Transaction Management

Dr Na Yao

Objectives

- Understand function and importance of transactions.
- Be able to explain properties of transactions.
- Understand and be able to explain the following for Concurrency Control
  - Meaning of serialisability.
  - How locking can ensure serialisability.
  - Deadlock and how it can be resolved.
  - How timestamping can ensure serialisability.
  - Optimistic concurrency control.
  - Granularity of locking.

Objectives

- Understand and be able to explain the following for Recovery Control
  - Some causes of database failure.
  - Purpose of transaction log file.
  - Purpose of checkpointing.

Transaction Support

- **Transaction:** Action, or series of actions, carried out by user or application, which reads or updates contents of database.

  - Logical unit of work on the database.
  - An application program can be thought as a series of transactions with non-database processing in between.
  - Transforms database from one consistent state to another, although consistency may be violated during transaction.

Properties of Transactions

Four basic (**ACID**) properties of a transaction are:

ie, all transactions are completed or non is completed

- <u>**A**tomicity</u> ‘All or nothing’ property
- <u>**C**onsistency</u> Must transform database from one valid state to another.
- <u>**I**solation</u> Partial effects of incomplete transactions should not be visible to other transactions.

  ie, transactions execute **independently** of one another
- <u>**D**urability</u> Effects of a committed transaction are permanent and must not be lost because of later failure.

Transaction example

- Transfer £50 from account A to account B

```
Read(A)
A = A - 50
Write(A)
Read(B)
B = B+50
Write(B)
```

- Atomicity - shouldn't take money from A without giving it to B
- Consistency - money isn't lost or gained
- Isolation - other queries shouldn't see A or B change until completion
- **Durability** - the money does not go back to A

Transaction Example (DreamHome)

```
read(staffNo = x, salary)
salary = salary * 1.1
write(staffNo = x, new_salary)
```

(a)

```
delete(staffNo = x)
for all PropertyForRent records, pno
begin
    read(propertyNo = pno, staffNo)
    if (staffNo = x) then
    begin
        staffNo = newStaffNo
        write(propertyNo = pno, staffNo)
    end
end
```

(b)

Transaction outcomes

Transaction outcomes

a) **Success** - transaction *commits* and database reaches a new consistent state.
   - Any changes made by the transaction should be saved
   - These changes are now visible to other transactions

b) **Failure** - transaction *aborts*, and database must be restored to consistent state before it started.
   - Such a transaction is *rolled back* or *undone*.
   - Any changes made by the transaction should be undone
   - It is now as if the transaction never existed


WHAT ABOUT
CONCURRENCY?

Concurrency

- Large databases are used by many people
  - Many transactions to be run on the database
  - It is desirable to let them run at the same time as each other
  - Need to preserve isolation
- If we don't allow for concurrency then transactions are run sequentially
  - Have a queue of transactions
  - Long transactions (e.g. backups) will make others wait for long periods

Concurrency problems

- In order to run transactions concurrently we interleave their operations
- Each transaction gets a share of the computing time
- This leads to several problems
  - Lost updates problem
  - Uncommitted updates problem
  - Incorrect analysis problem
- All arise because isolation is broken → violates ACID properties!!!

Lost Update Problem

- Successfully completed update is **overridden** by another user.
- While T1 (transaction 1) reads the value of an item, the value of that item is changed by T2 (transaction 2)
- This can lead to situations where one of the changes (updates) of one transaction are disregarded (lost)
- Example:
  - T1 withdrawing £10 from an account with X, initially £100.
  - T2 depositing £100 into same account.
  - Serially, final balance would be £190.

Lost Update Problem

Time	T1	T2	X
t_1		`begin_transaction`	100
t_2	`begin_transaction`	`read(X)`	100
t_3	`read(X)`	`X = X+100`	100
t_4	`X = X-10`	`write(X)`	200
t_5	`write(X)`	`commit`	90
t_6	`commit`		90

Solution: preventing T1 from reading X until after the update.

Uncommitted Dependency Problem

- Occurs when one transaction can see **intermediate results** of another transaction before it has committed.
- The reasons for not committing vary (cancelled by the user, connection problems, system crashes, etc.)
- Failure to commit causes a **rollback**, but other transactions are unaware of the rollback
- Example:
  - T4 updates X to £200 but it aborts, so X should be back at original value of £100.
  - T3 has read new value of X (£200) and uses value as basis of £10 reduction, giving a new balance of £190, instead of £90.

Uncommitted Dependency Problem

Time	T3	T4	X
t_1		`begin_transaction`	100
t_2		`read(X)`	100
t_3		`X = X+100`	100
t_4	`begin_transaction`	`write(X)`	200
t_5	`read(X)`	…	200
t_6	`X = X-10`	`rollback`	100
t_7	`write(X)`		190
t_8	`commit`		190

- **Solution:** preventing T3 from reading X until after T4 commits or aborts.

Inconsistent Analysis Problem

- Occurs when a transaction reads several values from the database but a second transaction **updates** some of them **during** the execution of the first.
- Sometimes referred to as *dirty read* or *unrepeatable read*.

- Example:
  - T6 is totaling balances of account X (£100), account Y (£50), and account Z (£25).
  - Meantime, T5 has transferred £10 from X to Z, so T6 now has wrong result (£10 too high).

Inconsistent Analysis Problem

Time	T5	T6	X	Y	Z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	Sum = 0	100	50	25	0
t_3	read(X)	read(X)	100	50	25	0
t_4	X = X-10	sum = sum + X	100	50	25	100
t_5	write(X)	read(Y)	90	50	25	100
t_6	read(Z)	sum = sum + Y	90	50	25	150
t_7	Z = Z + 10		90	50	25	150
t_8	write(Z)		90	50	35	150
t_9	commit	read(Z)	90	50	35	150
t_{10}		sum = sum + Z	90	50	35	185
t_{11}		commit	90	50	35	185

Solution: preventing T6 from reading X (and Z) until after T5 completed updates.

Need for Concurrency Control

- Transactions running concurrently may **interfere** with each other, causing various problems (lost updates problems etc.)

- **Concurrency control**: process of managing simultaneous operations on the database without having them interfere with one another.

Schedule

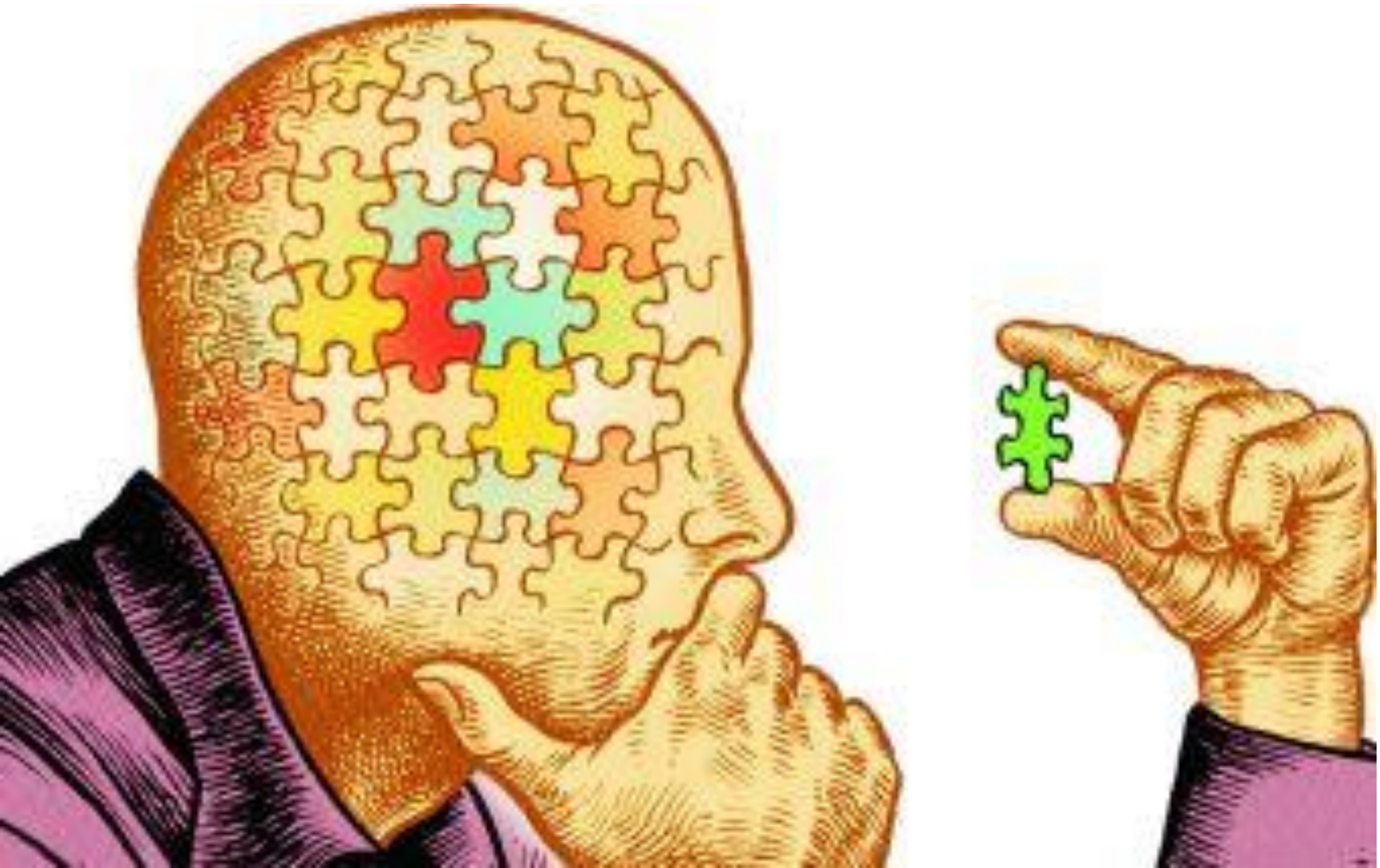
- A *schedule* is a sequence of the operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions.
- A *serial schedule* is a Schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions.

Schedule

- Serial schedules are guaranteed to avoid interference and keep the database consistent
  - No guarantee that results of all serial executions of a given set of transactions will be identical.

- However databases **need concurrent access** which means interleaving operations from different transactions

What can we do???



Serialisability

- Objective of a concurrency control protocol is to schedule transactions to avoid any interference.

- Could run transactions serially, but this limits degree of concurrency or parallelism in system.

- serialisability identifies those executions of transactions guaranteed to ensure consistency.

Serialisability

- **Nonserial** Schedule: Schedule where operations from set of concurrent transactions are **interleaved**. i.e., they overlap, transactions might conflict!

- ***Objective*** of serialisability is to find **nonserial** schedules that allow transactions to execute concurrently without interfering with one another.

- In other words, The objective of serialisability is to find nonserial schedules that are equivalent to some serial schedule. Such a schedule is called ***serialisable***.

Serialisability

- In serialisability, ordering of read/writes is important:
  1. If two transactions only read a data item, they do not conflict and order is not important.
  2. If two transactions either read or write completely separate data items, they do not conflict and order is not important.
  3. If one transaction writes a data item and another reads or writes same data item, order of execution is important.

Serialisable schedule?

Nonserial schedule

```
T1 Read(X)
T2 Read(X)
T2 Read(Y)
T1 Read(Z)
T1 Read(Y)
T2 Read(Z)
```

Serial Schedule

```
T2 Read(X)
T2 Read(Y)
T2 Read(Z)

T1 Read(X)
T1 Read(Z)
T1 Read(Y)
```

Conflict Serialisable Schedule

Nonserial Schedule

```
T1 Read(X)
T1 Write(X)
T2 Read(X)
T2 Write(X)
T1 Read(Y)
T1 Write(Y)
T2 Read(Y)
T2 Write(Y)
```

Serial Schedule

```
T1 Read(X)
T1 Write(X)
T1 Read(Y)
T1 Write(Y)

T2 Read(X)
T2 Write(X)
T2 Read(Y)
T2 Write(Y)
```

Conflict Serialisability

- Two transactions have a **conflict**:
  - ~~If they refer to different resources~~ **NO**
  - ~~If they are reads~~ **NO**
  - If at least one is a write and they use the same resource **YES**
- A schedule is *conflict serialisable* if transactions in the schedule have a conflict but the schedule is still serializable.

Concurrency Control Techniques

- Two basic concurrency control techniques:
  - Locking,
  - Timestamping
- Both are **conservative** approaches: delay transactions in case they conflict with other transactions.
- Optimistic methods assume conflict is rare and only check for conflicts at commit.

Locking

- Transaction uses locks to deny access to other transactions and so prevent incorrect updates.
- Two types of lock:
  - Shared lock (read-lock)
  - Exclusive lock (write-lock)
- Generally, a transaction must claim a *shared* (*read*) or *exclusive* (*write*) lock on a data item before read or write.

Locking - Basic Rules

- Read-lock allows several transactions simultaneously to read a resource (but no transactions can change it at the same time)
- Write-lock allows one transaction exclusive access to write to a resource. No other transaction can read this resource at the same time.
- Some systems allow transaction to upgrade read-lock to an exclusive-lock, or downgrade exclusive-lock to a shared-lock.

Locking

- Before reading from a resource a transaction must acquire a read-lock
- Before writing to a resource a transaction must acquire a write-lock
- Locks are released on commit/rollback
- A transaction may not acquire a lock on any resource that is write-locked by another transaction
- A transaction may not acquire a write-lock on a resource that is locked by another transaction
- If the requested lock is not available, transaction waits

Now let’s look at an example using locking

Time	T1	T2
$t_{1,2}$	`Write_lock(x); read(x)`	
t_3	`x = x + 100`	
$t_{4,5}$	`Write(x); unlock(x)`	
$t_{6,7}$		`Write_lock(x), read(x)`
$t_{11,12}$		`Write_lock(y); read(y)`
t_{13}		`y = y * 1.1`
$t_{14,15}$		`Write(y); unlock(y)`
$t_{16, 17}$	`Write_lock(y);read(y)`	`Commit`
t_{18}	`y = y - 100`	
$t_{19,20}$	`Write(y); unlock (y)`	
t_{21}	`commit`	

Is this a serializable schedule?

If T1 executes before T2

Time	T1	T2	X	Y
$t_{1,2}$	`Write_lock(x); read(x)`		100	400
t_3	`x = x + 100`		100	400
$t_{4,5}$	`Write(x); unlock(x)`		200	400
$t_{6,7}$	`Write_lock(y);read(y)`		200	400
t_8	`y = y - 100`		200	400
$t_{9,10}$	`Write(y); unlock (y)`		200	300
$t_{11,12}$	`commit`	`Write_lock(x), read(x)`	200	300
t_{13}		`x = x * 1.1`	200	300
$t_{14,15}$		`Write(x); unlock(x)`	220	300
$t_{16, 17}$		`Write_lock(y); read(y)`	220	300
t_{18}		`y = y * 1.1`	220	300
$t_{19,20}$		`Write(y); unlock(y)`	220	330
t_{21}		`Commit`	**220**	**330**

If T2 executes before T1

Time	T1	T2	X	Y
$t_{1,2}$		`Write_lock(x), read(x)`	100	400
t_3		`x = x * 1.1`	100	400
$t_{4,5}$		`Write(x); unlock(x)`	110	400
$t_{6,7}$		`Write_lock(y); read(y)`	110	400
t_8		`y = y * 1.1`	110	400
$t_{9,10}$		`Write(y); unlock(y)`	110	440
$t_{11,12}$	`Write_lock(x); read(x)`	`Commit`	110	440
t_{13}	`x = x + 100`		110	440
$t_{14,15}$	`Write(x); unlock(x)`		210	440
$t_{16, 17}$	`Write_lock(y);read(y)`		210	440
t_{18}	`y = y – 100`		210	440
$t_{19,20}$	`Write(y); unlock (y)`		210	340
t_{21}	`commit`		**210**	**340**

If T1 and T2 interleaved

Time	T1	T2	X	Y
$t_{1,2}$	`Write_lock(x); read(x)`		100	400
t_3	`x = x + 100`		100	400
$t_{4,5}$	`Write(x); unlock(x)`		200	400
$t_{6,7}$		`Write_lock(x), read(x)`	200	400
t_8		`x = x * 1.1`	220	400
$t_{9,10}$		`Write(x); unlock(x)`	220	400
$t_{11,12}$		`Write_lock(y); read(y)`	220	400
t_{13}		`y = y * 1.1`	220	400
$t_{14,15}$		`Write(y); unlock(y)`	220	440
$t_{16, 17}$	`Write_lock(y);read(y)`	`Commit`	220	440
t_{18}	`y = y - 100`		220	440
$t_{19,20}$	`Write(y); unlock (y)`		220	340
t_{21}	`commit`		**220**	**340**

Example - Incorrect Locking Schedule

- **Problem**: transactions release locks too soon, resulting in loss of total isolation and atomicity.

- **Solution:** to guarantee serialisability, we need an additional protocol concerning the positioning of the lock and unlock operations in every transaction.

2 PHASE LOCKING (2PL) TO THE RESCUE!

Two-Phase Locking (2PL)

- All locking operations precede unlock operation in the transaction.
- Principle of 2PL
  - Every transaction must lock an item (read or write) before accessing it
  - Once a lock has been released, no new items can be locked
- Two phases for transaction:

1. Growing phase - acquires all locks but cannot release any locks.

2. Shrinking phase - releases locks but cannot acquire any new locks.

Remember the Lost Update Problem?

- Successfully completed update is **overridden** by another user.
- While T1 reads the value of an item, the value of that item is changed by T2
- This can lead to situations where one of the changes (updates) of one transaction are disregarded (lost)

Lost Update Problem

Time	T1	T2	X
t_1		`begin_transaction`	100
t_2	`begin_transaction`	`read(X)`	100
t_3	`read(X)`	`X = X+100`	100
t_4	`X = X-10`	`write(X)`	200
t_5	`write(X)`	`commit`	90
t_6	`commit`		90

Preventing the Lost Update Problem using 2PL

Time	T1	T2	X
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(X)	100
t_3	write_lock(X)	read(X)	100
t_4	WAIT	X:=X+100	100
t_5	WAIT	write(X)	200
t_6	WAIT	commit/unlock(X)	200
t_7	read(X)		200
t_8	X:=X-10		200
t_9	write(X)		190
$t_1 0$	commit/unlock(X)		190

Remember the Uncommitted Dependency Problem?

- Occurs when one transaction can see **intermediate results** of another transaction before it has committed.
- The reasons for not committing vary (cancelled by the user, connection problems, system crashes, etc.)
- Failure to commit causes a **rollback**, but other transactions are unaware of the rollback

Uncommitted Dependency Problem

Time	T3	T4	X
t_1		`begin_transaction`	100
t_2		`read(X)`	100
t_3		`X = X+100`	100
t_4	`begin_transaction`	`write(X)`	200
t_5	`read(X)`	…	200
t_6	`X = X-10`	`rollback`	100
t_7	`write(X)`		190
t_8	`commit`		190

Preventing the Uncommitted Dependency Problem using 2PL

Time	T3	T4	X
t_1		begin_transaction	100
t_3		write_lock(X)	100
t_4		read(X)	100
t_5	begin_transaction	X:=X+100	100
t_4	write_lock(X)	write(X)	200
t_5	WAIT	rollback/unlock(X)	100
t_4	read(X)		100
t_6	X:=X-10		100
t_7	write(X)		90
t_8	commit/unlock(X)		90

Remember Inconsistent Analysis Problem?

- Occurs when a transaction reads several values from the database but a second transaction **updates** some of them **during** the execution of the first.
- Sometimes referred to as *dirty read* or *unrepeatable read*.

How to prevent the Inconsistent Analysis Problem using 2PL

Apply 2PL on the example below to prevent inconsistent analysis problem:

Time	T5	T6	X	Y	Z	sum
t_1		`begin_transaction`	100	50	25	
t_2	`begin_transaction`	`Sum = 0`	100	50	25	0
t_3	`read(X)`	`read(X)`	100	50	25	0
t_4	`X = X-10`	`sum = sum + X`	100	50	25	100
t_5	`write(X)`	`read(Y)`	90	50	25	100
t_6	`read(Z)`	`sum = sum + Y`	90	50	25	150
t_7	`Z = Z + 10`		90	50	25	150
t_8	`write(Z)`		90	50	35	150
t_9	`commit`	`read(Z)`	90	50	35	150
t_{10}		`sum = sum + Z`	90	50	35	185
t_{11}		`commit`	90	50	35	185

Does 2PL solve all the problems????

Nope…

- Cascading rollback
- Deadlocks

Cascading Rollback

- If **<u>every</u>** transaction in a schedule follows 2PL, schedule is serializable.
- However, problems can occur with the interpretation of when locks can be released.
- **Definiton:** a transaction (T1) causes a failure and a rollback must be performed. Other transactions dependent on T1's actions must also be rollbacked, thus causing a cascading effect.
- *One transaction's failure causes many to fail*.

Cascading Rollback

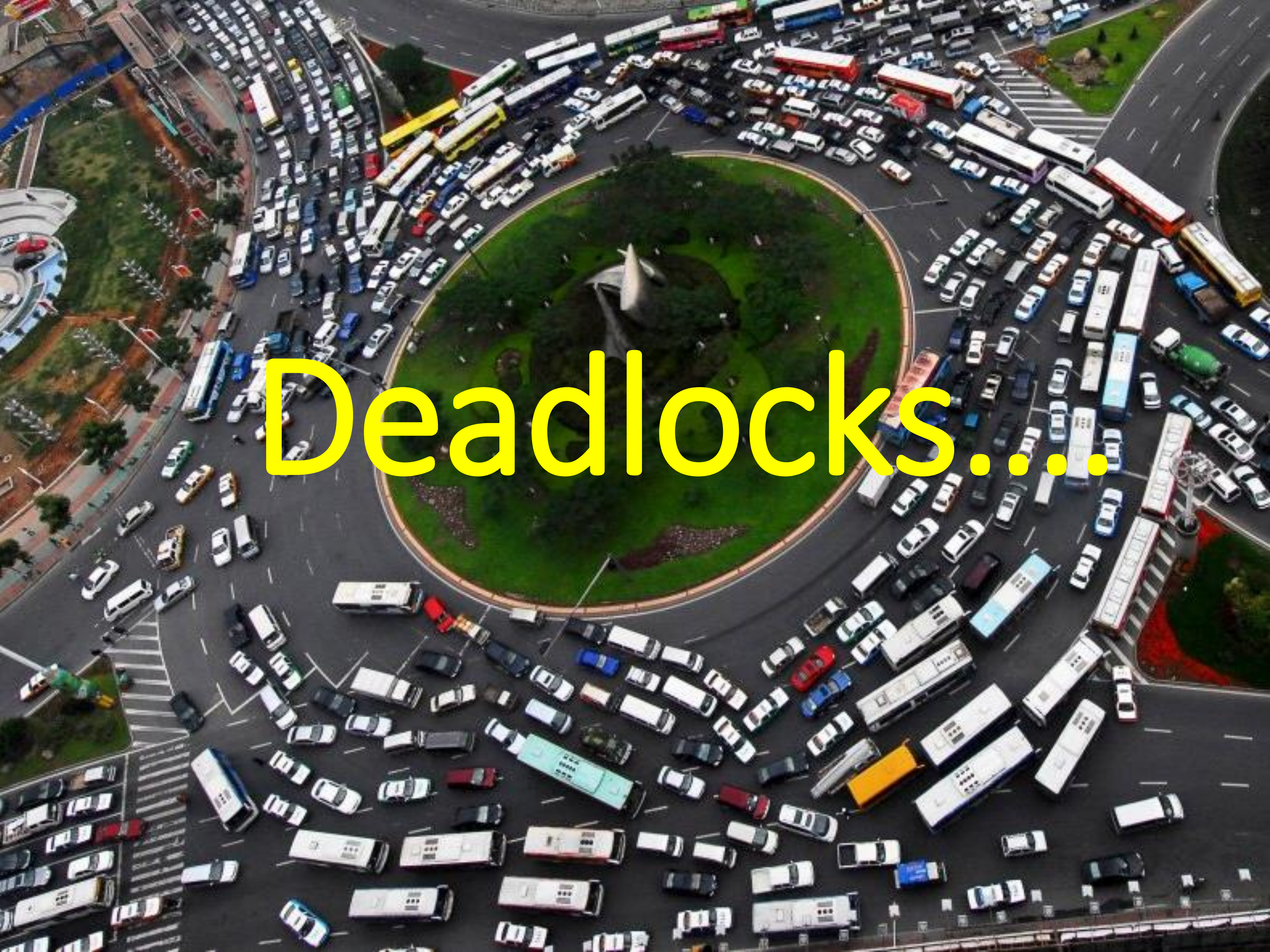
Time	T_{14}	T_{15}	T_{16}
t_1	begin_transaction		
t_2	write_lock($\mathbf{bal_x}$)		
t_3	read($\mathbf{bal_x}$)		
t_4	read_lock($\mathbf{bal_y}$)		
t_5	read($\mathbf{bal_y}$)		
t_6	$\mathbf{bal_x} = \mathbf{bal_y} + \mathbf{bal_x}$		
t_7	write($\mathbf{bal_x}$)		
t_8	unlock($\mathbf{bal_x}$)	begin_transaction	
t_9	⋮	write_lock($\mathbf{bal_x}$)	
t_{10}	⋮	read($\mathbf{bal_x}$)	
t_{11}	⋮	$\mathbf{bal_x} = \mathbf{bal_x} + 100$	
t_{12}	⋮	write($\mathbf{bal_x}$)	
t_{13}	⋮	unlock($\mathbf{bal_x}$)	
t_{14}	⋮	⋮	
t_{15}	rollback	⋮	
t_{16}		⋮	begin_transaction
t_{17}		⋮	read_lock($\mathbf{bal_x}$)
t_{18}		rollback	⋮
t_{19}			rollback

Cascading Rollback

- Cascading rollbacks are undesirable since they potentially lead to the undoing of a significant amount of work.

- How to prevent the cascading rollback with 2PL?

Postpone the release of *all locks* until end of the transaction.


Deadlocks....

Deadlock

- A deadlock occurs when two (or more) transactions are each waiting for locks held by the other to be released.

Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock(**bal**$_x$)	begin_transaction
t_3	read(**bal**$_x$)	write_lock(**bal**$_y$)
t_4	**bal**$_x$ = **bal**$_x$ − 10	read(**bal**$_y$)
t_5	write(**bal**$_x$)	**bal**$_y$ = **bal**$_y$ + 100
t_6	write_lock(**bal**$_y$)	write(**bal**$_y$)
t_7	WAIT	write_lock(**bal**$_x$)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

What can we do?

Deadlock

- Only one way to break deadlock: abort one or more of the transactions.
- Deadlock should be transparent to the user, so DBMS should restart transaction(s).
- General techniques for handling deadlock:
  - Timeouts.
  - Deadlock prevention.

Timeouts

- Transaction that requests lock will only wait for a system-defined period of time.
- If lock has not been granted within this period, lock request times out.
- In this case, DBMS assumes transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction.

Deadlock Prevention

- DBMS looks ahead to see if the transaction would cause a deadlock and never allows a deadlock to occur.
- Conservative 2PL: All data items have to be locked at the beginning of a transaction
  - Hard to predict what locks are needed
  - Low ‘lock utilisation’ - transactions can hold on to locks for a long time, but not use them much

So far…

- Concurrency control
- Serialisability
- How locking can ensure serialisability.
- Alternatives???

Timestamping

- Timestamp

  A unique identifier created by DBMS that indicates relative starting time of a transaction.

- Can be generated by using system clock at time transaction started, or by incrementing a logical counter every time a new transaction starts.

Timestamping

- Transactions are ordered globally so that older transactions (i.e., with smaller timestamps) get priority in the event of conflict.
- Read/write proceeds only if last update on that data item was carried out by an older transaction.
- Otherwise, transaction requesting read/write is restarted and given a new timestamp.
- No locks so no deadlock.

Timestamping

- There are also timestamps for data items:
  - read-timestamp - timestamp of last transaction to read item;
  - write-timestamp - timestamp of last transaction to write item.
- When transaction T asks to read/write a data item x, database will compare T's timestamp with the read/write timestamp of x, then decide whether to proceed T or abort/restart T.

Optimistic Concurrency Control Technique (OCC)

- Assumption: conflict is rare → more efficient to let transactions proceed without delays to ensure serialisability.
  - E.g., All transactions are readers.
- At commit, a check is made to determine whether conflict has occurred.
- If there is a conflict, transaction must be rolled back and restarted.
- Potentially allows greater concurrency than traditional protocols.

OCC Phases

1. **Read:**
   i. reads the values of all data items it needs from the database and stores them in local variables.
   ii. Writes in the local variables.
2. **Validation:** ensure there are no conflicts
3. **Write: (**if validation was successful) the updates in the local variable are applied to the public database

Some other aspects

- Granularity of data items
- Database recovery:
  - Logs
  - checkpoints

Granularity of Data Items

- Size of data items chosen as unit of protection by concurrency control protocol.
- Ranging from coarse to fine:
  - The entire database.
  - A file.
  - A page (or area or database space).
  - A record.
  - A field value of a record.

Granularity of Data Items

- Tradeoff:
  - coarser, the lower the degree of concurrency;
  - finer, more locking information that is needed to be stored.
- Best item size depends on the types of transactions.

Database Recovery

- Process of restoring database to a correct state in the event of a failure.
- Types of Failures:
  - System crashes, resulting in loss of main memory.
  - Power failures
  - Disk crashes, resulting in loss of parts of secondary storage.
  - Application software errors.
  - Natural physical disasters.
  - User mistakes.
  - Sabotage.

Transactions and Recovery

- Transactions represent basic unit of recovery.
- Recovery manager responsible for atomicity and durability.
- If failure occurs between commit and database buffers being flushed to secondary storage then, to ensure durability, recovery manager has to redo (rollforward) transaction’s updates.
- If transaction had not committed at failure time, recovery manager has to undo (rollback) any effects of that transaction for atomicity.

Recovery Facilities

- DBMS should provide following facilities to assist with recovery:
  - Backup mechanism, which makes periodic backup copies of database.
  - Logging facilities, which keep track of current state of transactions and database changes.
  - Checkpoint facility, which enables updates to database in progress to be made permanent.
  - Recovery manager, which allows DBMS to restore database to consistent state following a failure.

Log File

- Contains information about all updates to database:
  - Transaction records.
  - Checkpoint records.
- Often used for other purposes (for example, auditing).

Log File

- Transaction records contain:
  - Transaction identifier.
  - Type of log record, (transaction start, insert, update, delete, abort, commit).
  - Identifier of data item affected by database action (insert, delete, and update operations).
  - Before-image of data item.
  - After-image of data item.
  - Log management information.

Sample Log File

Tid	Time	Operation	Object	Before image	After image	pPtr	nPtr
T1	10:12	START				0	2
T1	10:13	UPDATE	STAFF SL21	(old value)	(new value)	1	8
T2	10:14	START				0	4
T2	10:16	INSERT	STAFF SG37		(new value)	3	5
T2	10:17	DELETE	STAFF SA9	(old value)		4	6
T2	10:17	UPDATE	PROPERTY PG16	(old value)	(new value)	5	9
T3	10:18	START				0	11
T1	10:18	COMMIT				2	0
	10:19	CHECKPOINT	T2, T3				
T2	10:19	COMMIT				6	0
T3	10:20	INSERT	PROPERTY PG4		(new value)	7	12
T3	10:21	COMMIT				11	0

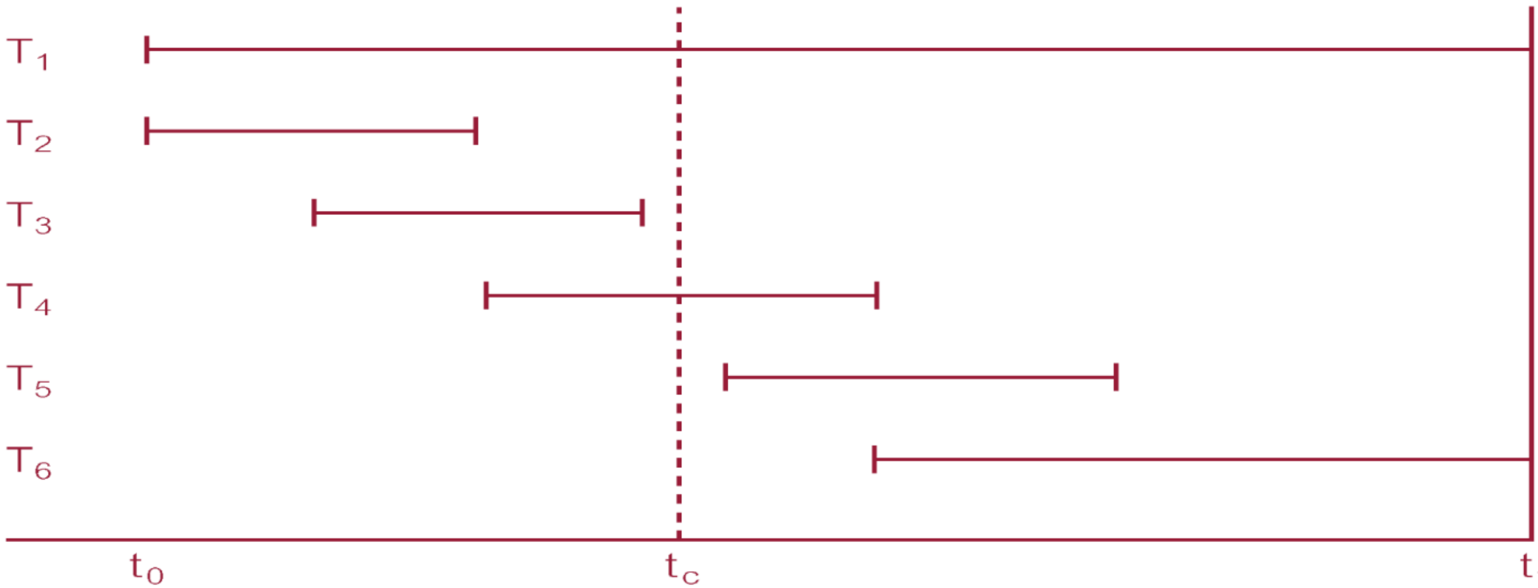
Checkpointing

- Checkpoint

  Point of synchronization between database and log file. All buffers are force-written to secondary storage.

- Checkpoint record is created containing identifiers of all active transactions.
- When failure occurs, redo all transactions that committed since the checkpoint and undo all transactions active at time of crash.

Example



- checkpoint at time t_c, changes made by T_2 and T_3 have been written to secondary storage.
- Thus:
  - only redo T_4 and T_5,
  - undo transactions T_1 and T_6.

Summary

- Transactions
- Schedules (serial, non-serial, serialisability)
- Concurrency problems (lost update problem, etc.)
- Conflicts and conflict serialisability
- Locking (2PL)
- Deadlocks
- Recovery (log file, checkpointing)