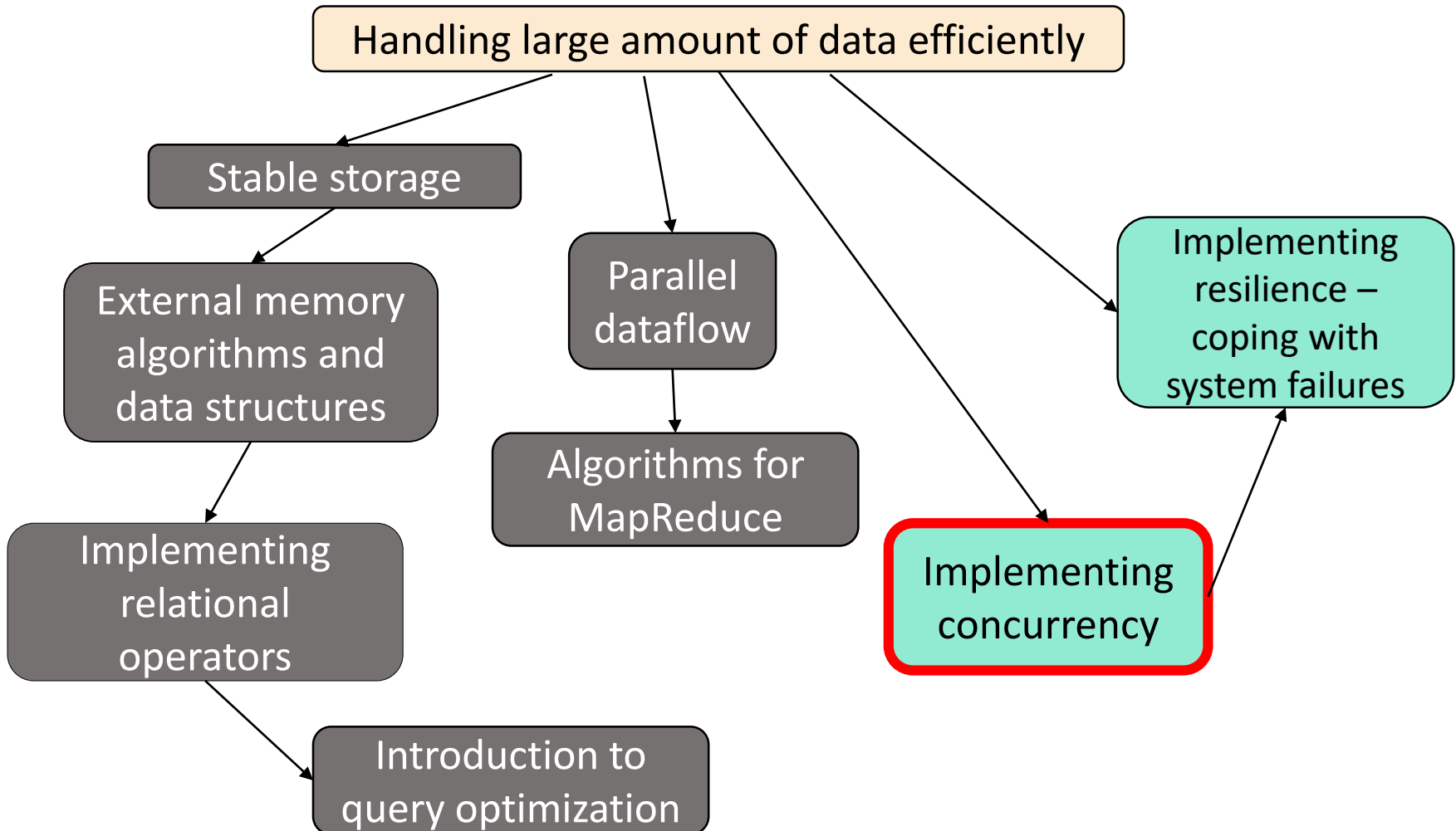


Roadmap



Concurrency

By Marina Barsky
Winter 2017, University of Toronto

We have learned:

- Where and how data is stored
- How to process large amounts of data efficiently
- ~~How to ask questions about data~~ – and how to implement efficient answers to these questions

next

How to preserve data integrity: dealing with crashes and concurrency

Why concurrent execution

- It is possible for multiple queries to be submitted at approximately the same time
- Many queries are both complex and time consuming: finishing these queries would make other queries wait a long time for a chance to execute
- Disk usage can be optimized for several queries running in parallel (recall – elevator algorithm)



So, in practice, the DBMS may be running **many different queries at about the same time**

Problems with concurrency: two people—one bank account

- Before withdrawing money, each needs to check if the balance is sufficient
- Initially there is **100**\$ on the account

Ryan

Monica

READ(X, b)

$b=100$

READ(X, c)

$c = 100$

$c - = 50$

WRITE (X, c)

$b - = 100$

WRITE (X, b)

Monica: thinks
50 \$ left

Ryan: thinks
0 \$ left

In fact, the withdrawn amount is 150\$

Problems with concurrency: two people—one bank account

- Before withdrawing money, each needs to check if the balance is sufficient
- Initially there is **100**\$ on the account

Ryan	Monica
READ(X, b)	
b=100	
	READ(X, c)
	c = 100
	c - = 50
	WRITE (X, c)
b - = 100	
WRITE (X, b)	

The problem is that the reading and writing operations should be performed as one *transaction*, their combination should be **atomic**

Transactions: recap

- DBMS groups SQL statements into ***transactions***.
- The ***transaction*** is the atomic unit of execution of database operations
- By default, each query or DML statement is a transaction
- User program can group multiple SQL statements into a single transaction

Transaction ends when:

- A **COMMIT** or **ROLLBACK** are issued
- A DDL (CREATE, ALTER, DROP ...) or DCL (GRANT, REVOKE) statement is issued
- A user properly exits (COMMIT)
- System crashes (ROLLBACK)

Subsets of SQL

Queries: SELECT

Data Manipulation Language (DML): INSERT, UPDATE, DELETE

Data Definition Language (DDL): CREATE, ALTER, DROP, RENAME

Transaction control: **COMMIT, ROLLBACK**

Data Control Language (DCL): GRANT, REVOKE

AUTOCOMMIT (Oracle syntax)

- Environment variable **AUTOCOMMIT** is by default set to OFF
- A user can set it to

SET AUTOCOMMIT ON

SET AUTOCOMMIT IMMEDIATE

and every SQL statement becomes a single transaction

- In the middle of transaction, the user can see the changed data by issuing SELECT queries.
- The user is getting the data from the temporary storage area.
- Other users cannot see the changes until transaction has been committed

Transaction properties: ACID

- **Atomicity**: **Whole transaction or none** is done.
- **Consistency**: Database constraints preserved. Transaction, executed completely, takes database from one *consistent state* to another
- **Isolation**: It appears to the user as if only one (his) process executes at a time.
 - That is, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after another in some **serial order**.
- **Durability**: Effects of a process survive a crash.

Interleaving

- DBMS has to interleave the actions of several transactions (see [slide](#))
- **Interleaving** of transactions may **lead to anomalies** even if each individual transaction preserves all the database constraints

Recording transactions

- To reason about the order of interleaving transactions, we can abstract each transaction into a **sequence of reads and writes of disk data**
- For example, withdrawing of money from the account can be written as:

$r_1(A); w_1(A)$

That means that transaction T_1 reads database element A, does something with it in main memory and writes it back to the database

- Then we can record the sequence of commands received by DBMS as:

$r_1(A); w_1(A); r_2(A); w_2(A)$

Recording sequence of commands

- Then we can record the sequence of commands from 2 transactions received by DBMS as:

$r_1(A); w_1(A); r_2(A); w_2(A)$

Transactions and Schedules: notation

- To ensure that interleaving does not lead to anomalies, DBMS *schedules* the execution of each action in a certain way
- A *schedule* is a list of actions for a set of interleaved transactions

Possible schedule:

<u>T1</u>	<u>T2</u>
r(A)	
	r(A)
	w(A)
	commit
w(A)	
commit	

Anomalies of interleaving: case 1

- Consider two transactions T1 and T2, each of which, when running alone preserves database consistency:
 - T1 transfers \$100 from A to B (e.g. from checking to saving account)
 - T2 increments both A and B by 1% (e.g. daily interest)
- The list of actions received by DBMS:
r1(A); w1(A);r1(B);w1(B);r2(A);w2(A);r2(B);w2(B)

Anomalies of interleaving transactions: possible schedule

DBMS decides on the following schedule:

T1	T2
r(A)	
w(A)	
	r(A)
	w(A)
	r(B)
	w(B)
	commit
r(B)	
w(B)	
commit	

What is the problem?

Anomalies of interleaving transactions: case 1

T1 T2

r(A)

w(A)

T1 deducted \$100 from A

r(A)

w(A)

r(B)

w(B)

commit

T2
incremented
both A and B
by 1%

r(B)

w(B)

commit

T1 added \$100 to B

Anomalies: case 1

reading uncommitted data

<u>T1</u>	<u>T2</u>	
r(A)		T1 deducted \$100 from A
w(A)		
	r(A)	T2
	w(A)	incremented
	r(B)	both A and B
	w(B)	by 1%
	commit	
r(B)		T1 added \$100 to B
w(B)		
commit		

The problem is that the bank didn't pay interest on the \$100 that was being transferred. This happened because T2 was **reading uncommitted** values.

Anomalies of interleaving transactions: case 2

- Suppose that A is the number of copies available for a book.
- Transactions $T1$ and $T2$ both place an order for this book. First they check the availability of the book.
- Consider the following scenario:
 1. $T1$ checks whether A is greater than 1.
Suppose $T1$ sees (reads) value 1.
 2. $T2$ also reads A and sees 1.
 3. $T2$ decrements A to 0.
 4. $T2$ commits.
 5. $T1$ tries to decrement A , which is now 0, and gets an error because some integrity check doesn't allow it.

Anomalies: case 2

unrepeatable reads

1. T1 checks whether A is greater than 1.
Suppose T1 sees (reads) value 1.
2. T2 also reads A and sees 1.
3. T2 decrements A to 0.
4. T2 commits.
5. T1 tries to decrement A, which is now 0, and gets an error because some integrity check doesn't allow it.

The problem is that because value of A has been changed by T1, when T2 reads A for the second time, before updating it, **the value is different from that when T2 started.**

Anomalies of interleaving transactions: case 3

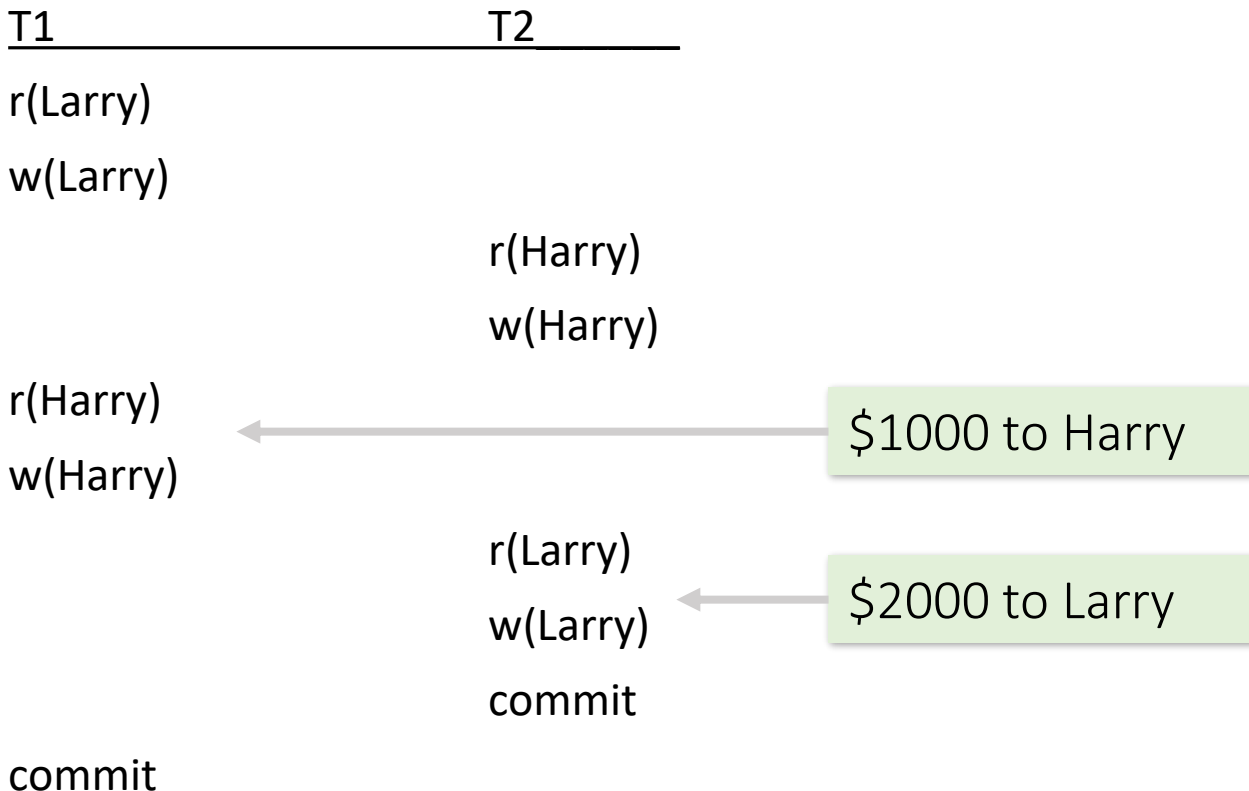
- Suppose that Larry and Harry are two employees, and their salaries **must be kept equal**. T1 sets their salaries to \$1000 and T2 sets their salaries to \$2000.
- Now consider the following schedule:

<u>T1</u>	<u>T2</u>
r(Larry)	
w(Larry)	
	r(Harry)
	w(Harry)
r(Harry)	
w(Harry)	
	r(Larry)
	w(Larry)
	commit
commit	

What is the problem?

Anomalies of interleaving transactions: case 3

- Suppose that Larry and Harry are two employees, and their salaries **must be kept equal**. T1 sets their salaries to \$1000 and T2 sets their salaries to \$2000.
- Now consider the following schedule:



Anomalies: case 3

overwriting uncommitted data

<u>T1</u>	<u>T2</u>
r(Larry)	
w(Larry)	
	r(Harry)
	w(Harry)
r(Harry)	
w(Harry)	
	r(Larry)
	w(Larry)
	commit
commit	

The problem is that T1 has overridden the result of T2, while T2 has not yet been committed.

Anomalies of interleaving

- Reading uncommitted data
- Unrepeatable reads
- Overriding uncommitted data

None of these would happen if we were executing transactions one after another: **serial schedules**

Notations

- A **transaction** (model) is a *sequence* of r and w requests on database elements
- A **schedule** is a *sequence* of reads/writes actions performed by a DBMS: to achieve interleaving and at the same time preserve consistency
- **Serial Schedule** = All actions for each transaction are consecutive.
 $r1(A); w1(A); r1(B); w1(B); r2(A); w2(A); r2(B); w2(B); \dots$
- **Serializable Schedule**: A schedule whose “**effect**” is equivalent to that of some serial schedule.

Serializable schedules

Sufficient condition for serializability

Equivalent schedules and conflicts

- Two transactions **conflict** if they access **the same data** element and **at least one of the actions is a write**.
- $r_i(X); r_j(Y) \equiv r_j(Y); r_i(X)$ (even when $X=Y$) No conflict
- We can flip $r_i(X); w_j(Y)$ as long as $X \neq Y$ No conflict
- However, $r_i(X); w_j(X) \neq w_j(X); r_i(X)$ Conflict!
- We can flip $w_i(X); w_j(Y)$; provided $X \neq Y$ No conflict
- However, $w_i(X); w_j(X) \neq w_j(X); w_i(X)$; Conflict!
The final value of X may be different depending on which write occurs last.

Conflicts: summary

There is a **conflict** if one of these two conditions hold:

1. A read and a write of the same X, or
 2. Two writes of the same X
- Such actions conflict in general and *may not be swapped in order*.
 - All other events (reads/writes) of 2 different transactions *may be swapped* without changing the **effect** of the schedule.

Sufficient condition for serializable schedule

A schedule is ***conflict-serializable*** if it can be converted into a **serial** schedule by a series of **non-conflicting swaps** of adjacent elements

Example:

What can we say about the original schedule?

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

Non-conflicting swaps:

$r_1(A); w_1(A); r_2(A); \underline{r_1(B)}; \underline{w_2(A)}; w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); \underline{r_1(B)}; \underline{r_2(A)}; w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); r_2(A); \underline{w_1(B)}; \underline{w_2(A)}; r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); \underline{w_1(B)}; \underline{r_2(A)}; w_2(A); r_2(B); w_2(B)$

Result: serial schedule

Conflict-serializability

Sufficient condition for serializability but **not necessary**.

Example

S1: $w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X);$ ← This is serial

S2: $w_1(Y); w_2(Y); w_2(X); w_1(X); w_3(X);$ ←

This is called **view-serializable**, and requires from scheduler to understand what each action is doing, not just its type

S2 isn't conflict-serializable, but it is serializable. It has the same effect as S1.

Intuitively, the values of X written by T1 and T2 have no effect, since T3 overwrites them.

Serializability/precedence Graphs

- Non-swappable pairs of actions represent potential conflicts between transactions.
- The existence of non-swappable actions enforces an **ordering** on the transactions that include these actions.

We can represent this order by a **graph**

- **Nodes**: transactions $\{T_1, \dots, T_k\}$
- **Arcs**: There is a directed edge from T_i to T_j if they have conflicting access to the same database element X and T_i is first:

written $T_i <_s T_j$.

Precedence graphs: example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Note the following:

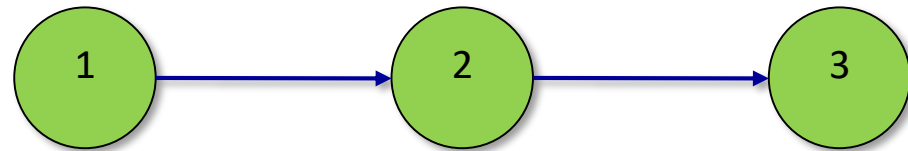
- $w_1(B) <_s r_2(B)$

- $r_2(A) <_s w_3(A)$

➤ These are conflicts since they contain a read/write on the same element

➤ They cannot be swapped.

Therefore $T_1 < T_2 < T_3$



Conflict-serializable

Precedence graphs: example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

Note the following:

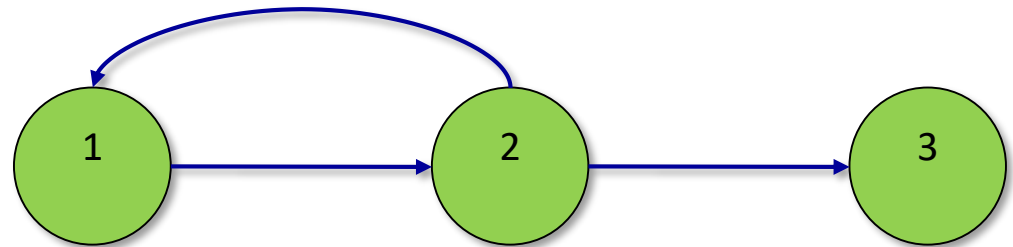
- $r_1(B) <_S w_2(B)$
- $w_2(A) <_S w_3(A)$
- $r_2(B) <_S w_1(B)$

➤ Here, we have

$T_1 < T_2 < T_3$,

but we also have

$T_2 < T_1$



Not conflict-serializable

Precedence graphs: test for conflict-serializability


- **If there is a cycle in the graph**, then there is **no** serial schedule which is conflict-equivalent to S.
 - Each arc represents a requirement on the order of transactions in a conflict-equivalent *serial schedule*.
 - A cycle puts too many requirements on any *linear order* of transactions.
- **If there is no cycle in the graph**, then *any topological order** of the graph suggests a conflict-equivalent schedule.

*A topological ordering of a directed acyclic graph (DAG) is a linear ordering of its nodes in which **each node comes before all nodes to which it has outbound edges**

Theorem: If a Precedence-Graph is acyclic, it represents a conflict-serializable schedule

Proof idea:

if the precedence graph is acyclic, then we can swap actions to form serial schedule.

- Given that the precedence graph is acyclic, there exists T_k such that there is no T_m that T_k depends on, i.e. T_k does not have any incoming edges.
- In this case, we can bring all actions of T_k to the front of the schedule.
(Actions of T_k)(Actions of the other $n-1$ transactions)
- The **tail** is a precedence graph that is the same as the original without T_k , i.e. it has $n-1$ nodes.
- Repeat for the tail. 

Enforcing serializability by locks

- If scheduler allows multiple transactions access the same element, this may result in non-serializable schedule
- To prevent this, before reading or writing an element X , a transaction T_i requests a lock on X from the scheduler.
- The scheduler can either grant the lock to T_i or make T_i wait for the lock.
- If granted, T_i should eventually unlock (release) the lock on X .
- Notations:
 - $L_i(X)$ = “transaction T_i requests a lock on X ”
 - $u_i(X)$ (or $uL_i(X)$) = “ T_i unlocks/releases the lock on X ”

Legal schedule with locks

Schedule with locks - constraints:

Consistency of Transactions:

- Read or write X only when hold a lock on X .
 $r_i(X)$ or $w_i(X)$ must be preceded by some $L_i(X)$ with no intervening $u_i(X)$.
- If T_i locks X , T_i must eventually unlock X .
Every $L_i(X)$ must be followed by $u_i(X)$.

Legality of Schedules:

- Two transactions may not have locked the same element X without one having first released the lock.
A schedule with $L_i(X)$ cannot have another $L_j(X)$ until $u_i(X)$ appears *in between*.

T_1	T_2	A	B
		25	25
$L_1(A); r_1(A)$			
$A = A + 100$			
$w_1(A); u_1(A)$		125	
<div style="border: 1px solid black; background-color: #fff9c4; padding: 5px; width: fit-content;"> T1 unlocks A so T2 is free to lock it </div>	$L_2(A); r_2(A)$		
	$A = A * 2$		
	$w_2(A); u_2(A)$	250	
	$L_2(B); r_2(B)$		
	$B = B * 2$		
	$w_2(B); u_2(B)$		50
$L_1(B); r_1(B)$			
$B = B + 100$			
$w_1(B); u_1(B)$			150

Legal schedule doesn't mean serializable!

- T1 adds 100 to both A and B
- T2 doubles both A and B
- Expected result: A=B, and should be 250 for both by the end

Two-Phase Locking

There is a simple condition, which guarantees conflict-serializability:

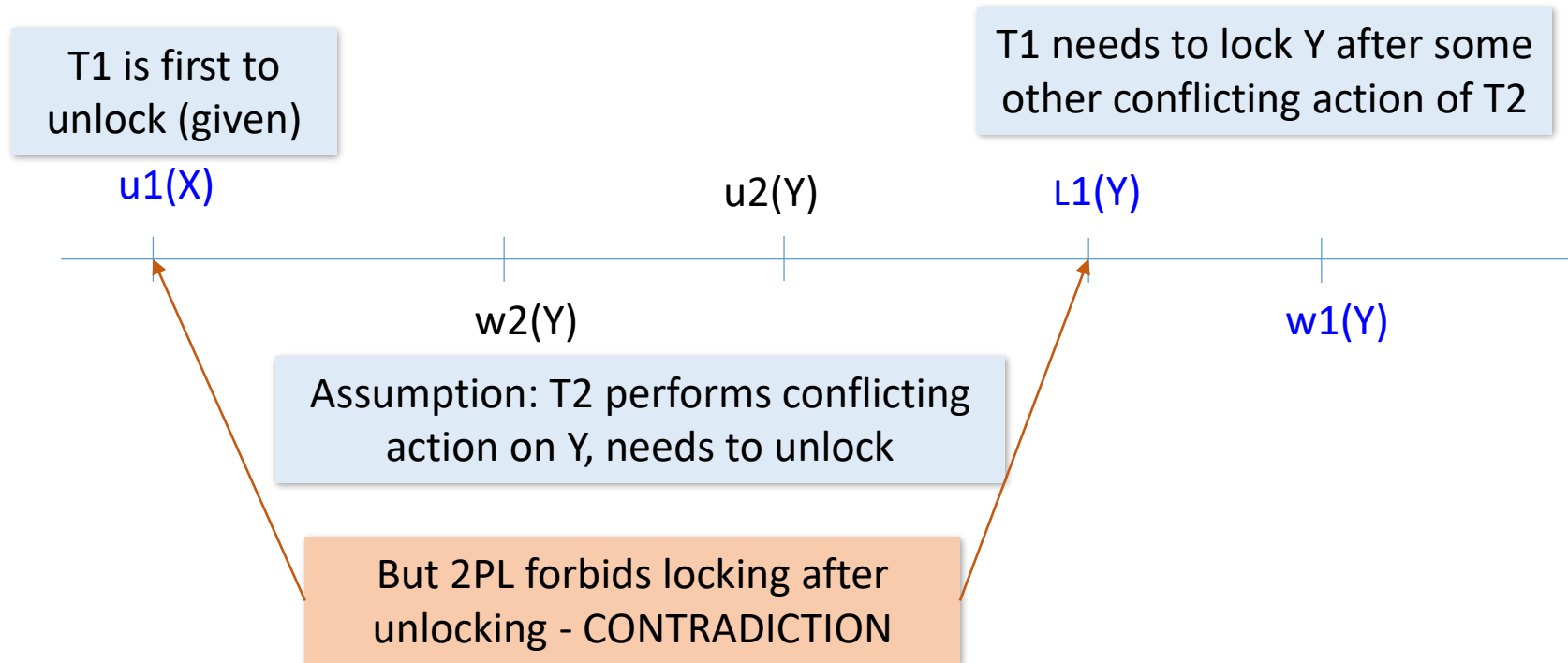
In every transaction, all lock requests (**phase 1**) precede all unlock requests (**phase 2**).

T_1	T_2	A	B
		25	25
$L_1(A); r_1(A)$			
$A = A + 100$			
$w_1(A); L_1(B); u_1(A)$		125	
	$L_2(A); r_2(A)$		
	$A = A * 2$		
	$w_2(A)$	250	
	$L_2(B)$ Denied		
$r_1(B)$			
$B = B + 100$			125
$w_1(B); u_1(B)$			
	$L_2(B); u_2(A); r_2(B)$		
	$B = B * 2$		
	$w_2(B); u_2(B)$		250

Theorem: A legal schedule with 2PL locking is conflict-serializable

- **Proof idea:** in 2PL each transaction that starts unlocking, has already acquired all locks on its elements, and thus all its actions can be moved to the front of the schedule
- **Proof by contradiction:** $S = \{T_1, \dots, T_n\}$. Find the **first** transaction, say T_1 , to perform an **unlock** action, say $u_1(X)$. We show that the r/w actions of T_1 can be moved to the front of the other transactions without conflict.
- Consider some action such as $w_1(Y)$. Let assume that it is preceded by some conflicting action $w_2(Y)$ or $r_2(Y)$. **In such a case we cannot swap them.**
 - If so, then $u_2(Y)$ and $L_1(Y)$ must interleave, as
 $w_2(Y) \dots u_2(Y) \dots L_1(Y) \dots w_1(Y)$
 - Since T_1 is the first to unlock, $u_1(X)$ appears before $u_2(Y)$.
 - But then $L_1(Y)$ appears after $u_1(X)$, **contradicting 2PL.**

A legal schedule with 2PL locking is conflict-serializable



- **Conclusion:** $w_1(Y)$ can slide forward in the schedule without conflict; similar argument for a $r_1(Y)$ action. ■

Simple locks are too restrictive

- While simple locks + 2PL guarantee conflict-serializability, **they do not allow two readers of DB element X at the same time.**
- **But having multiple readers is not a problem for conflict-serializability (since read actions commute)!**

Shared/Exclusive Locks

Solution: Two types of locks:

- I. **Shared lock $sL_i(X)$** allows T_i to read, but not write X .
It prevents other transactions from writing X but not from reading X .

- II. **Exclusive lock $xL_i(X)$** allows T_i to read and/or write X .
No other transaction may read or write X .

Shared/Exclusive Locks: changes

Consistency of transactions:

- A read $r_i(\mathbf{X})$ must be preceded by $sl_i(\mathbf{X})$ or $xl_i(\mathbf{X})$, with no intervening $u_i(\mathbf{X})$.
- A write $w_i(\mathbf{X})$ must be preceded by $xl_i(\mathbf{X})$, with no intervening $u_i(\mathbf{X})$.

Legal schedules:

- No two exclusive locks on the same element.
If $xl_i(\mathbf{X})$ appears in a schedule, then there cannot be a $xl_j(\mathbf{X})$ until after a $u_i(\mathbf{X})$ appears.
- No shared locks on exclusively locked element.
If $xl_i(\mathbf{X})$ appears, there can be no $sl_j(\mathbf{X})$ until after $u_i(\mathbf{X})$.
- No writing in shared lock mode
If $sl_i(\mathbf{X})$ appears, there can be no $w_j(\mathbf{X})$ until after $u_i(\mathbf{X})$.

2PL condition:

- No transaction may have a $sl(\mathbf{X})$ or $xl(\mathbf{X})$ after a $u(\mathbf{Y})$.

Scheduler rules for shared/exclusive locks

- When there is more than one kind of lock, the scheduler needs a rule that says **“if there is already a lock of type A on DB element X, can I grant a lock of type B on X?”**
- The compatibility matrix answers the question.

Compatibility Matrix for Shared/Exclusive Locks

	S	X
S	yes	no
X	no	no

Scheduling with locks: example

r1(A); r2(B); r3(C); r1(B); r2(C); r3(D); w1(A); w2(B); w3(C);

T1	T2	T3
xl(A); r1(A)		
	xl(B); r2(B)	
sl(B) denied		xl(C); r3(C)
	sl(C) denied	
w1(A);		sl(D); r3(D); ul(D)
	w2(B);	
	sl(C); r2(C); ul(B); ul(C)	w3(C); ul(C)
sl(B); r1(B); ul(A); ul(B)		

Upgrading Locks

- Instead of taking an exclusive lock immediately, a transaction can take a *shared* lock on X, read X, and then upgrade the lock to *exclusive* so that it can write X.

T_1	T_2
$sl_1(A); r_1(A);$	
	$sl_2(A); r_2(A);$ $sl_2(B); r_2(B);$
$sl_1(B); r_1(B);$ $xl_1(B)$ Denied	
	$u_2(A); u_2(B)$
$xl_1(B); w_1(B);$ $u_1(A); u_2(B);$	

Upgrading Locks allows more concurrent operations:

Had T1 asked for an exclusive lock on B before reading B, the request would have been denied, because T2 already has a shared lock on B.

Scheduling with upgrade locks: example

r1(A); r2(B); r3(C); r1(B); r2(C); r3(D); w1(A); w2(B); w3(C);

T1	T2	T3
sl(A); r1(A);		
	sl(B); r2(B);	
sl(B); r1(B);		sl(C); r3(C);
	sl(C); r2(C);	
xl(A); w1(A); ul(A); ul(B);		sl(D); r3(D);
	xl(B); w2(B); ul(B); ul(C);	
		xl(C); w3(C); ul(C); ul(D);

Compared to slide 47: no waiting

Possibility of Deadlocks

Example: T1 and T2 each reads X and later writes X.

T1	T2
SL1(X)	
	SL2(X)
xL1(X) denied	
	xL2(X) denied

Problem: when we allow upgrades, it is easy to get into a deadlock situation.

“When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”

Solution: Update Locks

Update lock $udl_i(X)$

- Only an **update lock** (**not shared lock**) can be upgraded to **exclusive lock** (if there are no shared locks anymore).
- A transaction that will read and later on write some element A, **asks initially for an update** lock on A, and then asks for an exclusive lock on A. Such transaction doesn't ask for a shared lock on A.

Legal schedules

- Read action permitted when there is either a shared or update lock.
- An update lock can be granted while there is a shared lock, but the scheduler will not grant a shared lock when there is an update lock.

2PL condition

- No transaction may have an $sl(X)$, $udl(X)$ or $xl(X)$ after a $u(Y)$.

Update Locks: scheduler rules

Compatibility Matrix for Shared/Exclusive/Update Locks

	S	X	U
S	yes	no	yes
X	no	no	no
U	no	no	no

Schedule with update locks: example

T1	T2	T3
SL(A); r(A)	<u>udL</u> (A); r(A)	SL(A) Denied
	xL(A) Denied	
u(A)	xL(A); w(A)	
	u(A)	SL(A); r(A)
		u(A)

(No) Deadlock Example

T_1 and T_2 each read X and later write X.

T1	T2
SL1(X);	
	SL2(X);
XL1(X); denied	
	XL2(X); denied

Deadlock when using **SL** and **XL** locks only.

T1	T2
udl1(X); r(X);	
	udl2(X); denied
XL1(X); w(X); u(X);	
	udl2(X); r2(X); xl2(X); w2(X); u2(X)

Fine when using update locks.

Scheduling with 3 types of locks: example

r1(A); r2(B); r3(C); r1(B); r2(C); r3(D); w1(A); w2(B); w3(C);

T1	T2	T3
uL(A); r1(A);		
	uL(B); r2(B);	
SL(B); denied		uL(C); r3(C);
	SL(C); denied	
xL(A); w1(A);		SL(D); r3(D);
	xL(B); w2(B);	
		xL(C); w3(C); uL(D); uL(C);
	SL(C); r2(C); uL(B); uL(C);	
SL(B); r1(B); uL(A); uL(B);		

Transaction control in SQL

Gives control over the locking overhead

- **Access mode:**
 - READ ONLY
 - READ WRITE
- **Isolation level** (to which extent transaction is exposed to actions of other transactions):
 - SERIALIZABLE (Default)
 - REPEATABLE READ
 - READ COMMITTED
 - READ UNCOMMITTED

What we can allow at different isolation levels

1. Reading uncommitted (dirty) data:
 - A transaction reads data written by a concurrent uncommitted transaction
2. Unrepeatable reads
 - A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read)
3. Phantom read
 - A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

Transaction Isolation Levels

SET TRANSACTION ISOLATION LEVEL X READ WRITE

Where X can be

SERIALIZABLE (Default)

REPEATABLE READ

READ COMMITTED

READ UNCOMMITTED



Decreasing isolation level

With a scheduler based on locks:

- A *SERIALIZABLE* transaction obtains locks before reading and writing objects, including locks on sets (e.g. table) of objects that it requires to be unchangeable and holds them until the end, according to 2PL.
- A *REPEATABLE READ* transaction sets the same locks as a *SERIALIZABLE* transaction, except that it doesn't lock sets of objects, but only individual objects.

Transaction Isolation Levels

- A **READ COMMITTED** transaction T obtains exclusive locks before writing objects and keeps them until the end. However, it obtains shared locks before reading values and then immediately releases them;

That is to ensure that the transaction that last modified the values is complete.

- T reads only the changes made by committed transactions.
 - No value written by T is changed by any other transaction until T is completed.
 - However, a value read by T may well be modified by another transaction (which eventually commits) while T is still in progress.
 - T is also exposed to the *phantom* problem.
-
- A **READ UNCOMMITTED** transaction doesn't obtain any lock at all. So, it can read data that is being modified. Such transactions are allowed to be READ ONLY only.

Problems at each isolation level

Level	Reading Uncommitted Data (Dirty Read)	Unrepeatable Read (different values in the same rows)	Phantom (different collections of rows)
READ UNCOMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

Summary: ACID transactions

- **Consistency**: Database constraints preserved. Transaction, executed completely, takes database from one *consistent* state to another: **serializable schedules**
- **Isolation**: It appears to the user as if only one process executes at a time: **locking**

Finally we talk how to ensure:

- **Atomicity**: Whole transaction or none is done.
- **Durability**: Effects of a process survive a crash.