Databases

Jörg Endrullis

VU University Amsterdam

2015

Example Transaction

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

ATM Transaction

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

The account is properly updated to reflect the new balance.

Concurrent Access

- My wife has a credit card for the same account.
- What if we use the cards at the same time (concurrently)?

Concurrent ATM Transaction

me	**my wife**	**DB state**
$bal \leftarrow \text{read}(acct)$		1200
	$bal \leftarrow \text{read}(acct)$	1200
$bal \leftarrow bal - 100$		1200
	$bal \leftarrow bal - 200$	1200
	$\text{write}(acct, bal)$	1000
$\text{write}(acct, bal)$		1100

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

Interrupted Transactions (e.g. the Plug is Pulled)

I want to **transfer money from checking to saving**:

Money Transfer

```
// Subtract money from source (checking) account
1. $checking\_balance \leftarrow \text{read\_balance}(checking\_account\_no)$
2. $checking\_balance \leftarrow checking\_balance - 500$
3. $\text{write\_balance}(checking\_account\_no, checking\_balance)$

// Add money to the target (saving) account
4. $saving\_balance \leftarrow \text{read\_balance}(saving\_account\_no)$
5. $saving\_balance \leftarrow saving\_balance + 500$
6. System crash!
7. $\text{write\_balance}(saving\_account\_no, saving\_balance)$
```

Before the transaction gets to step 7, the execution is interrupted. (power outage, disk failure or software bug)

My money is lost!

ACID Properties

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

Anomalies: Lost Update

Lost Update Anomaly

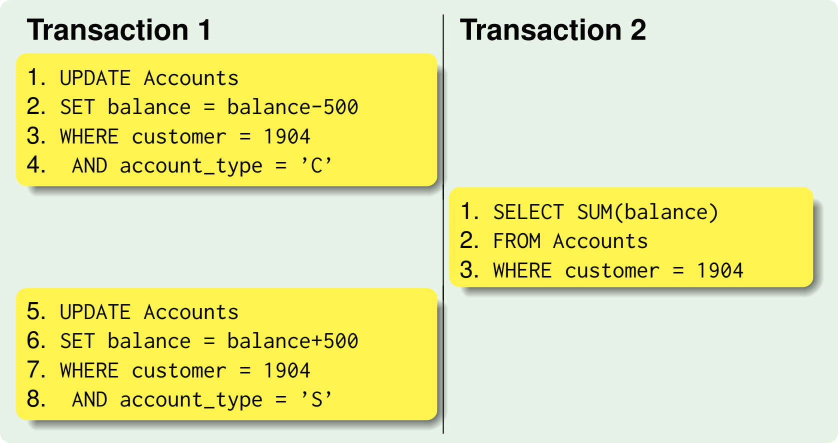
The effects of one transaction are lost due to an uncontrolled overwrite performed by a second transaction.

Inconsistent / Dirty Read

A transaction reads the partial result of another transaction.

Anomalies: Inconsistent Read

Reconsider the money transfer example, now in SQL syntax:



Transaction 1

```
UPDATE Accounts
SET balance = balance-500
WHERE customer = 1904
 AND account_type = 'C'
```

Transaction 2

```
SELECT SUM(balance)
FROM Accounts
WHERE customer = 1904
```

Transaction 1

```
UPDATE Accounts
SET balance = balance+500
WHERE customer = 1904
 AND account_type = 'S'
```

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

Anomalies: Dirty Read

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

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

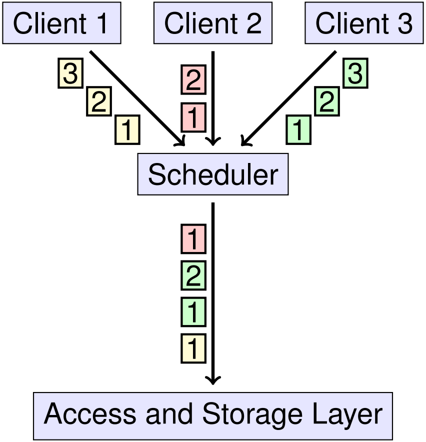
Concurrent ATM Transaction

me	**my wife**	**DB state**
$bal \leftarrow \mathrm{read}(acct)$		1200
$bal \leftarrow bal - 100$		1200
$\mathrm{write}(acct, bal)$		1100
	$bal \leftarrow \mathrm{read}(acct)$	1100
	$bal \leftarrow bal - 200$	1100
abort		1200
	$\mathrm{write}(acct, bal)$	900

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

Scheduler

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



Client 1

Client 2

Client 3

Scheduler

Access and Storage Layer

Transaction

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

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

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.

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

We assume that there are no effects other than the effects to the datbase, i.e. no writing to the screen.

Quiz Serializable Schedules

We usually only want to allow serializable schedules. Why?

Conflicts

There are several types of **conflicts** between transactions:

- **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 write 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 RW conflicts in the following schedule

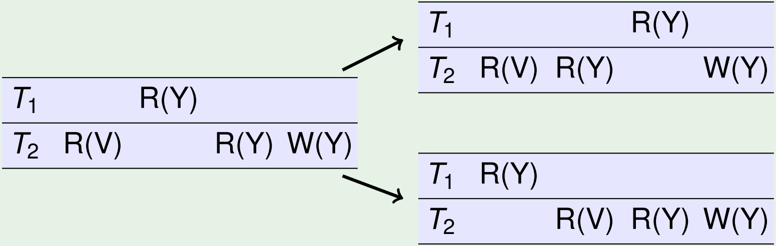
T_1	W(Y)					
T_2		W(V)		R(Z)	W(Y)	W(Z)
T_3			W(V)			

Swapping Actions

Actions **conflict** if they:

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

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



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

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

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

Conflict Equivalent Scheuldes

Two schedules are **conflict equivalent** if they can be turned into each other by a sequence of non-conflicting swaps of adjacent actions.

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 Scheuldes

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)			

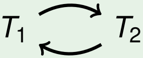
Yes

Checking Conflict-Serializability

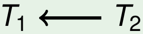
Given a schedule we can create a **precedence graph**:

- 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.

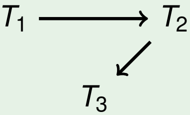
T_1	W(V)		W(V)
T_2		R(V)	



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)			

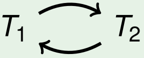


Checking Conflict-Serializability

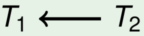
Checking Conflict-Serializability

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

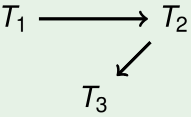
T_1	W(V)		W(V)
T_2		R(V)	



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)			

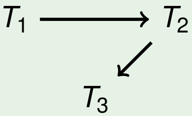


The schedules 2 and 3 have no cycles in their precedence graph. They are conflict serializable!

Checking Conflict-Serializability

If the precedence graph contains 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:

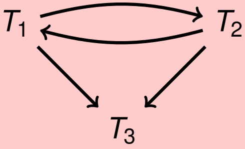
T_1	W(Y)				
T_2		R(V)	R(Y)	W(Z)	
T_3					W(V)

Checking Conflict-Serializability

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 write of T_1 and T_2 are called **blind writes.**

Ensuring Serializability

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

We now consider ensuring serializability during runtime.

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

Ensuring Serializability

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**
    - eliminate concurrency control overhead for read-only queries

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.
- Transactions unlocks objects when they are no longer needed.

Pessimistic: Lock-based Concurrency Control

In the following schedule the lock and unlock commands are written explicitly:

Schedule with explicit lock actions

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

Here we use the following abbreviations:

- S(. . .) = shared lock on . . .
- X(. . .) = exclusive lock on . . .
- U(. . .) = unlock . . .

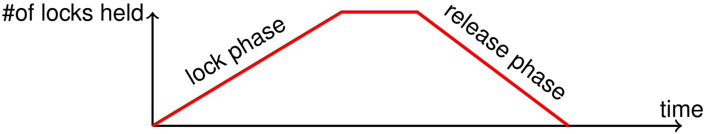
2 Phase Locking Protocol

2 Phase Locking

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 request additional locks once it releases any lock.



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

2 PL is the concurrency control protocol used in DBMSs today.

Example: ATM Transaction

Concurrent ATM Transaction

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

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

Examples

Which of the following conforms to the 2PL protocol?

T_1			X(B)	W(B)		U(B)				No
T_2	S(A)	R(A)			U(A)		X(B)	W(B)	U(B)	

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

Example: ATM Transaction

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

A 2PL-compliant ATM withdrawal transaction

1. $\mathrm{xlock}(acct)$
2. $bal \leftarrow \mathrm{read\_bal}(acct)$
3. $bal \leftarrow bal - 100$
4. $\mathrm{write\_bal}(acct, bal)$
5. $\mathrm{unlock}(acct)$

Example: ATM Transaction

Concurrent ATM Transaction

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

Transaction blocked until other transaction releases the lock.

Note: now both transactions are correctly executed!

Deadlocks

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

A Deadlock Situation

Transaction 1	**Transaction 2**
$\text{xlock}(A)$	
⋮	$\text{xlock}(B)$
do something	⋮
⋮	do something
$\text{lock}(B)$ (waiting for T_2 to unlock B)	⋮
	$\text{lock}(A)$ (waiting for T_1 to unlock A)

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

Deadlock Handling

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

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.**

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)		

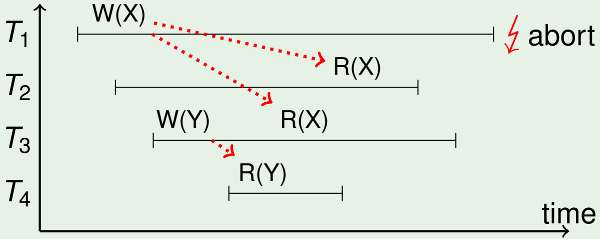
Assume that:

- 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 yield 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: Cascadeless

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!

Definition: Recoverable

Delay commits:

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

Note that transactions should be recoverable!

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

Strickt 2 Phase Locking Protocol

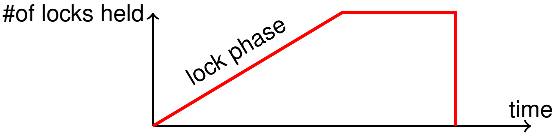
String 2 Phase Locking

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).

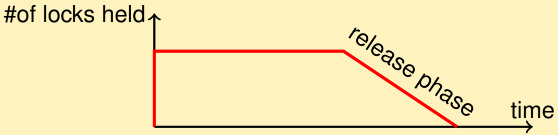


This protocol is **cascadeless**, avoids cascading aborts.

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.

Advantages:

- No deadlocks!

Disadvantages:

Not applicable in multi-query transactions. ↯

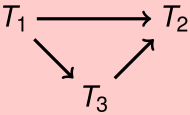
(Queries might depend on the results of the previous queries)

Example

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

- 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)		

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.

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)

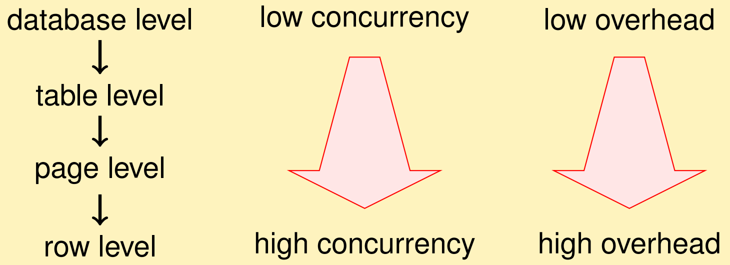
Impossible with Strict 2 PL: T_1 must hold lock on V until commit.

This schedule is 2 PL !

Can it be achieved using Preclaiming 2 PL? No

Granularity of Locking

The granularity of locking is a trade-off



database level
↓
table level
↓
page level
↓
row level

low concurrency → high concurrency

low overhead → high overhead

Idea: multi-granularity locking…

Multi-Granularity Locking

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

Depending on the characteristics of the transaction.

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

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

```
SELECT *
FROM CUSTOMERS
WHERE CUSTKEY = 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)**

A lock IS (or IX) on a coarser level of granularity means that there is some S (or X) lock on a finer level 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 `CUSTKEY = 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 CUSTKEY = 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 `CUSTKEY = 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)

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.

Isolation Levels

Sometimes, some degree of inconsistency may be acceptable for specific applications.

- E.g. occasional “off-by-one errors” will not considerably affect the outcome of an aggregation over a huge table.
- Accept inconsistent read anomaly and be awarded with improved concurrency.

SQL-92 offers different isolation levels. For example

```
SET ISOLATION SERIALIZABLE
SET ISOLATION DIRTY READ
SET ISOLATION READ UNCOMMITTED
...
```

Relaxed consistency guarantees can lead to increased throughput!

Isolation Levels

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 only held for as long as the application cursor sits on a particular, **current row**. Other rows may be changed by other transactions. Write locks as usual.

- **`READ STABILITY`**

  A transaction may read **phantom rows** if it executes an aggregation query twice.

- **`SERIALIZABLE`**

  Strict 2 PL locking. **No phantom rows**.

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!

Although both transactions properly follow the 2 PL protocol, 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

Resulting Consistency Guarantees

Isolation levels vs. consistency guarantees

isolation level	**dirty read**	**non-repeat. read**	**phantom rows**
`READ UNCOMMITTED`	possible	possible	possible
`READ COMMITTED`	not possible	possible	possible
`READ STABILITY`	not possible	not possible	possible
`SERIALIZABLE`	not possible	not possible	not possible

- Different DBMS support different levels of isolation.

Many applications do not need full serializability

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

SQL Transaction Control

- `SET AUTOCOMMIT ON/OFF`
    - `ON`: each SQL query is one transaction
- `START TRANSACTION`
- `COMMIT`
- `ROLLBACK`
- `SET TRANSACTION ISOLATION LEVEL ...`

Optimistic Concurrency Control

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

- Assume that transaction **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 immediately. Instead, collect updates in the transaction's **private workspace**.

2. **Validation phase:**

   When the transaction wants to **commit**, 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 need to be performed in a non-interruptible 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** $\mathrm{RS}(T_k) \cap \mathrm{WS}(T_i) = \varnothing$

Forward-oriented optimistic concurrency control (FOCC)

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

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

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

With old **object versions** still around, **read-only** transactions never need to be blocked!

- Might see **outdated**, but **consistent** version of the data.
- As if everything in the query happened at the moment it started.

Problems:

- Versioning requires **space** and **management overhead**.
- Update transactions still need concurrency control!

Multiversion Concurrency Control: Snapshot Isolation

Snapshot Isolation

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

With snapshot isolation:

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

Snapshot isolation does not guarantee serializability! But

- The anomalies dirty read, unrepeatable read, phantom rows do not occur. However, write skew occurs.

- Used in Oracle SQL Server (Oracle does not have real serializability).

Multiversion Concurrency Control: Snapshot Isolation

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 $Y = 1$ ↯ $X + Y \geq 2$

This problem does not occur (Oracle will detect the conflict) if the constraint is a `CHECK` constraint, and compares values of the same row. The finest locking granularity are rows.

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

Often not a problem since most databases do not support complex constraints anyway.

Transactions: Objectives

After completing this chapter, you should understand:

- ACID properties, transactions
- anomalies (lost update, dirty read, unrepeatable read, phantoms)
- transaction schedules, serializability, conflicts (rw, wr, ww)
- conflict equivalent, conflict serializability
- lock base concurrency control: 2 PL (Strict/ Preclaiming)
- cascading rollbacks, deadlocks, deadlock detection
- cascadeless, recoverable
- granularity of locking, intention locks
- SQL isolation levels: `READ UNCOMMITTED`, `READ COMMITTED`, `READ STABILITY`, `SERIALIZABLE`
- optimistic concurrency approach
- multiversion concurrency control, snapshot isolation