Relational Database Systems 2
10. Transaction Processing 2

Christoph Lofi
Benjamin Köhncke
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
Basic database transactions
The ACID principle
Transaction schedules
Conflict serializability
Locking schedulers
10.1 Transactions

Data accesses that are performed concurrently can potentially lead to database problems.

Concurrency control is a major issue in database implementation and enforced by the transaction manager.
10.1 Transactions

• Scheduling tries to **arrange operations** in concurrent transactions such that the **ACID principle** is maintained
  – **Serializability** means equivalence to some serial schedule, but cannot be efficiently tested
  – **Conflict serializability** means having the same effect and conflicts like a serial schedule and can be tested in polynomial time
    • Testing the conflict graph for cycles
  – **Two phase locking protocols** generate legal schedules which are conflict serializable
    • Also avoid dirty read problems
10 Transaction Processing

10.1 Locking schedulers
10.2 Altruistic locking
10.3 Predicate-oriented locking
10.4 Non-locking schedulers
10.5 Implementation details
10.6 Isolation levels
10.1 Lock Modes

- For conflict-free data access, there are **two types** of locks enforced by the DBMS
  - Read locks can be **shared** by several transactions
  - Write locks are **exclusive** locks

- **Compatibility** of locks
  - Remember: serializability conflicts always include at least one write operation

<table>
<thead>
<tr>
<th>Lock held</th>
<th>Lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>read lock</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>no</td>
</tr>
<tr>
<td>write lock</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>no</td>
</tr>
</tbody>
</table>
10.1 Deadlocks

- Locking protocols also introduce new problems
- Imagine following schedule within a 2PL scheduler
  \[ r_1(x) \rightarrow w_2(y) \rightarrow w_2(x) \rightarrow w_1(y) \]
- Results into following situation: **DEADLOCK**
10.1 Deadlocks

- A deadlock happens when transactions **mutually wait** to obtain locks from each other.
- Other scenario:
  - **Deadlock by Lock Conversation in 2PL**
  - \( r_1(x) \) \( r_2(x) \) \( w_1(x) \) \( w_2(x) \)
10.1 Deadlocks

• Why do deadlocks happen? Four criteria
  – **Mutual exclusion** (locking)
    • Resources can not be shared
    • Requests are delayed until resource is released
  – **Hold-and-wait**
    • Thread holds one resource while waits for another
  – **No preemption**
    • Resources are only released voluntarily after completion
  – **Circular wait**
    • Circular dependencies exist in “waits-for” graph
  – All conditions need to be fulfilled for a deadlock to happen
10.1 Deadlocks

• How to deal with deadlocks
  – **Ignore**
    • Easiest, but may stop the system
  – **Deadlock Detection**
    • Allow deadlocks, detect them, and then resolve them
  – **Deadlock Prevention**
    • Prevent that deadlocks can happen
    • Ensure that at least one of the 4 criteria is not fulfilled
  – **Deadlock Avoidance**
    • Prevent that deadlocks can happen
    • Use additional information about the request to dynamically prevent unsafe situations
10.1 Ignore Deadlocks

- **Solution 1**: Ignoring deadlocks
  - So called “Ostrich Algorithm”
  - Reasonable when deadlocks occur only *rarely* and are *expensive* to prevent or resolve

- Commonly used within threads in operating systems (i.e. Windows, Unix, …)
  - Not a good idea for critical database systems…..
10.1 Deadlock Detection

• **Solution 2: Deadlock Detection and Resolution**
  – If deadlocks occur, they need to be detected and resolved

• **Detection technique: Waiting-For Graphs (WFG)**
  – Every time a transaction waits for another, denote this fact in a waiting graph
    • **Vertices:** transactions
    • **Edges:** “waiting for”-relation
  – A deadlock occurs if there is a **cycle** within the waiting graph
10.1 Deadlock Detection

• Cycle Detection is within $O(n^2)$
  – e.g. Floyd–Warshall algorithm (which is $O(n^3)$)

• When to test for cycles?
  – **Continuously**: Check immediately whenever a transaction has to wait
    • Might be more expensive
    • Smaller freedom in choosing deadlock resolution
  – **Periodic**: Check periodically within a given time cycle
    • Determining the correct time interval is critical for approaches performance
10.1 Deadlock Resolving

• Resolving a usually involves **aborting** at least one transaction
  
  – Which one?

![Diagram showing the concept of deadlock resolving with transactions t1, t2, t3, t4, t5, and t6 with arrows indicating dependencies and two abort states: abort t2 and abort t1.]
Victim Selection

- **Last blocked**: Abort the last transaction which created a cycle
- **Random**: Just abort any random transaction
- **Youngest**: Abort the transaction which started most recently
  - Aims for minimizing wasted work
- **Minimum locks**: Abort transaction with fewest locks
  - Aims for minimizing wasted work
10.1 Deadlock Resolving

- **Minimum work**: Abort transaction which did the least amount of work (CPU, I/O, etc)
  - Aims for minimizing wasted work

- **Most cycles**: Abort transaction breaking the largest number of cycles

- **Most edges**: Abort the transaction with most edges
10.1 Livelocks

- Resolving deadlocks by eliminating a transaction poses a new danger
  - Livelocks (Starvation)
  - Imagine two persons in a narrow floor sidestepping each other forever…

- Livelock within transactions
  - A single transaction is chosen repeatedly as deadlock resolution victim and thus will never finish
  - Resolve by ensuring that the same transaction is not always the victim
    - Introduce priorities
    - Increase priority of victim transactions
10.1 Livelocks

- Livelocks can also occur isolated from deadlocks
  - Transaction $t_1$ and $t_2$ wait for a lock on $x$
    - Lock is freed and granted to $t_2$
  - $t_3$ enters and also waits for a lock on $x$
    - Lock is freed and granted to $t_3$
  - $t_4$ enters and waits for a lock on $x$
    - ...
  - $t_1$ starves and never finishes
10.1 Deadlock Prevention

• **Solution 3: Deadlock Prevention**
  – There are several techniques for deadlock prevention
    • **Wait and die**
      – Transaction can only be blocked by younger transactions
    • **Wound and wait**
      – Transactions can only be blocked by older ones and can kill conflicting younger transactions
    • **Immediate restart**
      – Restart a conflicting transaction immediately to avoid conflict
### 10.1 Deadlock Prevention

#### Running Priority

- If there is a new conflict with an already waiting transaction, abort the waiting transaction and transfer locks to the new one.
- Blocked transaction may not hinder running transactions.

#### Timeouts

- Use timers to abort transactions which are probably involved in a deadlock.
10.1 Deadlock Prevention

• **Wait-and-Die**
  – *Use timestamps per transaction*
    • monotonically increasing number
    • unique
    • priority of a transaction is the inverse of its timestamp:
    • older transaction ⇒ higher priority

• **Scenario:** \( t_i \) requests a lock on which \( t_j \) has a conflicting lock
  – \textbf{If} \( \text{ts}(t_i) < \text{ts}(t_j) \) 
    // true when \( t_i \) is older
    • \textbf{then} \( t_i \) waits
    • \textbf{else} abort \( t_i \) 
      // \( t_i \) dies

• Terms wound, wait, and die are used from \( t_i \)’s viewpoint
• Transactions can only be blocked by \textbf{younger} ones
10.1 Deadlock Prevention

• **Wound-and-Die**

• Scenario: $t_i$ requests a lock on which $t_j$ has a conflicting lock

  – **If** $ts(t_i) < ts(t_j)$  
    // true when $t_i$ is older
  
    • **then** abort $t_j$  
    // ”$t_i$ wounds $t_j””
  
    • **else** $t_i$ waits

• Transactions can only be blocked by **older** ones

• Younger ones can be killed
10.1 Deadlock Prevention

• **Timeouts**
  
  – Each transaction starts a timer as soon as they are blocked
  
  – When the timer **times out**, the system **assumes** a deadlock and terminates the transaction
    
    • Assumption may very well be wrong
  
  – Easy to implement and check
  
  – Time-out threshold crucial for effective performance
• **Solution 4: Deadlock Avoidance**

  – Deadlock avoidance usually involves simulation and trajectories
    
    • Systems tries to avoid “unsafe states”
    
    • If potential of a unsafe state is detected, the schedule is changed
    
    • Example: Dijkstra‘s **Banker’s Algorithm**
      – Check liquidity constraints before scheduling…

  – Usually too expensive and rarely used
10.1 Deadlock Avoidance

• Banker’s Algorithm in short
  – **Safe State:**
    • There is no deadlock
    • There is a scheduling order in which every process can complete even if they request all their locks immediately
10.2 Altruistic Locking

• **Two-phase locking** is a common locking protocol
  – 2PL means that for each transaction all necessary locks are acquired before the first lock is released
  – **Disadvantage:**
    • Imagine a long-running transaction requiring many short-lived locks…
    • Too many locks are held unnecessarily
  – **One Solution:**
    • **Altruistic Locking**
    • Transactions willingly return locks if they do not need them anymore
10.2 Altruistic Locking

• Problem Example with 2PL:
  
  – $t_1: w(a)w(b)w(c)w(d)w(e)w(f)w(g)$
    • Long-running transaction
  
  – $t_2$: $w(a)w(b)$
  – $t_3$: $w(c)w(e)$
  – $t_4$: $w(f)w(g)$
  
  – $t_2 - t_4$ enter when $t_1$ currently accesses $d$

  – In the following we will abbreviate (read/write) locking and unlocking operations by $(r/w)l(x)$ and $(r/w)u(x)$
• \( t_2 \) and \( t_3 \) cannot be executed until the end of \( t_1 \)
• Only \( t_4 \) can be executed in parallel
10.2 Altruistic Locking

- **Idea:** Allow short transactions needing only a subset of items of a long-running transaction obtaining locks
  - Transactions inform the scheduler when they do not need a lock anymore and may **donate** it
  - A donated item may be locked by another transaction
  - Otherwise, 2PL does not change
    - Donating does not count as unlocking, thus additional locks may be acquired
    - Special rules apply to donated items
10.2 Altruistic Locking

• Terminology:
  – **Being in the wake**
    • An *operation* in the wake of a transaction \( t_i \) when it uses an item donated by \( t_i \)
    • A *transaction* \( t_j \) is in the wake of a transaction \( t_i \) when one of it’s operations is in the wake of \( t_i \)
    • A *transaction* \( t_j \) is *completely* in the wake of a transaction \( t_i \) when *all* it’s operations are in the wake of \( t_i \)
  – **Being indebted**
    • A transaction \( t_j \) is *indebted* to \( t_i \) if it obtains a donated lock of \( t_i \) and there is either a *conflict* between those two or a third transaction on that item
10.2 Altruistic Locking

• Rollback Policies
  – When a transaction has to roll back, all transactions in its wake also have to roll back
    • So called cascading roll-backs
  – Cascading roll-backs may be very expensive for large transactions
10.2 Altruistic Locking

- All transactions may be executed in parallel.
10.2 Altruistic Locking

• Rules:

1. When a transaction donates a lock on a data item, it may **not access** that item again

2. The transaction originally putting the lock remains responsible for **unlocking**

3. Transactions may not hold **conflicting locks simultaneously** unless the respective data items were donated

4. If a transaction $t_j$ is **indebted** to $t_i$, it must remain **completely in wake** of $t_i$ until the unlocking phase
10.2 Altruistic Locking

• What happens if rule 4 is violated?

– \( rl_1(a) \ r_1(a) \ d_1(a) \) \( \rightarrow \) \( w_3(a) \ w_3(a) \ wu_3(a) \) \( \rightarrow \) \( rl_2(a) \ r_2(a) \) \( \rightarrow \) \( wl_2(b) \)

– \( ru_2(a) \ w_2(b) \ wu_2(b) \) \( \rightarrow \) \( rl_1(b) \ r_1(b) \) \( \rightarrow \) \( ru_1(a) \) \( \rightarrow \) \( ru_1(b) \)

– Plan not conflict serializable!

– Potential lost updates after

• Correctly: Green stays in the wake of red

– \( rl_1(a) \ r_1(a) \ d_1(a) \) \( \rightarrow \) \( w_3(a) \ w_3(a) \ wu_3(a) \) \( \rightarrow \) \( rl_2(a) \ r_2(a) \)

– \( rl_1(b) \ r_1(b) \) \( \rightarrow \) \( ru_1(a) \) \( \rightarrow \) \( ru_1(b) \) \( \rightarrow \) \( wl_2(b) \) \( \rightarrow \) \( ru_2(a) \ w_2(b) \) \( \rightarrow \) \( wu_2(b) \)
10.2 Altruistic Locking

- AL is not the same than unlocking directly after an item is not used anymore
  - Stay-In-Wake rule ensures consistency
- 2PL schedules are a proper subset of AL schedules
- AL schedules are a proper subset of conflict serializable schedules
Now, how implement locking in a relational database?

– Usually data operations are triggered by SQL commands

– Example: SELECT name, position FROM TABLE employee WHERE salary < 25000

• Semantic entities have to be locked for reading
  – either the entire tablespace of ‘employee’
  – or only those records addressed by the WHERE condition

• Basically all SQL statements can be transformed into read/write statements, but what happens to INSERT/DELETE?
• If only individual records are locked the **phantom problem** may occur
  
  – **Transaction 1**: (fire and hire)
    DELETE FROM employee WHERE position = ‘Manager’
    INSERT INTO employee VALUES (‘Smith’, ‘Manager’, 50000)
  
  – **Transaction 2**: (find the manager)
    SELECT FROM employee WHERE position = ‘Manager’
  
• **Locking**
  
  – Locks on the **entire tablespace** yield correct results, but heavily restrict concurrency
  
  – Otherwise transaction 1 might lock a **single tuple** (the old manager), delete it, return the lock, then insert a new manager
  
  – If transaction 2 is **interleaved** with transaction 1, it might not read any manager?!
10.3 Predicate Locking

• Locking semantic entities from SQL statements leads to **predicate-oriented concurrency control**
  
  – **Basic idea:** Do not lock the entire table space, but lock a **subset of the database** referring to a **descriptive predicate** independently of the current table contents

  • Locking of **intensional** entities instead of extensional entities

  – Usually the predicates needed for locking are given by the **WHERE condition** of a query, update, delete statement

    • (conjunctions of) attribute \( \theta \) value, with \( \theta \in \{<, \leq, =, \geq, >\} \)
    
    • e.g., … WHERE position = ‘Manager’ AND salary < 25000
• The conditions on predicates define a **hyperplane** in the vectorspace spanned by the attributes

  – For a set of conditions $C$ and attributes $A_1,…,A_n$

    $$H(C) := \{x \in \text{dom}(A_1) \times \ldots \times \text{dom}(A_n) \mid x \text{ satisfies } C\}$$

    • The hyperplane may be a superset or subset of the respective relation, they might have a non-empty intersection, or may even be disjoint

  – The phantom example showed that locking only the intersection is **not a good idea**

    • Locked records may vanish or new records may be added
10.3 Predicate Locking

• Every **update** or **retrieval** operation of a transaction comes with
  – A (set of) predicate condition(s) specifying **which records to lock**
  – A **lock mode** stating whether the lock is shared or exclusive

• Two lock requests are **compatible**, if
  – Both request a **shared** lock mode
  – Or the intersection of their two respective hyperplanes is **empty**
10.3 Predicate Locking

• When a new lock request arrives the scheduler performs a **compatibility test**
  – The lock is granted, only if the test does not show a conflict, i.e. only **disjoint** predicates are locked in conflicting mode
  – When a transaction commits the predicate locks are released and blocked transactions are resumed

• The testing is, however, **far more expensive** as in the case of discrete (individually named) items
  – Actually the test is again an NP-complete problem
  – Predicate locking is currently not supported by commercial systems
10.3 Predicate Locking

- Avoiding the complex satisfiability test can be done with so-called **precision locking**
  - The scheduler grants **all lock requests** right away
  - Once a transaction reads/writes a record, for **this specific record** the scheduler tests, whether it lies on one of the locked hyperplanes
  - The **read/write is rejected**, if the record intersects some hyperplane and the lock modes are in conflict

- This avoids the NP-hard testing, but still is not really a lightweight protocol...
10.4 Non-Locking Schedulers

- Transactions can also be serialized without locking
  - Timestamp ordering
  - Serialization graph testing
  - Optimistic protocols

- Non-Locking schedulers tend to **abort transactions** often and are thus less efficient
  - Rarely used in commercial applications
  - However, very suitable for **distributed systems** due to difficulty in distributed lock handling
10.4 Timestamp Ordering

• **Timestamp ordering**
  – Each transaction is annotated with a unique and monotonically increasing **time-stamp**
  – Every operation of the transaction inherits the time-stamp
  – Conflicting operations are ordered by their timestamps

• **Rule:** For two conflicting operations $p_i(x)$ and $q_j(x)$
  – $p_i(x)$ is executed before $q_j(x)$ iff $ts(t_i) < ts(t_j)$
    • smaller timestamp $\Leftrightarrow$ older transaction
10.4 Timestamp Ordering

• Operations are directly submitted to the data manager in their natural order unless they are too late
  – $p_i(x)$ is too late if there had already been a $q_j(x)$
    $ts(t_i) < ts(t_j)$
• If an operation $p_i(x)$ is too late, the whole transaction $t_i$ is aborted
  – Pessimistic approach – works only well if there are few conflicts
  – Performance rapidly decreases with number of latecomers
• To detect latecomers, two timestamps are needed for each data item $x$
  
  – $\text{max-read-scheduled}(x)$: Latest timestamp of already executed read operation on that item
  
  – $\text{max-write-scheduled}(x)$: Latest timestamp of already executed write operation on that item

• When a $p_i(x)$ arrives, its timestamp is compared to the respective max-schedule timestamp
  
  – If $ts(t_i) < \text{max-scheduled}$, $p_i$ is too late and the transaction is aborted
• **Example:** old transactions with conflicts are aborted

```
read₁(x)  
  t₁

write₂(x)  
  t₂

write₂(y)  
  abort

read₃(y)  
  t₃

write₃(z)  
  commit
```

```
10.4 Serialization Graphs

• **Serialization Graph Testing**
  
  – **Idea:** Dynamically maintain **Conflict Graphs** (i.e. wait-for-graph) and **check it for cycles**
  
  – Extend graph proactively before actually performing any operations
    
    • If extension would result to a **cycle**, **deny** the responsible transaction
    
    • **Otherwise, accept** transaction and execute transaction in its natural order
    
    • Implements a **deadlock avoidance** scheme
• Serialization graph testing is nice and simple from a theoretical point of view
  – However, impractical in real applications
  – Space is in $O(\#\text{transaction}^2)$
    • Also including inactive transaction as long as they are part of conflict graph
  – Continuous cycle detection computationally very expensive
    • Needs to be performed very often
10.4 Optimistic Protocols

• Assumption up to now: **Transactional Environment**
  – Conflicts occur often
  – Coping with conflicts is important
  – It is important to immediately detect and resolve conflicts

➢ **Pessimistic approach**
• Consider an online-shop
  – 99% of all transactions just read product data and descriptions
    • No conflict potential
  – Only very rarely, some prices are updated
    • Conflict potential

• Assumption: “Probably no conflict will happen anyway”
  – Optimistic Approach
  – A full-fledged locking protocol like 2PL would be a waste…
  – Only when an accident happens, actions are taken
• Basic Idea: **Three phases**

1. **Read Phase:** Just execute the transaction without any additional checks.
   - Use a private isolated copy of the data for write operations
   - All reads can be done directly on the DB or the private copy

2. **Validate Phase:** When a transaction is ready to commit, validate if execution was correct regarding conflict serialization
   - If not, abort transaction and delete copy

3. **Write Phase:** Write the private copy back into the DB
• For simplification, assume that validate and write are executed undividable and uninterruptable
  – All other transactions may not interfere or are even suspended
  – Difficult in real implementations when a large amount of data was modified

• In general, there are two validation protocols
  – Backward-/forward- oriented optimistic concurrency control
10.4 Optimistic Protocols

- **Backward-oriented optimistic concurrency control**
  - Conflict test against all already committed transactions
  - Validated *positively*, if
    - All previous transaction *finished before* the current one started
    - *Or read set* of current transaction does *not intersect* with *write set* of all previous interactions

- **Forward-oriented optimistic concurrency control**
  - Conflict test against all parallel transactions still in read phase
  - Validated *positively*, if
    - *Write set* of current transaction does *not intersect* with the *up-to-now read sets* of all parallel transactions
• **Overview** over protocols for concurrency control

**pessimistic**
- **locking**
  - two-phase locking
    - 2 PL
      - strict 2PL
      - conservative 2PL
    - altruistic locking
    - predicate/precision locking

**optimistic**
- **non-locking**
  - timestamp ordering
  - serialization
  - graph testing
- backward-/forward-oriented optimistic concurrency control
• The concurrency protocols in the overview are the most common ideas
  – Of course, there exists a large variety of special or hybrid algorithms, but the ideas are fairly similar
  – In today’s commercial systems strict two-phase locking is the prevalent protocol
    • Main reason is its versatility and robustness
    • Implementing strict 2PL needs some considerations about the efficiency
A lock manager needs some efficient data structures for bookkeeping:

- Whenever a lock is requested, checking whether another transaction holds a lock is necessary.
- Whenever a transaction terminates all its locks should be released at once.
- Once a lock is released, all transactions with a respective lock conflict should be resumed.
• **Checking for conflicts, releasing and resuming**
  
  – Needs an in memory search structure for locks
  
  – **Single key hash tables** are a good choice
    
    • All locking protocols have to map whatever abstract resources they need to lock (e.g. key ranges in a table) to concrete resources (e.g. a set of individual keys)
    
    • Entries in the hash table are pointers to **resource control blocks (RCBs)**
    
    • RCBs can represent pages, records, index entries,…
    
    • Hash conflict are resolved by linking all RCBs with the same value in a chain anchored to the respective hash table entry
– For **shared locks** multiple locks can be held simultaneously and other lock requests may still wait

- There is either a sequence of granted read locks or a single granted exclusive lock (in either case followed by a sequence of waiting read and write locks)
- Bookkeeping amounts to managing a queue of **lock control blocks (LCBs)** containing granted and requested locks attached to some RCB
- Ordering in the list reflects the arrival time (and the sequence in which waiting lock requests should be resumed)
- To avoid starvation shared locks should not be allowed to pass previously queued exclusive locks
10.5 Transaction Control Blocks

- For releasing all locks of a transaction all LCBs that belong to the same transaction have to be identified

  • A transaction control block (TCB) can be maintained for each active transaction

  • Upon commit or abort simply the respective list of LCBs is traversed and the LCBs are removed from the queue of the corresponding RCB

  • The removal initiates a check whether the following LCBs in the chain can be resumed
10.5 Data Structures

Hash Table indexed by resource ID

Transaction Control Block

Resource Control Block

Transaction ID
Transaction Status
Number of Locks
LCBs

Lock Control Blocks

Transaction ID
Resource ID
Lock Mode
Lock Status
LCB queue

Transaction ID
Resource ID
Lock Mode
Lock Status
LCB queue

Transaction ID
Resource ID
Lock Mode
Lock Status
LCB queue
• In large systems the lock manager ends up with a large number of RCBs, TCB, and LCBs
  – The **lock granularity** influences the number of RCBs and LCBs
    • The smaller the granularity the more bookkeeping overhead, but the less conflicts
    • Especially long running transactions might acquire a lot of locks, if a small lock granularity is chosen

• **Multiple granularity locking** allows to assign different granularities to different transactions
• But when locking with multiple granularities, how can conflicts be detected?
  – **Example:** A transaction requests a lock for an entire tablespace. Are pages of that table space already locked?
  – A common solution are **intention locks**
    • They are set on coarser granularities of the hierarchy by transactions acquiring a fine-grained lock
    • Any transaction can now easily check whether there is a read/write lock of finer granularity
    • Before a lock can be granted, the transaction has to hold intention locks on all coarser granularities
– If a transactions does not know a-priori how many locks it will need (and therefore what is the best granularity), it can try lock escalation

  • Many fine grained locks are converted into a single coarse grained lock
  • First the coarse grained lock has to be granted, then the fine grained locks can be released
  • Commercial systems trigger lock escalation, if a certain threshold for the lock manager’s memory is reached
(Strict) two-phase locking is widely used in commercial databases

- It is simple, versatile, and robust

Depending on the application, a certain relaxation may be helpful

- Concurrent read e.g. for statistical analysis may not need full-fledged serializability

- Application programmers may therefore decide to lock items manually, and unlock them whenever the context is considered safe enough
  
  - Still, this might result in inconsistencies

10.6 Isolation Levels
10.6 Isolation Levels

• Manually controlling locks on a per application level (or even per transaction) is made safer in commercial systems by allowing only a limited number of locking style options
  – So-called isolation levels
  – Controlled deviations from strict two-phase locking (S2PL)
  – Part of the SQL standard: “SET ISOLATION LEVEL ...“
• **Read uncommitted** (dirty-read or browse-level)
  – All *write locks* are acquired and released according to S2PL, especially they are held until the commit point

• **Read committed** (cursor stability level)
  – All *write locks* are acquired and released following S2PL
  – All *read locks* are held at least for the duration of each data server operation

• **Serializable** (conflict-serializable)
  – All *locks* are generated by S2PL
Since **read uncommitted levels** can cause database inconsistencies, they should only be used for mere **browsing or statistical evaluations**

- A consistent view of data is not always required

**Example:**

- A bookstore computes the **average amount for purchases** of the week
  
  - For serializability the **entire purchases table** would need to be locked and nobody could buy a book during that time
  
  - Even if a **couple of purchases** are made during the statistical evaluation, how will it affect the average amount?
10.6 Isolation Levels

• Difference between isolation levels is the lock duration for shared and exclusive locks
  – Read uncommitted does not require any read locks
    • All (even uncommitted) changes are immediately visible
    • But since data may be read in a dirty fashion, later writes may cause inconsistencies
  – Read committed eliminates dirty read problems, but may still face problems of lost updates
    • Especially useful for long read and short update transactions
    • But waiting periods between read and writes may lose updates,
      e.g. \( r_1(x) \) \( r_2(x) \) \( w_2(x) \) \( \text{commit}_2 \) \( w_1(x) \) \( \text{commit}_1 \)
10.6 Isolation Levels

– **Serializable** severely restricts concurrency, but is safe
  
  • **Pessimistic concurrency control** guarantees that no data inconsistencies can occur

– A slightly relaxed form of serializability tolerating the phantom problem, but no other type of inconsistency, is often referred to as **repeatable read**
  
  • Repeated read operation on the same record within a transaction will **always** result in the same value
  
  • Supported by most commercial database systems
In any case, **long running transactions** should be avoided as much as possible

- Especially transaction involving dialogs with the user (user I/O) are performance killers

- Application semantics often allows to divide long transactions into shorter steps (**transaction chopping**)

  - **Example:** booking a flight can be divided into a read transaction (what flights are available?), and a later write transaction for the actual reservation once the user is decided.
• Allowing more concurrency for transactions increases the system’s **throughput**
  – All transaction **compete** for access to (and locks on) data records (often called **data contention**)

• However, when data contention becomes **too high**, performance disasters can occur
  – **Data contention thrashing**
  – The probability of locks not granted (and thus waiting times until conflicts are resolved) **increases superlinearly**
• If too many transactions run concurrently…
  – **Most** transactions may be blocked because of locking conflicts
  – Deadlocks pose an additional threat adding **CPU and disk I/O contention**, if too many transactions have to be restarted

• Databases have an adjustable **multiprogramming level** (MPL) stating the maximum number of transactions allowed to run concurrently
  – Once the limit is reached, all new transaction are held in a **transaction admission queue** (usually FIFO)
10.6 MPL

- MPL should be set as high as possible, but sufficiently low to avoid thrashing
  - When thrashing occurs transaction throughput drops sharply
• **Tuning** the MPL is a difficult task and depends on the *workload*
  
  – Short and frequently arriving transactions with high degree of shared locks → **high MPL possible**
  
  – Long transactions holding locks for extended time spans → **low MPL needed**
  
  – Access patterns of transactions are highly skewed towards some records (hot spots) → **low MPL needed**
Variability of transaction length in mixed workloads influences MPLs

- Rather constant transaction lengths allows high MPLs
- Highly varying transaction lengths (even with the same mean) need low MPLs

Sometimes different MPLs can even be defined per transaction class

- Example: A bookstore allows many concurrent payment transactions, but only few concurrent market analysis transactions
10.6 Feedback-driven MPL

• The MLP can also be **dynamically adapted** according to the current workload
  – **Conflict-ratio** reflects the current degree of data contention
    • As long as the conflict ration is below a critical threshold, all transactions are **immediately admitted** for execution
    • If the conflict ratio surpasses the threshold, newly arriving transactions are queued in an **admission control** queue → the current MPL is enforced
    • If the conflict ratio still increases, **cancellation control** can be used, where constantly some active transactions are forced to abort and (usually after a short waiting period) are restarted → this lowers the current MPL
• Good measures for the conflict ratio are
  – The fraction of blocked transactions
    • Usually good for homogeneous workloads
    • Critical value around 25%; over 30% the system is thrasing
  – For inhomogeneous workloads transactions with few locks rarely cause thrashing, hence even if they block others they should be weighted down
    • Better measure implicit weighting blocked transactions:
      \[
      \frac{\text{# of locks currently held by all transactions}}{\text{# of locks currently held by all non-blocked transactions}}
      \]
Locking schedulers
Altruistic locking
Predicate-oriented locking
Non-locking schedulers
Implementation details
Isolation levels
Outlook: Recovery

Introduction to recovery
Basic concepts
Recovery without logs
Recovery with logs
Catastrophic recovery
Application recovery