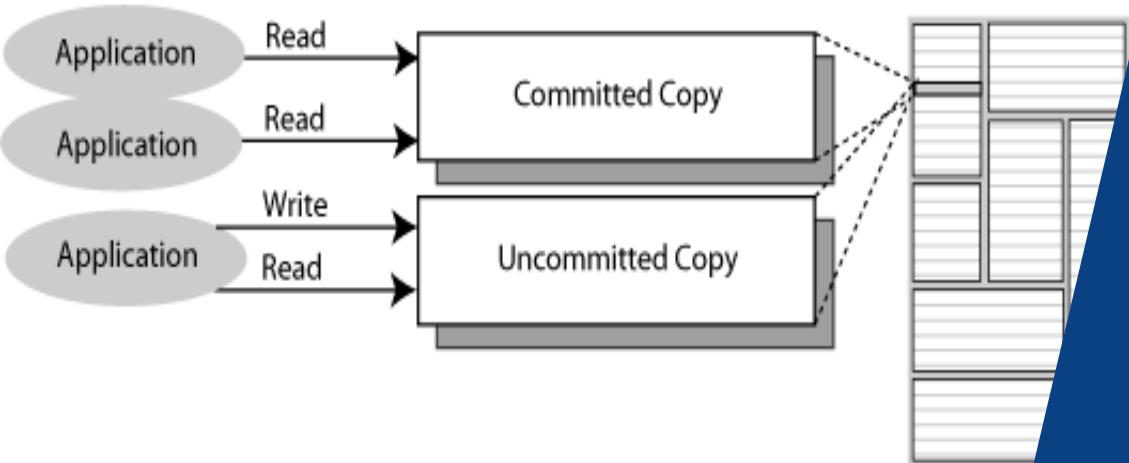
Action	ActionID	ActionDateTime	ActionType	ProdID	ProdQty	ProdCost
	1008	21/01/2021 10:24	Purchase	G43546	2	2160.00
	1026	23/01/2021 13:28	Return	G43546	-1	1080.00

Create an SQL transaction that records a sale of the product G43546 with the quantity being 2.

We need 2 updates: a new purchase activity in the Action table and stock update in ProductStockLevel

```

START TRANSACTION;
SET @qty = 2;          → variable
SET @price = (SELECT RetailPrice FROM Product WHERE ProdID='G43546');
SET @amount = @price*@qty;
INSERT INTO Action VALUES (1028, '2024-04-15', "Purchase", "G43546", @qty, @amount);
UPDATE ProductStockLevel SET QtyInStock = QtyInStock - @qty WHERE ProdID='G43546';
COMMIT;
  
```



Concurrency

What happens if there is more than one user?



Problem 2: Concurrent access

What happens if we have multiple users accessing the database at the same time...

multiple user may cause inconsistent
↓
so need to solve this problem

Concurrent execution of DML against a shared database

Note that the sharing of data among multiple users is where much of the benefit of databases derives – users communicate and collaborate via shared data

But what could go wrong?

Concurrency

- ① • lost updates
- ② • uncommitted data
- ③ • inconsistent retrievals

The Lost Update problem

Alice



Read account
balance
(balance = \$1000)

$t1a$

Withdraw \$100
(balance = \$900)

$t2a$
 $t1b$

Write balance
balance = \$900

$t3a$
 $t3b$

Bob



Read account
balance
(balance = \$1000)

Withdraw \$800
(balance = \$200)

Write balance
balance = \$200

Balance should be \$100

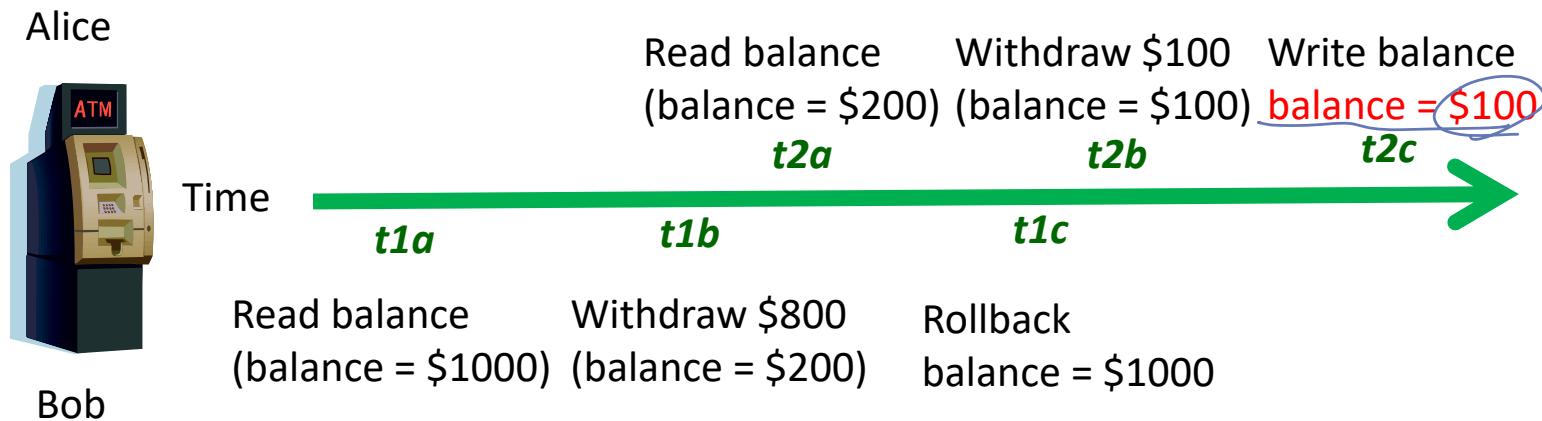
Two concurrent transactions are updating the same piece of data and one of the updates is lost because it is overwritten by the other transaction.

Example: Alice and Bob accessing the same account affecting its balance.

lost an update

The Uncommitted Data problem

Uncommitted data occurs when two transactions execute concurrently and the first is rolled back after the second has already accessed the uncommitted data
(Isolation property is violated)



Balance should be \$900

The Inconsistent Retrieval problem

Occurs when one transaction calculates some aggregate functions over a set of data, while other transactions are updating the data

- Some data may be read after they are changed and some before they are changed, yielding inconsistent results

while she is doing . Bob do ,

Alice	Bob
SELECT SUM(Salary) FROM Employee;	UPDATE Employee SET Salary = Salary * 1.01 WHERE EmpID = 33;
	UPDATE Employee SET Salary = Salary * 1.01 WHERE EmpID = 44;
(finishes calculating sum)	COMMIT;

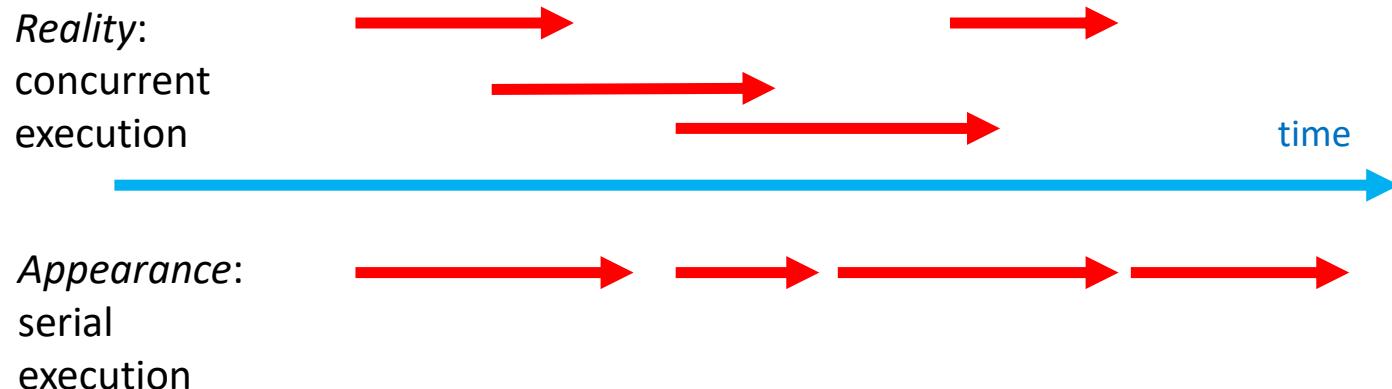
Example: Inconsistent Retrieval

Time	Trans- action	Action	Value	T1 SUM	Comment
1	T1	Read Salary for EmpID 11	10,000	10,000	
2	T1	Read Salary for EmpID 22	20,000	30,000	
3	T2	Read Salary for EmpID 33	30,000		
4	T2	Salary = Salary * 1.01			
5	T2	Write Salary for EmpID 33	30,300		
6	T1	Read Salary for EmpID 33	30,300	60,300	after update
7	T1	Read Salary for EmpID 44	40,000	100,300	before update
8	T2	Read Salary for EmpID 44	40,000		
9	T2	Salary = Salary * 1.01			
10	T2	Write Salary for EmpID 44	40,400		we want either before \$210,000 or after \$210,700
11	T2	COMMIT			
12	T1	Read Salary for EmpID 55	50,000	150,300	
13	T1	Read Salary for EmpID 66	60,000	210,300	but not this

Serializability

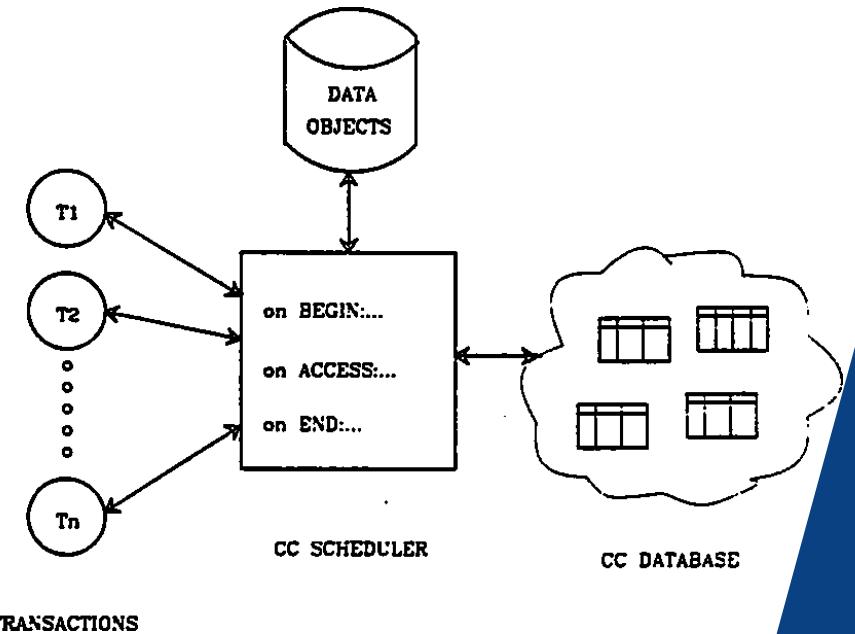
Transactions ideally are “serializable”

- Multiple, concurrent transactions *appear as if they were executed one after another*
- Ensures that the concurrent execution of several transactions yields consistent results



Scheduler – special DBMS process that schedules in which order operations within concurrent transactions execute, ensuring serializability and isolation.

True serial execution (i.e. no concurrency) is very expensive!



Concurrency Control Methods

Locking
Timestamps
Optimistic Method



Concurrency control methods

To achieve efficient execution of transactions, the DBMS creates a schedule of read and write operations for concurrent transactions

Interleaves the execution of operations, based on concurrency control algorithms such as locking and time stamping

Several methods of concurrency control

- 1 • *Locking* is the main method used
- 2 • Time Stamping
- 3 • Optimistic Methods

Concurrency Control with Locking

Lock

- Guarantees exclusive use of a data item to a current transaction
 - T1 acquires a lock prior to data access; the lock is released when the transaction is complete
 - T2 does not have access to data item currently being used by T1
 - T2 has to wait until T1 releases the lock
- Required to prevent another transaction from reading inconsistent data

Lock manager

- Responsible for assigning and policing the locks used by the transactions

Question: at what granularity should we apply locks?

Lock Granularity: options



Database-level lock

- Entire database is locked
- Good for batch processing but **unsuitable for multi-user DBMSs**
- T1 and T2 can not access the same database concurrently even if they use different tables
- (SQLite, Access)



Table-level lock

- Entire table is locked - as above but not quite as bad
- Good when updating most of the rows in a table
- **T1 and T2 can access the same database concurrently as long as they use different tables**
- Can cause bottlenecks, even if transactions want to access different parts of the table and would not interfere with each other
- **Not suitable for highly multi-user DBMSs**

Lock Granularity: options

T1: page 1

T2: on page two → it can process concurrency

only certain rows be locked
↓ rows separate to different pages.

③

Page-level lock (most frequently used)

- An entire disk page is locked (a table can span several pages and each page can contain several rows of one or more tables)

④

Row-level lock

- Allows concurrent transactions to access different rows of the same table, even if the rows are located in the same page
- Improves data availability but with high overhead (each row has a lock that must be read and written to)
- Currently the most popular approach (MySQL, Oracle)
- Higher overhead than Page level lock
- Automatically escalated to page level if several rows in the same page need to be locked.

But when on same page
↳ locked

Field-level lock (rarely use)

- Allows concurrent transactions to access the same row, as long as they access different attributes within that row
- A user may see some columns but not others (e.g. in online banking transactions listed but running balance not available)
- Most flexible lock but requires an extremely high level of overhead
- Not commonly used

lock certain column.

Types of Locks

Binary Locks

- has only two states: locked (1) or unlocked (0)
 - Every transaction will apply a lock and then terminate a lock (release the locked data)
- eliminates “Lost Update” problem
 - the lock is not released until the statement is completed
- considered too restrictive to yield optimal concurrency, as it locks even for two READs (when no update is being done)

The alternative is to allow both *Shared* and *Exclusive* locks

- often called Read and Write locks

Shared and Exclusive Locks

Exclusive lock *intended to write*

- Access is reserved for the transaction that locked the object
- Must be used when transaction intends to WRITE
- Granted if and only if no other locks are held on the data item
- In MySQL: “SELECT ... FOR UPDATE” or “LOCK TABLES ... WRITE”;

Shared lock (also called Read lock)

- Other transactions are also granted Read access
- Issued when a transaction wants to READ data, and no Exclusive lock is held on that data item
 - multiple transactions can each have a shared lock on the same data item if they are all just reading it
- In MySQL: “SELECT ... LOCK IN SHARE MODE” or “LOCK TABLES ... READ”;

<https://www.geeksforgeeks.org/lock-based-concurrency-control-protocol-in-dbms/>

Basic Lock Compatibility Matrix

1:32:0b

	S	X
S	✓	✗
X	✗	✗

S – shared
X - exclusive

If the transaction T1 is holding a shared lock in data item A, then the control manager can grant the shared lock to transaction T2 as compatibility is TRUE, but it cannot grant the exclusive lock.

In simple words, if transaction T1 is reading a data item A, then same data item A can be read by another transaction T2 but cannot be written by another transaction.

Similarly, if an exclusive lock (i.e. lock for read and write operations) is held on the data item in some transaction then no other transaction can acquire a lock, neither Shared nor Exclusive lock.

MySQL: FOR UPDATE OF versus LOCK IN SHARE MODE

- LOCK IN SHARE MODE does not prevent another transaction from reading the same row that was locked. FOR UPDATE prevents other locking reads of the same row



Locking Mechanisms

slow down

With simpler locking mechanisms, as depicted in the previous slide, writers block readers and other writers and readers block writers.

With more advanced multi-version concurrency control (MySQL is a DBMS that implements this), writers need not completely block readers and vice versa.

Version tracking via transaction id (sequential approach allows to know which one started earlier) + log of changes

Readers can see 'old' version of record while writer is doing an update.

More details (optional)

https://medium.com/@ajones1_999/understanding-mysql-multiversion-concurrency-control-6b52f1bd5b7e



Strict Two-Phase Locking (Strict 2PL)

Two Phases

1. A transaction T must *obtain all locks* before it can start reading or writing data (growing phase). *No unlocking can happen if another lock is required*.
2. All locks held by transaction T are released when the transaction is completed (shrinking phase).

If two transactions are accessing completely different parts of the database, they concurrently obtain the locks and proceed.

If two transactions T_1, T_2 are accessing the same object and one wants to modify it, their actions are ordered serially – all actions of one transaction T_1 must complete before the transaction T_2 can proceed.



ChatGPT

"Strict two-phase locking" 是一种严格的两阶段锁定协议，用于控制并发访问数据库中的数据。这种协议分为两个阶段：Growing Phase（增长阶段）和 Shrinking Phase（缩减阶段）。

- 在增长阶段，事务可以获取锁定，但不能释放任何锁定。这意味着事务可以获取它需要的所有锁定，但不能释放它们。
- 在缩减阶段，事务可以释放已经获取的所有锁定，但不能再获取任何新的锁定。这意味着事务在释放锁定后不能再获取新的锁定。

在这种协议下，事务必须严格地遵循这两个阶段。这种严格的控制有助于确保事务之间的隔离性，防止出现各种并发问题，如丢失更新、不可重复读和幻读。然而，它可能会导致事务等待时间增加，从而影响系统的性能。



Exercise

product → action → productstocklevel

Product					ProductStockLevel	
ProdID	Brand	Description	Retail Price	PurchasePrice	ProdID	QtyInStock
G43546	Gucci	Leather mid-heel pump	1080.00	450.00	G43546	12
Action	ActionID	ActionDateTime	ActionType	ProdID	ProdQty	ProdCost
	1008	21/01/2021 10:24	Purchase	G43546	2	2160.00
	1026	23/01/2021 13:28	Return	G43546	-1	1080.00

The SQL transaction below records a sale of the product G43546 with the quantity being 2.

Assuming that pessimistic locking with the two-phase locking protocol is being used with row-level lock granularity, create a chronological list of the locking, unlocking, and data manipulation activities that would occur during the complete processing of the transaction.

```
START TRANSACTION;
SET @qty = 2;
SET @price = (SELECT RetailPrice FROM Product WHERE ProdID='G43546');
SET @amount = @price*@qty;
INSERT INTO Action VALUES (1028, '2024-04-15', "Purchase", "G43546", @qty, @amount);
UPDATE ProductStockLevel SET QtyInStock = QtyInStock - @qty WHERE ProdID='G43546';
COMMIT;
```

Exercise - solution

```

START TRANSACTION;
SET @qty = 2;
SET @price = (SELECT RetailPrice FROM Product WHERE ProdID='G43546');
SET @amount = @price*@qty;
INSERT INTO Action VALUES (1028, '2024-04-15', "Purchase", "G43546", @qty, @amount);
UPDATE ProductStockLevel SET QtyInStock = QtyInStock - @qty WHERE ProdID='G43546';
COMMIT;
  
```

Time	Action
1	Lock Product row where ProdID = G43546
2	Lock Action
3	Lock ProductInStock row where ProdID = G43546
4	@qty = 2
5	@price = get RetailPrice from Product for ProdID = G43546
6	Insert row 1008 INTO Action
7	Update ProductInStock QtyInStock decreased by @qty for ProdID = G43546
8	Unlock Product row where ProdID = G43546
9	Unlock Action
10	Unlock ProductInStock row where ProdID = G43546

Transaction Isolation Levels



Isolation levels provide the level any given transaction is exposed to the actions of other executing transactions

Purpose – increase transaction concurrency

Isolation levels are

- *READ UNCOMMITTED*,
- *READ COMMITTED*,
- *REPEATABLE READ* and
- *SERIALIZABLE*

<https://www.geeksforgeeks.org/transaction-isolation-levels-dbsms/>

<https://dev.mysql.com/doc/refman/5.6/en/innodb-transaction-isolation-levels.html>

SET TRANSACTION ISOLATION LEVEL <level>;

MySQL can allow that uncommit data to be read.

Transaction Isolation Levels (cont)

1

READ UNCOMMITTED

- Does not obtain shared locks before reading, which increases performance but at the cost of data consistency
- May read uncommitted data from other transactions (**dirty read**). A transaction could be reading data that may become non-existent because the other transaction that was updating the data rolled-back the changes. *Sometime we will violate isolation*
- No isolation between transactions → *because we want faster process*

2

READ COMMITTED (default for Oracle, SQL server)

when T₁ process, not allow reading

- Only reads committed changes by other transactions
- No value written by T_1 is changed by another transaction until T_1 completes (commits)
- However, a value read by T_1 can be modified by other transactions T_n while T_1 is in progress.
- Read committed holds **exclusive locks (x)** before writing objects and **shared locks(s)** before reading objects.
will not allow reading

Example of Dirty Read

Timeline	Process 1	Process 2													
t1	<pre>SELECT * FROM account;</pre> <table border="1"> <thead> <tr> <th>Id</th><th>AccountType</th><th>Balance</th></tr> </thead> <tbody> <tr> <td>1</td><td>savings</td><td>200.00</td></tr> <tr> <td>2</td><td>credit</td><td>700.00</td></tr> <tr> <td>NULL</td><td>NULL</td><td>NULL</td></tr> </tbody> </table>	Id	AccountType	Balance	1	savings	200.00	2	credit	700.00	NULL	NULL	NULL		
Id	AccountType	Balance													
1	savings	200.00													
2	credit	700.00													
NULL	NULL	NULL													
t2		<pre>START TRANSACTION; /* transfer chosen amount from one account to another */ SET @amount = 100; UPDATE account SET balance = balance - @amount WHERE id = 1; UPDATE account SET balance = balance + @amount WHERE id = 2;</pre>	At what point in time does the dirty read happen in the above example?												
t3	<pre>SELECT * FROM account;</pre> <table border="1"> <thead> <tr> <th>Id</th><th>AccountType</th><th>Balance</th></tr> </thead> <tbody> <tr> <td>1</td><td>savings</td><td>100.00</td></tr> <tr> <td>2</td><td>credit</td><td>800.00</td></tr> <tr> <td>NULL</td><td>NULL</td><td>NULL</td></tr> </tbody> </table>	Id	AccountType	Balance	1	savings	100.00	2	credit	800.00	NULL	NULL	NULL	<p>/* note, there is no COMMIT in the transaction */</p> <p>uncommitted</p> <p>dirty read</p>	If roll back, it will make error (not the real data)
Id	AccountType	Balance													
1	savings	100.00													
2	credit	800.00													
NULL	NULL	NULL													

You

explain read uncommitted in database in traditional mandarin



ChatGPT

"Read uncommitted" 在数据库中是指一个事务可以读取另一个事务未提交的数据。简单来说，就是即使其他事务对数据进行了修改，未提交的修改也可以被当前事务读取到。这种隔离级别下，读取的数据可能是不一致的，因为它们可能会反映出其他事务中未完成的更改。

read committed



ChatGPT

"Read committed" 是数据库中的一种隔离级别，指的是一个事务只能读取到已经提交的数据。这意味着在读取数据时，事务会等待其他事务提交后才能看到其所做的更改，确保读取到的数据是稳定和一致的。相比于"Read uncommitted"，这种隔离级别下读取到的数据更加可靠，不会包含其他事务未提交的更改。



repeatable read



ChatGPT

"Repeatable read" 是数据库中的一种隔离级别，它在 "Read committed" 的基础上提供了更高的隔离性。在 "Repeatable read" 隔离级别下，一个事务在读取数据时可以确保在整个事务过程中多次读取同一组数据时，数据的一致性不会受到其他事务的影响。

换句话说，如果在一个事务中读取了某些数据，即使其他事务对这些数据进行了修改并提交，该事务在后续读取这些数据时仍然会看到一致的结果，即不会发生所谓的"幻读"问题。这种隔离级别下，事务会锁定读取的数据，以确保在事务完成之前不会受到其他事务的影响。



serializable

ChatGPT

"Serializable" 是数据库中最高级别的隔离级别之一，它提供了最严格的隔离性。在 "Serializable" 隔离级别下，事务之间的操作就好像是串行执行的，即使是在高并发环境下也会保证数据的一致性和可预测性。

在 "Serializable" 隔离级别下，事务会对读取的数据进行锁定，以确保其他事务不能修改或插入相关数据，直到当前事务完成。这样可以防止出现各种并发问题，如幻读、不可重复读等。

虽然 "Serializable" 提供了最高的数据隔离级别，但它也可能导致性能下降，因为它需要在事务之间加锁，从而可能导致事务的等待时间增加。因此，在选择隔离级别时，需要权衡数据的一致性和性能之间的平衡。



Transaction Isolation Levels (cont)

Repeatable Read (default for MySQL)

- The first read (SELECT) of the transaction T1 establishes a *snapshot* of results. *don't work back to table*
- Every next SELECT statement within the same transaction (T1) will return the same result for the *snapshot rows*. These SELECT statements are consistent also with respect to each other – even if the values have been modified by other transactions (T2,T3,...Tn) until a commit is issued.
- A transaction running in this isolation level does not take into account any changes to data made by other transactions, regardless of whether the changes have been committed or not.
- However, it could return phantom reads, i.e. additional rows that satisfy the repeated SELECT query.

Serializable - the highest degree of isolation

- Locks are obtained for both reads (R) and writes (W)
- Transaction T only reads committed transactions
- Similar to Repeatable Read, with the additional restriction that a row selected by transaction T1 cannot be changed by T2, T3, ... until T1 finishes.

Deadlock

Condition that occurs when two transactions wait for each other to unlock data

- T1 locks data item X, then wants Y
- T2 locks data item Y, then wants X
- Each waits to get a data item which the other transaction is already holding
- Could wait forever if not dealt with

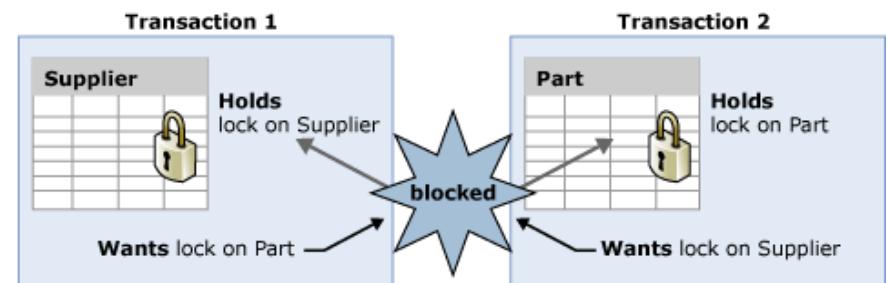
Can happen with several transactions being in a circular lock

- T1 waits for T2, T2 waits for T3, T3 waits for T1

Deadlock can only happen with exclusive locks

Deadlocks cannot happen with shared locks

↳ We can read



Dealing with Deadlocks

Techniques to avoid, detect and resolve deadlocks:

- **Timeout mechanisms**
- Prevention
 - preventing deadlocks by **careful design of transactions**, such as always acquiring locks in the **same order or releasing locks as soon as possible**.
- **Detection**
 - detection algorithms, which periodically scan the transaction log for deadlock cycles and then choose a transaction to abort to resolve the deadlock.
- **Avoidance**
 - Avoiding circular wait conditions
 - Using an algorithm to identify whether running some transactions could result in a deadlock

Deadlock demo

Two separate sessions

In order

- Tx1 Update row 3 (Green)
- Tx2 Update row 2 (White)
- Tx3 Update row 2 (Green)
- Tx4 Update row 3 (White)

Note: Only the session which detects the deadlock rolls back the transaction. The Green session still holds locks on row 2 and 3

```

decelles2 — mysql -u root -p — 80x24
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SELECT * FROM DEPT;
+-----+-----+-----+
| deptno | name   | location |
+-----+-----+-----+
| 10    | ACCOUNTING | NEW YORK |
| 20    | RESEARCH   | DALLAS   |
| 30    | SALES     | CHICAGO  |
| 40    | OPERATIONS | BOSTON   |
+-----+-----+-----+
4 rows in set (0.00 sec)

1 mysql> UPDATE DEPT set location = 'SYDNEY' where deptno = 30;
Query OK, 1 row affected (0.08 sec)
Rows matched: 1  Changed: 1  Warnings: 0

3 mysql> UPDATE DEPT set location = 'MELBOURNE' where deptno = 20;
Query OK, 1 row affected (13.07 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> [REDACTED]

decelles2 — mysql -u root -p — 71x24
mysql> USE SCOTT;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SELECT * FROM DEPT;
+-----+-----+-----+
| deptno | name   | location |
+-----+-----+-----+
| 10    | ACCOUNTING | NEW YORK |
| 20    | RESEARCH   | DALLAS   |
| 30    | SALES     | CHICAGO  |
| 40    | OPERATIONS | BOSTON   |
+-----+-----+-----+
4 rows in set (0.00 sec)

2 mysql> UPDATE DEPT set location = 'AUCKLAND' WHERE deptno = 20;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

4 mysql> Update dept set location = 'WELLINGTON' where deptno = 30;
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
mysql> [REDACTED]

```

Alternative concurrency control methods

Timestamp

- Assigns a global unique timestamp to each transaction
- Each data item accessed by the transaction gets the timestamp
- Thus for every data item, the DBMS knows which transaction performed the last read or write on it
- When a transaction wants to read or write, the DBMS compares its timestamp with the timestamps already attached to the item and decides whether to allow access

Optimistic

- Based on the assumption that the majority of database operations do not conflict
- Transaction is executed without restrictions or checking
- Then when it is ready to commit, the DBMS checks whether any of the data it read has been altered – if so, rollback

Logging transactions

If transaction cannot be completed, it must be aborted and any changes rolled back

To enable this, DBMS tracks all updates to data

This *transaction log* contains:

- a record for the beginning of the transaction
- for each SQL statement
 - operation being performed (update, delete, insert)
 - objects affected by the transaction
 - “before” and “after” values for updated fields
 - pointers to previous and next transaction log entries
- the ending (COMMIT) of the transaction

ChatGPT

"Logging transaction" 指的是数据库管理系统 (DBMS) 中记录事务操作的过程。这种记录是通过使用日志 (log) 来实现的，其中包含了对数据库进行更改的所有操作。

在执行事务期间，数据库系统会将事务执行的所有操作记录到事务日志中，包括插入、更新、删除等操作。这些日志记录在事务提交之前就已经完成，以确保在事务提交后，如果系统崩溃或发生故障，可以使用日志进行回滚或恢复。

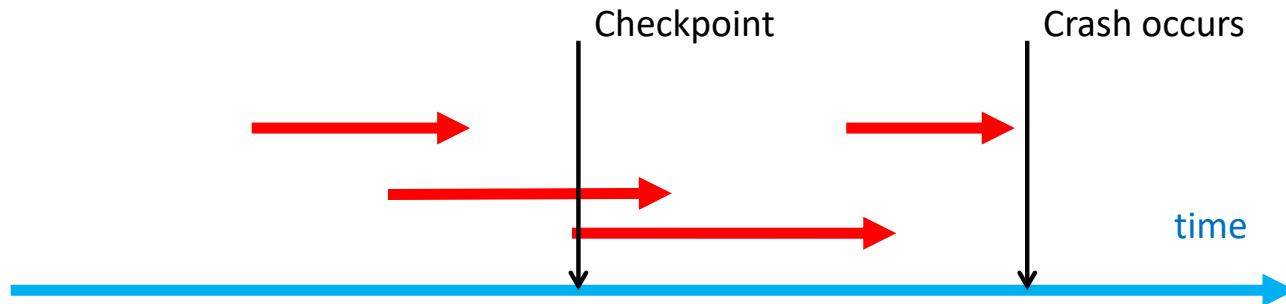
通过记录事务操作，日志可以提供数据的持久性和一致性，确保数据库的完整性。在系统崩溃或发生故障时，可以使用日志来还原数据库到事务提交之前的状态，从而避免数据丢失或损坏。



Transaction log

Also provides the ability to restore a corrupted database

If a system failure occurs, the DBMS will examine the log for all uncommitted or incomplete transactions and it will restore the database to its previous state



Example transaction log

TRL ID	TRX NUM	PREV PTR	NEXT PTR	OPERATION	TABLE	ROW ID	ATTRIBUTE	BEFORE VALUE	AFTER VALUE
341	101	Null	352	START	****Start Transaction				
352	101	341	363	UPDATE	PRODUCT	54778-2T	PROD_QOH	45	43
363	101	352	365	UPDATE	CUSTOMER	10011	CUST_BALANCE	615.73	675.62
365	101	363	Null	COMMIT	**** End of Transaction				
397	106	Null	405	START	****Start Transaction				
405	106	397	415	INSERT	INVOICE	1009			1009,10016, ...
415	106	405	419	INSERT	LINE	1009,1			1009,1, 89-WRE-Q,1, ...
419	106	415	427	UPDATE	PRODUCT	89-WRE-Q	PROD_QOH	12	11
423				CHECKPOINT					
427	106	419	431	UPDATE	CUSTOMER	10016	CUST_BALANCE	0.00	277.55
431	106	427	457	INSERT	ACCT_TRANSACTION	10007			1007,18-JAN-2004, ...
457	106	431	Null	COMMIT	**** End of Transaction				
521	155	Null	525	START	****Start Transaction				
525	155	521	528	UPDATE	PRODUCT	2232/QWE	PROD_QOH	6	26
528	155	525	Null	COMMIT	**** End of Transaction				
***** C *R*A* S* H *****									



What's examinable

Why we need user-defined transactions

Properties of transactions

ACID

How to use transactions

BEGIN; START TRANSACTION; COMMIT; ROLLBACK;

Concurrent access to data

Concurrent access strategies

Locking and deadlocking

Types of Locks (Binary; Shared)

Database recovery

Fundamentals of transaction recovery

* All material is examinable – these are the suggested key skills you would need to demonstrate in an exam scenario



THE UNIVERSITY OF
MELBOURNE

Thank you