

Part VI

Transaction Management and Recovery

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.

```
1 bal ← read_bal (acct_no) ;  
2 bal ← bal - 100 CHF ;  
3 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
$bal \leftarrow \text{read}(acct);$		1200
	$bal \leftarrow \text{read}(acct);$	1200
$bal \leftarrow bal - 100;$		1200
	$bal \leftarrow bal - 200;$	1200
$\text{write}(acct, bal);$		1100
	$\text{write}(acct, bal);$	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 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) ;
```

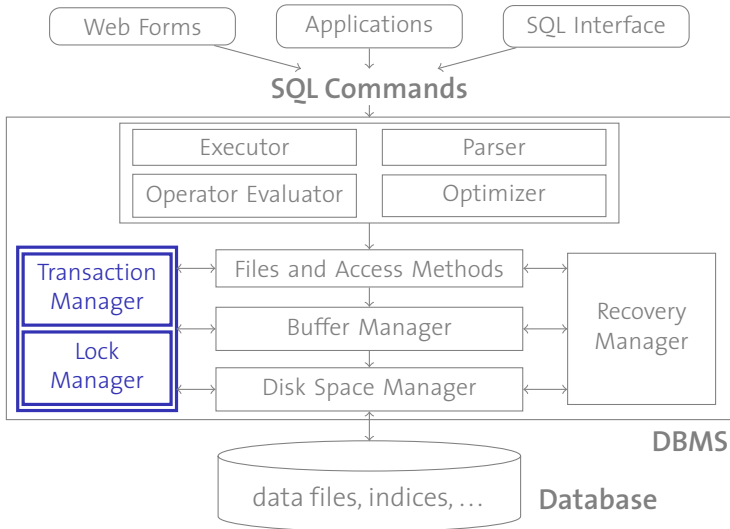
- ▶ Before the transaction gets to step **6**, its execution is **interrupted/cancelled** (power outage, disk failure, software bug, ...). My money is **lost** ☹️.

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.
- 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.

Concurrency Control



Anomalies: Lost Update

- ▶ We already saw a **lost update** example on slide 201.
- ▶ The effects of one transaction are lost, because an uncontrolled overwriting by the second transaction.

Anomalies: Inconsistent Read

Consider the money transfer example (slide 202), expressed in SQL syntax:

```
Transaction 1
UPDATE Accounts
  SET balance = balance - 500
  WHERE customer = 4711
     AND account_type = 'C';
```

```
UPDATE Accounts
  SET balance = balance + 500
  WHERE customer = 4711
     AND account_type = 'S';
```

```
Transaction 2

SELECT SUM(balance)
  FROM Accounts
 WHERE customer = 4711;
```

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

Anomalies: Dirty Read

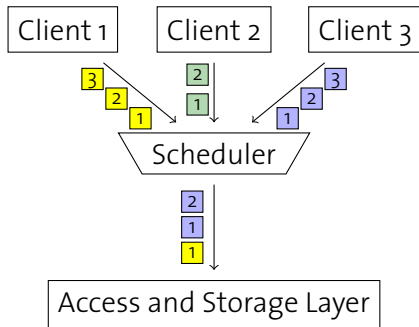
At a different day, my wife and me again end up in front of an ATM at roughly the same time:

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

object \equiv attribute .

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

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

We could write the money transfer transaction as

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



or, more concisely,

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

Schedules

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

$$S(k) = (T_j, a_i, e_i) \quad k = 1 \dots m ,$$



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

$$\begin{aligned} S(1) &= (T_1, \text{read}, B) & S(3) &= (T_1, \text{write}, B) \\ S(2) &= (T_2, \text{read}, B) & S(4) &= (T_2, \text{write}, B) \end{aligned}$$

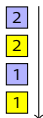
Serial Execution

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).

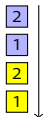
Consider again the ATM example from slide 201.

- ▶ $S = \langle r_1(B), r_2(B), w_1(B), w_2(B) \rangle$
- ▶ 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 201 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 211).
 - ▶ Rearranging operations must not lead to a different **result**.
- ▶ Two operations (a, e) and (a', e') are said to be **in conflict** $(a, e) \leftrightarrow (a', e')$ 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** \prec_S :

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

$$(a, e) \leftrightarrow (a', e') \wedge (T_i, a, e) \text{ occurs before } (T_j, a', e') \text{ in } S \wedge T_i \neq T_j$$

Conflict Serializability

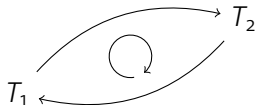
- ▶ A schedule S is **conflict serializable** iff it is conflict equivalent to **some** serial schedule S' .
- ▶ **The execution of a conflict-serializable S schedule is correct.**
 - ▶ S does **not** have to be a serial schedule.
- ▶ This allows us to **prove** the correctness of a schedule S based 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**.¹⁴

¹⁴A serial execution of S could be obtained by sorting $G(S)$ **topologically**.

Serialization Graph

Example: ATM transactions (↗ slide 201)

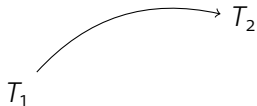
- ▶ $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$
- ▶ Conflict relation:
 - $(T_1, r, A) \prec_S (T_2, w, A)$
 - $(T_2, r, A) \prec_S (T_1, w, A)$
 - $(T_1, w, A) \prec_S (T_2, w, A)$



→ **not** serializable

Example: Two money transfers (↗ slide 202)

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



→ serializable

Query Scheduling

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

Idea:

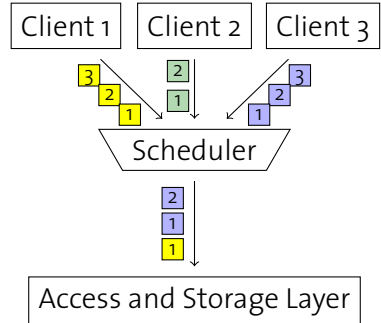
- ▶ Require each transaction to obtain a **lock** before it accesses a data object o :

```

1 lock o ;
2 ...ACCESS o ...;
3 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?

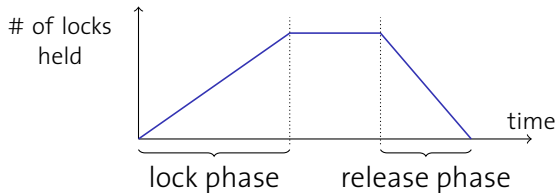
ATM Transaction with Locking

Transaction 1	Transaction 2	DB state
lock (<i>acct</i>) ; read (<i>acct</i>) ; unlock (<i>acct</i>) ;		1200
lock (<i>acct</i>) ; write (<i>acct</i>) ; unlock (<i>acct</i>) ;	lock (<i>acct</i>) ; read (<i>acct</i>) ; unlock (<i>acct</i>) ;	1100
	lock (<i>acct</i>) ; write (<i>acct</i>) ; unlock (<i>acct</i>) ;	1000

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 (<i>acct</i>) ; read (<i>acct</i>) ; unlock (<i>acct</i>) ;		1200
lock (<i>acct</i>) ; ⚡ write (<i>acct</i>) ; unlock (<i>acct</i>) ;	lock (<i>acct</i>) ; read (<i>acct</i>) ; unlock (<i>acct</i>) ;	1100
	lock (<i>acct</i>) ; ⚡ write (<i>acct</i>) ; unlock (<i>acct</i>) ;	1000

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.

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

} lock phase

} unlock phase

Resulting Schedule

Transaction 1	Transaction 2	DB state
lock (<i>acct</i>) ;		1200
read (<i>acct</i>) ;		
write (<i>acct</i>) ;	lock (<i>acct</i>) ;	
unlock (<i>acct</i>) ;	↓ Transaction ↓ blocked	1100
	read (<i>acct</i>) ;	
	write (<i>acct</i>) ;	900
	unlock (<i>acct</i>) ;	

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

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

lock (A) ;

⋮

do something

⋮

lock (B)

[wait for T_2 to release lock]

Transaction 2

lock (B)

⋮

do something

⋮

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:

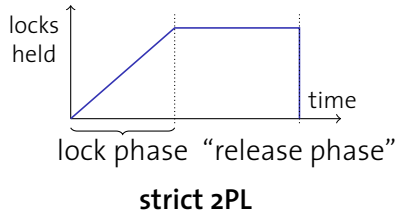
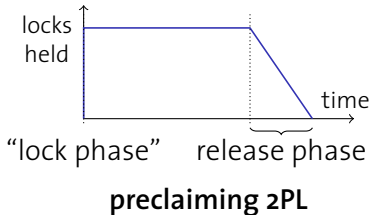
- ▶ **Deadlock prevention:** *e.g.*, by treating handling lock requests in an **asymmetric** way:
 - ▶ **wait-die:** A transaction is never blocked by an **older** transaction.
 - ▶ **wound-wait:** A transaction is never blocked by a **younger** transaction.
- ▶ **Timeout:** Only wait for a lock until 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 be acquired and released.
- ▶ Possible variants:



- ▶  What could motivate either variant?

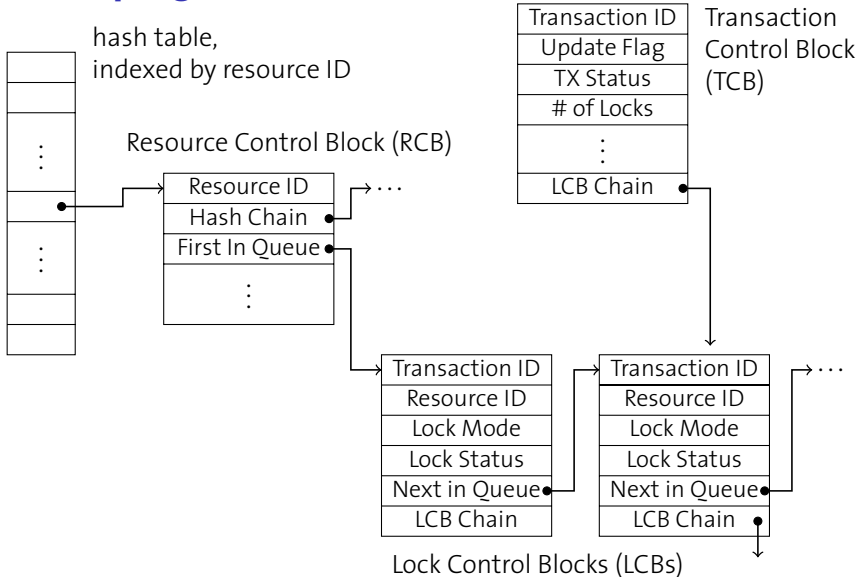
Implementing a Lock Manager

We'd like the Lock Manager to do 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.

What is a good data structure to accommodate these needs?

Bookkeeping

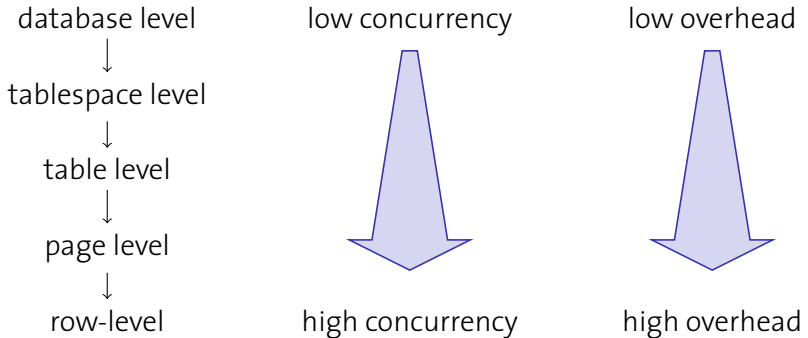


Implementing Lock Manager Tasks

1. The locks held for a given **resource** can be found using a **hash lookup**.
 - ▶ Linked list of Lock Control Blocks via ‘First In Queue’/‘Next in Queue’
 - ▶ The list contains **all** lock requests, granted or not.
 - ▶ The transaction(s) at the **head** of the list are the ones that currently hold a lock on the resource.
2. When a lock is **released** (*i.e.*, its LCB removed from the list), the next transaction(s) in the list are considered for granting the lock.
3. All locks held by a single **transaction** can be identified via the linked list ‘LCB Chain’ (and easily released upon transaction termination).

Granularity of Locking

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



► **Idea:** multi-granularity locking

Multi-Granularity Locking

- ▶ Decide 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.

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 $\square \in \{S, X\}$ mode.
2. Before a granule g can be locked in \square mode, it has to obtain an $I\square$ lock on **all** coarser granularities than contain g .

Query Q_1 would, *e.g.*,

- ▶ obtain an IS lock on **table CUSTOMERS**
(also on on tablespace and database) and
- ▶ obtain an S lock on the **tuple(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 a write query comes in:

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

Q_3

It'll want to place

- ▶ an IX lock on **table** CUSTOMER (and ...) 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.
 - ↷ 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')

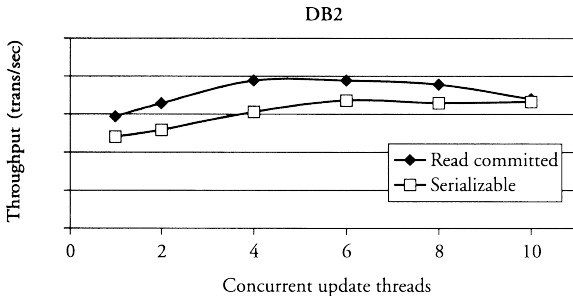
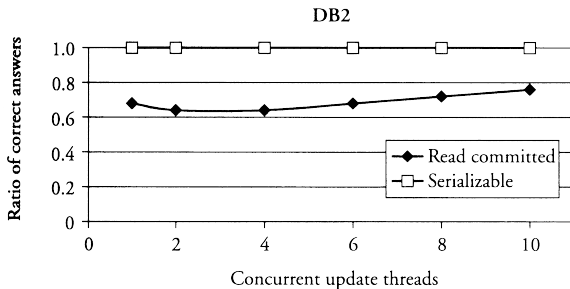
Read locks are only held for as long as a cursor sits on the particular row. **Write locks** acquired according to strict 2PL.

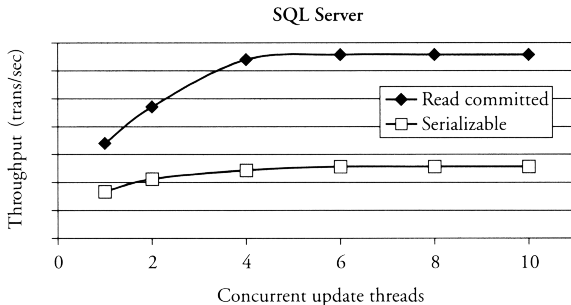
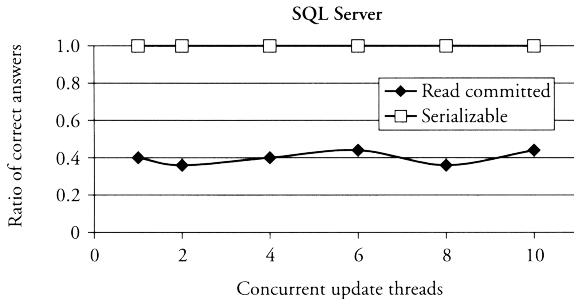
repeatable read (also: 'read stability')

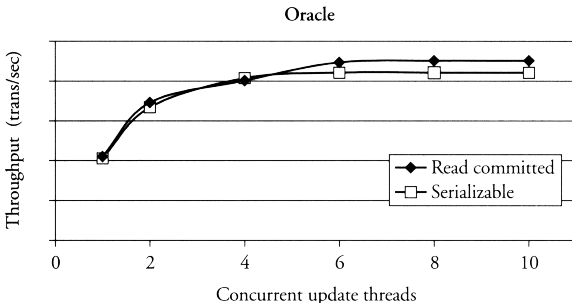
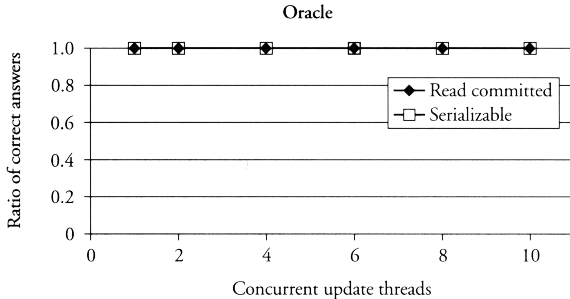
Acquires **read** and **write locks** according to strict 2PL.

serializable

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







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.

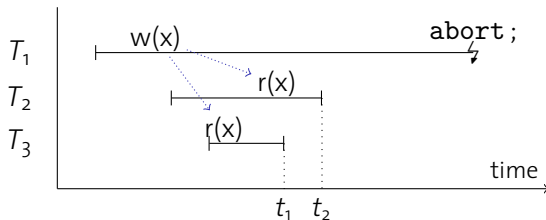
Phantom Problem

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

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 207)
- ▶ 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.
- ▶ two-phase locking vs. strict two-phase locking

Concurrency in B-tree Indices

Consider an **insert** transaction T_w into a B⁺-tree that resulted in a leaf node split, as on slide 58.

- ▶ 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**.

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 son 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 **fill 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 S/X locks when we walk down the tree¹⁵ and
- ▶ **keep** all locks until we're finished.

¹⁵Note that **lock conversion** is not a good idea. It would increase the likeliness of **deadlocks** (read locks acquired top-down, write locks bottom-up).

Locking and B-tree Indices

- ▶ This strategy would seriously **reduce concurrency**.
- ▶ **All** transactions will 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 (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**.

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 $2d$ 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**.

Lock Coupling Protocol (Variant 1)

1 place S lock on *root* ; readers
 2 *current* \leftarrow *root* ;
 3 **while** *current* is not a leaf node **do**
 4 place S lock on appropriate son of *current* ;
 5 release S lock on *current* ;
 6 *current* \leftarrow son of *current* ;

1 place X lock on *root* ; writers
 2 *current* \leftarrow *root* ;
 3 **while** *current* is not a leaf node **do**
 4 place X lock on appropriate son of *current* ;
 5 *current* \leftarrow son of *current* ;
 6 **if** *current* is safe **then**
 7 release all locks held on ancestors of *current* ;

Increasing Concurrency for Common Scenarios

- ▶ 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:
 $d = 50 \Rightarrow$ 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.

Lock Coupling Protocol (Variant 2)

Modified protocol for **writers**:¹⁶

```

1 place S lock on root ;
2 current ← root ;
3 while current is not a leaf node do
4   | son ← appropriate son of current ;
5   | if son is a leaf then
6   |   | place X lock on son ;
7   |   | else
8   |   |   | place S lock on son ;
9   |   |   | release lock on current ;
10  |   |   | current ← son ;
11  | if current is unsafe then
12  |   | release all locks and repeat with protocol Variant 1 ;

```

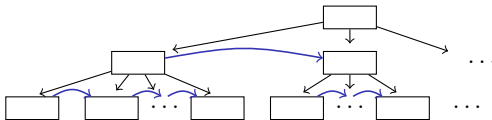
¹⁶Reader protocol remains unchanged.

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 Variant 2 is that, in case of a leaf node split, it throws away all the work it invested in the first attempt.
- ▶ A number of protocol variations try to improve on that.
 - exercises

B⁺-trees Without Read Locks

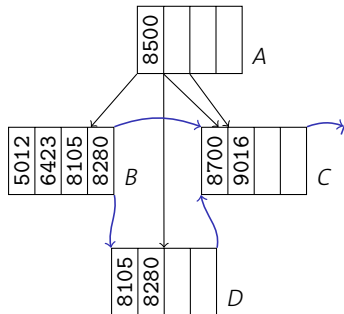
- ▶ Lehman and Yao (*TODS* vol. 6(4), 1981) proposed a protocol that does **not** need any read locks on B-tree nodes.
- ▶ Requirement: a **next** pointer, pointing to the **right sibling**.



- ▶ Chapter II: 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.


Insertion With Inner Node Split

- 1 lock & read page B ;
- 2 create new page D and lock it ;
- 3 populate page D ;
- 4 set next pointer $D \rightarrow C$;
- 5 write D ;
- 6 set next pointer $B \rightarrow D$;
- 7 adjust content of B ;
- 8 write B ;
- 9 lock & read A ;
- 10 adjust content of A ;
- 11 write A ;



- ▶ All index entries remained reachable at all times.

B-tree Access By Readers

- ▶ The `next` pointers give readers the chance to “find” entries even in the middle of a split, when some entries have already moved to the new page.
 - ▶ During `tree_search()`, reading transactions move from n to the `next` sibling as long as
 - (a) No appropriate entry can be found in n and
 - (b) `next` is not a **null** pointer.
 - ▶ Note that `next` pointers only point to siblings that have the same parent.
 - ▶ Write locks do **not** prevent readers from accessing a page! (This is because readers do **not** acquire read locks.)
-  **PostgreSQL**, *e.g.*, uses this protocol for B⁺-tree access.

Locks and Latches



Which assumption is necessary for lock-free read access?

- ▶ We do **not** want to use locks to guarantee atomicity (which would, after all, defeat our whole idea).
- ▶ Systems typically provide special **light-weight locks**, so-called **latches**, to allow for short-term atomic operations.
- ▶ Latches incur only **little overhead** (they are sometimes even implemented as **spinlocks**).
- ▶ However, they are **not** under control of the **lock manager**. Hence, they are not covered by **deadlock detection** or, depending on the system, automatic unlocking in case of a transaction rollback.

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

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

forward-oriented optimistic concurrency control (FOCC):

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

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

Multiversion Concurrency Control

Consider the schedule

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

t
↓



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

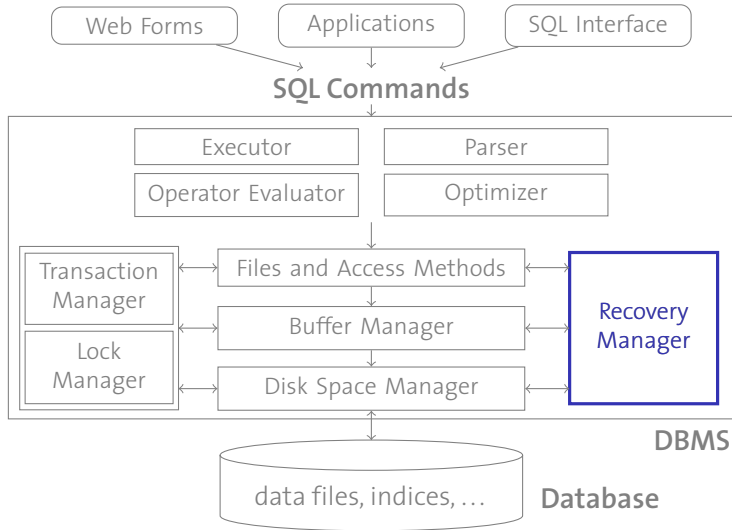
$r_1(x), w_1(x), r_2(x), r_1(y), w_2(y), w_1(z)$,

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** (↪ garbage collection).
- ▶ Some systems support **snapshot isolation**.
 - 🏰 Oracle, SQL Server, PostgreSQL

Recovery



Failure Recovery

We want to deal with **three types of failures**:

transaction failure (also: 'process failure')

A transaction voluntarily or involuntarily **aborts**. All of its updates need to be **undone**.

system failure

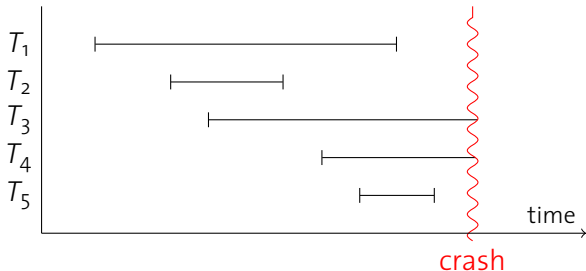
Database or operating **system crash**, power outage, etc. All information in main memory is lost. Must make sure that **no committed transaction is lost** (or **redo** their effects) and that all other transactions are **undone**.

media failure (also: 'device failure')

Hard disk crash, catastrophic error (fire, water, ...). Must **recover database** from stable storage.

In spite of these failures, we want to guarantee **atomicity** and **durability**.

Example: System Crash (or Media Failure)



- ▶ Transactions T_1 , T_2 , and T_5 were committed before the crash.
 - **Durability:** Ensure that updates are **preserved** (or **redone**).
- ▶ Transactions T_3 and T_4 were not (yet) committed.
 - **Atomicity:** All of their effects need to be **undone**.

Types of Storage

We assume three different types of storage:

volatile storage

This is essentially the **buffer manager**. We are going to use volatile storage to **cache** the **write-ahead log** in a moment.

non-volatile storage

Typical candidate is a **hard disk**.

stable storage

Non-volatile storage that survives all types of failures. Stability can be improved using, *e.g.*, (network) **replication** of disk data. Backup tapes are another example.

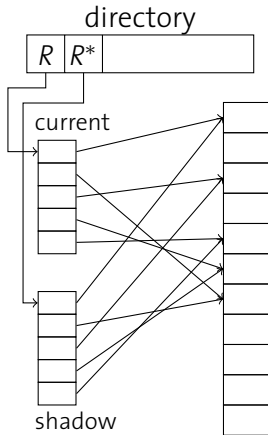
Observe how these storage types correspond to the three types of failures.

Shadow Pages

- ▶ Since a failure could occur **at any time**, it must be made sure that the system can **always** get back to a consistent state.
- ▶ Need to keep information **redundant**.
- ▶ System R: **shadow pages**. Two versions of every data page:
 - ▶ The **current version** is the system's “working copy” of the data and may be inconsistent.
 - ▶ The **shadow version** is a consistent version on stable storage.
- ▶ Use operation **SAVE** to save the current version as the shadow version.
 - ▶ **SAVE** ↔ **commit**
- ▶ Use operation **RESTORE** to recover to shadow version.
 - ▶ **RESTORE** ↔ **abort**

Shadow Pages

1. Initially: shadow \equiv current.
2. A transaction T now changes the **current** version.
 - ▶ Updates are **not** done in-place.
 - ▶ Create new pages and alter current page table.
- 3a. If T **aborts**, overwrite current version with shadow version.
- 3b. If T **commits**, change information in **directory** to make current version persistent.
4. Reclaim disk pages using **garbage collection**.



Shadow Pages: Discussion

- ▶ Recovery is instant and fast for **entire files**.
- ▶ To guarantee **durability**, all modified pages must be **forced** to disk when a transaction **commits**.
- ▶ As we discussed on slide 31, this has some undesirable effects:
 - ▶ high I/O cost, since writes cannot be cached,
 - ▶ high response times.
- ▶ We'd much more like to use a **no-force** policy, where write operations can be deferred to a later time.
- ▶ To allow for a no-force policy, we'd have to have a way to **redo** transactions that are committed, but haven't been written back to disk, yet.

↗ Gray *et al.*. The Recovery Manager of the System R Database Manager. *ACM Comp. Surv.*, vol. 13(2), June 1981.

Shadow Pages: Discussion

- ▶ Shadow pages do allow **frame stealing**: buffer frames **may** be written back to disk (to the “current version”) **before** the transaction T commits.
- ▶ Such a situation occurs, *e.g.*, if another transaction T' wants to use the space to bring in its data.
 - ▶ T' “**steals**” a frame from T .
 - ▶ Obviously, a frame may only be stolen if it is **not pinned**.
- ▶ Frame stealing means that **dirty** pages are written back to disk. Such writes have to be **undone** during recovery.
 - ▶ Fortunately, this is easy with shadow pages.

Effects on Recovery

- ▶ The decisions **force**/**no force** and **steal**/**no steal** have implications on what we have to do during recovery:

	force	no force
no steal	no redo no undo	must redo no undo
steal	no redo must undo	must redo must undo

- ▶ If we want to use **steal** and **no force** (to increase concurrency and performance), we have to implement **redo** and **undo** routines.

Write-Ahead Log

- ▶ The **ARIES**¹⁷ recovery method uses a **write-ahead log** to implement the necessary redundancy. Data pages are updated **in place**.

↗ Mohan *et al.* ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, vol. 17(1), March 1992.

- ▶ To prepare for **undo**, undo information must be written to stable storage **before** a page update is written back to disk.
- ▶ To ensure **durability**, **redo** information must be written to stable storage **at commit time** (no-force policy: the on-disk data page may still contain old information).


¹⁷Algorithm for Recovery and Isolation Exploiting Semantics

Content of the Write-Ahead Log

LSN	Type	TX	Prev	Page	UNxt	Redo	Undo
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

LSN (Log Sequence Number)

Monotonically increasing number to identify each log record.

Trick: Use byte position of log record  **Why?**

Type (Log Record Type)

Indicates whether this is an **update record** (UPD), **end of transaction record** (EOT), **compensation log record** (CLR), ...

TX (Transaction ID)

Transaction identifier (if applicable).

Content of the Write-Ahead Log (cont.)

Prev (Previous Log Sequence Number)

LSN of the preceding log record written by the same transaction (if applicable). Holds ‘-’ for the **first** record of every transaction.

Page (Page Identifier)

Page to which updates were applied (only for UPD and CLR).

UNxt (LSN Next to be Undone)

Only for CLR. Next log record of this transaction that has to be processed during **rollback**.

Redo

Information to **redo** the operation described by this record.

Undo

Information to **undo** the operation described by this record.
Empty for CLR.

Example

Transaction 1	Transaction 2	LSN	Type	TX	Prev	Page	UNxt	Redo	Undo
<code>a ← read(A);</code>	<code>c ← read(C);</code>								
<code>a ← a - 50;</code>	<code>c ← c + 10;</code>								
<code>write(a,A);</code>	<code>write(c,C);</code>	1	UPD	T ₁	-	...		A := A - 50	A := A + 50
<code>b ← read(B);</code>		2	UPD	T ₂	-	...		C := C + 10	C := C - 10
<code>b ← b + 50;</code>									
<code>write(b,B);</code>		3	UPD	T ₁	1	...		B := B + 50	B := B - 50
<code>commit;</code>		4	EOT	T ₁	3	...			
<code>a ← read(A);</code>	<code>a ← a - 10;</code>								
<code>write(a,A);</code>	<code>commit;</code>	5	UPD	T ₂	2	...		A := A - 10	A := A + 10
		6	EOT	T ₂	5	...			

Redo/Undo Information

ARIES assumes **page-oriented redo**:

- ▶ No other pages need to be examined to **redo** a log entry.
- ▶ *E.g.*, **physical logging**:
 - ▶ store byte images for (parts of) a page
 - ▶ **before image**: byte image before update was performed
 - ▶ **after image**: byte image after update was performed
- ▶ Makes recovery **independent amongst objects**
 - ▶ Recovery system does not need to know what type of page it deals with: data pages, index pages, ...
- ▶ Contrast to **logical redo** (“set tuple with *rid* to value *v*”):
 - ▶ Redo requires accessing and changing indices, may cause tuple relocation to another page, etc.

Redo/Undo Information

ARIES **does** support **logical undo**:

- ▶ Page-oriented undo can cause **cascading rollback** situations.
 - ▶ Even if a transaction T_1 does not directly inspect tuples written by another transaction T_2 , it may still see T_2 's effects on, *e.g.*, page layout.
 - ▶ T_1 could then not commit before T_2 commits.
- ▶ Logical undo, therefore, **increases concurrency**.
- ▶ By using **operation logging**, transactions can even run fully concurrent if their actions are **semantically compatible**.
 - ▶ *E.g.*, increment and decrement operations (The WAL example on slide 275 used operational logging.)
 - ▶ This is particularly interesting in indices and meta data.

Writing Log Records

- ▶ For performance reasons, all log records are first written to **volatile storage**.
- ▶ At certain times, the log is **forced to stable storage** up to a certain LSN:
 - ▶ All records until T 's EOT record are forced to disk when T **commits** (to prepare for a **redo** of T 's effects).
 - ▶ When a **data page p is written** back to disk, log records up to the last modification of p are forced to disk (such that uncommitted updates on p can be **undone**).
- ▶ The log is an ever-growing file (but see later).

Normal Processing

During normal transaction processing, keep two pieces of information in each Transaction Control Block (slide 232):

LastLSN (Last Log Sequence Number)

LSN of the last log record written for this transaction.

UNxt (LSN Next to be Undone)

LSN of the next log record to be processed during **rollback**.

Whenever an update to a page p is performed,

- ▶ a **log record** r is written to the WAL and
- ▶ The **LSN** of r is recorded in the **page header** of p .

Transaction Rollback

To **roll back** a transaction T after a **transaction failure**:

- ▶ Process the log in a **backward** fashion.
- ▶ Start the **undo** operation at the log entry pointed to by the UNxt field in the transaction control block of T .
- ▶ Find the remaining log entries for T by following the Prev and UNxt fields in the log.



Undo operations modify pages, too!

- **Log** all undo operations to the WAL.
- Use **compensation log records (CLRs)** for this purpose.

Transaction Rollback

```
1 Function: rollback (SaveLSN, T)
2 UndoNxt  $\leftarrow$  T.UNxt ;
3 while SaveLSN < UndoNxt do
4   LogRec  $\leftarrow$  read log entry with LSN UndoNxt ;
5   switch LogRec.Type do
6     case UPD
7       perform undo operation LogRec.Undo on page LogRec.Page ;
8       LSN  $\leftarrow$  write log entry
9          $\langle$ CLR, T, T.LastLSN, LogRec.Page, LogRec.Prev,  $\dots$ ,  $\emptyset$  $\rangle$  ;
10      set LSN = LSN in page header of LogRec.Page ;
11      T.LastLSN  $\leftarrow$  LSN ;
12     case CLR
13       UndoNxt  $\leftarrow$  LogRec.UNxt ;
14   T.UNxt  $\leftarrow$  UndoNxt ;
```

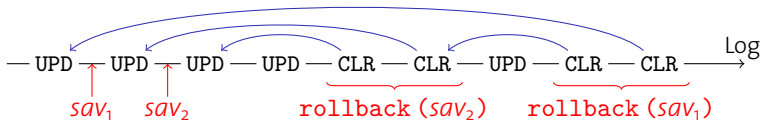
Transaction Rollback

- ▶ Transaction can be rolled back **partially** (back to *SaveLSN*).



Why is this useful?

- ▶ The UNxt field in a CLR points to the log entry before the one that has been undone.



Crash Recovery

Restart after a **system failure** is performed in **three phases**:

1. Analysis Phase:

- ▶ Read log in **forward** direction.
- ▶ Determine all transactions that were **active** when the failure happened. Such transactions are called **losers**.

2. Redo Phase:

- ▶ **Replay** the log (in **forward** direction) to bring the system into the state as of the time of system failure.

3. Undo Phase:

- ▶ **Roll back** all loser transactions, reading the log in a **backward** fashion (similar to “normal” rollback).

Analysis Phase

```
1 Function: analyze ()
2 foreach log entry record LogRec do
3   switch LocRec.Type do
4     create transaction control block for LogRec.TX if necessary ;
5     case UPD or CLR
6       LogRec.TX.LastLSN ← LogRec.LSN ;
7       if LocRec.Type = UPD then
8         | LogRec.TX.UNxt ← LogRec.LSN ;
9       else
10        | LogRec.TX.UNxt ← LogRec.UNxt ;
11     case EOT
12     | delete transaction control block for LogRec.TX ;
```

Redo Phase

```
1 Function: redo ()
2 foreach log entry record LogRec do
3   switch LogRec.Type do
4     case UPD or CLR
5        $v \leftarrow \text{pin}(\text{LogRec.Page}) ;$ 
6       if  $v.\text{LSN} < \text{LogRec.LSN}$  then
7         perform redo operation LogRec.Redo on  $v ;$ 
8          $v.\text{LSN} \leftarrow \text{LogRec.LSN} ;$ 
9       unpin ( $v, \dots$ ) ;
```



System crashes can occur **during** recovery!

- ▶ Undo and redo of a transaction T must be **idempotent**:

$$\text{undo}(\text{undo}(T)) = \text{undo}(T)$$

$$\text{redo}(\text{redo}(T)) = \text{redo}(T)$$

- ▶ Check LSN before performing the redo operation (line 6).

Redo Phase

- ▶ Note that we redo **all** operations (even those of losers) and in **chronological order**.
- ▶ After the redo phase, the system is in the **same state as it was at the time of the system failure**.

Some **log entries** may not have found their way to the disk before the failure. Committed operations would have been written to disk, though (slide 280). All others would have to be undone anyway.

- ▶ We'll have to **undo** all effects of **loser transactions** afterwards.
- ▶ As an optimization, the analyze pass could instruct the buffer manager to **prefetch** dirty pages.

Undo Phase

- ▶ The **undo phase** is similar to the rollback during “normal processing”.
- ▶ This time we roll back **several transactions** (all losers) at once.
- ▶ All loser transactions are rolled back completely (not just up to some savepoint).

```
1 Function: undo ()
2 while transactions (i.e., TCBs) left to roll back do
3    $T \leftarrow$  TCB of loser transaction with greatest UNxt ;
4    $LogRec \leftarrow$  read log entry with LSN  $T.UNxt$  ;
5   switch  $LogRec.Type$  do
6     case UPD
7       perform undo operation  $LogRec.Undo$  on page  $LogRec.Page$  ;
8        $LSN \leftarrow$  write log entry
9          $\langle CLR, T, T.LastLSN, LogRec.Page, LogRec.Prev, \dots, \emptyset \rangle$  ;
10      set LSN =  $LSN$  in page header of  $LogRec.Page$  ;
11       $T.LastLSN \leftarrow LSN$  ;
12     case CLR
13        $UndoNxt \leftarrow LogRec.UNxt$  ;
14     if  $T.UNxt = '-'$  then
15       write EOT log entry for  $T$  ;
16       delete TCB for  $T$  ;
```

Checkpointing

- ▶ We've considered the WAL as an ever-growing log file that we read **from the beginning** during crash recovery.
- ▶ In practice, we do not want to replay a log that has grown over days, months, or years.
- ▶ Every now and then, write a **checkpoint** to the log.
 - (a) **heavyweight checkpoints**
Force all dirty buffer pages to disk, then write checkpoint. Redo pass may then start at the checkpoint.
 - (b) **lightweight checkpoints** (or “fuzzy checkpoints”)
Do not force anything to disk, but write information about dirty pages to the log. Allows redo pass to start from a log entry shortly **before** the checkpoint.

Fuzzy Checkpointing

Periodically write checkpoint in three steps:

1. Write **begin checkpoint** log entry BCK.
2. Collect information about
 - ▶ all **dirty pages** in the buffer manager and the LSN of the **oldest** update operation that modified them and
 - ▶ all **active transactions** (and their LastLSN and UNxt TCB entries).

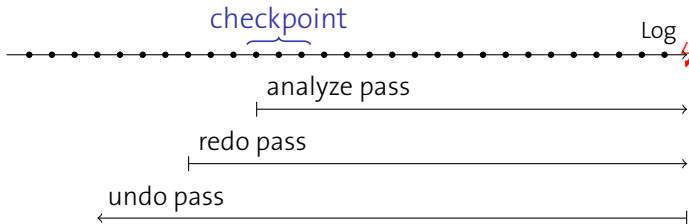
Write this information into the **end checkpoint** log entry ECK.

3. Set **master record** at a known place on disk to point to the LSN of the BCK log entry.

Recovery with Fuzzy Checkpointing

During crash recovery

- ▶ start **analyze pass** at the BCK entry recorded in the master record (instead of from the beginning of the log).
- ▶ When reading the ECK log entry,
 - ▶ Determine **smallest LSN** for **redo** processing and
 - ▶ Create TCBs for all transactions in the checkpoint.



Media Recovery

- ▶ To allow for recovery from **media failure**, periodically **back up** data to stable storage.
- ▶ Can be done **during normal processing**, if WAL is archived, too.
- ▶ If the backup process uses the **buffer manager**, it is sufficient to archive the log starting from the moment when the backup started.
 - ▶ Buffer manager already contains freshest versions.
 - ▶ Otherwise, log must be archived starting from the oldest write to any page that is dirty in the buffer.
- ▶ Other approach: Use log to **mirror** database on a remote host (send log to network **and** to stable storage).

Wrap-Up

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 practicable technique to guarantee serializability. Most systems implement **strict 2PL**. SQL 92 allows explicit **relaxation** of the ACID isolation constraints in the interest of performance.

Concurrency in B-trees

Specialized protocols exist for concurrency control in B-trees (the root would be a locking bottleneck otherwise).

Recovery (ARIES)

The ARIES technique aids to implement **durability** and **atomicity** by use of a **write-ahead log**.