Architecture and Implementation of Database Systems (Winter 2013/14)

Jens Teubner, DBIS Group
jens.teubner@cs.tu-dortmund.de

Winter 2013/14
Part VII

Transaction Management and Recovery
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. $\text{bal} \leftarrow \text{read_bal}(\text{acct_no})$;
2. $\text{bal} \leftarrow \text{bal} - 100 \text{ CHF}$;
3. $\text{write_bal}(\text{acct_no}, \text{bal})$;

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

<table>
<thead>
<tr>
<th>me</th>
<th>my wife</th>
<th>DB state</th>
</tr>
</thead>
<tbody>
<tr>
<td>( bal \leftarrow \text{read}(acct) ; )</td>
<td>( bal \leftarrow \text{read}(acct) ; )</td>
<td>1200</td>
</tr>
<tr>
<td>( bal \leftarrow bal - 100 ; )</td>
<td>( bal \leftarrow bal - 200 ; )</td>
<td>1200</td>
</tr>
<tr>
<td>\text{write}(acct, bal) ;</td>
<td>\text{write}(acct, bal) ;</td>
<td>1200</td>
</tr>
</tbody>
</table>

- The first update was lost during this execution. Lucky me!
Another Example

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

```plaintext
// 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 ☹️**.
One of the key benefits of a database system are the transaction properties guaranteed to the user:

A  Atomicity  Either all or none of the updates in a database transaction are applied.

C  Consistency  Every transaction brings the database from one consistent state to another.

I  Isolation  A transaction must not see any effect from other transactions that run in parallel.

D  Durability  The effects of a successful transaction maintain persistent and may not be undone for system reasons.

A challenge is to preserve these guarantees even with multiple users accessing the database concurrently.
We already saw a **lost update** example on slide 213.
The effects of one transaction are lost, because of an uncontrolled overwriting by the second transaction.
Anomalies: Inconsistent Read

Consider the money transfer example (slide 214), expressed in SQL syntax:

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPDATE Accounts</td>
<td>SELECT SUM(balance)</td>
</tr>
<tr>
<td>SET balance = balance - 500</td>
<td>FROM Accounts</td>
</tr>
<tr>
<td>WHERE customer = 4711</td>
<td>WHERE customer = 4711</td>
</tr>
<tr>
<td>AND account_type = 'C';</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UPDATE Accounts</td>
</tr>
<tr>
<td></td>
<td>SET balance = balance + 500</td>
</tr>
<tr>
<td></td>
<td>WHERE customer = 4711</td>
</tr>
<tr>
<td></td>
<td>AND account_type = 'S';</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>me</th>
<th>my wife</th>
<th>DB state</th>
</tr>
</thead>
<tbody>
<tr>
<td>( bal \leftarrow \text{read}(acct) );</td>
<td>( bal \leftarrow \text{read}(acct) );</td>
<td>1200</td>
</tr>
<tr>
<td>( bal \leftarrow bal - 100 );</td>
<td>( bal \leftarrow bal - 200 );</td>
<td>1200</td>
</tr>
<tr>
<td>write((acct, bal));</td>
<td>write((acct, bal));</td>
<td>1100</td>
</tr>
<tr>
<td>abort;</td>
<td>1100</td>
<td>1100</td>
</tr>
<tr>
<td></td>
<td>write((acct, bal));</td>
<td>1200</td>
</tr>
<tr>
<td></td>
<td>900</td>
<td></td>
</tr>
</tbody>
</table>

- My wife’s transaction has already read the modified account balance before my transaction was rolled back.
The **scheduler** decides the execution order of concurrent database 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 $\text{read } o$ and $\text{write } o$.

In a relational DBMS we have that

\[ \text{object} \equiv \text{attribute}. \]
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** $a_i \in \{ r(\text{ead}), w(\text{rite}) \}$

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

We could write the money transfer transaction as

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

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

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

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

$$S(1) = (T_1, \text{read}, B) \quad S(3) = (T_1, \text{write}, B)$$
$$S(2) = (T_2, \text{read}, B) \quad S(4) = (T_2, \text{write}, B)$$
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 213.

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

If my wife had gone to the bank one hour later, “our” schedule probably would have been serial.

- $S = \langle r_1(B), w_1(B), r_2(B), w_2(B) \rangle$
Anomalies such as the “lost update” problem on slide 213 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.
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 223).
  - 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:

<table>
<thead>
<tr>
<th></th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>write</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

- Conflict relation \(\prec_S\):

\[
(T_i, a, e) \prec_S (T_j, a', e')
\]

\[
:=
\]

\[
(a, e) \leftrightarrow (a', e') \land (T_i, a, e) \text{ occurs before } (T_j, a', e') \text{ in } S \land T_i \neq T_j
\]
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**.\(^{17}\)

---

\(^{17}\)A serial execution of $S$ could be obtained by sorting $G(S)$ topologically.
Example: ATM transactions (↗ slide 213)

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

\( T_1 \rightarrow \text{not serializable} \)

Example: Two money transfers (↗ slide 214)

- \( S = \langle r_1(C), w_1(C), r_2(C), w_2(C), r_1(S), w_1(S), r_2(S), w_2(S) \rangle \)
- Conflict relation:
  - \((T_1, r, C) S (T_2, w, C)\)
  - \((T_1, w, C) S (T_2, r, C)\)
  - \((T_1, w, C) S (T_2, w, C)\)
  - ... \( \rightarrow \text{serializable} \)
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. $\ldots\ access\ o\ \ldots;$
  3. $unlock\ o\ ;$

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

![Diagram of client interactions with a scheduler and access and storage layer.](image-url)
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.
Does locking guarantee serializable schedules, yet?

```plaintext
1. lock (acct);
2. bal ← read bal (acct);
3. unlock (acct);
4. bal ← bal - 100 CHF;
5. lock (acct);
6. write bal (acct, bal);
7. unlock (acct);
```
### ATM Transaction with Locking

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction 2</th>
<th>DB state</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lock (acct)</code>;</td>
<td><code>lock (acct)</code>;</td>
<td>1200</td>
</tr>
<tr>
<td><code>read (acct)</code>;</td>
<td><code>read (acct)</code>;</td>
<td></td>
</tr>
<tr>
<td><code>unlock (acct)</code>;</td>
<td><code>unlock (acct)</code>;</td>
<td></td>
</tr>
<tr>
<td><code>lock (acct)</code>;</td>
<td><code>lock (acct)</code>;</td>
<td></td>
</tr>
<tr>
<td><code>write (acct)</code>;</td>
<td><code>write (acct)</code>;</td>
<td>1100</td>
</tr>
<tr>
<td><code>unlock (acct)</code>;</td>
<td><code>unlock (acct)</code>;</td>
<td>1000</td>
</tr>
</tbody>
</table>
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.

![Diagram showing lock and release phases](image)

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

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction 2</th>
<th>DB state</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lock (acct) ;</code></td>
<td><code>lock (acct) ;</code></td>
<td>1200</td>
</tr>
<tr>
<td><code>read (acct);</code></td>
<td><code>read (acct);</code></td>
<td></td>
</tr>
<tr>
<td><code>unlock (acct) ;</code></td>
<td><code>unlock (acct) ;</code></td>
<td></td>
</tr>
<tr>
<td><code>lock (acct) ;</code></td>
<td><code>lock (acct) ;</code></td>
<td>1100</td>
</tr>
<tr>
<td><code>write (acct);</code></td>
<td><code>write (acct);</code></td>
<td></td>
</tr>
<tr>
<td><code>unlock (acct) ;</code></td>
<td><code>unlock (acct) ;</code></td>
<td>1000</td>
</tr>
<tr>
<td><code>lock (acct) ;</code></td>
<td><code>lock (acct) ;</code></td>
<td></td>
</tr>
<tr>
<td><code>write (acct);</code></td>
<td><code>write (acct);</code></td>
<td></td>
</tr>
<tr>
<td><code>unlock (acct) ;</code></td>
<td><code>unlock (acct) ;</code></td>
<td></td>
</tr>
</tbody>
</table>
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) ; } lock phase
2 bal ← read_bal (acct) ;
3 bal ← bal − 100 CHF ;
4 write_bal (acct, bal) ;
5 unlock (acct) ; } unlock phase
```
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:

\[
\begin{array}{ccc}
\text{shared (S)} & \text{exclusive (X)} \\
\text{shared (S)} & \times \\
\text{exclusive (X)} & \times & \times \\
\end{array}
\]

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:

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock ( (A) );</td>
<td>lock ( (B) )</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>do something</td>
<td>do something</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>lock ( (B) ) \hspace{1cm} [wait for ( T_2 ) to release lock]</td>
<td>lock ( (A) ) \hspace{1cm} [wait for ( T_1 ) to release lock]</td>
</tr>
</tbody>
</table>

- Both transactions would wait for each other indefinitely.
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.
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
```
The two-phase locking protocol does not prescribe exactly when locks have to acquired and released.

Possible variants:

- **preclaiming 2PL**
  - Locks held during the lock phase.
  - The lock is released during the release phase.

- **strict 2PL**
  - Locks held during the lock phase.
  - The lock is held until the end of the release phase.

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

- Resource Control Block (RCB)
  - Resource ID
  - Hash Chain
  - First In Queue

- Lock Control Blocks (LCBs)
  - Transaction ID
  - Resource ID
  - Lock Mode
  - Lock Status
  - Next in Queue
  - LCB Chain

- Transaction Control Block (TCB)
  - Transaction ID
  - Update Flag
  - TX Status
  - # of Locks
  - LCB Chain
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:

- **Database level**: low concurrency, low overhead
- **Tablespace level**: low concurrency, low overhead
- **Table level**: high concurrency, high overhead
- **Page level**: high concurrency, high overhead
- **Row-level**: high concurrency, high overhead

**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
    
    ```sql
    SELECT * FROM CUSTOMERS
    WHERE C_CUSTKEY = 42
    ```
    
    and a table lock for
    
    ```sql
    SELECT * FROM CUSTOMERS
    ```
    
- How do such transactions know about each others’ locks?
  - Note that locking is **performance-critical**. Q2 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:

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
<th>IS</th>
<th>IX</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>×</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>×</td>
<td>×</td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>IS</td>
<td></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IX</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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 \textbf{CUSTOMERS} (also on on tablespace and database) and
- obtain an S lock on the \textbf{tuple(s)} with \texttt{C\_CUSTKEY} = 42.

Query \( Q_2 \) would place an
- S lock on table \textbf{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
```

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).
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**.
DB2

Ratio of correct answers

Concurrent update threads

Read committed
Serializable

DB2

Throughput (trans/sec)

Concurrent update threads

Read committed
Serializable

© Jens Teubner · Architecture & Implementation of DBMS · Winter 2013/14
**SQL Server**

**Ratio of correct answers**
- **Read committed**
- **Serializable**

**Throughput (trans/sec)**
- **Read committed**
- **Serializable**

**Concurrent update threads**

_Dennis Shasha and Philippe Bonnet. Database Tuning. Morgan Kaufmann, 2003._
Oracle

Ratio of correct answers

Concurrent update threads

Read committed

Serializable

Oracle

Throughput (trans/sec)

Concurrent update threads

Read committed

Serializable
### Resulting Consistency Guarantees

<table>
<thead>
<tr>
<th>isolation level</th>
<th>dirty read</th>
<th>non-repeat. rd</th>
<th>phantom rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>read uncommitted</td>
<td>possible</td>
<td>possible</td>
<td>possible</td>
</tr>
<tr>
<td>read committed</td>
<td>not possible</td>
<td>possible</td>
<td>possible</td>
</tr>
<tr>
<td>repeatable read</td>
<td>not possible</td>
<td>not possible</td>
<td>possible</td>
</tr>
<tr>
<td>serializable</td>
<td>not possible</td>
<td>not possible</td>
<td>not possible</td>
</tr>
</tbody>
</table>

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

Consider three transactions:

\[
\begin{align*}
T_1 &; w(x) &; \text{abort} ; \\
T_2 &; r(x) &; \\
T_3 &; r(x) &;
\end{align*}
\]

- When transaction \( T_1 \) aborts, transactions \( T_2 \) and \( T_3 \) have already read data written by \( T_1 \) (↗ dirty read, slide 219)
- \( T_2 \) 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
Transaction 1

```
SELECT COUNT(*)
FROM Customers
WHERE Name = 'Sam'
```

Transaction 2

```
INSERT INTO Customers
VALUES (..., 'Sam', ...)
```

```
SELECT COUNT(*)
FROM Customers
WHERE Name = 'Sam'
```

| Result | 2 | ok | 3 |

Transaction 1 “sees” the concurrent insert done by Transaction 2.

→ **Isolation property violated.**

This is an instance of the **phantom problem**.
Avoiding Phantoms

Locking only tuples cannot avoid the phantom problem.

- The tuple added by $T_2$ is new; $T_1$ could never have locked it before.
- To avoid the phantom problem, we also have to lock absent tuples.

Phantoms can be avoided with:

- **Predicate Locking**: For each query, lock the predicates that it uses.
  
  Representing, finding, and comparing predicates can be difficult and inefficient.

- **Key-Range Locking**: Lock index entries that match the predicate.
  
  *E.g.*, in the previous example, lock the index key `Sam`. 
Key-Range Locking

- Use B-trees to lock **key values, not tuples**!
  → This is somewhat orthogonal to regular data locking.
- In general, we want to lock **ranges** of key values.
  → Including **absence** of key values.
  → Lock existing **key values** and **gaps**.

→ The current index content determines which ranges can be locked.
**Typically:**

- Acquire **one lock** to mean a key value **and** its neighboring gap:

  
  ![Diagram of key-range locking]

  

  → **Previous key locking:**  
  Lock covers key value $x$ and the gap that **follows** $x$.

  → **Next key locking:**  
  Lock covers key value $x$ and the gap that **precedes** $x$.

This way, existing key values can be used as lookup keys in the system’s **lock manager** (which is typically organized as a hash table).
Reading Transactions

Idea:

- Queries acquire S locks for all key ranges that intersect with ranges in query predicates.

E.g., scan range \([4200, 5000]\):

\[
\begin{array}{ccccccc}
4104 & 4123 & 4222 & 4450 & 4528 & 5012 & 6330 & 6423
\end{array}
\]

→ Ranges \([4123, 4200]\) and \([5000, 5012]\) locked “too much”!
Inserts

- Inserts need to acquire a lock on the gap into which they want to insert.
- Thus, with next key locking: acquire lock on next-largest key.

*E.g.*, insert 4500:

→ Acquire X lock on 4528 (which covers range [4450, 4528]).
→ If the reading transaction from the previous slide still holds its locks, a conflict on 4528 will be detected (and the insert will have to wait).
→ Insert new key and X lock it immediately.
Lock Duration

Readers:
- Keep the range locked until the transaction commits. This is to make sure the range can be re-read at any time without seeing phantoms.

Inserts:
- Keep newly inserted entry $X$ locked until commit time. This prevents others from reading un-committed data.
- The lock on the next key (4528 here), however, can be released immediately.
  - Acquiring the lock with “instant duration” ensures there is no co-running reader for that range.
  - Once the new key is inserted, readers (or writers) are free to lock the next key (4528), since its associated range ($[4500, 4528]$ now) only covers the gap without the newly inserted key.

© Jens Teubner · Architecture & Implementation of DBMS · Winter 2013/14
This ability to lock with instant duration is very relevant in practice.

- Inserts at the **right end** of a B-tree are a very common pattern.
  - Next key locking requires an extra $+\infty$ index entry, by the way.
  - All append queries will lock this $+\infty$ entry.
  - When the lock on $+\infty$ is an instant lock, other inserts can proceed immediately.
- Note how this also favors next key locking over previous key locking.
To **delete** an entry $x$, the transaction has to obtain

- an $X$ lock on the to-be-deleted entry $x$,
  - Make sure no other transaction still depends on $x$.
  - The lock is effectively instant, since the transaction is about to remove $x$ anyway.

- an $X$ lock on $x$’s **next key** until **commit time**.
  - **Why?**
IBM DB2 does not lock index entries explicitly.

- Instead, DB2 performs **data-only locking**.
- A locked tuple **implies** a key-range lock in **all** indexes on the table.
- When checking for lock compatibility, DB2 looks for already held locks, but also considers the **isolation level** of the lock holder.

Data-only locking may lead to unexpected **side effects**:

- *E.g.*, a scan criterion on one column may lead to locks in scattered regions of other attributes.

On the positive side, deriving key-range locks from row locks reduces the number of locks to maintain (and thus the complexity of the lock manager).
Support for **ghost records** may ease key-range locking considerably.

- **Deletes** will not actually remove the index entry, but only turn the record into a ghost.
- The ghost still represents a valid range boundary (locks can be acquired on ghosts just as on normal records).
- Flipping the ghost bit is merely a form of **value update** of the record.
  - Value updates do not need range locks as long as they do not modify the key value.
The same advantages also hold for **inserts** if a ghost with the right key value already exists.

→ Need to lock only the key value itself (neighboring range is often implicit, but not strictly required).

Existence of a matching ghost need **not** be a coincidence.

**Trick:**

- Invoke a short, **separate transaction** that creates the ghost for us.
- The transaction will have to acquire range locks. But it will commit immediately (and release its locks).
So far we looked at ill effects of transactions. Locks on data objects helped to isolate transactions.

Parallel threads might cause additional problems:

- Two writers, different data objects, same page → corrupted data.
- Locks will not isolate threads that belong to the same transaction.
- How do we protect internal data structures (lock table, buffer pool, etc.)?
  - Lock manager can only lock user data objects!

This calls for a mechanism to isolate threads (not transactions).

- Short-lived, in-memory “locks” or latches.
  (The term “lock” is reserved for transaction-level locking.)
Page Latches

Latches protect data at a page granularity.

→ This has also been called storage-layer concurrency.

To achieve high concurrency:

■ Hold latches as short as possible.
■ Hold few latches only (and/or latch at fine granularities).

In addition:

■ Choose a fast implementation for latches.
  → no frills like deadlock checking
  → instead: avoid deadlocks by coding discipline
Page Latches (Data Pages)

Example:

- Latches on data pages make page modifications appear as an atomic operation.
  → Protect from, e.g., observing a corrupt page.

- Latching is in-memory only.
  → No I/O while holding a page latch.
  → Latches are not flushed to disk.

- Only hold one latch at a time.
  → Why?
Order for latch acquisition/release during a B-tree search?

1. n ← root page;
2. read-latch n;
3. while n is not a leaf do
   4. locate child n′ of n;
   5. read-latch n′ obtain new latch first!
   6. un-latch n;
   7. n ← n′;
8. return matching records (if any);
9. un-latch n;

This is also known as latch coupling (or lock coupling).
Latch Coupling

With latch coupling, a thread may hold more than one latch at a time.

→ A deadlock still cannot occur:
  - Every thread will navigate/acquire latches top-down.
  - All threads acquire latches in same order → no deadlock.
Updates to B-trees operate **bottom-up**.

**Possible strategy:**

- Acquire read latches as during search, but **keep** all latches.
  - Ensure that the parent (grandparent, ...) is still the parent during bottom-up processing.

- Acquire **write latches bottom-up**.
  - Latch conversion: read latch $\sim$ write latch.
  - Write-latch parent before splitting a child.

- Release write latches when all necessary changes to the page are applied; release ancestor read latches when no more splits are necessary.

If the B-tree implementation uses **sibling pointers**, additional locks may have to be acquired on **sibling nodes**.
The strategy on the previous slide guarantees **correctness**.

- All tree modifications are write-latched, and released latches always leave behind a consistent B-tree.

**But:**

⚠️ The strategy entails a danger of **deadlocks**.

- **Searches** acquire their latches **top-down**.
- **Updates** acquire their (write) latches **bottom-up**.

**Remember:** We want latches to be lightweight → no deadlock checking.
Deadlocks can be avoided when all operations acquire latches either top-down or bottom-up.

Thus:

- Let insert operations acquire write latches right away.

What do you think of this strategy?
Chances that a write latch on a parent is actually needed are really low. 
→ E.g., B-tree with up to 100 entries/node → chance of a split: 2%

**Idea:** (Try to) keep write latch only when really necessary.
- During tree descent, observe **space utilization** in visited nodes.
- When a node $n$ has **enough space** to hold another entry, $n$ definitely won’t have to be split.
- For such nodes $n$, the **parent node** $p$ will not have to be updated.
  → $p$ is then called **split safe**.
- The latch on that parent $p$ can be released safely.
Lock Coupling Protocol (Variant 1)

**Readers**

1. place $S$ lock on *root*;
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 $current$;

**Writers**

1. place $X$ lock on *root*;
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:
    \[ d = 50 \Rightarrow \text{every 50th insert causes a split (2\% chance)}. \]
- 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**:\(^{18}\)

1. place $S$ lock on *root* ;
2. $current \leftarrow root$ ;
3. **while** $current$ is not a leaf node **do**
   4. $son \leftarrow$ 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. $current \leftarrow son$ ;
11. **if** $current$ is unsafe **then**
12. release all locks and repeat with protocol Variant 1 ;

\(^{18}\)Reader protocol remains unchanged.
Deciding **split safety** can be difficult for **variable-length keys**.

The strategy on the previous slide thus has to be **very conservative**.

Effectively, many latches are still held **unnecessarily**.

Ways to improve concurrency (by holding fewer latches):

- **split proactively**: When a node is not split safe, split it right away. At least the system then suffers the unnecessary latch only once.

- **repeated root-to-leaf passes**: Descend with only read latches first. Re-traverse the tree with full write latches when a split is necessary.

- **giveup technique**: hold only single-node read latches (and risk inconsistencies); detect conflicts and re-traverse in case of a conflict.

- **Blink-trees**: slightly relax some B-tree rules.
Giveup Technique

A deadlock can only arise when a thread acquires (or tries to) a new latch before releasing an old one.

→ A thread that always only holds a single latch at a time can never deadlock.

Search routine with only a single latch held at any time:

1. $n \leftarrow$ root page;
2. **while** $n$ is not a leaf **do**
3. read-latch $n$;
4. determine child $n'$ of $n$;
5. un-latch $n$;
6. $n \leftarrow n'$;
7. read-latch $n$;
8. return matching records (if any);
9. un-latch $n$;
There is a risk of inconsistencies when only a single latch is held.

- Between determining the child page \( n' \) and latching it, a concurrent update might have split \( n' \).
- The search might miss an entry that is now on a new page.

Thus: Detect when a conflicting update has happened.

- When descending, remember the two separator keys \( k_{min} \) and \( k_{max} \) in \( n \) that guided to \( n' \).
- When looking at \( n' \), first check whether \( k_{min} \) and \( k_{max} \) are still the correct separator keys for that page.
  → Keep copies of parent’s separator keys in each node.
  → Such copies are also called fence keys.
- If a conflict is detected, abort and re-try a moment later.
Lehman and Yao\textsuperscript{19} proposed a B-tree variant, usually referred to as \textit{Blink-tree}, where writes must latch at most two nodes at a time.

**Idea:**

- Assume a B-tree with \textit{forward sibling pointers}.
- \textbf{Relax B-tree structure:} Allow parent $\rightarrow$ child to be missing when the child is reachable via the sibling pointer of its predecessor.

With the relaxation, node splitting and parent updates can be separated.

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

→ Lines 9–11 can be deferred to a later time.
With the relaxation stated before, lines 1–8 already represent a correct B\textsuperscript{link}-tree.

- Lines 9–11 are, in a sense, only applied for performance reasons.

The parent could be updated also at a later time:
- As a “clean-up process” triggered when the update has completed.
- When the next search traverses the tree.
- During database maintenance.

In fact, even the page latches can be avoided when pointer updates and record deletions can be done atomically.

\*\*PostgreSQL\*, e.g., uses B\textsuperscript{link}-Trees.
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

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

**forward-oriented optimistic concurrency control (FOCC):**

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

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

\[ r_1(x), w_1(x), r_2(x), w_2(y), r_1(y), w_1(z) \].

Is this schedule serializable?

- Now suppose when \( T_1 \) wants to read \( y \), we’d still have the “old” value of \( y \), valid at time \( t \), around.
- We could then create a history equivalent to

\[ r_1(x), w_1(x), r_2(x), r_1(y), w_2(y), w_1(z) \],

which is serializable.
A simple form of MVCC is the **Read-Only MVCC**:

- **Read/write transactions** use concurrency control as before (e.g., 2PL)
- **Read-only transactions** do not acquire any locks. For each read operation \( r(x) \) of a read-only transaction \( T_{RO} \), read the version of \( x \) that existed when \( T_{RO} \) started.

That is, read-only transactions see a **snapshot** of the database as of the time when they started.

**Problem:**

- Must mark each data object with **commit time** of transaction.
Oracle implements “read committed” (↗ slide 252) using the “Read-Consistency” protocol:

- **read-only transactions** are treated as in the Read-Only protocol.
- **writes in read/write transactions** acquire long-duration write locks.
- **reads in read/write transactions** do not acquire read locks; they read the most recent version of any data object.

→ Reads only return committed values (↔ read committed).
→ Read-only transactions see consistent state (unlike in read committed).
→ Readers never block writers and vice versa.
A modification of the same idea yields **snapshot isolation**.

- All **reads** of any transaction $T$ see the version that was current when $T$ started.

- All **writes** must satisfy the **“first committer wins”** property. A transaction $T$ is allowed to commit only if there is no other transaction $T'$ such that
  1. $T'$ committed between the start and commit time of $T$ and
  2. $T'$ updated a data object that $T$ also updated.

Otherwise, $T$ aborts.

To test “first committer wins,” compare write sets of $T$ and $T'$.

Snapshot isolation is implemented, *e.g.*, in Oracle, SQL Server, PostgreSQL.
Recovery

SQL Commands

- Executor
- Operator Evaluator
- Parser
- Optimizer
- Files and Access Methods
- Buffer Manager
- Disk Space Manager
- Transaction Manager
- Lock Manager
- Recovery Manager

Web Forms → Applications → SQL Interface

DBMS

data files, indices, ...

© Jens Teubner · Architecture & Implementation of DBMS · Winter 2013/14
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.
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.
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 \textbf{current} version.
   - Updates are \textbf{not} done in-place.
   - Create new pages and alter current page table.

3a. If $T$ \textbf{aborts}, overwrite current version with shadow version.

3b. If $T$ \textbf{commits}, change information in \textbf{directory} to make current version persistent.

4. Reclaim disk pages using \textbf{garbage collection}.
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 34, 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.
The decisions force/no force and steal/no steal have implications on what we have to do during recovery:

<table>
<thead>
<tr>
<th></th>
<th>force</th>
<th>no force</th>
</tr>
</thead>
<tbody>
<tr>
<td>no steal</td>
<td>no redo</td>
<td>must redo</td>
</tr>
<tr>
<td></td>
<td>no undo</td>
<td>no undo</td>
</tr>
<tr>
<td>steal</td>
<td>no redo</td>
<td>must redo</td>
</tr>
<tr>
<td></td>
<td>must undo</td>
<td>must undo</td>
</tr>
</tbody>
</table>

If we want to use steal and no force (to increase concurrency and performance), we have to implement redo and undo routines.
The ARIES\textsuperscript{20} recovery method uses a write-ahead log to implement the necessary redundancy. Data pages are updated in place.


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).

\textsuperscript{20}Algorithm for Recovery and Isolation Exploiting Semantics
Content of the Write-Ahead Log

<table>
<thead>
<tr>
<th>LSN</th>
<th>Type</th>
<th>TX</th>
<th>Prev</th>
<th>Page</th>
<th>UNxt</th>
<th>Redo</th>
<th>Undo</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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.
<table>
<thead>
<tr>
<th>Transact. 1</th>
<th>Transact. 2</th>
<th>LSN</th>
<th>Type</th>
<th>TX</th>
<th>Prev</th>
<th>Page</th>
<th>UNxt</th>
<th>Redo</th>
<th>Undo</th>
</tr>
</thead>
<tbody>
<tr>
<td>a ← rd(A);</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c ← rd(C);</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a ← a − 50;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c ← c + 10;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wr(a,A);</td>
<td>1</td>
<td>UPD</td>
<td>T₁</td>
<td>−</td>
<td></td>
<td></td>
<td></td>
<td>A := A − 50</td>
<td>A := A + 50</td>
</tr>
<tr>
<td></td>
<td>wr(c,C);</td>
<td>UPD</td>
<td>T₂</td>
<td>−</td>
<td></td>
<td></td>
<td></td>
<td>C := C + 10</td>
<td>C := C − 10</td>
</tr>
<tr>
<td>b ← rd(B);</td>
<td>2</td>
<td>UPD</td>
<td>T₂</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b ← b + 50;</td>
<td></td>
<td>UPD</td>
<td>T₁</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>B := B + 50</td>
<td>B := B − 50</td>
</tr>
<tr>
<td>wr(b,B);</td>
<td></td>
<td>EOT</td>
<td>T₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit;</td>
<td>3</td>
<td>UPD</td>
<td>T₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a ← rd(A);</td>
<td>4</td>
<td>EOT</td>
<td>T₁</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a ← a − 10;</td>
<td></td>
<td>EOT</td>
<td>T₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wr(a,A);</td>
<td>5</td>
<td>UPD</td>
<td>T₂</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>A := A − 10</td>
<td>A := A + 10</td>
</tr>
<tr>
<td>commit;</td>
<td>6</td>
<td>EOT</td>
<td>T₁</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- rd is for “read”; wr is for “write”
What Redo/Undo Information to Log

Redo/undo information can be **encoded in different ways**.

In **physical logging**, the exact byte representation of every page is faithfully logged and preserved.

- *E.g.*, **before** and **after image** of the entire page.
- Typically try to be smarter: log only modified parts of the page and/or compress log entries.

**Advantages:**

- Recovery mechanism is **object-independent** (whether the page is an index/data/... page doesn’t matter).
- Recovery is **page-oriented** (and pages are the granularity for atomic data changes on disk).

**Disadvantages:**

- **Log volume** can become very large. Logs are a key limitation of today’s transaction processing systems.
Disadvantages (cont.):

- A **transaction abort** might force other transactions to abort when they altered the same pages.
  - This can happen even if the transactions do **not conflict** on the logical level.

Observe that

- Physical logging not only preserves the logical database content but also its **physical representation**.
- To this end, **every** page modification must be logged.
  - Even cleanup operations or internal page re-organizations.

As such, physical logging does **more than needed**.

- The physical representation of data is not visible to the user.
- ACID only refers to the logical representation.
Logical logging is an alternative.

- Log **high-level operations**
  - *E.g.*, “insert tuple ⟨...⟩ into table R”
- A single such log record often implies a **series of changes**.
  - Insert tuple in data pages, indexes, etc.
  - May have to split index pages, allocate new heap space, etc.
- Logical logging will **not** preserve the physical representation.
  - *E.g.*, don’t undo an index page split.
  - During redo, tuples or index entries might end up on completely different pages.
Logical Logging

**Advantages:**

- Log volume **very** small.
  - Individual log entries very small.
  - Maintenance operations need not be logged at all.
- Potential to improve undo/recovery performance
  - *E.g.*, don’t undo a page split

**Disadvantages:**

- Very hard to get right.
  - Logged operations are not atomic with respect to disk operations.
  - Idempotency of redo/undo operations?
Problems of Logical Logging

Example:
- The insertion of a tuple into table $T$ implies a new entry in indexes $A$ and $B$.

**Problem 1:** Partial Actions
- A **transaction failure** could occur at any of the three steps.
- Need to know which prefix of changes to $T$, $A$, $B$ has to be undone.

**Problem 2:** Action Consistency
- In the case of **crash recovery**, any subset of the affected pages could have reached the disk before the crash.

It is even worse:
- A B-tree insertion itself may affect multiple pages (splits, etc.).
Compromise: Physiological Logging

Idea:
- “physical to a page, logical within a page”

Physical part:
- Every log record refers to a particular **physical page**.

Logical part:
- Use logical logging to describe changes **within one page**.

Example log entry:

- \[\ldots, \text{insert}, \ldots, \text{page } 4711, \ldots, \text{record value } r\]\n- \[\ldots, \text{ix insert}, \ldots, \text{ix page } 0815, \ldots, \text{ix key: } k_1, \text{rid: } v\]\n- \[\ldots, \text{ix insert}, \ldots, \text{ix page } 4242, \ldots, \text{ix key: } k_2, \text{rid: } v\]\n
© Jens Teubner · Architecture & Implementation of DBMS · Winter 2013/14
Important: page action consistency

- Complex actions composed of single-page actions, each such action is logged.
- Pages may be inconsistent during single-page action, but this situation is protected by latches.

In practice:

- Write undo/redo log record before releasing the page latch.
- Page modifications are atomic from the perspective of the logging/recovery mechanism.
The **idempotency** challenge of logical logging remains.

*E.g.*, insert log record for new tuple $t$

$\rightarrow$ $t$ should be inserted exactly **once**.

$\rightarrow$ Must not re-insert $t$ (again and again), *e.g.*, in case of **crash recovery**.

**Thus:**

- Assign a **unique, monotone log sequence number (LSN)** to each log entry.
- Record the LSN of the **latest page update** in each page header.
- During redo, apply operation **only if** $\text{pageLSN} < \text{logLSN}$.

Redo instructions themselves need **not** be idempotent!
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 244):

**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 $p$. 
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.
Function: rollback (SaveLSN, T)

undoNxt ← T.UNxt;

while SaveLSN < undoNxt do

LogRec ← read log entry with LSN undoNxt;

switch LogRec.Type do

  case UPD

    perform undo operation LogRec.Undo on page LogRec.Page;
    LSN ← write log entry
    \( \langle \text{CLR, } T, T.\text{LastLSN}, \text{LogRec.Page}, \text{LogRec.Prev, \cdots, } \emptyset \rangle \);
    set LSN = LSN in page header of LogRec.Page;
    T.\text{LastLSN} ← LSN;

  case CLR

    undoNxt ← LogRec.UNxt;

    T.UNxt ← undoNxt;

© Jens Teubner · Architecture & Implementation of DBMS · Winter 2013/14
Undo Processing

Write compensation log records (CLRs) during undo.

- The redo information in the CLR describes the performed undo operation.
- The undo operation increases the page’s LSN.

Why?
Undo Processing

Undo need **not** precisely re-establish the page to the representation before the corresponding ‘do’ operation.

*E.g.*,

- The ‘undo’ for an ‘insert’ might be a ‘delete’.
  → The deleted record might remain as a ghost.
- A B-tree node split might not be un-done at all.
- An insert might have required page compaction.

Undo only needs to re-establish the **logical** contents of a page, but **not** its physical representation.

→ This is the “logical” aspect of “physiological”.
Transaction Rollback

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

ientes why is this useful?

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

```
  UPD  UPD  UPD  UPD  CLR  CLR  UPD  CLR  CLR  Log

sav1  sav2  rollback(sav2)  rollback(sav1)
```
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;

- In practice, systems also use the analyze phase to collect further information, e.g., to **prefetch** pages for redo.
Redo Phase

1. **Function:** redo()

2. **foreach** log entry record LogRec do

3. **switch** LocRec.Type do

4. **case** UPD or CLR

5. \[ v \leftarrow \text{pin}(\text{LogRec.Page}) ; \]

6. **if** v.LSN < LogRec.LSN **then**

7. perform redo operation LogRec.Redo on v ;

8. v.LSN \leftarrow \text{LogRec.LSN} ;

9. unpin(v, ...) ;

⚠️ System crashes can occur **during** recovery!

- Undo and redo of a transaction \( T \) must be **idempotent**:

\[
\text{undo(undo}(T)) = \text{undo}(T) \\
\text{redo(redo}(T)) = \text{redo}(T)
\]

- Check LSN before performing the redo operation (line 6).
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 316). 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.
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).
Function: undo ()

while transactions (i.e., TCBs) left to roll back do

\[ T \leftarrow \text{TCB of loser transaction with greatest UNxt} \; ; \]

\[ LogRec \leftarrow \text{read log entry with LSN } T.\text{UNxt} \; ; \]

switch \( LogRec.\text{Type} \) do

\[ \text{case UPD} \]

perform undo operation \( LogRec.\text{Undo} \) on page \( LogRec.\text{Page} \) ;

\[ LSN \leftarrow \text{write log entry} \]

\[ \langle \text{CLR}, T, T.\text{LastLSN}, LogRec.\text{Page}, LogRec.\text{Prev}, \cdots, \emptyset \rangle \; ; \]

set \( LSN = LSN \) in page header of \( LogRec.\text{Page} \) ;

\[ T.\text{LastLSN} \leftarrow LSN \; ; \]

\[ \text{case CLR} \]

\[ \text{UndoNxt} \leftarrow LogRec.\text{UNxt} \; ; \]

\[ T.\text{UNxt} \leftarrow \text{UndoNxt} \; ; \]

if \( T.\text{UNxt} = '-' \) then

\[ \text{write EOT log entry for } T \; ; \]

\[ \text{delete TCB for } T \; ; \]

© Jens Teubner · Architecture & Implementation of DBMS · Winter 2013/14
Forcing Log Records to Stable Storage

The effects of all *committed* transactions must be durable.

→ When committing a transaction $T$, **force** the log to stable storage (at least) up until the *commit record* of $T$.

Conversely, in case of a *crash recovery*, any effects caused by transactions that did **not yet commit** must be undone.

→ When **evicting** a page $p$ from the buffer pool, first **force** the log to stable storage until the LSN recorded in $p$.

*What about aborted transactions? Force abort records, too?*
Logical (and mostly also physiological) logging protects only the logical content of the database.

Physical logging, by contrast, also protects the physical representation.

We saw something similar before:

- **User transactions** perform changes to the logical content of the database.
- **System transactions** only change the physical representation.

Indeed, the distinction allows us to reduce the overhead of logging.
System Transactions

System transactions need to log their operations just like user transactions.

**Example:** B-tree node split
- Migration of tuples to new node changes (“logical”) content of those pages (even if not of the B-tree overall), which thus has to be logged.
- Space allocation must be logged.

**But:**
- Whether or not the system transaction $T_x$ **commits** is immaterial until a user transaction depends on changes done by $T_x$.
- System transactions do **not** have to **force** the log to stable storage at commit time.
- A dependent user transaction (upon commit) will implicitly force the effects of $T_x$ to stable storage.
In case the invoking user transaction **aborts**, system transactions do not have to be rolled back.\(^{21}\)

\(\rightarrow\) Reduced rollback overhead compared to alternative without system transactions.

\(\rightarrow\) **Other transactions** that already saw the effects of invoked system transactions need **not** be rolled back.

- *E.g.*, B-tree node split: other transactions might already have put new entries on new node.

\(^{21}\)In fact, they cannot be rolled back if they already committed.
Similar effects **can** be achieved also without system transactions.

**Trick:**
- When finishing a “nested top action,” write a **dummy CLR** to the log:

  ![Diagram](image)

  - UPD — UPD — UPD — UPD — UPD — CLR — UPD — UPD — Log

  nested top action

- In case of an **undo**, processing will **skip** over all actions of the nested top action, thus preserve its effects.
System transactions lead to additional log records. 
→ At a minimum, there is the added commit record.

However, system transactions may also reduce log volume.

Example: Tuple deletion

- **Without system transactions:**
  - Need to log deletion (for redo) and deleted tuple value (for undo).

- **With system transactions:**
  - Turn tuple in to ghost only (log only bit flip).
  - Once the user transaction has committed, the tuple is logically NULL. A clean-up system transaction thus need not log deleted tuple value (only the key).
Such delete operations can be optimized even further.

**Step 1:**
- Merge UPD for ghost deletion and EOT (commit record) into single log record.  
  (The operation occurs frequent enough to warrant a special entry type.)
- There is now only a **single log record** when a system transaction deletes a ghost.

**Step 2:**
- Since there is only a single log record, the system transaction cannot fail in-between deletion and commit.
- Thus: **omit** logging undo operations altogether.
Write Ordering

Log volume can further be reduced by careful **write ordering**.

**Example:** B-tree node split

→ A log record like “Move entries \( k_i, \ldots, k_j \) from page \( x \) to page \( y \)” covers all information needed for the two split nodes \( x \) and \( y \).

The operation is performed **in memory** first:

If \( x \) is **written back** to disk **before** \( y \), data behind \( k_i, \ldots, k_j \) is no longer persistent on disk!
Write Ordering

There is **no problem** if page $y$ is written back **first**.

**Thus:** Add **write order** information to buffer pool meta data.

1. Let destination page $y$ **depend** on $x$:
   - Add dependency pointer to $y$.
   - Increment a **reference counter** in $x$ for each dependence; decrement it when $y$ is flushed to disk.
   - Flush pages only when their reference counter is 0.

2. Alternatively: For each page maintain a **list of pages** that have to be flushed first.
   - Add pointer $x \rightarrow y$ to $x$.
   - When $x$ is chosen for replacement, flush all referenced pages first.

For **append-heavy indexes**, write ordering can lead to “write convoys”.
Checkpoints

- 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.
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.
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).
What locks have to be acquired during a transaction rollback?

- In strict two-phase locking, all locks are kept until the transaction commits.
  - → Locks are still held, no new locks have to be acquired during rollback.

- This also means that a transaction cannot run into a deadlock situation during rollback.
And what about locking after a crash?

- Concurrency issues have already been resolved when the transactions were normally running.
  - No need to isolate them again during recovery.

- New transactions, issued after restart, might conflict with those of the recovery process.
  - If new transactions are allowed to enter the system during the recovery process, locks must be acquired (for old and new transactions).
  - Since “old” transactions don’t conflict with each other, they can all run under the same recovery transaction.
The **log analysis pass** helps fast recovery/early restart of new transactions.

- With the analysis pass, determine which locks have to be acquired for recovery.
  - Note that log analysis runs relatively fast, because it does only a **sequential read** of the log.
  - Analysis runs faster if **checkpoints** are done more often.
  - To acquire locks, list locks held by **indoubt transactions** in **checkpoint information**.
  - Locks **cannot conflict** at this stage (→ speed-up analysis)
Once all locks are acquired, **new transactions** can be allowed into the system.

→ Locks for recovery need not (necessarily) be acquired at the **same granularity** as the original transactions did. (Again, this might help speed-up the analysis pass.)

The actual redo/undo takes much longer than log analysis.

→ Many **data pages** have to be fetched in **random order**.

Effectively, many pages will be read **unnecessarily**.

→ Often, the disk will already contain the **latest version** of the data. But we cannot tell that in advance just from analyzing the log.

→ Possible improvement: Log **write-back** of pages, so analysis pass can detect the situation and avoid unnecessarily page reads.
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.