Chapter 11

Transaction Management

Concurrent and Consistent Data Access

Architecture and Implementation of Database Systems
Summer 2016

Torsten Grust
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Multi-Version Concurrency Control

The “Hello World” of Transaction Management

- My bank issued me a debit card to access my account.
- Every once in a while, I use it at an ATM to withdraw some money from my account, causing the ATM to perform a **transaction** in the bank's database.

Example (ATM transaction)

1. $bal \leftarrow \texttt{read_bal}\,(acct_no)$;
2. $bal \leftarrow bal - 100$ € ;
3. $\texttt{write_bal}\,(acct_no, bal)$;

- My account is **properly updated** to reflect the new balance.

ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Concurrent Access

The problem is: My wife owns such a card for the very same account, too.

⇒ We might end up using our cards at different ATMs at the **same time**, *i.e.*, **concurrently**.

Example (Concurrent ATM transactions)

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

- The first **update was lost** during this execution. Lucky me!

ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

If the Plug is Pulled . . .

- This time, I want to **transfer** money over to another account.

Example (Money transfer transaction)

```
   // Subtract money from source (checking) account
1  chk_bal ← read_bal (chk_acct_no) ;
2  chk_bal ← chk_bal − 500 € ;
3  write_bal (chk_acct_no, chk_bal) ;

   // Credit money to the target (savings) account
4  sav_bal ← read_bal (sav_acct_no) ;
5  sav_bal ← sav_bal + 500 € ;
6     ↯
7  write_bal (sav_acct_no, sav_bal) ;
```

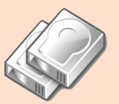
- Before the transaction gets to step `7`, its execution is **interrupted/cancelled** (power outage, disk failure, software bug, . . .). My money is **lost** ☹.

ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

ACID Properties

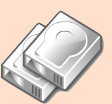
To prevent these (and many other) effects from happening, a DBMS guarantees the following **transaction properties**:

A	Atomicity	Either **all** or **none** of the updates in a database transaction are applied.
C	Consistency	Every transaction brings the database from one **consistent** state to another. (While the transaction executes, the database state may be temporarily inconsistent.)
I	Isolation	A transaction must not see any effect from other transactions that run in parallel.
D	Durability	The effects of a **successful** transaction remain persistent and may not be undone for system reasons.



ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Concurrency Control

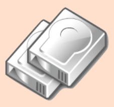


ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Anomalies: Lost Update

- We already saw an example of the **lost update** anomaly on slide 3:

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



ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Anomalies: Inconsistent Read

Reconsider the money transfer example (slide 4), expressed in SQL syntax:

Example

Transaction 1

```
1 UPDATE Accounts
2    SET balance = balance - 500
3  WHERE customer = 1904
4    AND account_type = 'C';
```

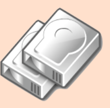
Transaction 2

```
1 SELECT SUM(balance)
2   FROM Accounts
3  WHERE customer = 1904;
```

Transaction 1 (continued)

```
5 UPDATE Accounts
6    SET balance = balance + 500
7  WHERE customer = 1904
8    AND account_type = 'S';
```

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



ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Anomalies: Dirty Read

On a different day, once more my wife and me end up in front of ATMs at roughly the same time. This time, my transaction is cancelled (**aborted**):

Example

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

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

ACID Properties

Anomalies

The Scheduler

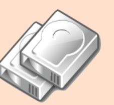
Serializability

Query Scheduling

Locking

Two-Phase Locking

Multi-Version Concurrency Control

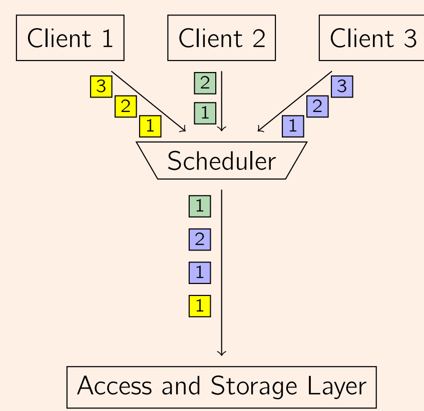


ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Concurrent Execution

- The **scheduler** decides the execution order of concurrent database accesses.

The transaction scheduler



Database Objects and Accesses

- We now assume a slightly simplified model of database access:
  1. A database consists of a number of named **objects**. In a given database state, each object has a **value**.
  2. Transactions access an object o using the two operations `read` o and `write` o.

- In a **relational** DBMS we have that

$$\text{object} \equiv \text{attribute} \enspace .$$

This defines the **granularity** of our discussion. Other possible granularities:

$$\text{object} \equiv \text{row}, \text{object} \equiv \text{table} \enspace .$$

ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Transactions

ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Database transaction

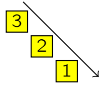
A **database transaction** T is a (strictly ordered) sequence of **steps**. Each **step** is a pair of an **access operation** applied to an **object**.

- Transaction $T = \langle s_1, \dots, s_n \rangle$
- Step $s_i = (a_i, e_i)$
- Access operation $a_i \in \{\mathtt{r(ead)}, \mathtt{w(rite)}\}$

The **length** of a transaction T is its number of steps $|T| = n$.

We could write the money transfer transaction as

$$T = \langle\ (\mathtt{read}, \mathit{Checking}), (\mathtt{write}, \mathit{Checking}),\\ (\mathtt{read}, \mathit{Saving}), (\mathtt{write}, \mathit{Saving})\ \rangle$$



or, more concisely,

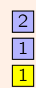
$$T = \langle r(C), w(C), r(S), w(S) \rangle \ .$$

Schedules

Schedule

A **schedule** S for a given set of transactions $\mathbf{T} = \{T_1, \ldots, T_n\}$ is an arbitrary sequence of execution steps

$$S(k) = (T_j, a_i, e_i) \qquad k = 1 \ldots m \ ,$$



such that

1. S contains all steps of all transactions and nothing else and
2. the order among steps in each transaction T_j is preserved:

$$(a_p, e_p) < (a_q, e_q) \text{ in } T_j \Rightarrow (T_j, a_p, e_p) < (T_j, a_q, e_q) \text{ in } S$$

(read "$<$" as: *occurs before*).

We sometimes write

$$S = \langle r_1(B), r_2(B), w_1(B), w_2(B) \rangle$$

to abbreviate

$$S(1) = (T_1, \texttt{read}, B) \quad S(3) = (T_1, \texttt{write}, B)$$
$$S(2) = (T_2, \texttt{read}, B) \quad S(4) = (T_2, \texttt{write}, B)$$

ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Multi-Version Concurrency Control

Serial Execution

Serial execution

One particular schedule is **serial execution**.

- A schedule S is **serial** iff, for each contained transaction T_j, all its steps are adjacent (no interleaving of transactions and thus **no concurrency**).

Briefly:

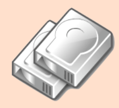
$$S = T_{\pi 1}, T_{\pi 2}, \dots, T_{\pi n} \qquad \text{(for some permutation } \pi \text{ of } 1, \dots, n\text{)}$$

Consider again the ATM example from slide 3.

- $S = \langle r_1(B), r_2(B), w_1(B), w_2(B) \rangle$
- This is a schedule, but it is **not** serial.

If my wife had gone to the bank one hour later (initiating transaction T_2), the schedule probably would have been serial.

- $S = \langle r_1(B), w_1(B), r_2(B), w_2(B) \rangle$



ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Correctness of Serial Execution

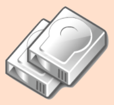
- Anomalies such as the "lost update" problem on slide 3 can **only** occur in multi-user mode.
- If all transactions were fully executed one after another (no concurrency), no anomalies would occur.

⇒ **Any serial execution is correct.**

- Disallowing concurrent access, however, is **not practical**.

Correctness criterion

Allow concurrent executions if their **overall effect is equivalent to an (arbitrary) serial execution**.



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Multi-Version Concurrency Control

Conflicts

What does it mean for a schedule S to be **equivalent** to another schedule S'?

- Sometimes, we may be able to **reorder** steps in a schedule.
  - We must not change the order among steps of any transaction T_j (↗ slide 13).
  - Rearranging operations must not lead to a different **result**.
- Two operations (T_i, a, e) and (T_j, a', e') are said to be **in conflict** $(T_i, a, e) \nleftrightarrow (T_j, a', e')$ if their order of execution matters.
  - When reordering a schedule, we must not change the relative order of such operations.
- Any schedule S' that can be obtained this way from S is said to be **conflict equivalent** to S.

ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Conflicts

Based on our `read`/`write` model, we can come up with a more machine-friendly definition of a conflict.

Conflicting operations

Two operations (T_i, a, e) and (T_j, a', e') are **in conflict** ($\nleftrightarrow$) in S if

1. they belong to two **different transactions** ($T_i \neq T_j$), and
2. they access the **same database object**, *i.e.*, $e = e'$, and
3. at least one of them is a `write` operation.

- This inspires the following **conflict matrix**:

	`read`	`write`
`read`		×
`write`	×	×

- **Conflict relation $\prec_S$:**

$$(T_i, a, e) \prec_S (T_j, a', e')$$
$$:=$$
$$(T_i, a, e) \nleftrightarrow (T_j, a', e') \land (T_i, a, e) \text{ occurs before } (T_j, a', e') \text{ in } S$$

ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Conflict Serializability

ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Conflict serializability

A schedule S is **conflict serializable** iff it is **conflict equivalent to some serial schedule** S'.

- **The execution of a conflict-serializable** S **schedule is correct.**
- Note: S does **not** have to be a serial schedule.

Serializability: Example

ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Example (schedules S_i for two transactions $T_{1,2}$, with S_2 serial)

Schedule S_1

T_1	T_2
read A	
write A	
	read A
	write A
read B	
write B	
	read B
	write B

Schedule S_2

T_1	T_2
read A	
write A	
read B	
write B	
	read A
	write A
	read B
	write B

Schedule S_3

T_1	T_2
read A	
write A	
	read A
	write A
	read B
	write B
read B	
write B	

- Conflict relations:

$$\left.\begin{array}{l}
(T_1,\mathtt{r},A) \prec_{S_1} (T_2,\mathtt{w},A),\ (T_1,\mathtt{r},B) \prec_{S_1} (T_2,\mathtt{w},B),\\
(T_1,\mathtt{w},A) \prec_{S_1} (T_2,\mathtt{r},A),\ (T_1,\mathtt{w},B) \prec_{S_1} (T_2,\mathtt{r},B),\\
(T_1,\mathtt{w},A) \prec_{S_1} (T_2,\mathtt{w},A),\ (T_1,\mathtt{w},B) \prec_{S_1} (T_2,\mathtt{w},B)\\
\\
(\text{Note: } \prec_{S_2} = \prec_{S_1})\\
\\
(T_1,\mathtt{r},A) \prec_{S_3} (T_2,\mathtt{w},A),\ (T_2,\mathtt{r},B) \prec_{S_3} (T_1,\mathtt{w},B),\\
(T_1,\mathtt{w},A) \prec_{S_3} (T_2,\mathtt{r},A),\ (T_2,\mathtt{w},B) \prec_{S_3} (T_1,\mathtt{r},B),\\
(T_1,\mathtt{w},A) \prec_{S_3} (T_2,\mathtt{w},A),\ (T_2,\mathtt{w},B) \prec_{S_3} (T_1,\mathtt{w},B)
\end{array}\right\} \Rightarrow S_1 \text{serializable}$$

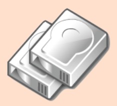
The Conflict Graph

- The serializability idea comes with an effective test for the correctness of a schedule S based on its **conflict graph** $G(S)$ (also: **serialization graph**):
  - The **nodes** of $G(S)$ are all transactions T_i in S.
  - There is an **edge** $T_i \rightarrow T_j$ iff S contains operations (T_i, a, e) and (T_j, a', e') such that
    $(T_i, a, e) \prec_S (T_j, a', e')$

    (read: *in a conflict equivalent serial schedule, T_i must occur before T_j*).

- S is conflict serializable iff $G(S)$ is **acyclic**.

  An equivalent serial schedule for S may be immediately obtained by sorting $G(S)$ **topologically**.



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

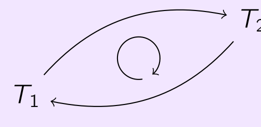
Two-Phase Locking

Multi-Version Concurrency Control

Serialization Graph

Example (ATM transactions (↗ slide 3))

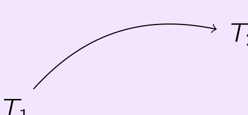
- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$
- Conflict relation:
  $(T_1, \mathtt{r}, A) \prec_S (T_2, \mathtt{w}, A)$
  $(T_2, \mathtt{r}, A) \prec_S (T_1, \mathtt{w}, A)$
  $(T_1, \mathtt{w}, A) \prec_S (T_2, \mathtt{w}, A)$



⇒ **not** serializable

Example (Two money transfers (↗ slide 4))

- $S = \langle r_1(C), w_1(C), r_2(C), w_2(C), r_1(S), w_1(S), r_2(S), w_2(S) \rangle$
- Conflict relation:
  $(T_1, \mathtt{r}, C) \prec_S (T_2, \mathtt{w}, C)$
  $(T_1, \mathtt{w}, C) \prec_S (T_2, \mathtt{r}, C)$
  $(T_1, \mathtt{w}, C) \prec_S (T_2, \mathtt{w}, C)$
  ⋮



⇒ serializable

ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Query Scheduling

Can we build a scheduler that **always** emits a serializable schedule?

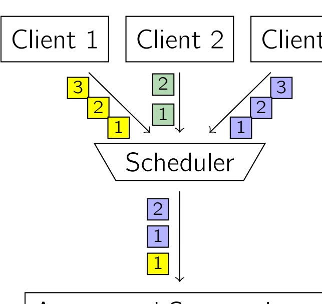
Idea:

- Require each transaction to obtain a **lock** before it accesses a data object o:

Locking and unlocking of o

```
1 lock o ;
2 ...access o ...;
3 unlock o ;
```

- This prevents **concurrent** access to o.



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

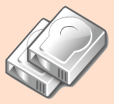
Two-Phase Locking

Multi-Version Concurrency Control

Locking

- If a lock cannot be granted (*e.g.*, because another transaction T' already holds a **conflicting** lock) the requesting transaction T gets **blocked**.
- The scheduler **suspends** execution of the blocked transaction T.
- Once T' **releases** its lock, it may be granted to T, whose execution is then **resumed**.

⇒ Since other transactions can continue execution while T is blocked, locks can be used to **control the relative order of operations**.



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Multi-Version Concurrency Control

Locking and Scheduling

ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Example (Locking and scheduling)

- Consider two transactions $T_{1,2}$:

T_1
`lock` A
`write` A
`lock` B
`unlock` A
`write` B
`unlock` B

T_2
`lock` A
`write` A
`lock` B
`write` B
`write` A
`unlock` A
`write` B
`unlock` B

- Two valid schedules (respecting **`lock`** and **`unlock`** calls) are:

Schedule S_1

T_1	T_2
`lock` A	
`write` A	
`lock` B	
`unlock` A	
	`lock` A
	`write` A
`write` B	
`unlock` B	
	`lock` B
	`write` B
	`write` A
	`unlock` A
	`write` B
	`unlock` B

Schedule S_2

T_1	T_2
	`lock` A
	`write` A
	`lock` B
	`write` B
	`write` A
	`unlock` A
`lock` A	
`write` A	
	`write` B
	`unlock` B
`lock` B	
`write` B	
`unlock` B	
`unlock` A	

- Note: Both schedules $S_{1,2}$ are serializable. **Are we done yet?**

Locking and Serializability

Example (Proper locking does not guarantee serializability yet)

Even if we adhere to a properly nested `lock`/`unlock` discipline, the scheduler might still yield **non-serializiable schedules**:

Schedule S_1

T_1	T_2
`lock` A	
`lock` C	
`write` A	
`write` C	
`unlock` A	
	`lock` A
	`write` A
	`lock` B
	`unlock` A
	`write` B
	`unlock` B
`unlock` C	
`lock` B	
`write` B	
`unlock` B	
	`lock` C
	`write` C
	`unlock` C

✎ **What is the conflict graph of this schedule?**

ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

ATM Transaction with Locking

Example (Two concurrent ATM transactions with locking)

Transaction 1	Transaction 2	DB state
`lock` (*acct*) ;		1200
`read` (*acct*) ;		
`unlock` (*acct*) ;		
	`lock` (*acct*) ;	
	`read` (*acct*) ;	
	`unlock` (*acct*) ;	
`lock` (*acct*) ;		
`write` (*acct*) ;		1100
`unlock` (*acct*) ;		
	`lock` (*acct*) ;	
	`write` (*acct*) ;	1000
	`unlock` (*acct*) ;	

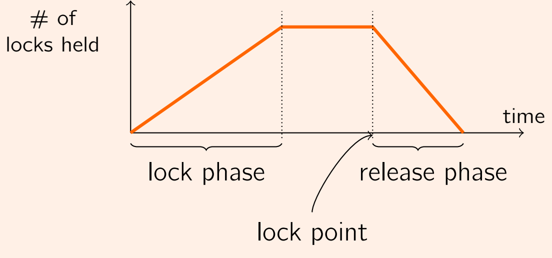
ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Two-Phase Locking (2PL)

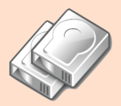
The **two-phase locking protocol** poses an additional restriction on how transactions have to be written:

Definition (Two-Phase Locking)

- Once a transaction has **released** any lock (*i.e.*, performed the first `unlock`), it must **not** acquire any new lock:



- Two-phase locking is **the** concurrency control protocol used in database systems today.

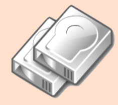


ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Again: ATM Transaction

Example (Two concurrent ATM transactions with locking, $\neg$ 2PL)

Transaction 1	Transaction 2	DB state
`lock` ($acct$) ;		1200
`read` ($acct$) ;		
`unlock` ($acct$) ;		
	`lock` ($acct$) ;	
	`read` ($acct$) ;	
	`unlock` ($acct$) ;	
`lock` ($acct$) ; ϟ		
`write` ($acct$) ;		1100
`unlock` ($acct$) ;		
	`lock` ($acct$) ; ϟ	
	`write` ($acct$) ;	1000
	`unlock` ($acct$) ;	



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Multi-Version Concurrency Control

A 2PL-Compliant ATM Transaction

- To comply with the two-phase locking protocol, the ATM transaction must not acquire any new locks after a first lock has been released:

A 2PL-compliant ATM withdrawal transaction

```
1 lock (acct) ;                          } lock phase
2 bal ← read_bal (acct) ;
3 bal ← bal − 100 € ;
4 write_bal (acct, bal) ;
5 unlock (acct) ;                        } unlock phase
```

ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

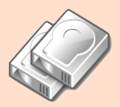
Two-Phase Locking

Multi-Version Concurrency Control

Resulting Schedule

Example

Transaction 1	Transaction 2	DB state
`lock` (*acct*) ;		1200
`read` (*acct*) ;		
	`lock` (*acct*) ;	
`write` (*acct*) ;	Transaction blocked	1100
`unlock` (*acct*) ;		
	`lock` (*acct*) ;	
	`read` (*acct*) ;	
	`write` (*acct*) ;	900
	`unlock` (*acct*) ;	



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Multi-Version Concurrency Control

Lock Modes

- We saw earlier that two **read** operations do not conflict with each other.
- Systems typically use different types of locks (**lock modes**) to allow read operations to run concurrently.
  - **read locks** or **shared locks**: mode S
  - **write locks** or **exclusive locks**: mode X
- Locks are only in conflict if at least one of them is an X lock:

Shared vs. exclusive lock compatibility

	shared (S)	exclusive (X)
shared (S)		×
exclusive (X)	×	×

- It is a safe operation in two-phase locking to (try to) **convert a shared lock into an exclusive lock during the lock phase** (lock upgrade) ⇒ improved concurrency.

ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

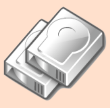
Deadlocks

- Like many lock-based protocols, two-phase locking has the risk of **deadlock** situations:

Example (Proper schedule with locking)

Transaction 1	Transaction 2
`lock` (A) ;	
⋮	`lock` (B) ;
do something	⋮
⋮	do something
`lock` (B) ;	⋮
[wait for T_2 to release lock]	`lock` (A) ;
	[wait for T_1 to release lock]

- Both transactions would wait for each other **indefinitely**.



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

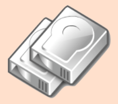
Two-Phase Locking

Multi-Version Concurrency Control

Deadlock Handling

- **Deadlock detection:**

  1. The system maintains a **waits-for graph**, where an edge $T_1 \rightarrow T_2$ indicates that T_1 is blocked by a lock held by T_2.
  2. Periodically, the system tests for **cycles** in the graph.
  3. If a cycle is detected, the deadlock is **resolved** by **aborting** one or more transactions.
  4. Selecting the **victim** is a challenge:
     - Aborting **young** transactions may lead to **starvation**: the same transaction may be cancelled again and again.
     - Aborting an **old** transaction may cause a lot of computational investment to be thrown away (but the **undo** costs may be high).



ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Deadlock Handling

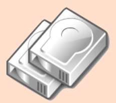
Other common technique:

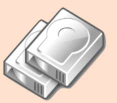
- **Deadlock detection via timeout:**
  Let a transaction T block on a `lock` request only until a **timeout** occurs (counter expires). On expiration, *assume* that a deadlock has occurred and **abort** T.

DB2 Timeout-based deadlock detection

```
db2 => GET DATABASE CONFIGURATION;
    ⋮
Interval for checking deadlock (ms)       (DLCHKTIME) = 10000
Lock timeout (sec)                      (LOCKTIMEOUT) = 30
    ⋮
```

- Also: lock-less **optimistic concurrency control** (↗ slide 0.0).



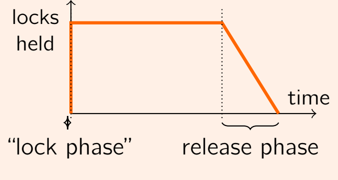


ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

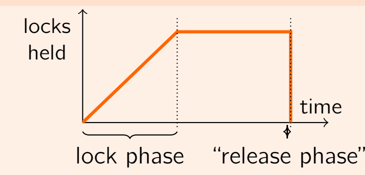
Variants of Two-Phase Locking

- The two-phase locking discipline does not prescribe exactly when locks have to be acquired and released.
- Two possible variants:

Preclaiming and strict 2PL



Preclaiming 2PL

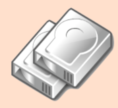


Strict 2PL

✎ What could motivate either variant?

1. Preclaiming 2PL:

2. Strict 2PL:

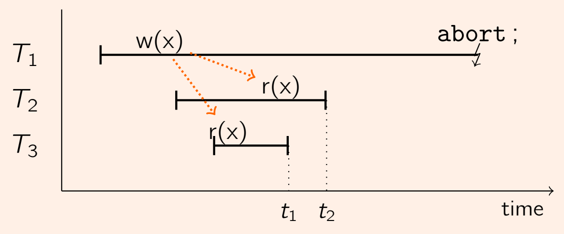


ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control

Cascading Rollbacks

Consider three transactions:

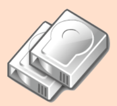
Transations $T_{1,2,3}$, T_1 fails later on



- When transaction T_1 aborts, transactions T_2 and T_3 have already read data written by T_1 (↗ dirty read, slide 9)
- T_2 and T_3 need to be **rolled back**, too (**cascading roll back**).
- T_2 and T_3 **cannot** commit until the fate of T_1 is known.

⇒ Strict 2PL can avoid cascading roll backs altogether. (**How?**)

Multi-Version Concurrency Control



ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Multi-Version Concurrency Control

Looking back at the concurrency control strategies discussed up to this point, we have seen

1. **Wait Mechanisms**, i.e., **locks** and the associated two-phase locking protocol.

We now add

2. **Timestamp Mechanisms** that use a **DBMS-wide clock** to order transactions and decide visibility of rows.

The resulting **Multi-Version Concurrency Control (MVCC)** protocol is lock-less but comes with a space overhead (that requires **garbage collection** of rows).

MVCC: Timestamps

- In MVCC, each transaction T_i is assigned a **timestamp** t_i that represents the point in time when T_i started.
- Can implement timestamps based on
  - **actual system clock** (resolution, uniqueness, portability across OSs?),
    or
  - **a DBMS-internal counter** used to assign transaction IDs (`xid`).
- Timestamp requirements:
  1. unique: $t_i \neq t_j$ if $i \neq j$,
  2. ordered: $t_i < t_j$ if T_i has started before T_j.

MVCC: Semantics

Under MVCC, a transaction T_i operates on **the consistent state of the database that was current at time t_i.** The resulting transaction semantics is also known as **snapshot isolation**.

ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Multi-Version Concurrency Control

MVCC: Versions and Snapshots

In a concurrent DBMS, operations can **conflict** if they write the **same database object** (recall relation $\not\leftrightarrow$). Thus:

MVCC: Versions

Under MVCC, **multiple versions of the same database object** may exist at one time. Different transactions may read/write different (not necessarily the most recent) object versions.

MVCC uses **snapshots** to identify exactly which version of each database object are visible to a transaction:

MVCC: Snapshot

To take a **snapshot**, gather the following information:

- the highest `xid` of all committed transactions,
- a list of `xid`s of all transactions currently executing.

Typically, a snapshot is taken at the time the transaction starts.

ACID Properties

Anomalies

The Scheduler

Serializability

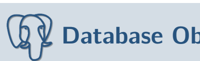
Query Scheduling

Locking

Two-Phase Locking

Multi-Version Concurrency Control

MVCC: Row Timestamps



Database Object ≡ Row

PostgreSQL implements MVCC at a granularity of rows: **multiple versions of the same row** may exist. Adopt this model in what follows.

- To help decide whether a particular row version is included in (or excluded from) a snapshot, attach to each row version two virtual/hidden attributes:
  1. `xmin`: the `xid` of the transaction that **created** this row,
  2. `xmax`: the `xid` of the transaction that **deleted** this row.
- Row **updates** are modellled as the two-step operation row deletion, then creation.

MVCC: No Physical Deletion!

Note: ② implies that rows are **not actually physically deleted**. Instead, their `xmax` attribute is modified to record the deleting/updating transaction (⇒ eventual row garbage).

ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Multi-Version Concurrency Control

MVCC: Row Visibility (1)

Example (Decide Row Visibility)

- Current snapshot:[1] $\langle \underbrace{100}_{\text{highest committed } \texttt{xid}} , \underbrace{[25, 50, 75]}_{\text{currently active } \texttt{xids}} \rangle$
- Row visible in snapshot?

1.

xmin	xmax	⋯ *data* ⋯
30	—	⋯

✓

2.

xmin	xmax	⋯ *data* ⋯
50	—	⋯

↯

3.

xmin	xmax	⋯ *data* ⋯
110	—	⋯

↯

[1] For simplicity: assume that all other `xids` have committed (and not rolled back their work).

ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Multi-Version Concurrency Control

MVCC: Row Visibility (2)

Example (Decide Row Visibility)

- Current snapshot: ⟨ 100 (highest committed `xid`), [25, 50, 75] (currently active `xids`) ⟩
- Row visible in snapshot?

1.

xmin	xmax	··· *data* ···
30	80	···

↯

2.

xmin	xmax	··· *data* ···
30	75	···

✓

3.

xmin	xmax	··· *data* ···
30	110	···

✓

- Given the current state of the system, may row 1 be considered garbage that can be collected?

ACID Properties

Anomalies

The Scheduler

Serializability

Query Scheduling

Locking

Two-Phase Locking

Multi-Version Concurrency Control

MVCC: Garbage Collection

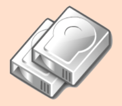
- The creation of new row versions during `UPDATE` (rather than replacing the existing row) requires the **reclamation of storage space** used by old row versions.
- Delay such **row garbage collection** until the old versions are guaranteed to be invisible to all current and future transactions.

Delayed Row Garbage Collection

Exactly when is it safe to declare a row as garbage and mark it for collection?

Row Cleanup

- **On-demand cleanup of a single page** when page is accessed during `SELECT`, `UPDATE`, `DELETE`.
- **Bulk cleanup** by scheduled auto-vacuum process or via an explicit `VACUUM` command.



ACID Properties
Anomalies
The Scheduler
Serializability
Query Scheduling
Locking
Two-Phase Locking
Multi-Version Concurrency Control