Information Systems
(Informationssysteme)

Jens Teubner, TU Dortmund
jens.teubner@cs.tu-dortmund.de

Summer 2018

Part VIII

Transaction Management

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.

$bal \leftarrow$ **read_bal** $(acct\_no)$;
$bal \leftarrow bal - 100\,\text{EUR}$;
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 account, too.

- We might end up using our cards at different ATMs at the **same time**.

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!

Another Example

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

```
  // Subtract money from source (checking) account
1 chk_bal ← read_bal (chk_acct_no) ;
2 chk_bal ← chk_bal − 500 EUR ;
3 write_bal (chk_acct_no, chk_bal) ;
  // Credit money to the target (saving) account
4 sav_bal ← read_bal (sav_acct_no) ;
5 sav_bal ← sav_bal + 500 EUR ;
6 write_bal (sav_acct_no, sav_bal) ;
```

- Before the transaction gets to step **6**, its execution is **interrupted or cancelled** (power outage, disk failure, software bug, ...). My money is **lost** ☹.

ACID Properties

One of the key benefits of a database system are the **transaction properties** guaranteed to the user:

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.
I	Isolation	A transaction must not see any effect from other transactions that run in parallel.
D	Durability	The effects of a **successful** transaction maintain persistent and may not be undone for system reasons.

A challenge is to preserve these guarantees even with **multiple users** accessing the database **concurrently**.

Anomalies: Lost Update

- We already saw a **lost update** example on slide 257.
- The effects of one transaction are lost, because of an uncontrolled overwriting by the second transaction.

Anomalies: Inconsistent Read

Consider the money transfer example (slide 258), expressed in SQL syntax:

Transaction 1	Transaction 2
`UPDATE Accounts SET balance = balance - 500 WHERE customer = 4711 AND account_type = 'C';`	
	`SELECT SUM(balance) FROM Accounts WHERE customer = 4711;`
`UPDATE Accounts SET balance = balance + 500 WHERE customer = 4711 AND account_type = 'S';`	

- Transaction 2 sees an **inconsistent** database state.

Anomalies: Dirty Read

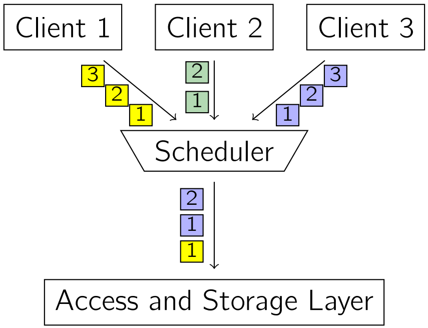
At a different day, my wife and me again end up in front of an ATM at roughly the same time:

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

- My wife's transaction has already read the modified account balance before my transaction was **rolled back**.

Concurrent Execution

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



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

$$\text{object} \equiv \text{tuple} \quad .$$

Transactions

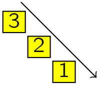
A **database transaction** T is a (strictly ordered) sequence of **steps**. Each **step** is a pair of an **access operation** applied to an **object**.

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

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

We could write the money transfer transaction as

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



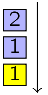
or, more concisely,

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

Schedules

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

We sometimes write

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

to mean

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

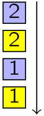
Serial Execution

One particular schedule is **serial execution**.

- A schedule S is **serial** iff, for each contained transaction T_j, all its steps follow each other (no interleaving of transactions).

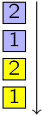
Consider again the ATM example from slide 257.

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



If my wife had gone to the bank one hour later, "our" 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 257 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**.
- Therefore, allow concurrent executions if they are **equivalent** to a 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 (↗ slide 266).
  - Rearranging operations must not lead to a different **result**.
- Two operations (a, e) and (a', e') are said to be **in conflict** $(a, e) \nleftrightarrow (a', e')$ 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.

- Two operations (T_i, a, e) and (T_j, a', e') are **in conflict** in S if
  1. they belong to two **different transactions** $(T_i \neq T_j)$,
  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 $\prec_S$:**

$$(T_i, a, e) \prec_S (T_j, a', e')$$
$$:=$$
$$(a, e) \not\leftrightarrow (a', e') \wedge (T_i, a, e) \text{ occurs before } (T_j, a', e') \text{ in } S \wedge T_i \neq T_j$$

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.**
  - S does **not** have to be a serial schedule.

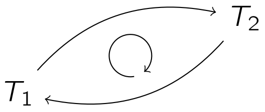
- This allows us to **prove** the correctness of a schedule S based on its **conflict graph** $G(S)$ (also: **serialization graph**).
  - **Nodes** are all transactions T_i in S.
  - There is an **edge** $T_i \rightarrow T_j$ iff S contains operations (T_i, a, e) and (T_j, a', e') such that $(T_i, a, e) \prec_S (T_j, a', e')$.
- S is conflict serializable if $G(S)$ is **acyclic**.[13]

[13] A serial execution of S could be obtained by sorting $G(S)$ **topologically**.

Serialization Graph

Example: ATM transactions (↗ slide 257)

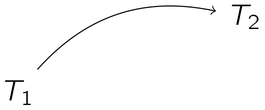
- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$
- Conflict relation:
  $(T_1, \mathtt{r}, A) \prec_S (T_2, \mathtt{w}, A)$
  $(T_2, \mathtt{r}, A) \prec_S (T_1, \mathtt{w}, A)$
  $(T_1, \mathtt{w}, A) \prec_S (T_2, \mathtt{w}, A)$



→ **not** serializable

Example: Two money transfers (↗ slide 258)

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

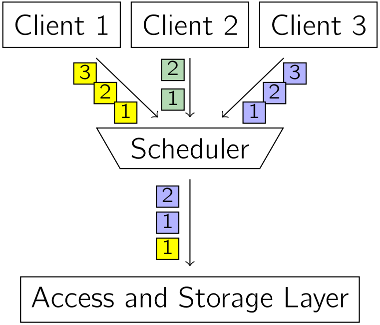
Query Scheduling

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*:

```
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_i 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**.

Locking and Serializability

✎ **Does locking guarantee serializable schedules, yet?**

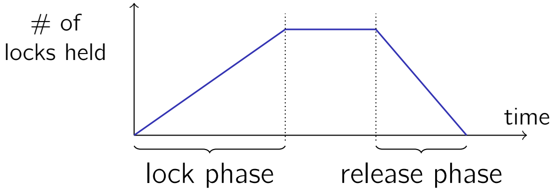
ATM Transaction with Locking

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) ;`	

Two-Phase Locking (2PL)

The **two-phase locking protocol** poses an additional restriction:

- Once a transaction has **released** any lock, it must **not** acquire any new lock.



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

Again: ATM Transaction

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*) ;	

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.

```
lock (acct) ;
bal ← read_bal (acct) ;
bal ← bal − 100 EUR ;
write_bal (acct, bal) ;
unlock (acct) ;
```

} lock phase

} unlock phase

Resulting Schedule

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

- The use of locking lead to a correct (and serializable) schedule.

Deadlocks

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

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

A typical approach to deal with deadlocks is **deadlock detection**:

- 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.
- Periodically, the system tests for **cycles** in the graph.
- If a cycle is detected, the deadlock is **resolved** by **aborting** one or more transactions.
- Selecting the **victim** is a challenge:
  - Blocking **young** transactions may lead to **starvation**: the same transaction is cancelled again and again.
  - Blocking an **old** transaction may cause a lot of investment to be thrown away.

Deadlock Handling

Other common techniques:

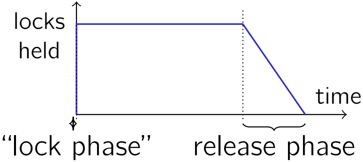
- **Deadlock prevention:** *e.g.*, by treating handling lock requests in an **asymmetric** way:
  - **wait-die**: A transaction is never blocked by an **older** transaction.
  - **wound-wait**: A transaction is never blocked by a **younger** transaction.
- **Timeout:** Only wait for a lock until a timeout expires. Otherwise assume that a deadlock has occurred and **abort**.

E.g., IBM DB2 UDB:

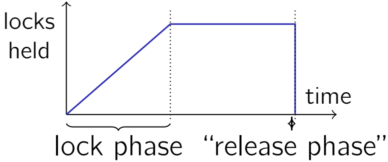
```
db2 => GET DATABASE CONFIGURATION;
  :
  :
Interval for checking deadlock (ms)      (DLCHKTIME) = 10000
Lock timeout (sec)                     (LOCKTIMEOUT) = -1
```

Variants of Two-Phase Locking

- The two-phase locking protocol does not prescribe exactly when locks have to acquired and released.
- Possible variants:



preclaiming 2PL

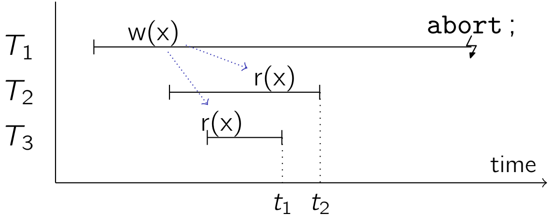


strict 2PL

- ✎ **What could motivate either variant?**

Cascading Rollbacks

Consider three transactions:



- When transaction T_1 aborts, transactions T_2 and T_3 have already read data written by T_1 (↗ dirty read, slide 262)
- T_2 and T_3 need to be **rolled back**, too.
- T_2 and T_3 **cannot** commit until the fate of T_1 is known.
- This problem cannot arise under strict two-phase locking.

Consistency Guarantees and SQL 92

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

- “Mistakes” in few data sets, *e.g.*, will not considerably affect the outcome of an aggregate over a huge table.
  - ⇝ Inconsistent read anomaly
- SQL 92 specifies different **isolation levels**.
- *E.g.*,

```
SET ISOLATION SERIALIZABLE;
```

- Obviously, less strict consistency guarantees should lead to increased throughput.

SQL 92 Isolation Levels

read uncommitted (also: 'dirty read' or 'browse')
: Only **write locks** are acquired (according to strict 2PL).

read committed (also: 'cursor stability')
: **Read locks** are only held for as long as a cursor sits on the particular row. **Write locks** acquired according to strict 2PL.

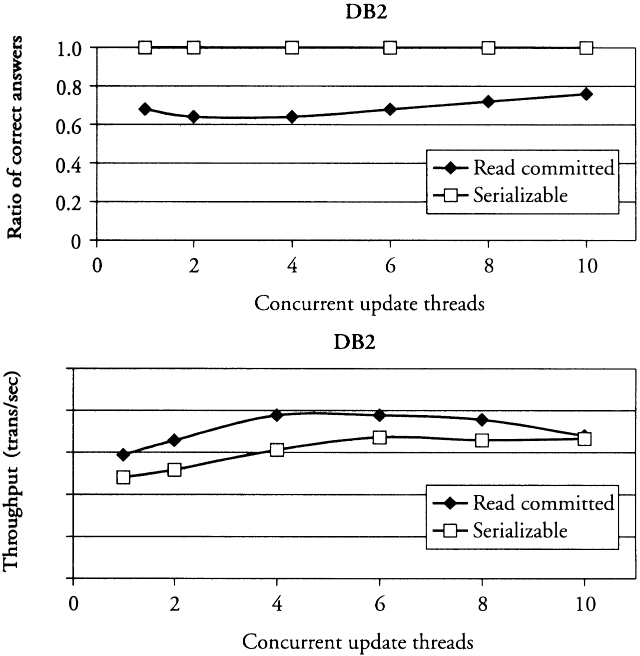
repeatable read (also: 'read stability')
: Acquires **read** and **write locks** according to strict 2PL.

serializable
: Additionally obtains locks to avoid **phantom reads**.

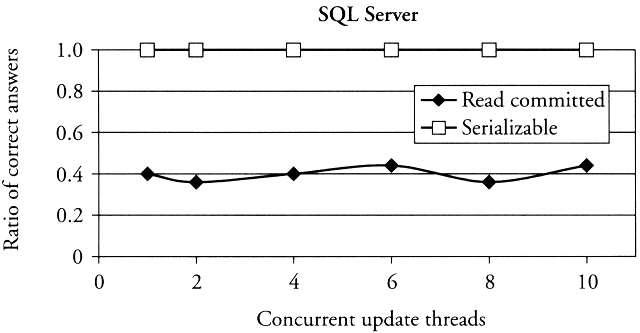
Phantom Problem

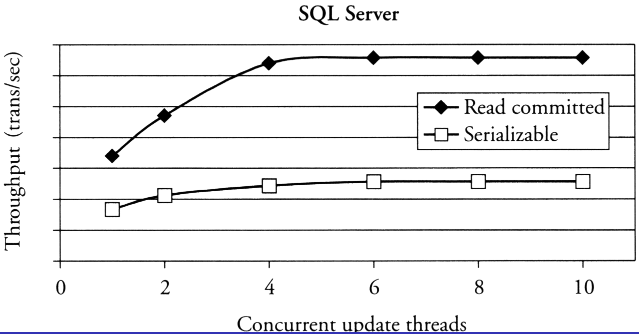
Transaction 1	Transaction 2	Effect
scan relation R ;		T_1 locks all rows
	insert new row into R ;	T_2 locks new row
	`commit` ;	T_2's lock released
scan relation R ;		reads **new** row, too!

- Although both transactions properly followed the 2PL protocol, T_1 observed an effect caused by T_2.
- Cause of the problem: T_1 can only lock **existing** rows.
- Possible solutions:
  - **Key range locking**, typically in B-trees
  - **Predicate locking**

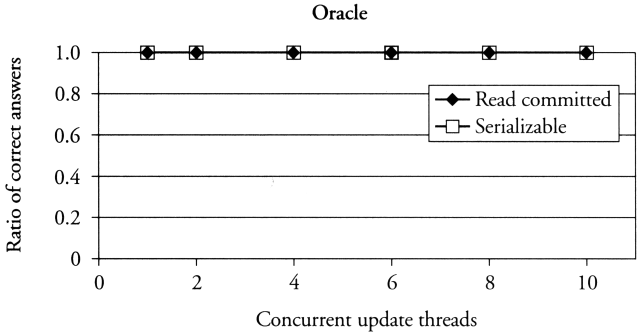


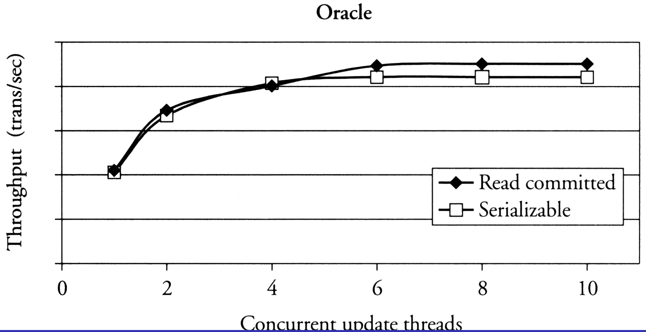
Dennis Shasha nad Philippe Bonnet. Database Tuning. Morgan Kaufmann, 2003.





Dennis Shasha nad Philippe Bonnet. Database Tuning. Morgan Kaufmann, 2003.





Dennis Shasha nad Philippe Bonnet. Database Tuning. Morgan Kaufmann, 2003.

Resulting Consistency Guarantees

isolation level	dirty read	non-repeat. rd	phantom rd
read uncommitted	possible	possible	possible
read committed	not possible	possible	possible
repeatable read	not possible	not possible	possible
serializable	not possible	not possible	not possible

- Some implementations support more, less, or different levels of isolation.
- Few applications really need serializability.

Optimistic Concurrency Control

- So far we've been rather **pessimistic**:
  - we've assumed the worst and prevented that from happening.
- In practice, conflict situations are not that frequent.
- **Optimistic concurrency control:** Hope for the best and only act in case of conflicts.

Optimistic Concurrency Control

Handle transactions in **three phases**:

1. **Read Phase.** Execute transaction, but do **not** write data back to disk immediately. Instead, collect updates in a **private workspace**.
2. **Validation Phase.** When the transaction wants to **commit**, test whether its execution was correct. If it is not, **abort** the transaction.
3. **Write Phase.** Transfer data from private workspace into database.

Validating Transactions

Validation is typically implemented by looking at transactions'

- **Read Sets** $RS(T_i)$**:** (attributes read by transaction T_i) and
- **Write Sets** $WS(T_i)$**:** (attributes written by transaction T_i).

backward-oriented optimistic concurrency control (BOCC):

Compare T against all **committed** transactions T_c.
Check **succeeds** if

$$T_c \text{ committed before } T \text{ started} \quad \textbf{or} \quad RS(T) \cap WS(T_c) = \varnothing \ .$$

forward-oriented optimistic concurrency control (FOCC):

Compare T against all **running** transactions T_r.
Check **succeeds** if

$$WS(T) \cap RS(T_r) = \varnothing \ .$$

Multiversion Concurrency Control

Consider the schedule

$$r_1(x), w_1(x), r_2(x), \overset{\color{red}{t \atop \downarrow}}{} w_2(y), r_1(y), w_1(z) \ .$$

✎ **Is this schedule serializable?**

- Now suppose when T_1 wants to read y, we'd still have the "old" value of y, valid at time t, around.
- We could then create a history equivalent to

$$r_1(x), w_1(x), r_2(x), r_1(y), w_2(y), w_1(z) \ ,$$

which is **serializable**.

Multiversion Concurrency Control

- With old **object versions** still around, **read** transactions need no longer be blocked.
- They might see **outdated, but consistent** versions of data.
- **Problem:** Versioning requires **space** and **management overhead** ($\rightsquigarrow$ garbage collection).

- Some systems support **snapshot isolation**.
  - Oracle, SQL Server, PostgreSQL