Databases – Transactions

Jörg Endrullis

VU University Amsterdam

Transactions :: Concurrency Anomalies

Example Transaction

A withdrawal of 100 euro causes the ATM to perform a **transaction in the bank's database**.

ATM Transaction

$balance \leftarrow \text{read}(account)$
$balance \leftarrow balance - 100$
$\text{write}(account, balance)$

The account is properly updated to reflect the new balance.

Interrupted Transactions

Money Transfer from Checking to Saving

Subtract money from source (checking) account:

1. $checking\_balance \leftarrow \text{read}(checking\_account)$
2. $checking\_balance \leftarrow checking\_balance - 500$
3. $\text{write}(checking\_account, checking\_balance)$

Add money to the target (saving) account:

4. $saving\_balance \leftarrow \text{read}(saving\_account)$
5. $saving\_balance \leftarrow saving\_balance + 500$
6. $\text{write}(saving\_account, saving\_balance)$

Interrupted Transactions

Money Transfer from Checking to Saving

Subtract money from source (checking) account:

1. $checking\_balance \leftarrow \text{read}(checking\_account)$
2. $checking\_balance \leftarrow checking\_balance - 500$
3. $\text{write}(checking\_account, checking\_balance)$

Add money to the target (saving) account:

4. $saving\_balance \leftarrow \text{read}(saving\_account)$
5. $saving\_balance \leftarrow saving\_balance + 500$

System crash!

6. $\text{write}(saving\_account, saving\_balance)$

Before the transaction gets to step 6, the system crashes.
(power outage, disk failure or software bug)

My money is lost!

Transactions should be **atomic** (executed fully or not at all).

Concurrent Access: Lost Update

My wife and I have **credit cards for the same account**.

What if we use the cards at the same time (**concurrently**)?

Concurrent ATM Transaction

me (withdraws 100)	my wife (withdraws 200)	state
$\underbrace{balance}_{=1200} \leftarrow \text{read}(account)$		1200
	$\underbrace{balance}_{=1200} \leftarrow \text{read}(account)$	1200
$\underbrace{balance}_{=1100} \leftarrow \underbrace{balance}_{=1200} - 100$		1200
	$\underbrace{balance}_{=1000} \leftarrow \underbrace{balance}_{=1200} - 200$	1200
	$\text{write}(account, \underbrace{balance}_{=1000})$	1000
$\text{write}(account, \underbrace{balance}_{=1100})$		1100

The update of my wife was lost during this execution. Lucky me!

This is known as **lost update anomaly**.

Concurrent Access: Inconsistent Read

Reconsider the **transfer from checking to saving account**:

Transaction 1

```
1. update Accounts
2. set    balance = balance - 500
3. where  customer = 1904
4.    and account_type = 'Checking'
```

Transaction 2

```
1. select sum(balance)
2. from   Accounts
3. where  customer = 1904
```

```
5. update Accounts
6. set    balance = balance + 500
7. where  customer = 1904
8.    and account_type = 'Saving'
```

Transaction 2 sees a temporary, **inconsistent database state**.

This is known as **inconsistent read anomaly**.

Concurrent Access: Dirty Read

Again, my wife and I are doing a transaction at the same time.

This time, **my transaction gets cancelled**!

Concurrent ATM Transaction

me	**my wife**	**state**
$balance \leftarrow \text{read}(account)$		1200
$balance \leftarrow balance - 100$		1200
$\text{write}(account, balance)$		1100
	$balance \leftarrow \text{read}(account)$	1100
	$balance \leftarrow balance - 200$	1100
abort		1200
	$\text{write}(account, balance)$	900

My wife's transaction has read the modified balance before my transaction was **rolled back** (i.e., the effects are undone).

This is known as **dirty read anomaly**.

Concurrency Anomalies

Lost Update Anomaly

The effects of one transaction are lost due to an uncontrolled overwrite performed by a second transaction.

Inconsistent Read

A transaction reads the partial result of another transaction.

Dirty Read

A transaction reads changes made by another transaction that is not yet committed (and might get aborted & rolled back).

Unrepeatable Read

A transaction reads a value which is afterwards changed by another transaction (before the former transaction is finished). So the first transaction operates on stale data.

ACID Properties

To prevent the mentioned problems. . .

Database management system ensures **ACID properties**

- **Atomicity:**
  transaction executes fully (commit) or not at all (abort)
- **Consistency:**
  transactions always leave the database in a consistent state where all defined integrity constraints hold
- **Isolation:**
  multiple users can modify the database at the same time but will not see each others partial actions
- **Durability:**
  once a transaction is committed successfully, the modified data is persistent, regardless of disk crashes

Transactions :: Transactions, Schedules and Serializability

Transactions

Formally, transactions are defined as:

A **transaction** is a list of actions.

The **actions** are

- reads (written R(O)) and
- writes (written W(O))

of database objects O.

Transactions end with **Commit** or **Abort**.

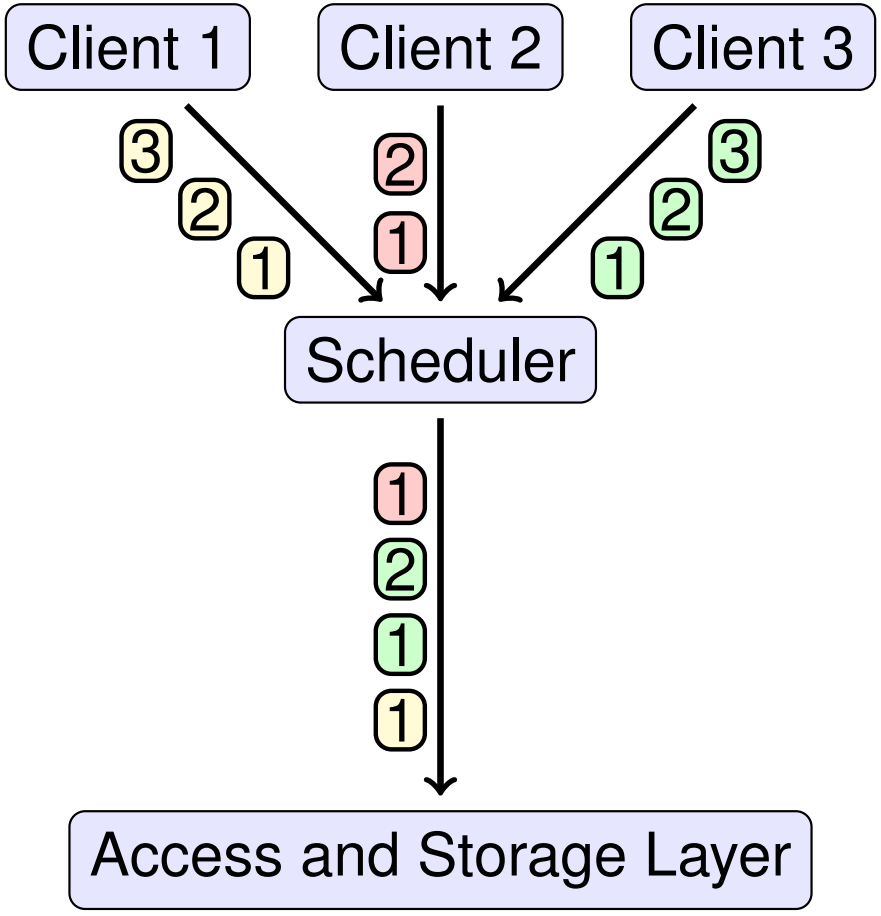
These are sometimes omitted if not relevant.

Example Transaction

$$T_1 \ : \ \mathrm{R(V)}, \mathrm{R(Y)}, \mathrm{W(V)}, \mathrm{W(C)}, \mathrm{Commit}$$

Scheduler

The **scheduler** decides the execution order of concurrent database access.



Schedules

A **schedule** is a list of actions from a **set of transactions**.

Intuitively, this is a plan on how to execute transactions.

The **order** in which 2 actions of a transaction T appear in a schedule must be the same order as they appear in T.

$$T_1 : \text{R(V) W(V)}$$
$$T_2 : \text{R(Y) W(Y)}$$

Which of the following is a schedule of these transactions?

S_1 :

T_1	R(V)			W(V)
T_2		R(Y)	W(Y)	

S_2 :

T_1	W(V)			R(V)
T_2		R(Y)	W(Y)	

Serializable Schedules

A schedule is **serial** if the actions of the different transactions are not interleaved; they are executed one after another.

S_1 :

T_1			R(V)	W(V)
T_2	R(Y)	W(Y)		

A schedule is **serializable** if its effect on the database is the same as that of some serial schedule.

Quiz Serializable Schedules

We usually only want to allow serializable schedules. Why?

Conflicts

Two actions in a schedule **conflict** if they:

- are from **different transactions**,
- involve the **same data item**, and
- **one of the actions is a write**.

T_1	R(Y)	W(Y)			W(X)
T_2			R(Y)	W(Z)	

There are several types of **conflicts**:

- **write read** (WR)
- **read write** (RW)
- **write write** (WW)

Such conflicts may cause a schedule to be not serializable.

WR Conflicts

There is a **WR conflict** between T_1 and T_2 if there is an item Y:

- T_1 writes Y and afterwards, T_2 reads Y

If T_1 has not committed this is a **dirty read**.

Find all WR conflicts in the following schedule

T_1			W(Y)		
T_2	R(V)			R(Y)	W(Z)
T_3		W(V)			

RW Conflicts

There is a **RW conflict** between T_1 and T_2 if there is an item Y:

- T_1 reads Y and afterwards, T_2 writes Y

This read becomes **unrepeatable**.

Find all RW conflicts in the following schedule

T_1			W(Y)			
T_2	R(V)			R(Y)	W(Z)	R(V)
T_3		W(V)				

WW Conflicts

There is a **WW conflict** between T_1 and T_2 if there is an item Y:

- T_1 writes Y and afterwards, T_2 writes Y

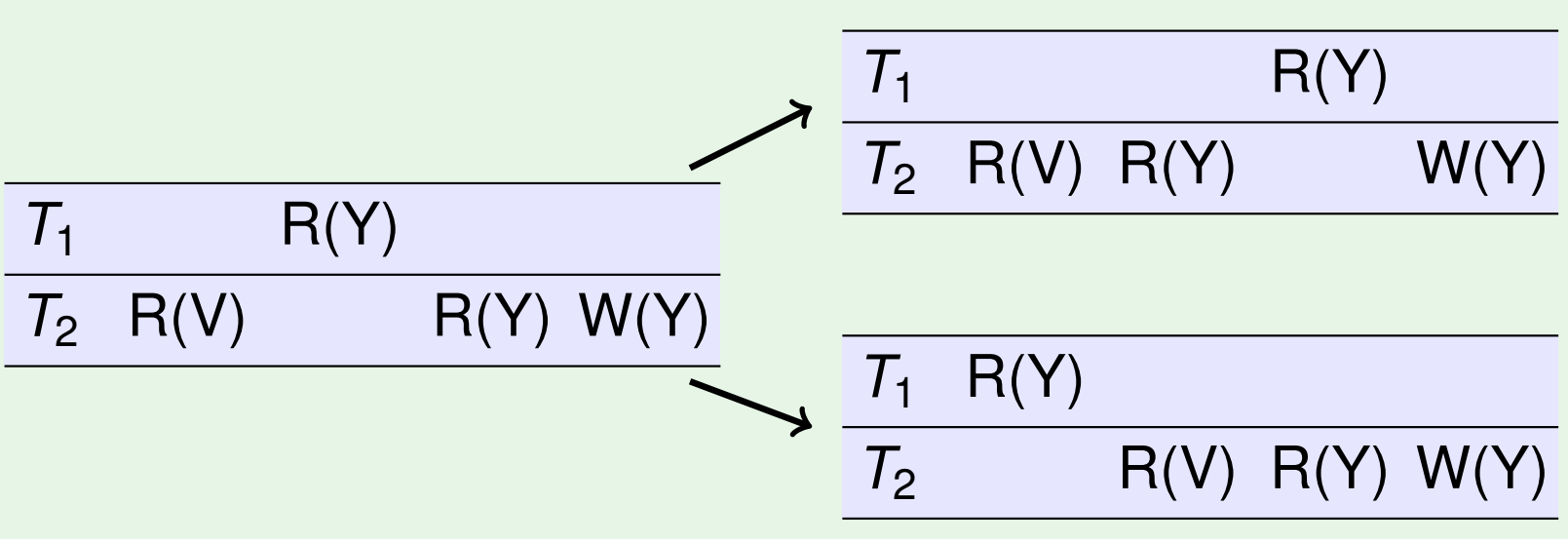
This write becomes **overwritten**.

Find all WW conflicts in the following schedule

T_1	W(Y)					
T_2		W(V)		R(Z)	W(Y)	W(Z)
T_3			W(V)			

Swapping Actions

We can **swap actions** (of different transactions) without changing the outcome, if the actions are **non-conflicting**.



Conflict Equivalent Schedules

Two schedules are **conflict equivalent** if they can be transformed into each other by a sequence of **swaps of non-conflicting, adjacent actions** (of different transactions).

Conflict Equivalent Schedules

Are any of the following schedules conflict equivalent?

S_1 :

T_1	W(V)		R(V)	W(V)
T_2		R(V)		

S_2 :

T_1	W(V)	R(V)		W(V)
T_2			R(V)	

S_3 :

T_1		W(V)	R(V)	W(V)
T_2	R(V)			

Schedules S_1 and S_2 are conflict equivalent (RR swap).

Conflict Serializable Schedules

A schedule is **conflict-serializable** if it is conflict equivalent to some serial schedule.

Conflict-serializable schedules are serializable (but not necessarily vice-versa).

Which of these schedules are conflict-serializable?

T_1	W(V)		W(V)
T_2		R(V)	

No

T_1	R(V)		W(V)
T_2		R(V)	

Yes

T_1			W(Y)		
T_2	R(V)			R(Y)	W(Z)
T_3		W(V)			

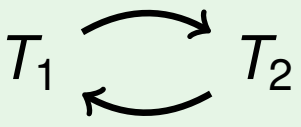
Yes

Checking Conflict-Serializability

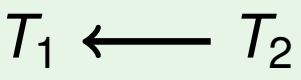
Given a schedule we can create a **precedence graph**:

- The graph has a node for each transaction.
- There is an edge from T_1 to T_2 if there is a conflicting action between T_1 and T_2 in which T_1 occurs first.

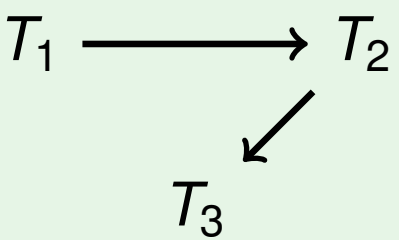
T_1	W(V)		W(V)
T_2		R(V)	



T_1	R(V)		W(V)
T_2		R(V)	



T_1		W(Y)			
T_2	R(V)			R(Y)	W(Z)
T_3			W(V)		

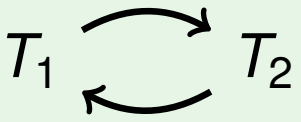


Checking Conflict-Serializability

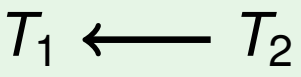
Checking Conflict-Serializability

A schedule is **conflict-serializable** if and only if there is **no cycle** in the precedence graph!

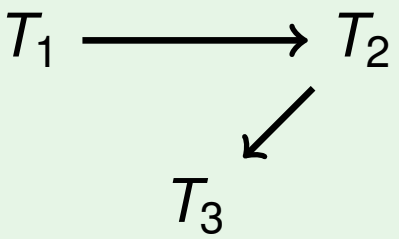
T_1	W(V)		W(V)
T_2		R(V)	



T_1	R(V)		W(V)
T_2		R(V)	



T_1			W(Y)		
T_2	R(V)			R(Y)	W(Z)
T_3		W(V)			

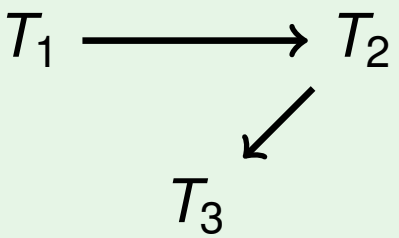


Schedules 2 and 3 have no cycles in their precedence graph. They are conflict serializable!

Checking Conflict-Serializability

If the precedence graph has no cycles, then an equivalent serial schedule is obtained by a **topological sort** of the precedence graph.

T_1		W(Y)			
T_2	R(V)			R(Y)	W(Z)
T_3			W(V)		



- There is an edge from T_1 to T_2 thus T_1 must be before T_2.
- There is an edge from T_2 to T_3 thus T_2 must be before T_3.

The sorting which fulfils these criteria is: T_1, T_2, T_3.

This yields the equivalent serial schedule:

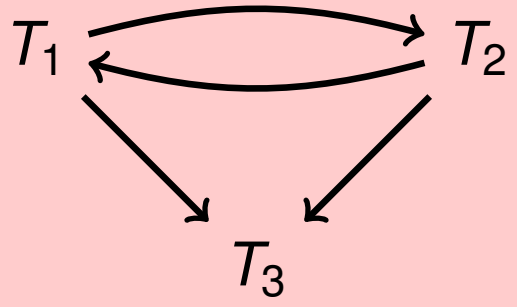
T_1	W(Y)				
T_2		R(V)	R(Y)	W(Z)	
T_3					W(V)

Example

Is the following schedule conflict-serializable?

T_1	R(V)		W(V)	
T_2		W(V)		
T_3				W(V)

The precedence graph is:



There is a cycle, thus not conflict-serializable!

However, the schedule is serializable: T_1, T_2, T_3!

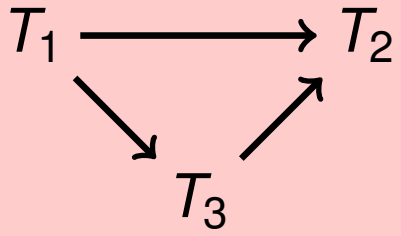
The writes of T_1 and T_2 are **blind writes**.

Example

T_1	R(V)				R(Z)		R(Y)	
T_2		R(Y)				W(V)		
T_3			W(V)					W(Z)

- Is this following schedule conflict-serializable?

The precedence graph is:



There is no cycle, thus the schedule is conflict-serializable!

T_1	R(V)	R(Z)	R(Y)					
T_2							R(Y)	W(V)
T_3					W(V)	W(Z)		

Transactions :: Strategies for Concurrency Control

Ensuring Serializability

So far, we have seen a sufficient condition that allows us to check whether a schedule is serializable.

But how to ensure serializability **during runtime**?

Challenge: the system does not know in advance which transactions will run and which items they will access.

Different **strategies** for ensuring serializability

1. **Pessimistic**
   - lock-based concurrency control (needs deadlock detection)
   - timestamp based concurrency control (not discussed here)
2. **Optimistic**
   - read-set/write-set tracking
   - validation before commit (transaction might abort)
3. **Multi-version techniques**
   - less concurrency control overhead for read-only queries

Transactions :: Two Phase Locking

Pessimistic: Lock-based Concurrency Control

Lock-based concurrency control

Transactions must **lock** objects before using them.

Types of **locks**

- **Shared lock (S-lock)** is acquired on Y before reading Y.

  Many transactions can hold a shared lock on Y.

- **Exclusive lock (X-lock)** is acquired on Y before writing Y.

  A transaction can hold an exclusive lock on Y only if no other transaction holds any lock on Y.

If a transaction has an X-lock on Y it can also read Y.

Pessimistic: Lock-based Concurrency Control

Schedule with explicit lock actions

T_1				X(B)	W(B)	U(B)			
T_2	S(A)	R(A)					X(B)	W(B)	U(A) U(B)

Here we use the following abbreviations:

- S(A) = shared lock on A
- X(A) = exclusive lock on A
- U(A) = unlock A, or if more precision is needed
  - US(A) = unlock shared lock on A
  - UX(A) = unlock exclusive lock on A

2 Phase Locking Protocol

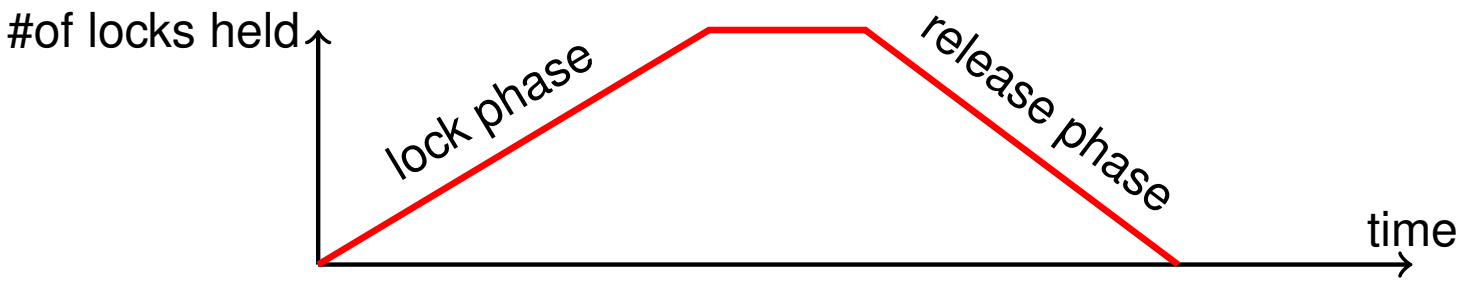
2 PL is the concurrency control protocol used in most DBMSs:

2 Phase Locking (2 PL)

Each transaction must get,

- an **S-lock** on an object **before reading** it, and
- an **X-lock** on an object **before writing** it.

A transaction cannot get new locks once it releases any lock.



Theorem

Any schedule that confirms to 2 PL is conflict-serializable.

2 Phase Locking Protocol: Examples

Which of the following conforms to the 2PL protocol?

T_1					X(B)	W(B)	U(B)			
T_2	S(A)	R(A)	U(A)					X(B)	W(B)	U(B)

No

T_1					X(B)	W(B)	U(B)			
T_2	S(A)	X(B)	R(A)					W(B)	U(A)	U(B)

No

T_1			X(B)	W(B)			U(B)			
T_2	S(A)	R(A)				U(A)		X(B)	W(B)	U(B)

No

T_1			X(B)	W(B)	U(B)					
T_2	S(A)	R(A)					X(B)	U(A)	W(B)	U(B)

Yes

Example: ATM Transaction

Concurrent ATM Transaction

Transaction 1	Transaction 2	DB state
slock(*account*)		1200
read(*account*)		
unlock(*account*)		
	slock(*account*)	
	read(*account*)	
	unlock(*account*)	
xlock(*account*) ↯		
write(*account*)		1100
unlock(*account*)		
	xlock(*account*) ↯	
	write(*account*)	1000
	unlock(*account*)	

↯ Once a lock has been released, no new lock can be acquired.

Example: ATM Transaction

To comply with the 2PL, the ATM transaction must not acquire new locks after a lock has been released.

ATM withdrawal with 2 Phase Locking

1. $\text{xlock}(account)$
2. $balance \leftarrow \text{read}(account)$
3. $balance \leftarrow balance - 100$
4. $\text{write}(account, balance)$
5. $\text{unlock}(account)$

Example: ATM Transaction

Concurrent ATM Transaction

Transaction 1	Transaction 2	DB state
xlock(*account*)		1200
read(*account*)		
	xlock(*account*)	
write(*account*)	Transaction	1100
unlock(*account*)	↓ blocked	
	xlock(*account*)	
	read(*account*)	
	write(*account*)	900
	unlock(*account*)	

Transaction 2 blocked until transaction 1 releases the lock.

Note: now both transactions are correctly executed!

Transactions :: Two Phase Locking - Deadlock Handling

Deadlocks

Like many lock-based protocols, 2PL has the risk of **deadlocks**.

A Deadlock Situation

Transaction 1	**Transaction 2**
xlock(A)	
⋮	xlock(B)
do something	⋮
⋮	do something
lock(B)	⋮
(waiting for T_2 to unlock B)	lock(A)
	(waiting for T_1 to unlock A)

Both transactions would wait for each other **indefinitely**.
We need to detect deadlocks!

Deadlock Handling via Wait-for-Graphs

Deadlock Detection via Wait-for-Graphs

The system maintains a **waits-for-graph**:

- Nodes of the graph are transactions.
- Edge $T_1 \rightarrow T_2$ means T_1 is blocked by a lock held by T_2.
  Hence T_1 waits for T_2 to release the lock.

The system checks periodically for **cycles** in the graph:

- If a cycle is detected, then the deadlock is **resolved** by **aborting** one or more transactions.

Selecting the **victim** is a challenge

- Aborting **young** transactions might lead to starvation. The same transaction may be cancelled again and again.
- Aborting **old** transactions may cause a lot of computational investment to be thrown away.

Deadlock Handling via Timeout

Deadlock Detection via Timeout

Let transactions block on a lock request only for a **limited time**.

After **timeout**, **assume** a deadlock has occurred and **abort T**.

Transactions :: Two Phase Locking - Cascading Rollbacks

Cascading Rollbacks

What is the problem here?

T_1	X(A)	S(B)	W(A)	U(A)														R(B)	U(B)
T_2									S(A)	R(A)		X(A)	W(A)		U(A)				

Assume that:

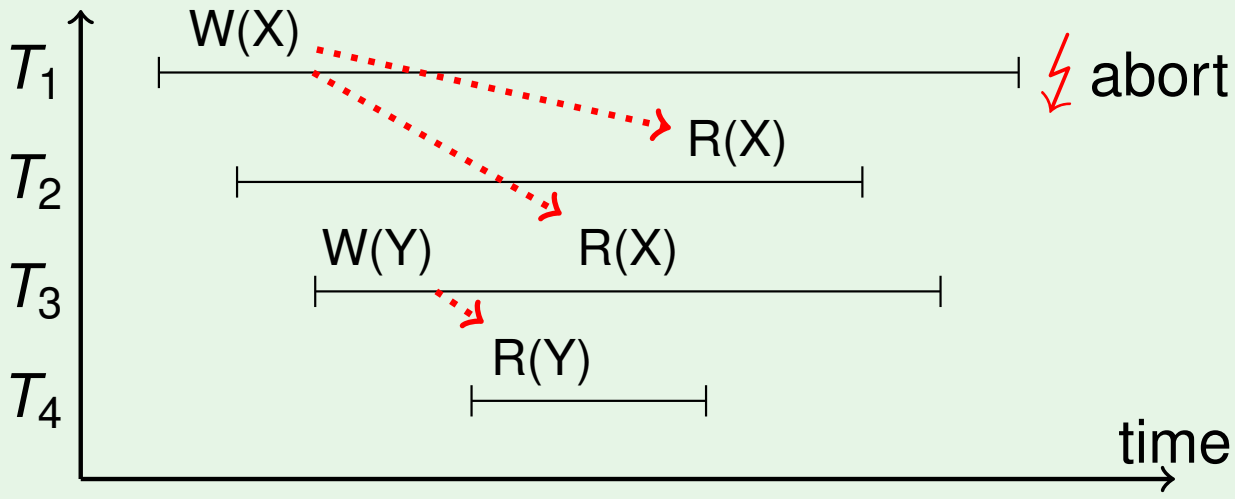
- T_1 is aborted (due to a conflict with another transaction)
- T_2 tries to commit

What is the problem here?

- T_2 has read a value written by T_1.
- Thus if T_1 is aborted, then T_2 needs to be aborted too.

  The commit will result in an abort.

Cascading Rollbacks



What happens here?

Note: T_2 and T_3 **cannot commit** until the fate of T_1 is known.

When T_1 aborts:

- T_2 and T_3 have already read data written by T_1 (dirty read)
- T_2 and T_3 need to be rolled back too (**cascading roll back**)

Since T_3 is aborted, T_4 needs to be aborted as well.

Cascades / Recoverable

Definition: Recoverable Schedule

Delay commits:

- If T_2 reads a value written by T_1, the commit of T_2 must be delayed until after the commit of T_1.

Schedules should always be recoverable!

Definition: Cascadeless Schedule

Delay reads: only read values produced by already committed transactions.

- If T_2 reads a value written by T_1, then the read is delayed until after the commit of T_1.

No dirty reads, thus abort (rollback) **does not cascade**!

All cascadeless schedules are recoverable.

Cascades / Recoverable

T_1	X(A)	S(B)	W(A)	U(A)						R(B)	U(B)	Commit
T_2					S(A)	R(A)	X(A)	W(A)	U(A)	Commit		

Is this schedule cascadeless? No

- If the commit of T_1 fails, then T_2 needs to be rolled back.

Is this schedule recoverable? No

- The commit of T_2 is not delayed until after commit of T_1.

Not Cascadeless, But **Recoverable**

T_1	X(A)	S(B)	W(A)	U(A)						R(B)	U(B)	Commit
T_2					S(A)	R(A)	X(A)	W(A)	U(A)			Commit

Transactions :: Strict & Preclaiming Two Phase Locking

Strict 2 Phase Locking Protocol

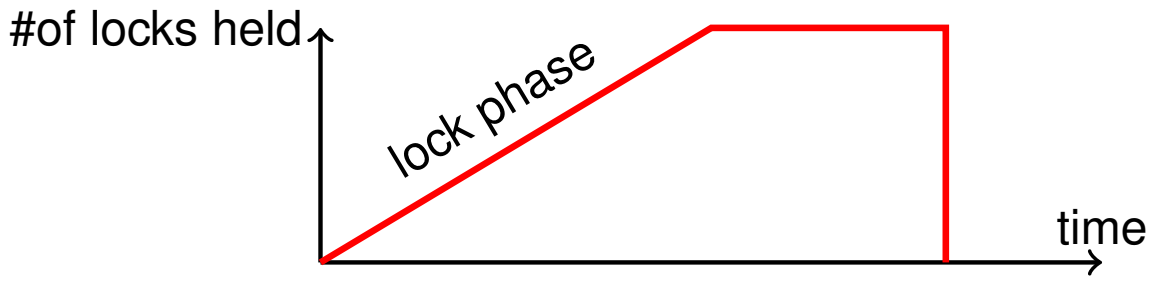
Strict 2 Phase Locking

Like in 2 PL, each transaction must get,

- an **S-lock** on an object **before reading** it, and
- an **X-lock** on an object **before writing** it.

But moreover:

- A transaction releases all locks only when the transaction is completed (i.e. when performing commit/rollback).

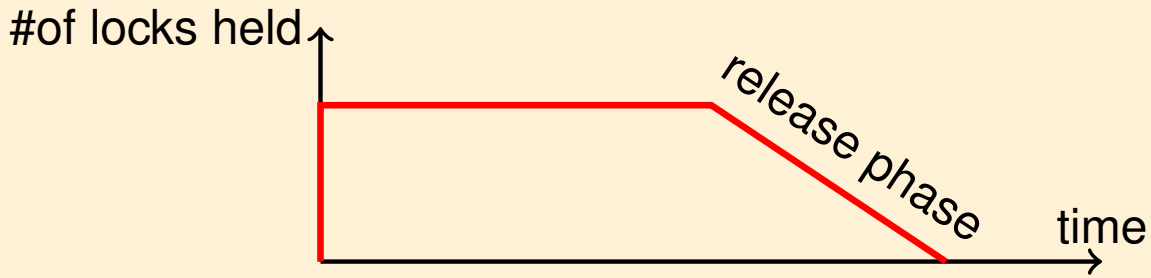


This protocol is **cascadeless**, avoids cascading aborts.

But there still are deadlocks!

Preclaiming 2 Phase Locking Protocol

Preclaiming 2 Phase Locking



All needed locks are declared at the beginning of the transaction.

Advantage: **No deadlocks!**
But rollbacks are cascading.

Disadvantage

Not applicable in multi-query transactions.

(Queries might depend on the results of the previous queries)

Example

T_1	R(V)					R(Z)			R(Y)	
T_2			R(Y)					W(V)		
T_3					W(V)					W(Z)

- Can it be achieved using 2 PL?
- Can it be achieved using Strict 2 PL?

Add the corresponding lock and unlock statements.

T_1	S(V)R(V)					S(Z)R(Z)			S(Y)R(Y)	
T_2			S(Y)R(Y)					X(V)W(V)		
T_3					X(V)W(V)					X(Z)W(Z)

Impossible with Strict 2 PL: T_1 must hold lock on V until commit.

Example

T_1	R(V)					R(Z)			R(Y)	
T_2			R(Y)					W(V)		
T_3					W(V)					W(Z)

- Can it be achieved using 2 PL?
- Can it be achieved using Strict 2 PL?

Add the corresponding lock and unlock statements.

T_1	S(VZY)	R(V)	U(V)				R(Z)		R(Y)	
T_2				S(Y)	R(Y)			X(V)	W(V)	
T_3						X(V)	W(V)			X(Z) W(Z)

Impossible with Strict 2 PL: T_1 must hold lock on V until commit.

Example

T_1	R(V)					R(Z)			R(Y)	
T_2			R(Y)					W(V)		
T_3					W(V)					W(Z)

- Can it be achieved using 2 PL?
- Can it be achieved using Strict 2 PL?

Add the corresponding lock and unlock statements.

T_1	S(VZY)	R(V)	U(V)			R(Z)			R(Y)	
T_2				S(Y)	R(Y)			X(V)	W(V)	
T_3					X(V)	W(V)	U(V)			X(Z) W(Z)

Impossible with Strict 2 PL: T_1 must hold lock on V until commit.

Example

T_1	R(V)					R(Z)			R(Y)	
T_2			R(Y)					W(V)		
T_3					W(V)					W(Z)

- Can it be achieved using 2 PL?
- Can it be achieved using Strict 2 PL?

Add the corresponding lock and unlock statements.

T_1	S(VZY)	R(V)	U(V)			R(Z)			R(Y)	
T_2				S(Y)	R(Y)			X(V)	W(V)	
T_3					X(V)	W(V)	X(Z)	U(V)		W(Z)

Impossible with Strict 2 PL: T_1 must hold lock on V until commit.

Example

T_1	R(V)					R(Z)			R(Y)	
T_2			R(Y)					W(V)		
T_3					W(V)					W(Z)

- Can it be achieved using 2 PL?
- Can it be achieved using Strict 2 PL?

Add the corresponding lock and unlock statements.

T_1	S(VZY)	R(V)	U(V)					R(Z)	U(Z)				R(Y)	U(Y)
T_2					S(Y)	R(Y)						X(V)	W(V)	U(VY)
T_3							X(V)	W(V)		X(Z)	U(V)		W(Z)	U(Z)

Impossible with Strict 2 PL: T_1 must hold lock on V until commit.

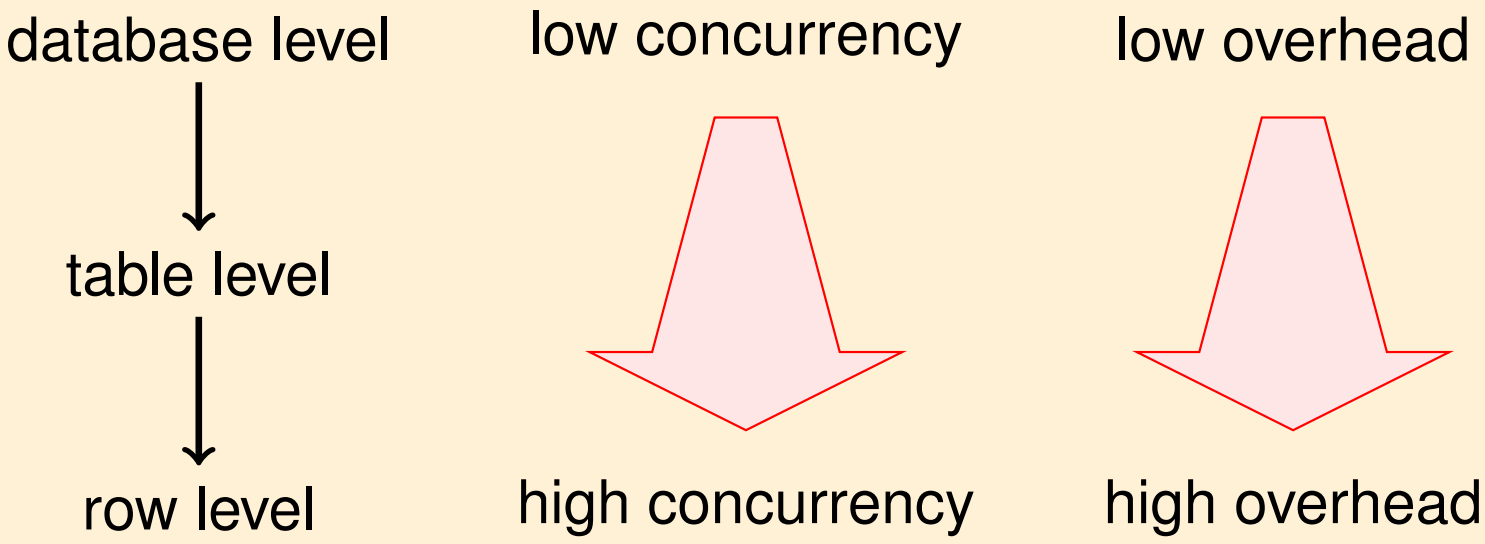
This schedule is 2 PL !

Can it be achieved using Preclaiming 2 PL? No

Transactions :: Granularity of Locking

Granularity of Locking

The granularity of locking is a trade-off



Idea: multi-granularity locking...

Multi-Granularity Locking

Decide the granularity of locks held **for each transaction**.

Depending on the characteristics of the queries.

For example, acquire a **row lock** for:

Q_1: row-selecting query (`id` is a key)

```
select *
from   Customers
where  id = 42
```

For example, acquire a **table lock** for:

Q_2: table scan query

```
select *
from   Customers
```

How do such transactions know of each others locks?

Note that the locks are on different granularity levels!

Intention Locks

Databases use an additional type of locks: **intention locks**.

- Lock mode **intention share (IS)**
- Lock mode **intention exclusive (IX)**

Before introducing S (or X) lock, first IS (or IX) locks on all coarser levels of granularity.

Extended lock conflict matrix

	S	X	IS	IX
S		x		x
X	x	x	x	x
IS		x		
IX	x	x		

Intention Locks

Multi-granularity Locking Protocol

- Before a granule *g* can be locked in S (or X) mode, the transaction has to obtain an IS (or IX) lock on **all coarser** granularities that contain *g*.
- After all intention locks are granted, the transaction can lock g in the announced mode.

The query Q_1 would for example:

- obtain an **IS lock** on the **database**
- obtain an **IS lock** on the **table** `Customers`

Afterwards obtain an **S lock** on the **row** with `id = 42`.

The query Q_2 would for example:

- obtain an **IS lock** on the **database**

Afterwards obtain an **S lock** on the **table** `Customers`.

Detecting Conflicts

Now suppose an updating query comes in:

Q_3: update request

```
update Customers
set    name = 'Pete'
where  id = 17
```

The query Q_3 will try to:

- obtain an **IX lock** on the **database**
- obtain an **IX lock** on the **table** `Customers`

Afterwards obtain an **X lock** on the **row** with `id = 17`.

- compatible with Q_1
  (no conflict between IS of Q_1 and IX lock of Q_3 on table)
- incompatible with Q_2
  (conflict between S lock of Q_2 and IX lock of Q_3 on table)

Transactions :: Isolation Levels

Isolation Levels

Some degree of **inconsistency** may be acceptable for specific applications to gain **increased concurrency & performance**.

E.g. accept inconsistent read anomaly and be rewarded with improved concurrency. Relaxed consistency guarantees can lead to increased throughput!

SQL-92 Isolation Levels & Consistency Guarantees

isolation level	dirty read	non-repeat. read	phantom rows
`read uncommitted`	possible	possible	possible
`read committed`	**not possible**	possible	possible
`repeatable read`	**not possible**	**not possible**	possible
`serializable`	**not possible**	**not possible**	**not possible**

Different DBMS support different levels of isolation.

Phantom Row Problem

Transaction 1	Transaction 2	Effect
scan relation R		T_1 locks all rows
	insert new row into R	T_2 locks new row
	commit	T_2's lock released
scan relation R		reads **new** row too!

Both transactions properly follow the 2 PL protocol!

Nevertheless, T_1 observed an effect caused by T_2.

- Isolation violated!
- Cause of the problem: T_1 **can only lock existing rows**.

Solutions

1. **multi-granularity locking** (locking the table)
2. **declarative locking**: key-range or predicate locking

Isolation Levels via Locking

Basic idea: use variations of strict 2 PL.

SQL-92 Isolation Levels

- **`read uncommitted`** (also `dirty read` or `browse`)

  Only write locks are acquired. Any row read may be concurrently changed by other transactions.

- **`read committed`** (also `cursor stability`)

  Read locks are held for as long as the application cursor sits on a particular, **current row**. Write locks as usual. Rows may be changed between repeated reads.

- **`repeatable read`**

  Strict 2 PL locking. Nevertheless, a transaction may read **phantom rows** if it executes an aggregation query twice.

- **`serializable`**

  Strict 2 PL + multi-granularity locking. **No phantom rows**.

Transactions :: SQL Transaction Control

SQL Transaction Control

SQL Transaction Control

- `set autocommit on/off`
  - `on`: each SQL query is one transaction
- `start transaction`
- `commit`
- `rollback`
- `set transaction isolation level ...`

Many applications do not need full serializability

Selecting a weaker, yet acceptable isolation level is important part of **database tuning**.

Transactions :: Optimistic Concurrency Control

Optimistic Concurrency Control

Up to now we have seen **pessimistic** concurrency control:

- assume that transactions **will conflict**
- protect the database integrity by **locks** and lock protocols

Optimistic concurrency control

- hope for the best
- let transactions freely proceed with read/write operations
- only **at commit, check that no conflicts have happened**

Rationale:

- non-serializable conflicts are not that frequent
- **save the locking overhead**
- only invest effort if really required

Optimistic Concurrency Control

Under **optimistic concurrency control**, transactions proceed in **three phases**:

1. **Read phase:**

   Execute transaction, but do **not** write data back to disk. Collect updates in the transaction's **private workspace**.

2. **Validation phase:**

   When the transaction wants to **commit**, the DBMS test whether its execution was correct (only acceptable conflicts happened). If not, **abort** the transaction.

3. **Write phase:**

   Transfer data from private workspace into database.

Note: phases 2 and 3 are performed in a uninterruptible critical section (also called val-write phase).

Optimistic Concurrency Control: Validation

Validation is typically implemented by maintaining:

- a **read set RS(T_k)** (attributes read by T_k), and
- a **write set WS(T_k)** (attributes written by T_k)

for every transaction T_k.

Backward-oriented optimistic concurrency control (BOCC)

On commit, compare T_k against all **committed** transactions T_i.
Check **succeeds** if

T_i committed before T_k started **or** $\mathrm{RS}(T_k) \cap \mathrm{WS}(T_i) = \varnothing$

If the check fails, abort T_k.

Forward-oriented optimistic concurrency control (FOCC)

On commit, compare T_k against all **running** transactions T_i.
Check **succeeds** if

$$\mathrm{WS}(T_k) \cap \mathrm{RS}(T_i) = \varnothing$$

If the check fails, abort T_k or T_i.

Transactions :: Multiversion Concurrency Control

Multiversion Concurrency Control

Is this schedule serializable?

T_1	R(X)	W(X)			R(Y)	W(Z)
T_2			R(X)	W(Y)		

No

But what if we had a copy of the old values available?

Then we could do:

Multi-version

T_1	R(X)	W(X)			R(Y-old)	W(Z)
T_2			R(X)	W(Y)		

This is can be serialised to:

T_1	R(X)	W(X)	R(Y)	W(Z)		
T_2					R(X)	W(Y)

Multiversion Concurrency Control: Snapshot Isolation

Multiversion Concurrency Control

Multiple versions of the data are stored. The isolation level determines what version a transaction sees.

Common Isolation Level: **Snapshot Isolation**

Each transaction sees a consistent snapshot of the database that corresponds to the state at the moment it started.

Read-only transactions never need to be blocked!

- might see **outdated**, but **consistent** version of the data
- as if the entire query happened at the moment it started

With snapshot isolation:

- Read-only transactions do not have to lock anything!
- Transactions **conflict if they write the same object**:
  - Pessimistic concurrency control: only writes are locked
  - Optimistic concurrency control: only write-sets interesting

Multiversion Concurrency Control: Snapshot Isolation

Disadvantages:

- versioning requires **space** and **management overhead**
- update transactions still need concurrency control!

Snapshot isolation does not guarantee serializability! But

- the anomalies
  - dirty read,
  - unrepeatable read,
  - phantom rows

  do not occur.
- however, write skew occurs!

Snapshot isolation is used in the Oracle SQL Server.
(Oracle has no real serializability).

Snapshot Isolation: Write Skew Anomaly

Write Skew Anomaly

- Constraint: $X + Y < 2$
- Initially: $X = 0$ and $Y = 0$
- $T_1 : X = X + 1$; it sees $X = 1$ and $Y = 0$ and commits
- $T_2 : Y = Y + 1$; it sees $X = 0$ and $Y = 1$ and commits
- T_1 and T_2 have an empty write intersection (**no conflict**).
- End result: $X = 1$ and $Y = 1$ $\lightning$ $X + Y \geq 2$

This problem does not occur if this is a `check` constraint comparing values of the same row since the finest locking granularity are rows.

Therefore write skew anomalies occur with complex assertions that involve multiple tuples.

Transactions :: Optimising Performance

Optimising Performance

Suppose you have a typical log of queries for your database.

For each query in the log:

- Analyse average time and variance for this type of query.
  - Long delays or frequent aborts may indicate contention.
- Is it is a read-only or updating query?
  - Compute the read-sets and write-sets.
  - Will it require row or table locks? Shared or exclusive?

How do read- and write-sets of the different queries intersect?

- What is the chance of conflicts? (delays/rollbacks)

Once you understand your query workload, you might improve performance by:

- Rewriting queries to have smaller read- and write-sets.
- Change scheduling of queries to reduce contention.

  E.g. rewrite applications to do large aggregation queries at night.
- Use a different isolation level for the queries.