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.

```
$bal \leftarrow$ read_bal $(acct\_no)$ ;
$bal \leftarrow bal - 100$ CHF ;
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 $chk\_bal \leftarrow \texttt{read\_bal}\ (chk\_acct\_no)$ ;
2 $chk\_bal \leftarrow chk\_bal - 500\ \mathrm{CHF}$ ;
3 $\texttt{write\_bal}\ (chk\_acct\_no, chk\_bal)$ ;
  // Credit money to the target (saving) account
4 $sav\_bal \leftarrow \texttt{read\_bal}\ (sav\_acct\_no)$ ;
5 $sav\_bal \leftarrow sav\_bal + 500\ \mathrm{CHF}$ ;
6 $\texttt{write\_bal}\ (sav\_acct\_no, sav\_bal)$ ;
```

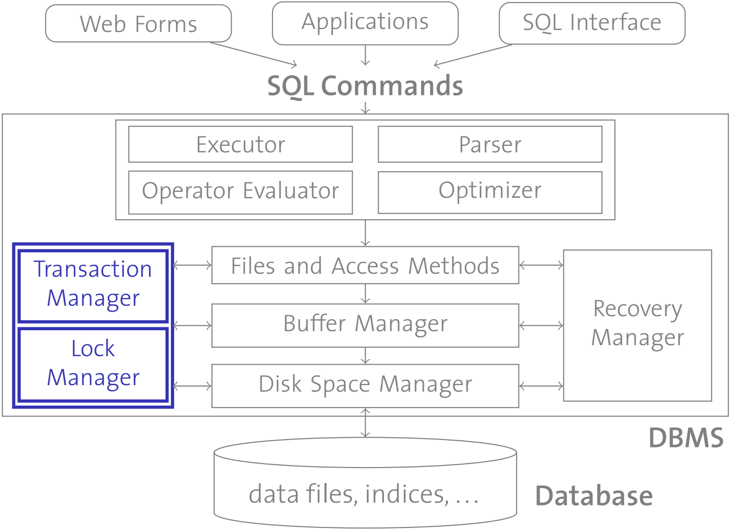
- Before the transaction gets to step **6**, its execution is **interrupted/cancelled** (power outage, disk failure, software bug, …). My money is **lost** ☹.

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                        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

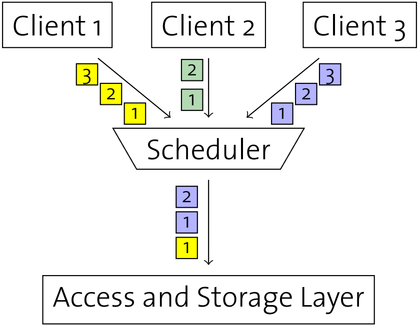
At a different day, my wife and me again end up in front of an ATM at roughly the same time:

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

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

Transactions

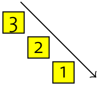
A **database transaction** T is a (strictly ordered) sequence of **steps**. Each **step** is a pair of an **access operation** applied to an **object**.

- 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

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

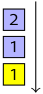


or, more concisely,

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

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 an 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

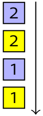
$$\begin{array}{ll} S(1) = (T_1, \texttt{read}, B) & S(3) = (T_1, \texttt{write}, B) \\ S(2) = (T_2, \texttt{read}, B) & S(4) = (T_2, \texttt{write}, B) \end{array}$$

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).

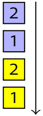
Consider again the ATM example from slide 201.



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

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

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]

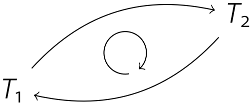
---

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

Serialization Graph

Example: ATM transactions (↗ slide 201)

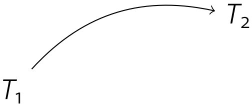
- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$
- Conflict relation:
  $(T_1, \mathtt{r}, A) \prec_S (T_2, \mathtt{w}, A)$
  $(T_2, \mathtt{r}, A) \prec_S (T_1, \mathtt{w}, A)$
  $(T_1, \mathtt{w}, A) \prec_S (T_2, \mathtt{w}, A)$



→ **not** serializable

Example: Two money transfers (↗ slide 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:
  $(T_1, \mathtt{r}, C) \prec_S (T_2, \mathtt{w}, C)$
  $(T_1, \mathtt{w}, C) \prec_S (T_2, \mathtt{r}, C)$
  $(T_1, \mathtt{w}, C) \prec_S (T_2, \mathtt{w}, C)$
  $\vdots$



→ serializable

Query Scheduling

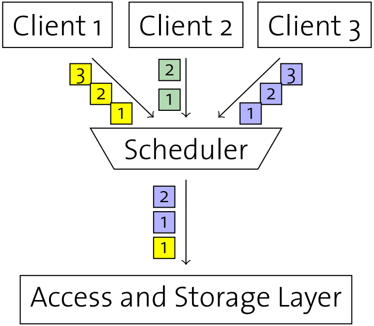
Can we build a scheduler that **always** emits a serializable schedule?

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*.



Client 1

Client 2

Client 3

Scheduler

Access and Storage Layer

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?

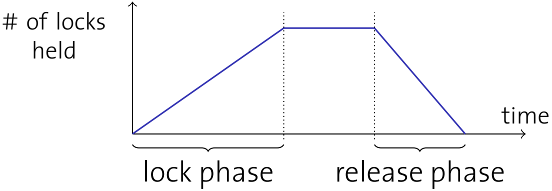
ATM Transaction with Locking

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 ($acct$) ;                          } lock phase
$bal \leftarrow$ read_bal ($acct$) ;
$bal \leftarrow bal - 100$ CHF ;
write_bal ($acct$, $bal$) ;
unlock ($acct$) ;                        } 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.

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]

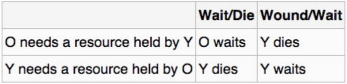
- Both transactions would wait for each other **indefinitely**.

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:
  - 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



	Wait/Die	Wound/Wait
O needs a resource held by Y	O waits	Y dies
Y needs a resource held by O	Y dies	Y waits

Other common techniques:

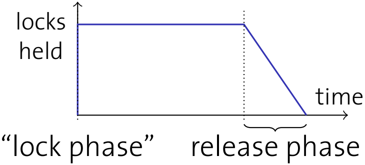
- **Deadlock prevention:** *e.g.*, by treating handling lock requests in an **asymmetric** way:
  - **wait-die**: A transaction is never blocked by an **older** transaction.
  - **wound-wait**: A transaction is never blocked by a **younger** transaction.
- **Timeout:** Only wait for a lock until a timeout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

E.g., IBM DB2 UDB:

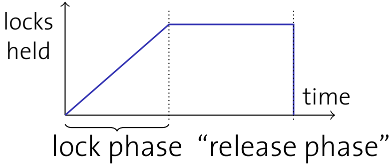
```
db2 => GET DATABASE CONFIGURATION;
   :
   :
Interval for checking deadlock (ms)       (DLCHKTIME) = 10000
Lock timeout (sec)                      (LOCKTIMEOUT) = -1
```

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?**

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!

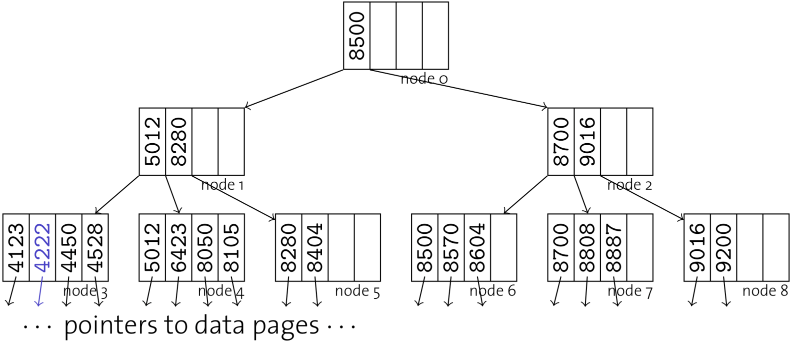
- Although both transactions properly followed the 2PL protocol, T_1 observed an effect caused by T_2.
- Cause of the problem: T_1 can only lock **existing** rows.

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.

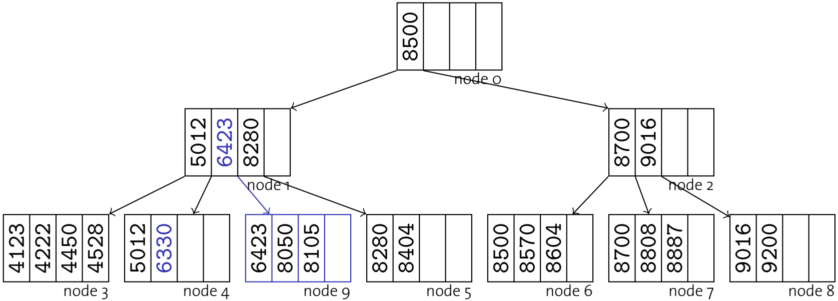
- Assume node 4 has just been split, but the new separator has **not yet** been inserted into node 1.
- Now a concurrent **read** transaction T_r tries to find 8050.
- The (old) node 1 guides T_r to node 4.
- Node 4 no longer contains entry 8050, T_r believes there is no data item with zip code 8050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

Insert: Examples (Insert without Split)



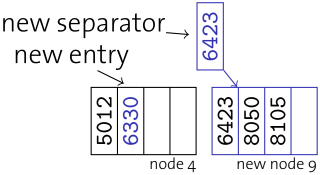
Insert new entry with key 6330.

Insert: Examples (Insert with Leaf Split)



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 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 tree[15] and
- **keep** all locks until we're finished.

---

[15] Note 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.

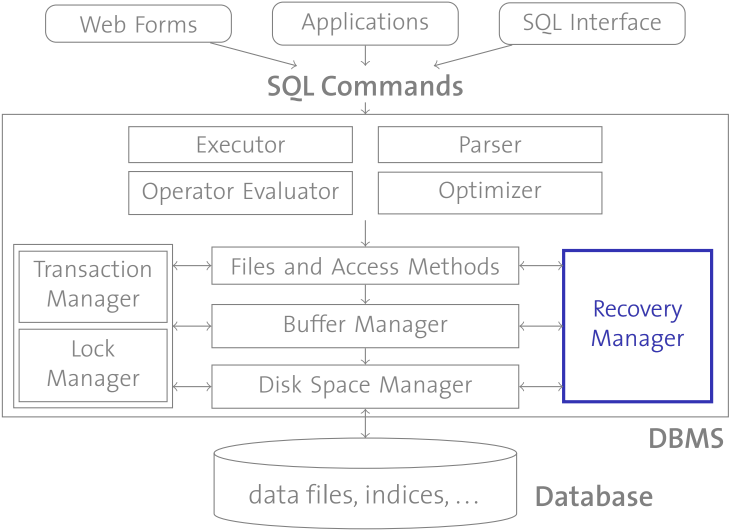
Effectively,

- 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 may 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**.

Recovery


Web Forms
Applications
SQL Interface
SQL Commands
Executor
Parser
Operator Evaluator
Optimizer
Transaction Manager
Lock Manager
Files and Access Methods
Buffer Manager
Disk Space Manager
Recovery Manager
DBMS
data files, indices, …
Database

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**.

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**

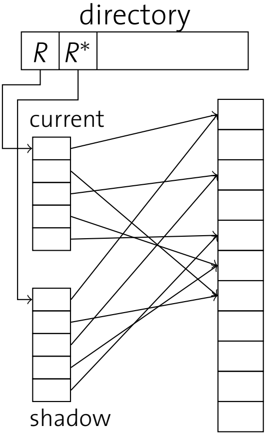
Shadow Pages

1. Initially: shadow ≡ 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).

---

[17] Algorithm for Recovery and Isolation Exploiting Semantics

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.

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**.