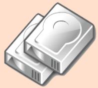
Chapter 11

Transaction Management

Concurrent and Consistent Data Access

Architecture and Implementation of Database Systems
Summer 2014

Torsten Grust
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen



The “Hello World” of Transaction Management

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

Example (ATM transaction)

$bal \leftarrow$ `read_bal` $(acct\_no)$;
$bal \leftarrow bal - 100$ € ;
`write_bal` $(acct\_no, bal)$;

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

Concurrent Access

The problem is: My wife has 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

$bal \leftarrow \texttt{read}\,(acct)\,;$

$bal \leftarrow bal - 100\,;$

$\texttt{write}\,(acct\,,\,bal)\,;$

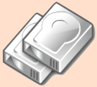
Concurrent Access

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

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

Example (Concurrent ATM transactions)

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



Concurrent Access

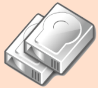
The problem is: My wife has a card for the very same account, too.

⇒ 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!



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) ;
```

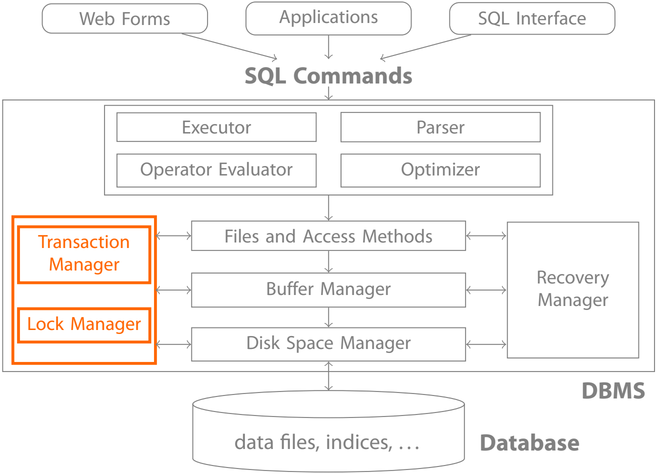
- Before the transaction gets to step **7**, its execution is **interrupted/cancelled** (power outage, disk failure, software bug, . . .). My money is **lost** ☹.

ACID Properties

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.

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.

Anomalies: Inconsistent Read

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

Example

Transaction 1	Transaction 2
`UPDATE Accounts SET balance = balance - 500 WHERE customer = 1904 AND account_type = 'C';`	
	`SELECT SUM(balance) FROM Accounts WHERE customer = 1904;`
`UPDATE Accounts SET balance = balance + 500 WHERE customer = 1904 AND account_type = 'S';`	

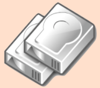
- Transaction 2 sees a temporary, **inconsistent** database state.

Anomalies: Dirty Read

At a different day, my wife and me again end up in front of an ATM 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

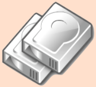


Anomalies: Dirty Read

At a different day, my wife and me again end up in front of an ATM 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



Anomalies: Dirty Read

At a different day, my wife and me again end up in front of an ATM at roughly the same time. This time, my transaction is cancelled (**aborted**):

Example

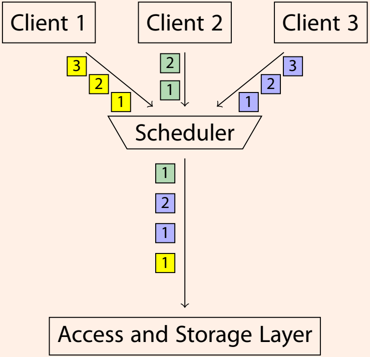
me	**my wife**	**DB state**
$bal \leftarrow \texttt{read}\,(acct)\,;$		1200
$bal \leftarrow bal - 100\,;$		1200
$\texttt{write}\,(acct, bal)\,;$		1100
	$bal \leftarrow \texttt{read}\,(acct)\,;$	1100
	$bal \leftarrow bal - 200\,;$	1100
`abort;`		1200
	$\texttt{write}\,(acct, bal)\,;$	900

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

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:

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

Transactions

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 $T = \langle s_1, \ldots, s_n \rangle$
- 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,

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

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

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

$\boxed{2}$
$\boxed{1}$
$\boxed{1}$

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

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

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:

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

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.

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

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**.

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 ($\nearrow$ 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.

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`		$\times$
`write`	$\times$	$\times$

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

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

Conflict Serializability

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

Example (Three 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**.

Serialization Graph

Example (ATM transactions (↗ slide 3))

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

$$\begin{tikzcd} T_1 \arrow[r, bend left] & T_2 \arrow[l, bend left] \end{tikzcd}$$

Serialization Graph

Example (ATM transactions (↗ slide 3))

- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$
- Conflict relation:

  $(T_1, \mathtt{r}, A) \prec_S (T_2, \mathtt{w}, A)$
  $(T_2, \mathtt{r}, A) \prec_S (T_1, \mathtt{w}, A)$
  $(T_1, \mathtt{w}, A) \prec_S (T_2, \mathtt{w}, A)$

$$\begin{tikzcd} & T_2 \arrow[dl, bend left] \\ T_1 \arrow[ur, bend left] & \end{tikzcd}$$

$\Rightarrow$ **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:

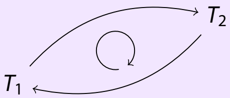
  $(T_1, \mathtt{r}, C) \prec_S (T_2, \mathtt{w}, C)$
  $(T_1, \mathtt{w}, C) \prec_S (T_2, \mathtt{r}, C)$
  $(T_1, \mathtt{w}, C) \prec_S (T_2, \mathtt{w}, C)$
  $\vdots$

$$\begin{tikzcd} & T_2 \\ T_1 \arrow[ur, bend left] & \end{tikzcd}$$

Serialization Graph

Example (ATM transactions (↗ slide 3))

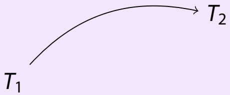
- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$
- Conflict relation:
  $(T_1, \mathtt{r}, A) \prec_S (T_2, \mathtt{w}, A)$
  $(T_2, \mathtt{r}, A) \prec_S (T_1, \mathtt{w}, A)$
  $(T_1, \mathtt{w}, A) \prec_S (T_2, \mathtt{w}, A)$



$\Rightarrow$ **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:
  $(T_1, \mathtt{r}, C) \prec_S (T_2, \mathtt{w}, C)$
  $(T_1, \mathtt{w}, C) \prec_S (T_2, \mathtt{r}, C)$
  $(T_1, \mathtt{w}, C) \prec_S (T_2, \mathtt{w}, C)$
  $\vdots$



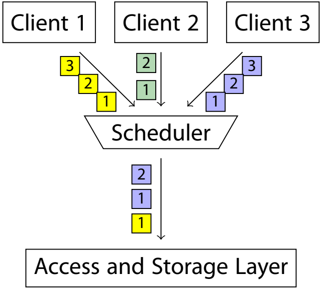
$\Rightarrow$ serializable

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*

```
lock o ;
...access o ...;
unlock o ;
```

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

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**.

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

Locking and Scheduling

Example (Locking and scheduling)

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

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

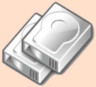
T_1	T_2
lock A	
lock C	
write A	
write C	
unlock A	
	lock A
	write A
	lock B
	unlock A
	write B
	unlock B
unlock C	
lock B	
write B	
unlock B	
	lock C
	write C
	unlock C

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

ATM Transaction with Locking

Example (Two concurrent ATM transactions with locking)

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



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*) ;	

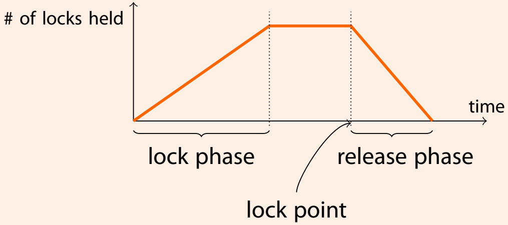
$\Rightarrow$ Again: on its own, proper locking does **not** guarantee serializability yet.

Two-Phase Locking (2PL)

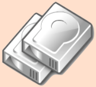
The **two-phase locking protocol** poses an additional restriction on how transactions have to be written:

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.



Again: ATM Transaction

Example (Two concurrent ATM transactions with locking, ¬ 2PL)

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*) ;	

Again: ATM Transaction

Example (Two concurrent ATM transactions with locking, ¬ 2PL)

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*) ;	

↯ These `lock`s violate the 2PL principle.

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:

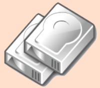
A 2PL-compliant ATM withdrawal transaction

```
lock (acct) ;                         } lock phase
bal ← read_bal (acct) ;
bal ← bal − 100 € ;
write_bal (acct, bal) ;
unlock (acct) ;                       } unlock phase
```

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*) ;	

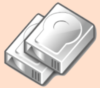


Resulting Schedule

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*`) ;`	

- **Theorem:** The use of 2PL-compliant locking **always** leads to a correct and **serializable** schedule or to a **deadlock**.



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.

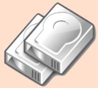
Deadlocks

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

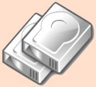
Example (Proper schedule with locking)

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**.



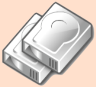
Deadlock Handling



- **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).

Deadlock Handling

- **Deadlock detection:**
  1. The system maintains a **waits-for graph**, where an edge $T_1 \to T_2$ 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).
- **Deadlock prevention:**
  Define an **ordering $\ll$ on all database objects**. If $A \ll B$, then order the `lock` operations in all transactions in the same way (`lock(`A`)` before `lock(`B`)`).



Deadlock Handling

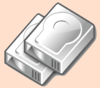
Other common technique:

- **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
    :
```

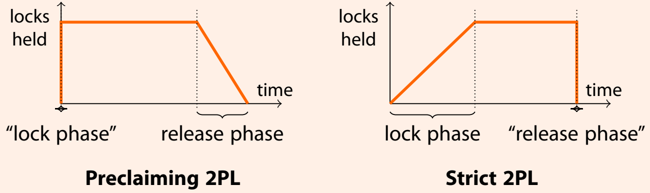
- Also: lock-less **optimistic concurrency control** (↗ slide 42).



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



✎ What could motivate either variant?

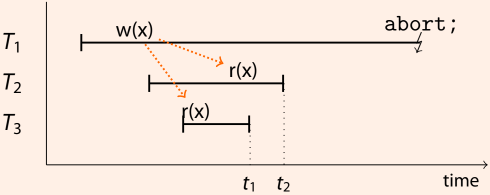
1. Preclaiming 2PL:

2. Strict 2PL:

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?**)

Granularity of Locking

The **granularity** of locking is a trade-off:

database level	low concurrency	low overhead
↓		
tablespace level[1]		
↓		
table level		
↓		
page level		
↓		
row-level	high concurrency	high overhead

⇒ **Idea: multi-granularity** locking.

[1] An DB2 tablespace represents a collection of tables that share a physical storage location.

Multi-Granularity Locking

- Decide the granularity of locks held **for each transaction** (depending on the characteristics of the transaction):
  - For example, aquire a **row lock** for

Row-selecting query (`C_CUSTKEY` is key)

```
SELECT *
FROM   CUSTOMERS
WHERE  C_CUSTKEY = 42
```

Q_1

and a **table lock** for

Table scan query

```
SELECT *
FROM   CUSTOMERS
```

Q_2

- How do such transactions know about each others' locks?
  - Note that locking is **performance-critical**. Q_2 does not want to do an extensive search for row-level conflicts.

Intention Locks

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

- Lock mode **intention share**: `IS`
- Lock mode **intention exclusive**: `IX`

Extended conflict matrix

	S	X	IS	IX
S		×		×
X	×	×	×	×
IS		×		
IX	×	×		

- A lock I□ on a coarser level of granularity means that there is some □ lock on a lower level.

Intention Locks

Multi-granularity locking protocol

1. Before a granule g can be locked in $\square \in \{\mathtt{S}, \mathtt{X}\}$ mode, the transaction has to obtain an $\mathtt{I}\square$ lock on **all** coarser granularities that contain g.
2. If all intention locks could be granted, the transaction can lock granule g in the announced $\square$ mode.

Example (Multi-granularity locking)

Query Q_1 (↗ slide 38) would, *e.g.*,

- obtain an `IS` lock on **table** `CUSTOMERS` (also on the containing tablespace and database) and
- obtain an `S` lock on the **row(s)** with `C_CUSTKEY = 42`.

Query Q_2 would place an

- `S` lock on `table CUSTOMERS` (and an `IS` lock on tablespace and database).

Detecting Conflicts

- Now suppose an updating query comes in:

Update request

Q_3

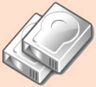
```
UPDATE CUSTOMERS
SET NAME = 'Seven␣Teen'
WHERE C_CUSTKEY = 17
```

- Q_3 will want to place
  - an `IX` lock on **table** `CUSTOMER` (and all coarser granules) and
  - an `X` lock on the **row** holding customer `17`.

  As such it is
  - **compatible** with Q_1 (there is no conflict between `IX` and `IS` on the table level),
  - but **incompatible** with Q_2 (the table-level `S` lock held by Q_2 is in **conflict** with Q_3's `IX` lock request).

Optimistic Concurrency Control

- Up to here, the approach to concurrency control has been **pessimistic**:
  - Assume that transactions **will conflict** and thus protect database objects by locks and lock protocols.
- The converse is a **optimistic concurrency control** approach:
  - Hope for the best and let transactions freely proceed with their `read`/`write` operations.
  - Only just before updates are to be committed to the database, perform a check to see whether conflicts indeed did not happen.
- Rationale: Non-serializable conflicts are not that frequent. **Save the locking overhead** in the majority of cases and only invest effort if really required.



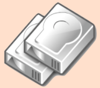
Optimistic Concurrency Control

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

Optimistic concurrency control

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 it is 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* (thus also called the **val-write phase**).



Validating Transactions

Validation is typically implemented by looking at transaction T_j's

- **Read Set** $RS(T_j)$ (attributes read by T_j) and
- **Write Set** $WS(T_j)$ (attributes written by T_j).

Backward-oriented optimistic concurrency control (BOCC)

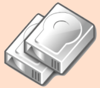
Compare T_j against all **committed** transactions T_i.
Check **succeeds** if

$$T_i \text{ committed before } T_j \text{ started} \quad \textbf{or} \quad RS(T_j) \cap WS(T_i) = \varnothing \enspace .$$

Forward-oriented optimistic concurrency control (FOCC)

Compare T_j against all **running** transactions T_i.
Check **succeeds** if

$$WS(T_j) \cap RS(T_i) = \varnothing \enspace .$$



Optimistic Concurrency Control in IBM DB2

- DB2 V9.5 provides SQL-level constructs that enable database applications to implement optimistic concurrency control:
  - `RID(`*r*`)`: return row identifier for row *r*,
  - `ROW CHANGE TOKEN FOR` *r*: unique number reflecting the time row *r* has last been updated.

DB2® Optimistic concurrency control

```
db2 => SELECT * FROM EMPLOYEES

ID          NAME DEPT SALARY
----------- ---- ---- -----------
          1 Alex DE           300
          2 Bert DE           100
          3 Cora DE           200
          4 Drew US           200
          5 Erik US           400

db2 => SELECT E.NAME, E.SALARY,
              RID(E) AS RID, ROW CHANGE TOKEN FOR E AS TOKEN
       FROM   EMPLOYEES E
       WHERE  E.NAME = 'Erik'

NAME SALARY      RID                  TOKEN
---- ----------- -------------------- --------------------------
Erik         400             16777224             74904229642240
```

Optimistic Concurrency Control in IBM DB2

DB2® Optimistic concurrency control

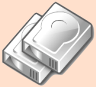
- The 'Erik' row belongs to our read set. Save its RID and TOKEN values to perform validation later.
- (…Time passes …) Now try to save our changes to the row. Perform BOCC-style validation:

```
db2 => UPDATE EMPLOYEES E
       SET E.SALARY = 450
       WHERE RID(E) = 16777224                      -- identify row
       AND ROW CHANGE TOKEN FOR E = 74904229642240 -- row changed?

SQL0100W  No row was found for FETCH, UPDATE or DELETE; or the
result of a  query is an empty table.  SQLSTATE=02000

db2 => SELECT E.ID, E.NAME
              RID(E) AS RID, ROW CHANGE TOKEN FOR E AS TOKEN
       FROM EMPLOYEES E

ID          NAME  RID                  TOKEN
----------- ----  -------------------- --------------------
          1 Alex              16777220       74904229642240
          2 Bert              16777221       74904229642240
          3 Cora              16777222       74904229642240
          4 Drew              16777223       74904229642240
          5 Erik              16777224   141378732653941710
```



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,
2. **Rollback Mechanisms**, i.e., a conditional write phase that makes it trivial to **take back any changes** made by a transaction.

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,
2. **Rollback Mechanisms**, i.e., a conditional write phase that makes it trivial to **take back any changes** made by a transaction.

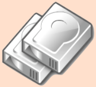
We now add

3. **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.

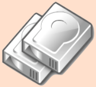


MVCC: Timestamps

- 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.**



MVCC: Versions and Snapshots

In a concurrent DBMS, operations can **conflict** if they write the **same database object** (recall relation $\nleftrightarrow$). 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: Versions and Snapshots

In a concurrent DBMS, operations can **conflict** if they write the **same database object** (recall relation $\nleftrightarrow$). 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.

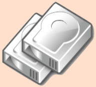
MVCC uses **snapshots** to identify exactly which version of each database object are visible to a transaction:

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.



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: 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!

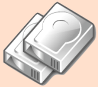
Note: 2 implies that rows are **not actually physically deleted**. Instead, their `xmax` attribute is modified to record the deleting/updating transaction (⇒ eventual row garbage).

MVCC: Row Visibility (1)

Example (Decide Row Visibility)

- Current snapshot:[2] $\langle \underbrace{100}_{\text{highest committed } \texttt{xid}} , \underbrace{[25, 50, 75]}_{\text{currently active } \texttt{xids}} \rangle$

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



MVCC: Row Visibility (1)

Example (Decide Row Visibility)

- Current snapshot:[2] $\left\langle \underbrace{100}_{\text{highest committed } \texttt{xid}}, \underbrace{[25, 50, 75]}_{\text{currently active } \texttt{xids}} \right\rangle$
- Row visible in snapshot?

1

xmin	xmax	$\cdots$ *data* $\cdots$
30	—	$\cdots$

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

MVCC: Row Visibility (1)

Example (Decide Row Visibility)

- Current snapshot:[2] $\langle \underbrace{100}_{\text{highest committed } \texttt{xid}}, \underbrace{[25, 50, 75]}_{\text{currently active } \texttt{xids}} \rangle$
- Row visible in snapshot?

1

xmin	xmax	⋯ *data* ⋯
30	—	⋯

✓

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

MVCC: Row Visibility (1)

Example (Decide Row Visibility)

- Current snapshot:[2] $\langle \underbrace{100}_{\text{highest committed } \mathtt{xid}} , \underbrace{[25, 50, 75]}_{\text{currently active } \mathtt{xids}} \rangle$
- Row visible in snapshot?

1

xmin	xmax	··· *data* ···
30	—	···

✓

2

xmin	xmax	··· *data* ···
50	—	···

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

MVCC: Row Visibility (1)

Example (Decide Row Visibility)

- Current snapshot:[2] $\langle \underbrace{100}_{\text{highest committed } \texttt{xid}} , \underbrace{[25, 50, 75]}_{\text{currently active } \texttt{xids}} \rangle$
- Row visible in snapshot?

1

xmin	xmax	$\cdots$ *data* $\cdots$
30	—	$\cdots$

$\checkmark$

2

xmin	xmax	$\cdots$ *data* $\cdots$
50	—	$\cdots$

$\lightning$

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

MVCC: Row Visibility (1)

Example (Decide Row Visibility)

- Current snapshot:[2] $\langle \underbrace{100}_{\text{highest committed } \texttt{xid}} , \underbrace{[25, 50, 75]}_{\text{currently active } \texttt{xids}} \rangle$
- Row visible in snapshot?

1.

xmin	xmax	· · · *data* · · ·
30	—	· · ·

$\checkmark$

2.

xmin	xmax	· · · *data* · · ·
50	—	· · ·

$\lightning$

3.

xmin	xmax	· · · *data* · · ·
110	—	· · ·

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

MVCC: Row Visibility (1)

Example (Decide Row Visibility)

- Current snapshot:[2] $\langle \underbrace{100}_{\text{highest committed xid}} , \underbrace{[25, 50, 75]}_{\text{currently active xids}} \rangle$
- Row visible in snapshot?

1

xmin	xmax	· · · *data* · · ·
30	—	· · ·

✓

2

xmin	xmax	· · · *data* · · ·
50	—	· · ·

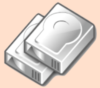
↯

3

xmin	xmax	· · · *data* · · ·
110	—	· · ·

↯

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



MVCC: Row Visibility (2)

Example (Decide Row Visibility)

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

1

xmin	xmax	$\cdots$ *data* $\cdots$
30	80	$\cdots$

MVCC: Row Visibility (2)

Example (Decide Row Visibility)

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

1

xmin	xmax	$\cdots$ *data* $\cdots$
30	80	$\cdots$

↯

MVCC: Row Visibility (2)

Example (Decide Row Visibility)

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

1

xmin	xmax	⋯ *data* ⋯
30	80	⋯

↯

2

xmin	xmax	⋯ *data* ⋯
30	75	⋯

MVCC: Row Visibility (2)

Example (Decide Row Visibility)

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

- Row visible in snapshot?

	xmin	xmax	⋯ *data* ⋯	
1	30	80	⋯	$\lightning$

	xmin	xmax	⋯ *data* ⋯	
2	30	75	⋯	$\checkmark$

MVCC: Row Visibility (2)

Example (Decide Row Visibility)

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

- Row visible in snapshot?

1

xmin	xmax	$\cdots$ *data* $\cdots$
30	80	$\cdots$

↯

2

xmin	xmax	$\cdots$ *data* $\cdots$
30	75	$\cdots$

✓

3

xmin	xmax	$\cdots$ *data* $\cdots$
30	110	$\cdots$

MVCC: Row Visibility (2)

Example (Decide Row Visibility)

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

1

xmin	xmax	⋯ *data* ⋯
30	80	⋯

↯

2

xmin	xmax	⋯ *data* ⋯
30	75	⋯

✓

3

xmin	xmax	⋯ *data* ⋯
30	110	⋯

✓

MVCC: Row Visibility (2)

Example (Decide Row Visibility)

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

- Row visible in snapshot?

1

xmin	xmax	··· *data* ···
30	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?

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.