Information Systems

Computer Science Department
ETH Zurich
Spring 2012

Lecture VI: Transaction Management (Concurrency Control)

An Example

- My bank issued a debit card for me 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.
- My account is properly updated to reflect the new balance.

```
1 bal ← read_bal (acct_no) ;
2 bal ← bal – 100 CHF ;
3 write_bal (acct_no, bal) ;
```

Concurrent Access

- Problem: My husband has a card for the same account, too.
  - We might end up using our cards at different ATMs at the **same time**.

me	my husband	DB state
$bal \leftarrow$ read $(acct)$;		1200
	$bal \leftarrow$ read $(acct)$;	1200
$bal \leftarrow bal - 100$;		1200
	$bal \leftarrow bal - 200$;	1200
write $(acct, bal)$;		1100
	write $(acct, bal)$;	1000

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

Failures

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

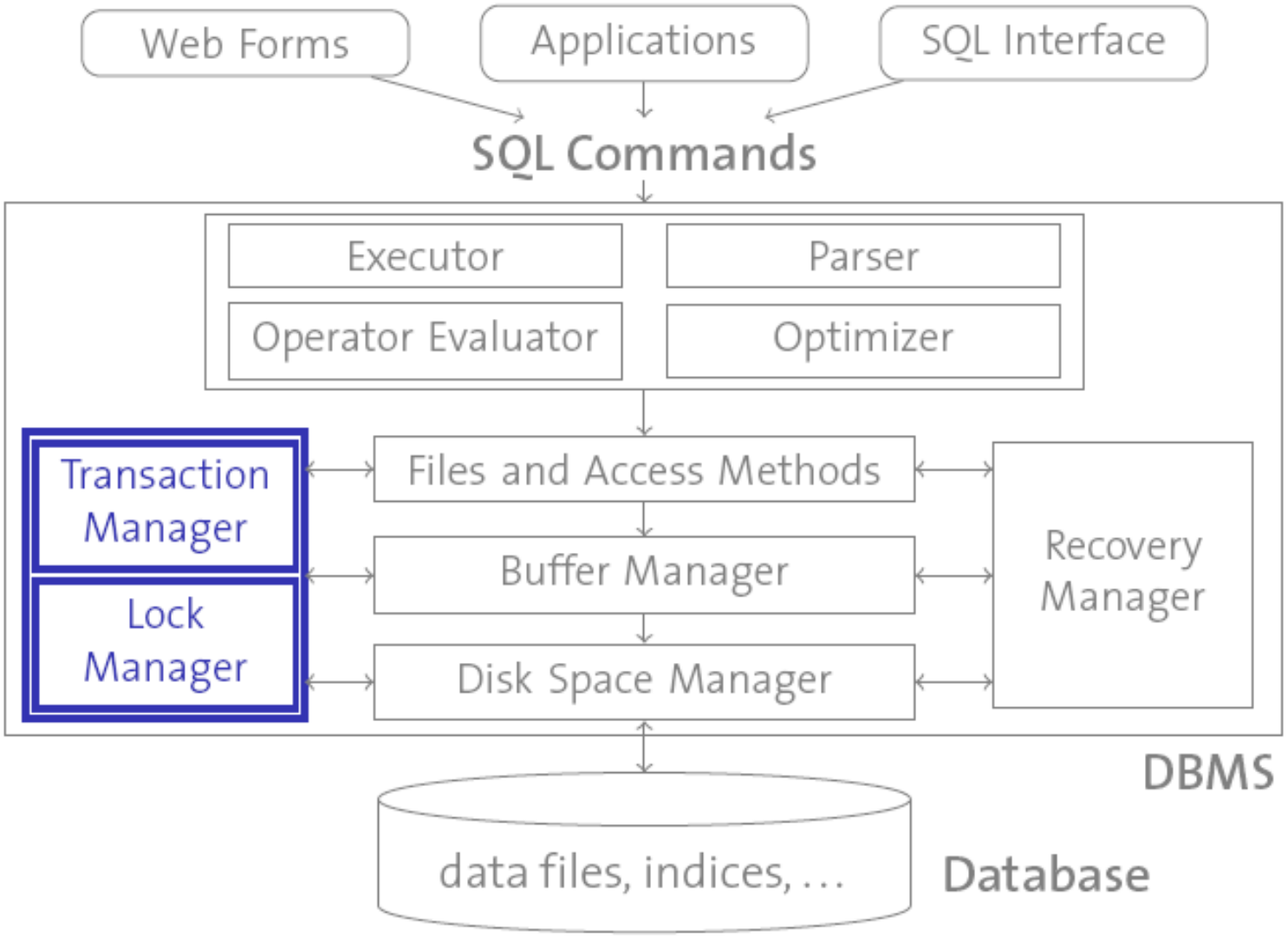
```
   // Subtract money from source (checking) account
1  chk_bal ← read_bal (chk_acct_no) ;
2  chk_bal ← chk_bal – 500 CHF ;
3  write_bal (chk_acct_no, chk_bal) ;
   // Credit money to the target (saving) account
4  sav_bal ← read_bal (sav_acct_no) ;
5  sav_bal ← sav_bal + 500 CHF ;
6  write_bal (sav_acct_no,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

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

ACID Properties

- To prevent these (and many other) effects from happening, a DBMS must guarantee the following transactional properties:
  - **A**tomicity : Either all or none of the updates in a database transaction are applied.
  - **C**onsistency : Every transaction brings the database from one consistent state to another.
  - **I**solation : A transaction must not see any effect from other transactions that run in parallel.
  - **D**urability : The effects of a successful transaction are made persistent and cannot be undone for system reasons.

Concurrency Control



Anomalies: Lost Update

- The effects of one transaction are lost because of an uncontrolled overwriting by another transaction.

- We saw an example of this anomaly in slide #4 (the effect of my money withdrawal transaction was overwritten by my husband's).

Anomalies: Inconsistent Read

- Consider the money transfer example in slide #5, 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

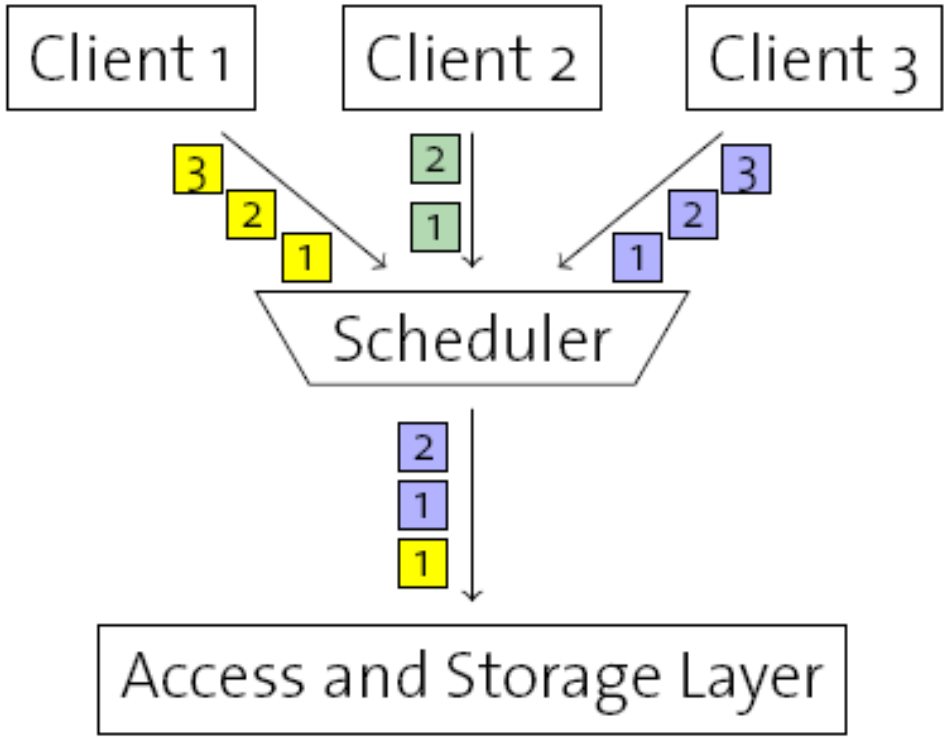
- At a different day, my husband and me again end up in front of two different ATMs at roughly the same time:

me	my husband	DB state
bal ← read (*acct*) ;		1200
bal ← *bal* − 100 ;		1200
write (*acct*, *bal*) ;		1100
	bal ← read (*acct*) ;	1100
	bal ← *bal* − 200 ;	1100
abort ;		1200
	write (*acct*, *bal*) ;	900

- My husband'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 typically have that:

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

Transactions

- A **database transaction** *T* is a (strictly ordered) sequence of **steps**. Each **step** (s_i) is a pair of an **access operation** (a_i) applied to an **object** (e_i).
  - Transaction $T = \langle s_1, \ldots, s_n \rangle$
  - Step $s_i = (a_i, e_i)$
  - Access operation $a_i \in \{r(ead), w(rite)\}$
- The **length** of a transaction *T* is its number of steps $|T| = n$.
- We could write the money transfer transaction as:

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

- More concisely:

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

Schedules

- A schedule S for a given set of transactions $T = \{T_1, \ldots, T_n\}$ 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 for the serial execution.
  - A schedule S is serial, iff for each contained transaction T_j, all its steps follow each other (i.e., no interleaving of transactions).
- Consider again the ATM example from slide #4.

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

This schedule is **not serial**.

- If my husband 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 #4 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**.

Conflict Equivalence

- 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.
  - Rearranging operations must not lead to a different result.
- Two operations (*a*, *e*) and (*a'*, *e'*) are said to be **in conflict** (*a*, *e*) $\not\leftrightarrow$ (*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**:

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

Conflict Serializability

- A schedule *S* is **conflict serializable**, iff it is conflict equivalent to **some** serial schedule *S'*.
- **The execution of a conflict serializable schedule *S* 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)* (a.k.a., **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:

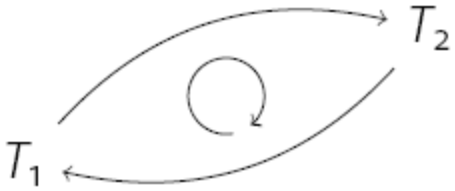
$$(T_i, a, e) \prec_S (T_j, a', e')$$

- *S* is conflict serializable if *G(S)* is **acyclic** (i.e., a serial execution of *S* could be obtained by sorting *G(S)* topologically.)

Serialization Graph

- Example: ATM transactions (slide #4)
  - $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$
  - 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: Money transfers
  - $S = \langle r_1(C), w_1(C), r_2(C), w_2(C), r_1(S), w_1(S), r_2(S), w_2(S) \rangle$
  - Conflict relation:

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

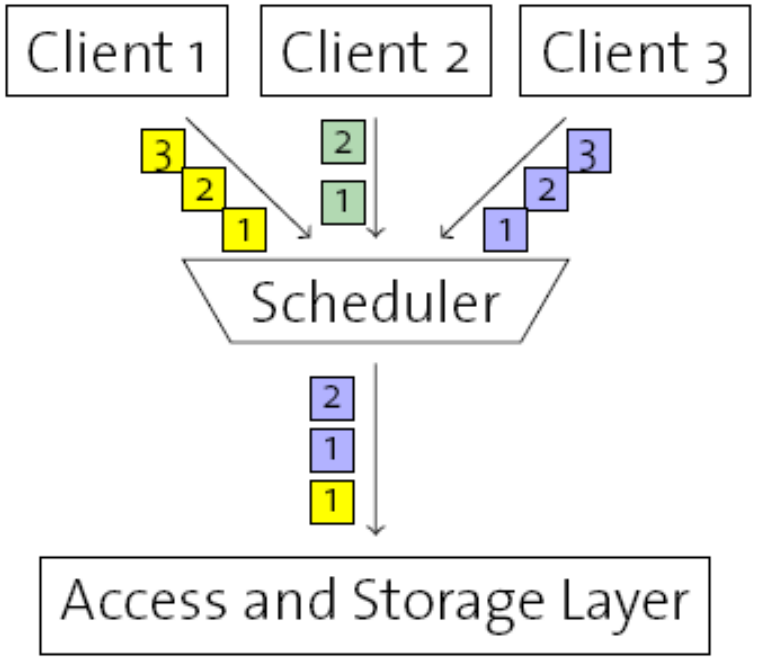
$\rightarrow$ serializable

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:*

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

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



Locking

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

ATM Transaction Example with Locking

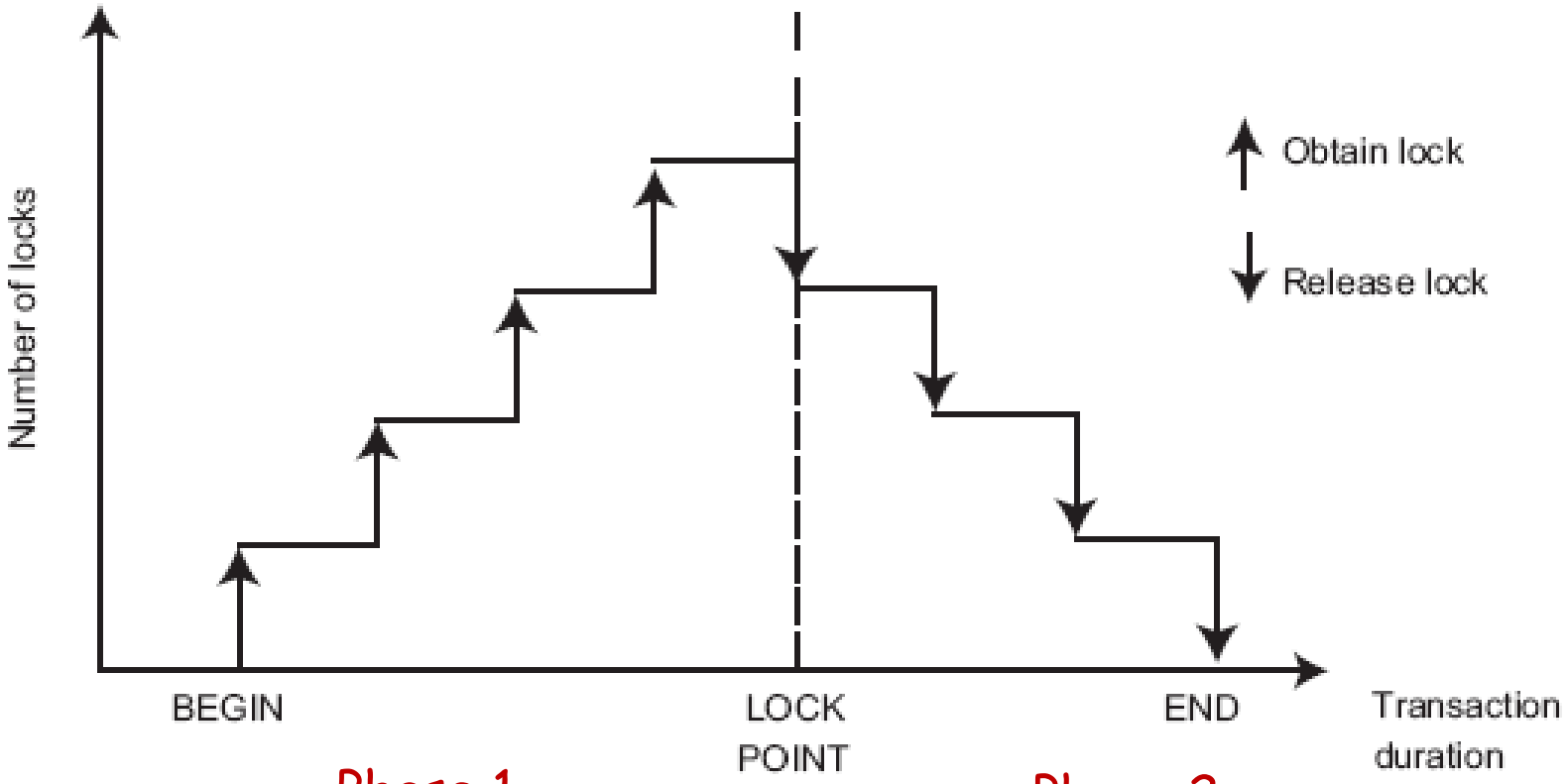
Transaction 1	Transaction 2	DB state
`lock (acct) ;`		1200
`read (acct) ;`		
`unlock (acct) ;`		
	`lock (acct) ;`	
	`read (acct) ;`	
	`unlock (acct) ;`	
`lock (acct) ;`		
`write (acct) ;`		1100
`unlock (acct) ;`		
	`lock (acct) ;`	
	`write (acct) ;`	1000
	`unlock (acct) ;`	

- Is locking by itself enough to guarantee serializable schedules?

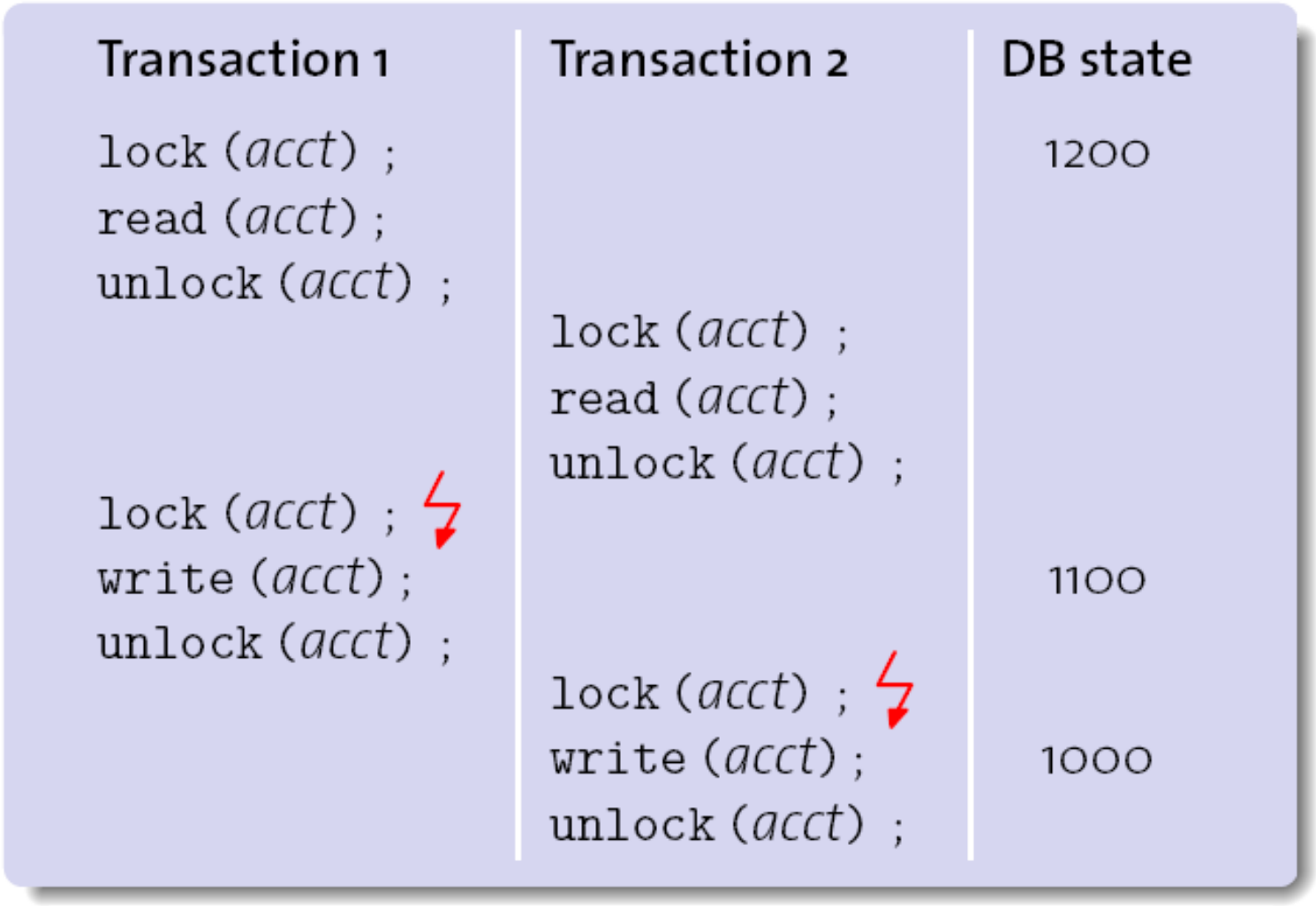
Two-Phase Locking (2PL)

- To generate serializable histories, the locking and releasing operations of transactions must be coordinated.
- 2PL achieves the coordination as follows:
  - A transaction must lock an object before using it.
  - When an object is locked by another transaction, the requesting transaction must wait.
  - Once a transaction releases any lock, it cannot request additional locks.
- Any history generated by a concurrency algorithm that obeys the 2PL rule is serializable.

Two-Phase Locking (2PL)



ATM Transaction Example violates 2PL



Making the ATM Transaction 2PL-Compliant

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

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

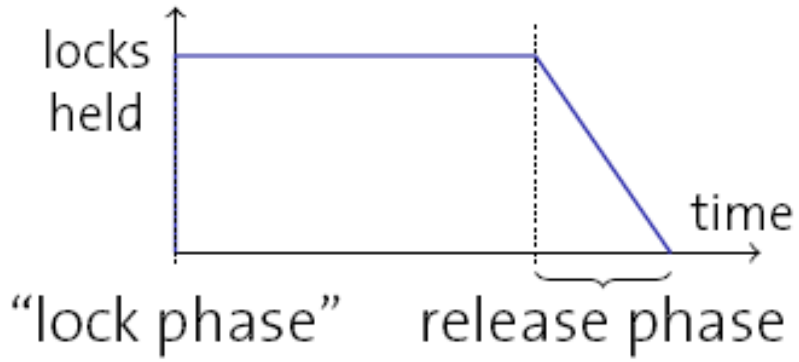
Resulting ATM Transaction Schedule

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) ;`	

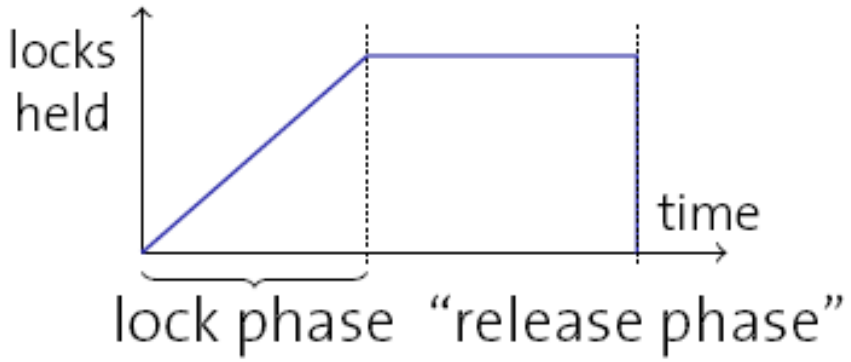
- Use of locking with 2PL leads to a correct (and serializable) schedule.

Variants of 2PL

- The two-phase locking protocol does not prescribe exactly when locks have to be acquired and released.
- Variants are possible. Examples:



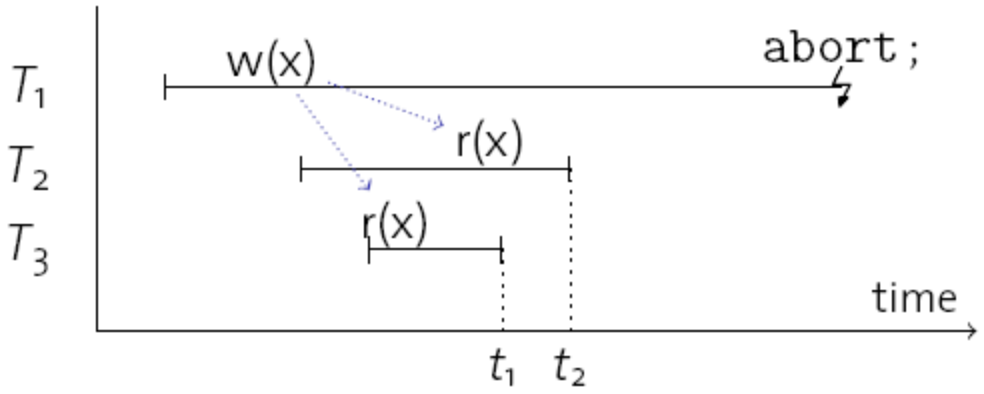
preclaiming 2PL



strict 2PL

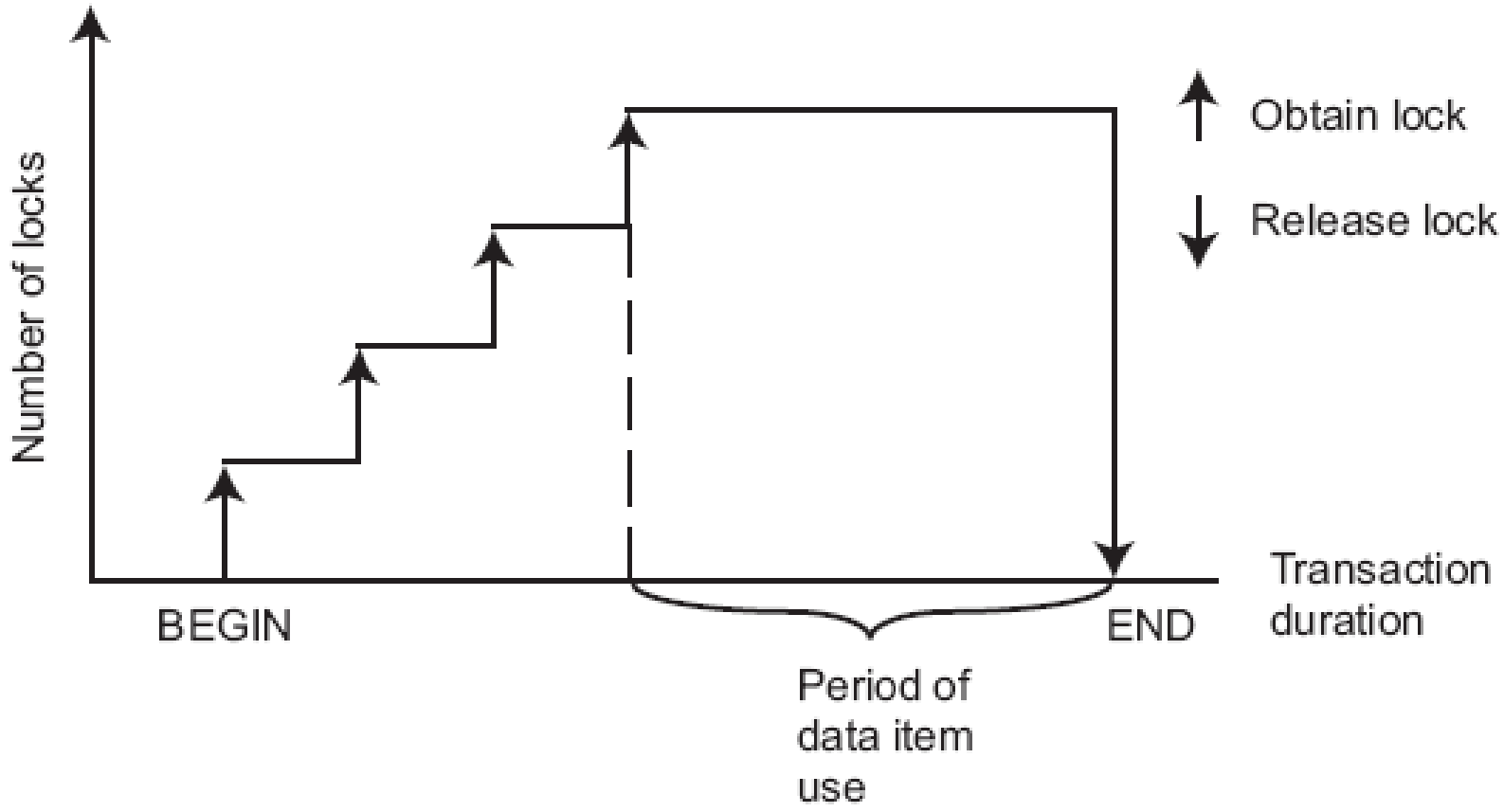
Cascading Rollbacks

- Consider three transactions:



- When transaction T_1 aborts, transactions T_2 and T_3 have already read data written by T_1 (dirty read).
- 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.
- **Strict two-phase locking avoids this problem.**

Strict Two-Phase Locking



- Facilitates lock management
- Avoids cascading aborts

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

- It is a safe operation in two-phase locking to convert a shared lock into an exclusive lock during the lock phase.

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:
    - Aborting **young** transactions may lead to **starvation**: the same transaction is cancelled again and again.
    - Aborting an **old** transaction may cause a lot of investment to be thrown away.

Deadlock Handling

- Other common techniques:
  - **Deadlock prevention:** e.g., by 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.
- Example: IBM DB2:

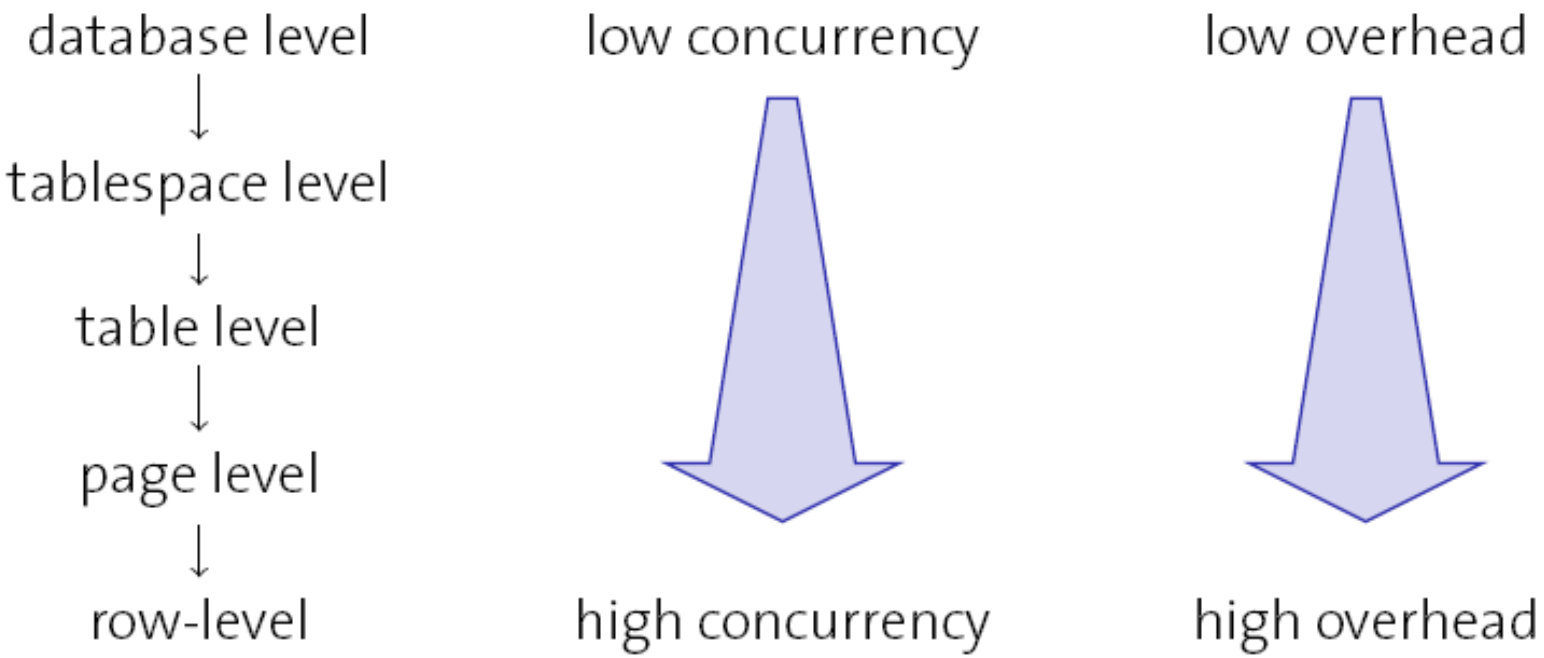
```
db2 => GET DATABASE CONFIGURATION;
   :
Interval for checking deadlock (ms)      (DLCHKTIME) = 10000
Lock timeout (sec)                      (LOCKTIMEOUT) = -1
```

Implementing a Lock Manager

- We'd like the lock manager to perform the following three tasks very efficiently:
  1. Check which locks are currently held for a given resource (in order to decide whether another lock request can be granted).
  2. When a lock is released, transactions that requested locks on the same resource have to be identified and granted the lock.
  3. When a transaction terminates, all held locks must be released.

Granularity of Locking

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



➢ Idea: **multi-granularity** locking

Multi-Granularity Locking

- Decide on the granularity of locks held **for each transaction** (depending on the characteristics of the transaction).
  - A row lock, e.g., for

```
SELECT * FROM CUSTOMERS                    Q1
 WHERE C_CUSTKEY = 42
```

  - and a table lock for

```
SELECT * FROM CUSTOMERS                    Q2
```

- How do such transactions know about each others' locks?
  - Note that locking is performance-critical. Q_2 doesn't want to do an extensive search for row-level conflicts.
  - Idea: Exploit the hierarchical nature of granularities.

Intention Locks

- Databases use an additional type of locks: intention locks
  - Lock mode intention share: IS
  - Lock mode intention exclusive: IX
  - Conflict matrix:

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

- A lock I* on a coarser level means that there's some lock * on a lower level.

Intention Locks

- Protocol for multi-granularity locking:
  1. A transaction can lock any granule *g* in * ϵ {S, X} mode.
  2. Before a granule *g* can be locked in * mode, it has to obtain an I* lock on all coarser granularities that contain *g*.
- Query Q_1 would, e.g.,
  - obtain an IS lock on table CUSTOMERS (also on tablespace and database levels), and
  - obtain an S lock on the row 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 a write query comes in:

```
UPDATE CUSTOMERS                                    Q3
  SET NAME = 'John Doe'
  WHERE C_CUSTKEY = 17
```

- It'll want to place
  - an IX lock on table CUSTOMER (and tablespace and database), and
  - an X lock on the row holding customer 17.
- As such it is
  - compatible with Q_1 (there's no conflict between IX and IS on the table level),
  - but incompatible with Q_2 (the S lock held by Q_2 is in conflict with Q_3's IX lock).

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.
  - SQL 92 specifies different isolation levels.
  - E.g.,

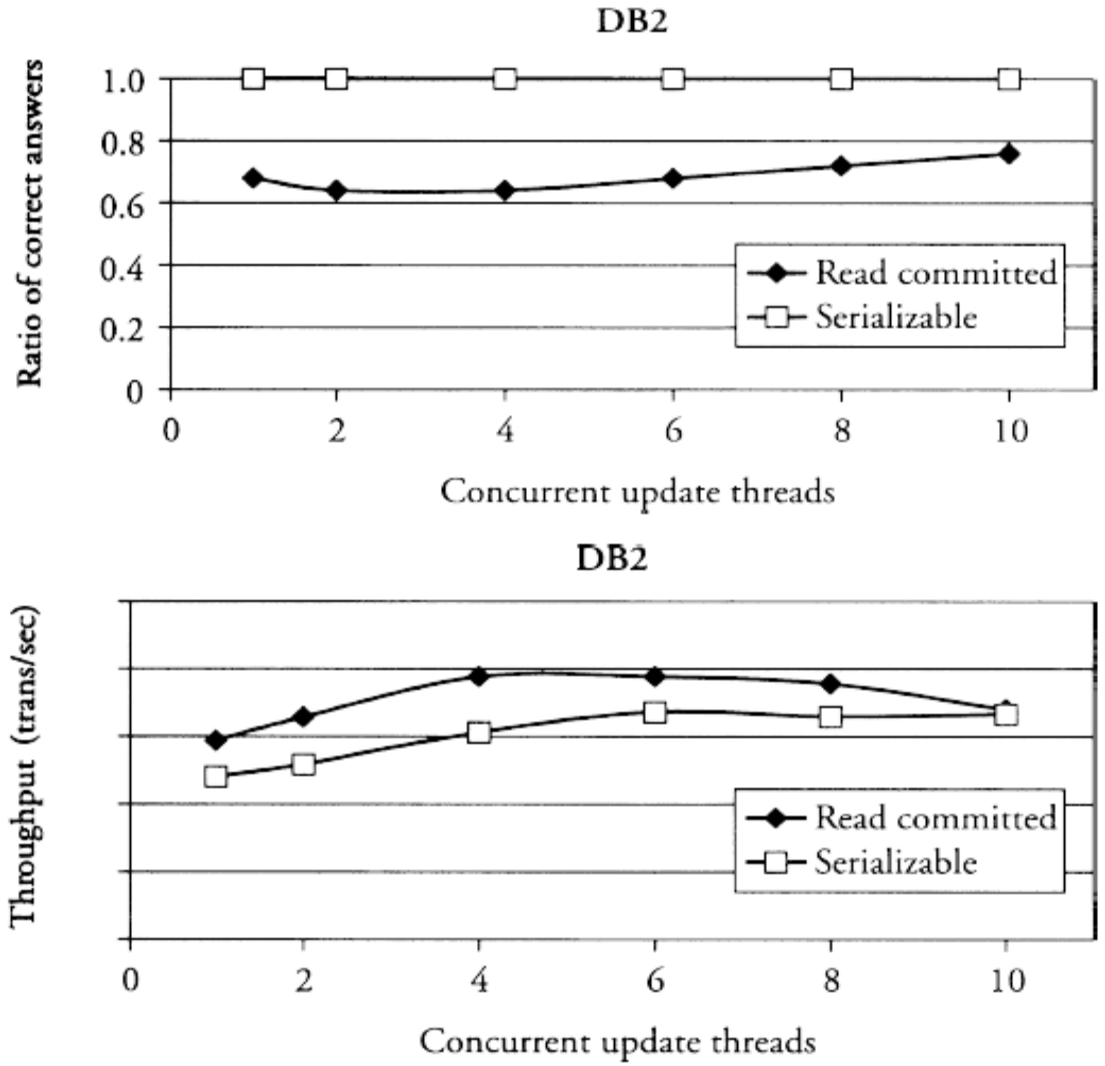
```
SET ISOLATION SERIALIZABLE;
```

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

SQL 92 Isolation Levels

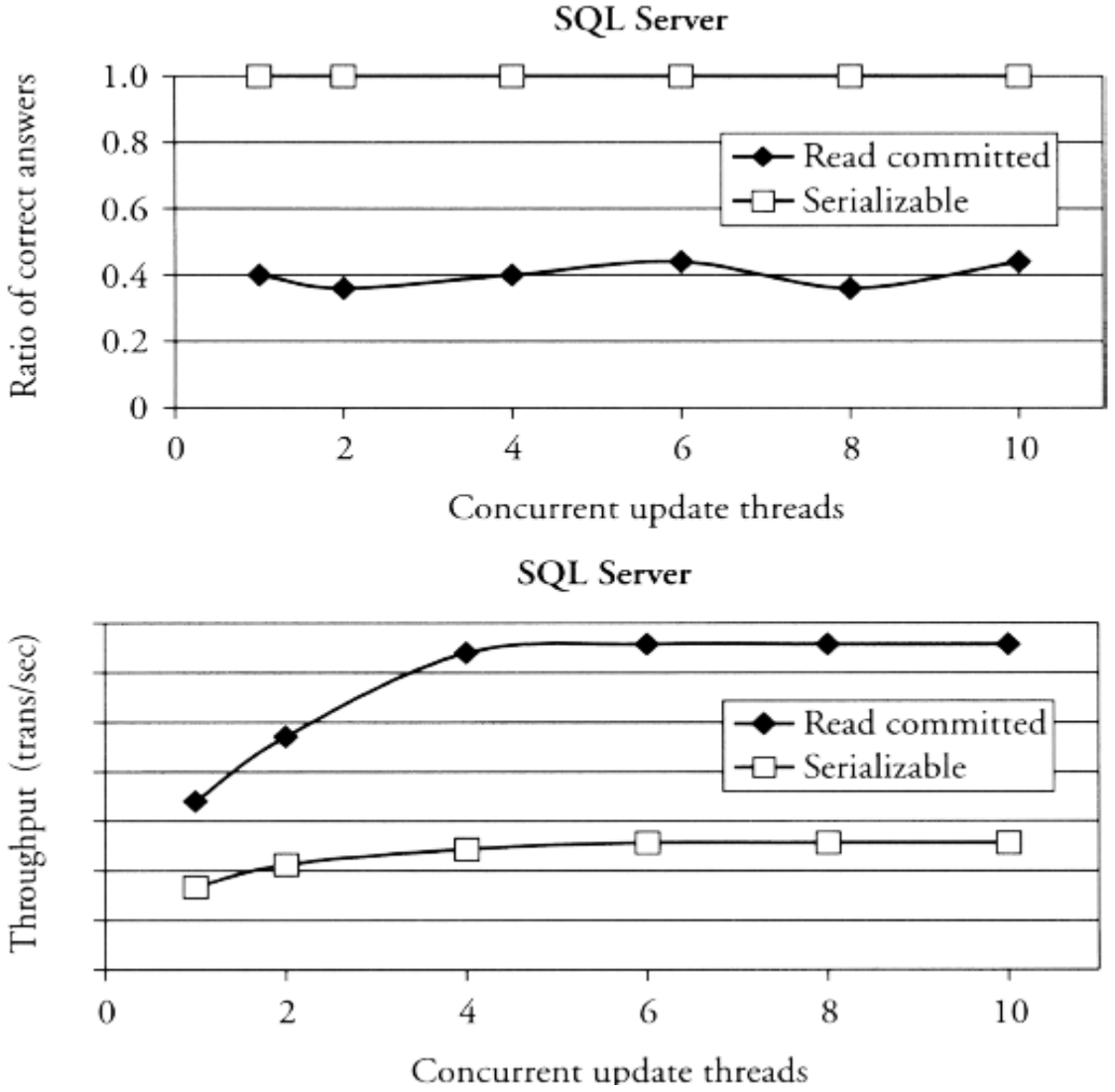
- **read uncommitted (also: ‘dirty read’ or ‘browse’)**
  - Only write locks are acquired according to strict 2PL.
- **read committed (also: ‘cursor stability’)**
  - Read locks are only held for as long as a cursor sits on the particular row. Write locks acquired according to strict 2PL.
- **repeatable read (also: ‘read stability’)**
  - Acquires read and write locks according to strict 2PL.
- **serializable**
  - Additionally, obtains locks to avoid **phantom reads**.

Performance Comparison



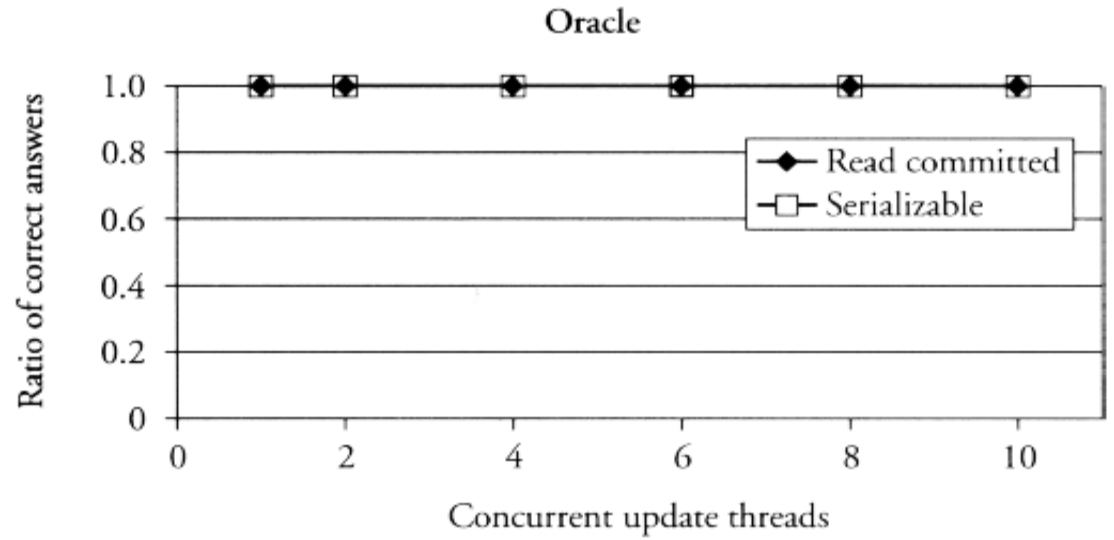
Dennis Shasha nad Philippe Bonnet. Database Tuning. Morgan Kaufmann, 2003.

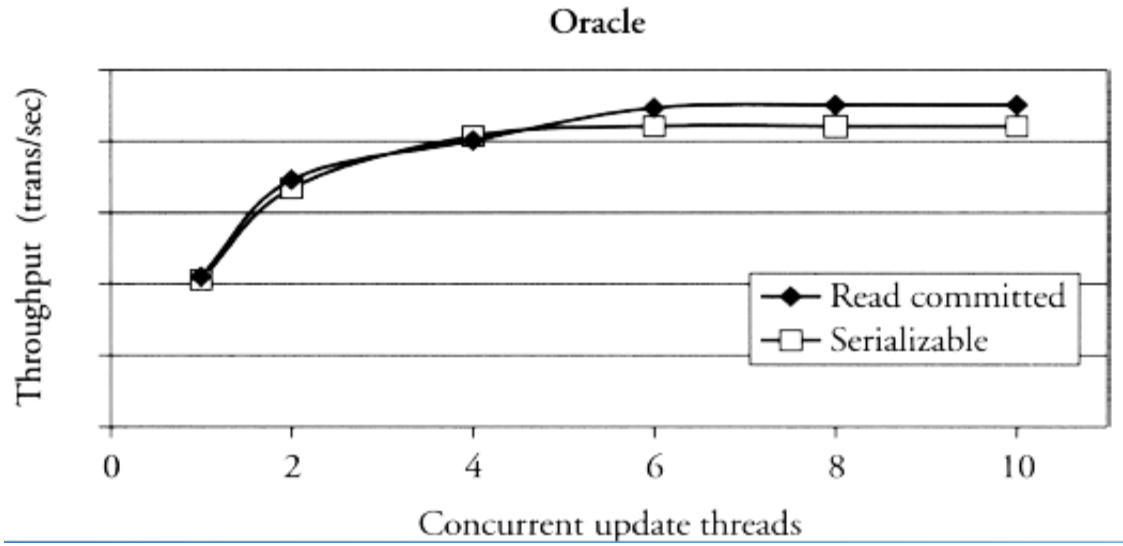
Performance Comparison



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

Performance Comparison





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.

Phantom Problem

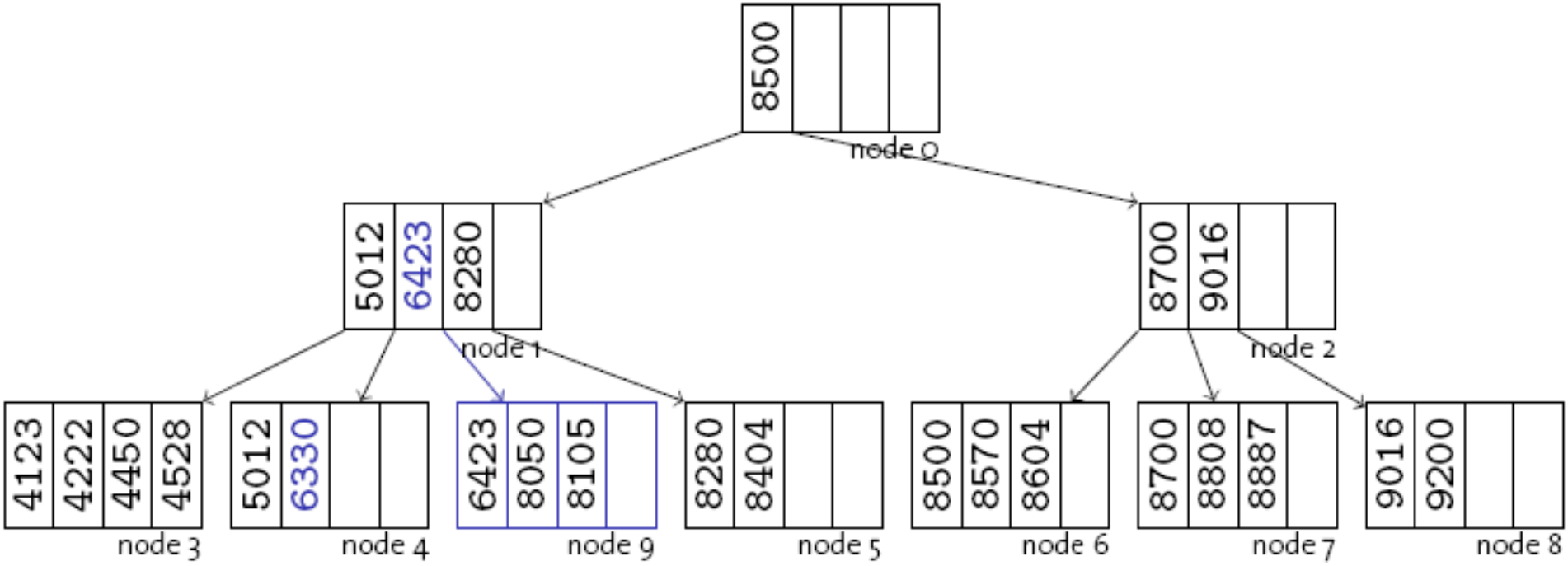
Transaction 1	Transaction 2	Effect
scan relation R ;		T_1 locks all rows
	insert new row into R ;	T_2 locks new row
	commit ;	T_2's lock released
scan relation R ;		reads **new** row, too!

- 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

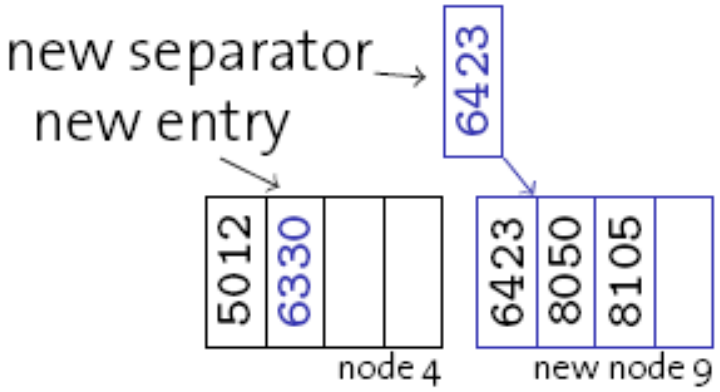
Concurrency in B-Tree Indices

- Consider an insert transaction T_w into a B$^+$-tree that resulted in a leaf node split.
  - Assume node 4 has just been split, but the new separator has not yet been inserted into node 1.
  - Now a concurrent read transaction T_r tries to find 8050.
  - The (old) node 1 guides T_r to node 4.
  - Node 4 no longer contains entry 8050, T_r believes there is no data item with zip code 8050.
  - This calls for concurrency control in B-trees.

Insertion to a B$^+$-tree: Example



- Insert key 6330.
  - Must **split** node 4.
  - **New separator** goes into node 1 (including pointer to new page).



Locking and B-Tree Indices

- Remember how we performed operations on B$^+$-trees:
  - To search a B$^+$-tree, we descended the tree top-down. Based on the content of a node *n*, we decided in which child *n'* to continue the search.
  - To update a B$^+$-tree, we
    - first did a search,
    - then inserted new data into the right leaf.
    - depending on the fullness levels of nodes, we had to split tree nodes and propagate splits bottom-up.
- According to the two-phase locking protocol, we'd have to
  - obtain $\mathbb{S}/\mathbb{X}$ locks when we walk down the tree and
  - keep all locks until we're finished.

Locking and B-Tree Indices

- This strategy would seriously reduce **concurrency**.
- All transactions would have to lock the tree root, which becomes a locking **bottleneck**.
- Root node locks, effectively, **serialize** all (write) transactions.
- Two-phase locking is **not practical** for B-trees.

Lock Coupling

- Let us consider the **write-only** case first (i.e., all locks conflict).
- The **write-only tree locking (WTL) protocol** is sufficient to guarantee serializability:
  1. For all tree nodes *n* other than the root, a transaction may only acquire a lock on *n* if it already holds a lock on *n*'s parent.
  2. Once a node *n* has been unlocked, the same *n* may not be locked again by the same transaction.
- Effectively,
  - All transactions have to follow a **top-down access pattern.**
  - No transaction can "bypass" any other transaction along the same path. Conflicting transactions are thus **serializable.**
  - The WTL protocol is **deadlock free.**

Lock Coupling with Split Safety

- We still have to keep as many write locks as nodes might be affected by node splits.
- It is easy to check for a node *n* whether an update might affect *n*'s ancestors:
  - if *n* contains less than 2*d* entries, no split will propagate above *n*.
- If *n* satisfies this condition, it is said to be **(split) safe**.
- We can use this definition to release write locks early:
  - if, while searching top-down for an insert location, we encounter a safe node *n*, we can release locks on all of *n*'s ancestors.
- Effectively, locks near the root are held for a shorter time.

The Optimistic Lock Coupling Variant

- Even with lock coupling there's a considerable amount of locks on inner tree nodes (reducing concurrency).
- Chances that inner nodes are actually affected by updates are very small.
  - Back-of-the-envelope calculation:

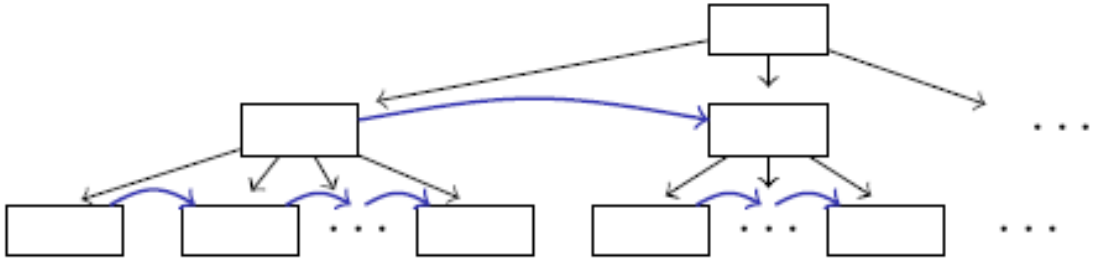
    d = 50 => every 50th insert causes a split (2% chance).
- An insert transaction could thus optimistically assume that no leaf split is going to happen.
  - On inner nodes, only read locks acquired during tree navigation (plus a write lock on the affected leaf).
  - If assumption is wrong, re-traverse the tree and obtain write locks.

Discussion

- If a leaf split happens, the writer bails out without having done any changes to the tree, then starts a whole new attempt from scratch.
- The resulting executions are **correct**, even though this looks like re-locking some nodes (which is disallowed by WTL).
- The **drawback** of the optimistic variant is that, in case of a leaf node split, it throws away all the work it invested in the first attempt.
- A number of other protocol variations try to improve on that.

B$^+$-Trees without Read Locks

- Lehman and Yao [TODS vol. 6(4), 1981] proposed a protocol that does not need any read locks on B-tree nodes.
- Requirement: a `next` pointer, pointing to the right sibling.



- Linked list along the leaf level.
- Pointers provide a second path to find each node.
- This way, mis-guided (by concurrent splits) read operations can still find the node that they need.

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
  - **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 committed before T started, or

$$RS(T) \cap WS(T_c) = \emptyset$$

- Forward-Oriented Optimistic Concurrency Control (FOCC):
  - Compare T against all running transactions T_r.
  - Check succeeds if

$$WS(T) \cap RS(T_r) = \emptyset$$

Multi-version Concurrency Control

- Consider the schedule

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

(t ↓ between $r_2(x)$ and $w_2(y)$)

- Now suppose that when T_1 wants to read y, we still had 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.

Multi-version 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 (garbage collection).

Multi-version Concurrency Control

- Maintain multiple versions of each database object o, each with a write timestamp (WTS(o)) and a read timestamp (RTS(o)).
- Transaction T_i can read the most recent version whose timestamp precedes $TS(T_i)$.
- If T_i wants to write an object o, we must ensure that o has not been read by another transaction T_j such that $TS(T_i) < TS(T_j)$. Otherwise, T_i's change should have been seen by T_j for serializability, but it's too late for that. To handle this:
  - Whenever a transaction T_j reads an object o, the RTS(o) is set to max(current RTS(o), $TS(T_j)$).
  - If T_i wants to write an object o and $TS(T_i) < RTS(o)$, T_i is aborted and restarted with a new larger timestamp.
  - Otherwise, T_i creates a new version of o (say o'), and sets RTS(o') = $TS(T_i)$ and WTS(o') = $TS(T_i)$.

Summary

- **ACID and Serializability**
  - To prevent from different types of anomalies, DBMSs guarantee ACID properties. Serializability is a sufficient criterion to guarantee isolation.
- **Two-Phase Locking**
  - Two-phase locking is a practical technique to guarantee serializability. Most systems implement strict 2PL. SQL 92 allows explicit relaxation of the ACID isolation constraints in the interest of performance.
- **Other Concurrency Control Issues**
  - B-trees: Specialized protocols exist for concurrency control in B-trees (the root would be a locking bottleneck otherwise).
  - Optimistic concurrency control
  - Multi-version concurrency control