Relational Database Systems 2
9. Transaction Processing

Christoph Lofi
Benjamin Köhncke
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
Basic join order optimization
Join cost and size estimations
Left-deep join trees
Dynamic programming
Greedy strategy
Randomized algorithms
9.1 Basic database transactions
9.2 The ACID principle
9.3 Transaction schedules
9.4 Conflict serializability
9.5 Locking schedulers
9.1 Introduction to Transactions

• Depending on the application, certain database interactions belong to the same business workflow
  – Queries (read operations)
  – Updates, deletes, inserts (write operations)

• Typical workflow examples
  – Money transfers in banking
  – Travel booking for vacations
  – …

• Database operations in workflows are often intertwined
Automatic teller machines (ATM)

- **User Interaction**
  - Insert your card and input PIN code
  - Select amount
  - Take card and cash

- **Basic business workflow**
  - Authenticate user
  - Ask for requested amount
  - **Query** for available balance (**read operation**): if balance is too low shred card and abort…
  - Else **deduct** amount from balance (**write operation**)
  - Return card and dispense cash
9.1 Workflow Example

- **Travel agency**
  - **User interaction**
    - “I want to go on vacations to Hawaii in the first week of May”
  - **Basic business workflow**
    - Check for flight availability during the week (**read operation**)
    - Check for hotel accommodation availability during the week (**read operation**)
    - Align dates for flights and hotels, shift it around a little for best prices
    - Reserve suitable room from hotel (**write operation**)
    - Buy flight ticket from airline (**write operation**)
9.1 Problems...

• Still, while processing workflows severe problems can occur
  – Even if we assume that individual workflows are always sensible and correct

• Examples
  – What if the ATM catches fire after withdrawing your money, but before dispensing it..?!  
  – What if you found the perfect flight and hotel, but when trying to buy the ticket, somebody else has already bought the last ticket in the meantime..?!
For avoiding these problems we need the concept of **transactions**

- A transaction is a **finite set of operations** (workflow, program) that has to be performed in a certain **order**, while ensuring certain **properties**

The properties are concerned with

- **Integrity**: transactions can always be executed safely, especially in concurrent manner, while ensuring data integrity
- **Fail Safety**: transactions are immune to system failures
9.1 Transactions

- **Application** programs should not be bothered with data integrity and protection issues
  - DBMS holds responsibility for ensuring the well-being of data
  - Transactions are an **interface contract** of an transaction-enabled server
    - **Start**: Starts an transaction, followed by a finite sequence of operations of a workflow or program
    - **Commit**: Executes all operations since transaction begin and ends the transaction
    - **Rollback**: Cancels the current transaction and reverts all effects of the transaction operations
9.1 Transactions - Example

• **Example:** Money Transfer
  – Assume system crashes during a transaction…

```plaintext
MoneyTransfer(acc1, amount, acc2)
start T1
Read balance of acc1
Deduct amount from acc1
Add amount to acc2
commit T1
```

**Constraints**
- Balance(acc1\_before) + Balance(acc2\_before) = Balance(acc1\_after) + Balance(acc2\_after)
- Balance(acc1\_after) ≥ 0
9.1 Applications

• Transactions in Applications

• Two interesting scenarios
  – Transaction within one data source
    • DBMS is responsible for managing transactions
    • Applications may connect directly to DBMS
    • Topic of this lecture
  – Transactions spanning several data sources
    • Distributed transactions
    • Additional transaction manager necessary abstracting from the data sources (e.g. several DBMS)
    • Topic of lecture “Distributed Data Management”
• Transaction Management within one DBMS
  – Applications connect directly to the DBMS
    • Using JDBC (Java DataBase Connectivity)
      – Java-API for abstracting SQL communications to DBMS
    • Using ODBC (Open DataBase Connectivity)
      – Platform and language independent API for abstracting SQL communication to DBMS
  – DBMS responsible for managing transactions
9.1 Applications

**Application Layer**

- ATM
- travel agency

**Applications Clients**

- bookkeeper

**DBMS Layer**

- Encapsulated data
- exposed data

- TransactionManager

- DB pages
9.1 Applications

• Example: **JDBC**
  
  – **Java Database Connectivity**
  
  – **JDBC Driver** provided by DBMS Vendor
  
  – **Driver Manager** provides driver to application
  
  – Transactions are completely handled by DBMS
9.1 Applications

- **Example: JDBC-Transactions**

```java
Connection conn = DriverManager.getConnection("jdbc://databaseUrl");
conn.setAutoCommit(false);

PreparedStatement updateBalance = conn.prepareStatement("UPDATE accounts SET balance = ? WHERE accNo=?");

updateBalance.setDouble(1, oldBalance1 - 50);
updateBalance.setInt(2, accNo1);
updateBalance.executeUpdate();

updateBalance.setDouble(1, oldBalance2 + 50);
updateBalance.setInt(2, accNo2);
updateBalance.executeUpdate();

conn.commit();
```
9.1 Applications

- **Enterprise applications** usually involve multiple data sources
  - Transaction may also span **multiple** data sources
    - e.g. book a flight within one system and an hotel in another
  - Need for **Distributed Transaction Processing (DTP)**
    - Additional coordination layer necessary, i.e. **transaction manager**
      - Usually provided by an **application server**
    - All participating databases need a common interface for coordinating transactions
      - e.g. XOpen **XA**
9.1 Applications

Applications Clients

ATM
travel agency
... bookkeeper

Application Layer

app_1
... app_n

Transaction Manager

Application Management Layer

DBMS Layer

Encapsulated data
exposed data

DBMS
DB pages

Transaction Manager

Encapsulated data
exposed data

DBMS
DB pages

Transaction Manager

Relational Database Systems 2 – Christoph Lofi - Benjamin Köhncke – Institut für Informationssysteme

17
9.1 Applications

• **Example: JTA**
• **Java Transaction API**
• **Uses Application Server**
  – e.g. **J2EE Server**
  – Provides centralized **Transaction Manager**
    • Provided by AppServer
  – **User Transaction** interface for applications
  – XOpen XA Adapter connecting to databases
9.1 Applications

• **J2EE Application Servers** with JTA Transaction Manager Implementations
  - JBoss
  - Apache Geronimo
  - Sun Glassfish
  - Bea WebLogic Server
  - IBM WASCE
  - Oracle Application Server
  - SAP NetWeaver
  - …
• Open Group XOpen XA
  – Vendor-spanning standard protocol for Distributed Transaction Processing
  – Each DBMS / data source participating within a transaction needs to support XA
  – Uses Distributed 2-Phase Commit
  – Each DBMS is responsible for maintaining integrity of its own data
    • Centralized transaction manager necessary to coordinate individual commits
9.1 Applications

- Example: JTA-Transactions

```java
UserTransaction ut = envCtx.lookup("jta/UserTransaction");
DataSource ds = envCtx.lookup("jdbc/Datasource");
// note: explicit transaction handling necessary!
ut.begin();
boolean success = false;
try {
    Connection conn = ds.getConnection();
    // do stuff here
}
finally {
    if (success)
        ut.commit();
    else
        ut.rollback();
}
```
Database transactions show certain properties, also known as the **ACID principle**

- **Atomicity**
- **Consistency**
- **Isolation**
- **Durability**

Every system handling **non-ACID transactions** has to take special precautions.
9.2 The ACID Principle

• **Atomicity**
  – Any transaction is either executed **completely**, or **not at all**
    • Complete transaction is to be treated as an uninterruptable single operation
    • That means, **all effects** of a transaction have to be materialized in the database once it has been executed
    • The effects will only become visible to other users or transactions **if and when** the transaction is committed
– No Atomicity: **Dirty Read**

- Data changes are only to be propagated after commit

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>start T1</td>
<td></td>
</tr>
<tr>
<td>a1 := read(A)</td>
<td>=1</td>
</tr>
<tr>
<td>a1 = a1 + 1</td>
<td>=2</td>
</tr>
<tr>
<td>write(a1,A)</td>
<td>=2</td>
</tr>
<tr>
<td></td>
<td>start T2</td>
</tr>
<tr>
<td>a2 := read(A)</td>
<td>=2</td>
</tr>
<tr>
<td>a2 := a2 + 1</td>
<td>=3</td>
</tr>
<tr>
<td>write(a2, A)</td>
<td>=3</td>
</tr>
<tr>
<td></td>
<td>commit T2</td>
</tr>
<tr>
<td></td>
<td>rollback T1</td>
</tr>
</tbody>
</table>
9.2 The ACID Principle

• Consistency Preservation
  – Transactions lead from one consistent state of the data instance to another
    • Especially, data constraints are always respected
    • Note that the consistency can be violated during a transaction, but not once the transaction is finished
    • Transactions that cannot reach a consistent state (any more) have to be aborted
No consistency: Inconsistent read problem

- Constraints need to be respected at commit time
- Constraint: $A = B$, at start time: $A = B = 5$
- $T_1$ increases $A$ and $B$, $T_2$ doubles $A$ and $B$,

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>start T1</strong></td>
<td></td>
</tr>
<tr>
<td>$a_1 := \text{read}(A)$</td>
<td>$=5$</td>
</tr>
<tr>
<td>$a_1 := a_1 + 1$</td>
<td>$=6$</td>
</tr>
<tr>
<td>\text{write}(a_1, A)</td>
<td>$=6$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>start T2</strong></td>
</tr>
<tr>
<td>$a_2 := \text{read}(A)$</td>
<td>$=6$</td>
</tr>
<tr>
<td>$a_2 := 2*a_2$</td>
<td>$=12$</td>
</tr>
<tr>
<td>\text{write}(a_2, A)</td>
<td>$=12$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>start T1</strong></td>
<td></td>
</tr>
<tr>
<td>$b_1 := \text{read}(B)$</td>
<td>$=10$</td>
</tr>
<tr>
<td>$b_1 := b_1 + 1$</td>
<td>$=11$</td>
</tr>
<tr>
<td>\text{write}(b_1, B)</td>
<td>$=11$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>commit T1</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>commit T2</strong></td>
</tr>
<tr>
<td>$b_2 := \text{read}(B)$</td>
<td>$=5$</td>
</tr>
<tr>
<td>$b_2 := 2*b_2$</td>
<td>$=10$</td>
</tr>
<tr>
<td>\text{write}(b_2, B)</td>
<td>$=10$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
9.2 The ACID Principle

• **Isolation**
  – Transactions are isolated from others, i.e. even in a concurrent scenario transactions do not interfere with each other
  • The effect of a transaction always has to be the same as if it had been executed on its own
  • Moreover, each transaction will read only consistent data from the data store
### No Isolation: Lost Updates

- Concurrent writes can lead to information loss

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>start T1</strong></td>
<td></td>
</tr>
<tr>
<td>a1 := read(A)</td>
<td>=1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>start T2</strong></td>
<td></td>
</tr>
<tr>
<td>a2 := read(A)</td>
<td>=1</td>
</tr>
<tr>
<td>a2 := a2 - 1</td>
<td>=0</td>
</tr>
<tr>
<td>write(s2, A)</td>
<td>=0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>commit T2</strong></td>
<td></td>
</tr>
<tr>
<td>a1 := a1 + 1</td>
<td>=2</td>
</tr>
<tr>
<td>write(a1, A)</td>
<td>=2</td>
</tr>
<tr>
<td><strong>commit T1</strong></td>
<td></td>
</tr>
</tbody>
</table>
### 9.2 Transactions - Problems

- **No Isolation:** **Non-Reproducible Read** ("Phantom Problem")
  - Transaction should not interfere with each other

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>start T1</strong></td>
<td></td>
</tr>
<tr>
<td>a1 := read(A)</td>
<td>=1</td>
</tr>
<tr>
<td>print(a1)</td>
<td>=1</td>
</tr>
<tr>
<td><strong>start T2</strong></td>
<td></td>
</tr>
<tr>
<td>a2 := read(A)</td>
<td>=1</td>
</tr>
<tr>
<td>a2 := a2 + 1</td>
<td>=2</td>
</tr>
<tr>
<td>write(a2, A)</td>
<td>=2</td>
</tr>
<tr>
<td><strong>Commit T2</strong></td>
<td></td>
</tr>
<tr>
<td>a1 := read(A)</td>
<td>=2</td>
</tr>
<tr>
<td>print(a1)</td>
<td>=2</td>
</tr>
<tr>
<td><strong>Commit T1</strong></td>
<td></td>
</tr>
</tbody>
</table>
9.2 The ACID Principle

• **Durability**
  
  – As soon as the transaction is completed (committed), all performed **data changes** are guaranteed to **survive** subsequent system failures
    • Either the data is permanently written on the disk, or specific means for recovery have been taken
  
  – Incomplete transactions need to **survive** system crashes
    • Transaction continues as soon as system is online again
9.2 Transactions - Problems

– No Durability: **System Crash**
  
  • Not fully completed transactions need to survive system failures
  
  • After a transaction has been committed, everything has to be permanently written to disk or has to be logged for recovery

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>start-TA T1</td>
<td></td>
</tr>
<tr>
<td>a1 := read(A)</td>
<td></td>
</tr>
<tr>
<td>a1 := a1 +1</td>
<td></td>
</tr>
<tr>
<td>write(a1, A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>System Crash</strong></td>
</tr>
<tr>
<td>Commit-TA T1</td>
<td></td>
</tr>
</tbody>
</table>
9.2 Page Model

• We are currently focusing only on the database
  – Exact semantics of transactions from a business perspective do not matter
  – It is only interesting how data is affected
    • i.e. read and write operations
  – Individual operations are usually on page level in the storage
    • Operations performed in buffer
    • Each operation accesses the complete database page with the changed record(s) on it
      Each read and write operation is considered indivisible
• **Definition:** A transaction is a totally-ordered finite sequence of actions of the form $r(x)$ or $w(x)$ where $x$ is some record from the database instance
  
  – $r(x)$ denotes a read operation
  – $w(x)$ denotes a write operation

• **Example:**
  
  – $T := r(x) r(y) r(z) w(u) w(x)$
9.2 Page Model

- Actually, the total order of operations can be relaxed to a partial order, if this does not affect the effect of the transaction
  - e.g., \( r(x) \ r(y) \ w(u) \equiv r(y) \ r(x) \ w(u) \)
  - But in any case: If two operations affect the same data record, an order has to be specified between them
    - Reading a page after a write operation on the same page generally results in different data than reading the page before the write operation
• There are more complex models, but the page model is usually sufficient for illustrating the important concepts

• Example: object model
  – Transactions can be seen as node-labeled trees with
    • Transaction identifier as root
    • Names and parameters of invoked operations as labels of inner nodes
    • Read/write operations in the leaf nodes, together with a partial order as in the page model

T1

deduct

r(x) w(x)
9.3 Transaction Manager

- 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**
9.3 Transaction Manager

• The transaction manager gets a set of transactions from users/applications
  – Derives a correct transaction schedule
  – Caters for recovery in case of system failures
9.3 Schedules

• For a set of **concurrently executed transactions**
  
  – A complete **schedule** (also called **history**) is a sequence of operations that
    
    • Contains all (and only) the operations of the involved transactions
    
    • Respects the (partial) order within each single transaction
  
  – A complete schedule is called **serial**, if it consists of any permutation of the transactions, where each transaction is **fully executed** before the next starts
    
    • i.e. no interleaving of transactions
  
  – Often complete schedules are distinguished from **general schedules**, where some transactions may be still active, i.e. not yet committed or aborted
9.3 Schedules

• Example (without starts and commits)
  - \( T_1 := r(x) \; r(y) \; w(u) \; w(x) \)
  - \( T_2 := r(p) \; r(q) \; w(p) \)
  - \( T_3 := r(z) \; w(z) \)

• Schedule
  - \( S := r(x) \; r(y) \; r(p) \; r(z) \; w(u) \; r(q) \; w(x) \; w(z) \; w(p) \)

• Serial schedule
  - \( S := r(p) \; r(q) \; w(p) \; r(x) \; r(y) \; w(u) \; w(x) \; r(z) \; w(z) \)
9.3 Consistency by Schedules

• Obviously, restricting the system to serial schedules will remove all problems of concurrency control

  – But… that has severe implications on performance

    • Imagine only one person at a time can get money from an ATM of some bank…?!

    • Somebody takes forever deciding for a hotel and blocks all other users from booking…?!
9.3 Schedule Equivalence

• Two complete schedules are called **equivalent**, if
  – They are comprised of the **same set of operations**
  – Every transaction in both schedules **reads the same values** from a given record
  – Eventually, every transaction in both schedule **writes the same values** to a given record

• That means, starting with a certain database instance the final database instance after executing either schedule will be **exactly the same** (final state equivalence)
9.3 Serializability

- A complete schedule is called **serializable**, if it is equivalent to any serial schedule of the respective set of transactions
  - i.e. **has the same effects** as executing the transactions serially, but it still **allows for concurrent execution**
    - More operations can be processed by **minimizing waiting times**
    - Serial schedules ensure atomicity, consistency, and isolation
9.3 Serializability

- **Serial** schedules are a proper subset of **serializable** schedules
9.3 Consistency by Schedules

• Restricting the system to **serializable schedules** will lead to consistent concurrency control

• **Can we test** a schedule for serializability?
  – Obviously, we can test equivalence to any possible serial schedules
    • … but that has complexity $O(#TA!)$
    • And, unfortunately this is also the best way to do it…
    • In fact, the problem can be shown to be **NP-complete**
      – The proof builds on graph theory and is rather complex…
      – Thus, testing schedules for real serializability is not possible in practical systems
9.4 Conflict Serializability

• We need to find safe schedules fast!
  – Define a easier-to-test more restrictive variant of serializability

• Problem with transaction occur when different transactions work on the same data item
9.4 Conflict Serializability

• Two operations in a schedule are **conflicting**, if
  – They access the **same** database record (or page)
  – And at least one of them **writes** data
    • i.e. dirty reads, phantoms, etc. possible

• All pairs of conflicting operations are called the **conflict relation** of a schedule
  – Aborted transactions can be removed from a schedule’s conflict relation
9.4 Conflict Serializability

• Two schedules are called **conflict equivalent**, if
  – They contain the **same** operations
  – And have the **same** conflict relations

• Example for conflict equivalent schedules

  – $S := r(x) \ r(y) \ w(z) \ w(y) \ r(z) \ w(x) \ w(y)$
  – $S' := r(y) \ r(x) \ w(y) \ w(z) \ w(x) \ r(z) \ w(y)$

  – Conflict relations: $\{r(y), w(y)\}, \{w(y), w(y)\}$
9.4 Conflict Serializability

- **Conflict serializable** schedules are a proper subset of all **serializable** schedules.
- **Serial** schedules are a proper subset of **conflict serializable** schedules.
9.4 Conflict Serializability

• A schedule is conflict serializable, if there exists a conflict equivalent serial schedule
  – This is a subset of all serializable schedules

• To test for conflict equivalence, construct a conflict graph
  – Transactions as nodes and conflicts as edges
  – Two schedules are conflict equivalent, if and only if their conflict graphs are identical
9.4 Conflict Serializability

- An example for a **non conflict serializable** schedule is the lost update problem
  
  \[ S := r(x) r(x) w(x) w(x) \]

  - Conflict relations:
    \[ \{(r(x), w(x)), (r(x), w(x)), (w(x), w(x))\} \]

- The two respective serial schedules are

  \[ S_1 := r(x) w(x) r(x) w(x) \]

  with \[ \{(r(x), w(x)), (w(x), r(x)), (w(x), w(x))\} \]

  \[ S_2 := r(x) w(x) r(x) w(x) \]

  with symmetric conflicts
A **conflict graph** of a schedule $S$ consists of

- All committed transactions of $S$ as nodes
- And between any two nodes $t$ and $t'$ there is a directed edge, if the pair of operations $(p, q)$ is in the conflict set of $S$ (with $p \in t$ and $q \in t'$)

**In conflict graphs**...

- If there is an edge between two transactions there is at least one conflict between them
- The direction of each edge respects the ordering of the conflicting steps
9.4 Conflict Graphs

• Example for a conflict serializable schedule
  
  – \( S := r(x) r(x) w(x) r(x) w(x) w(y) w(y) \)
  
  – \( S_s := r(x) w(y) r(x) w(x) w(y) r(x) w(x) \)

  – Conflict graph
• Now, how does the graph of some non-conflict serializable schedule look like?
  – **Intuition**: if an operation conflicts with some other operation, the operation should be performed after the other in a serial schedule
  – $S := r(x) r(x) w(x) w(x)$
  – As we can see the conflict graph contains a **cycle**!
A complete schedule is conflict serializable, if and only if the respective conflict graph is acyclic.

- The proof works by
  - Using the equivalent serial schedule of an conflict serializable schedule to build the respective conflict graph that is obviously acyclic
  - Topologically sorting transactions in an acyclic conflict graph such that an equivalent serial schedule can be derived

- Major advantage: finding cycles in a conflict graph can be done in polynomial time wrt. the number of transactions in the schedule
9.4 Restrictions

• Conflict serializability is a **good correctness criterion** for complete schedules, but it does not consider **commits/rollbacks** of transactions
  – This does for example not avoid **dirty reads**
  – It does also not take **system failures** into account

• Looking at the serializability of **prefixes** of complete schedules defines a correction for this
  – i.e. require that already committed transactions have been processed in a serializable way **at any point in time**
  – Often referred to as **commit serializability**
9.5 Building a Scheduler

• **Schedulers** are an integral part of every transaction manager
  
  – **Keeps lists** of active, committed, and aborted transactions
  
  – Accepts new transactions from applications / optimizer
  
  – Basic operations are handed on to the **scheduler** to build a consistent schedule of operations for the storage manager
9.5 Building a Scheduler

• The transactions manager **hands on** all operations to the scheduler
  – Read/write, commit, and rollback operations

• The scheduler can **autonomously decide to abort** transactions, whenever a non-serializable situation is detected

• For each transaction a scheduler can
  – **Output** operations: the transaction is running
  – **Reject** operations: the transaction is aborted
  – **Block** operations: the transaction is waiting
9.5 Locking Schedulers

• The largest class of practical schedulers are locking schedulers

  – Locks can be set on and removed from data items on behalf of transactions
    • Locking should be an atomic operation
    • The lock granularity defines what is actually locked (data records, pages, etc.)

  – Once a lock has been set on behalf of a transaction, the respective item is not available to other transactions
    • Concurrent transactions needing to access the same item are “locked out”
9.5 Lock Conflicts

• If a lock is requested by some transaction, the scheduler checks whether it has already been issued to another transactions
  – If not, the transaction can acquire the lock
    • Else a lock conflict arises, the requesting transaction is blocked by the scheduler, and has to wait
  – Eventually, the running transaction will release its lock and the scheduler can check whether the waiting transaction can be resumed
9.5 Locking in Schedules

- DBMS should only support **legal** schedules, i.e.
  - All data items are locked **before** a respective data access is performed
  - All locks are **released** at some point after the data accesses have been performed (items are **unlocked**)
  - **No unnecessary** locks are acquired

- The idea of **conflict serializability** can be extended by respective lock/unlock operations in a straightforward manner
  - Serializability graphs are analogous
9.5 Lock Modes

- For conflict-free data access there are **two types** of locks
  - **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
A generalization of this model allows transactions to prepare updates on a copy of the actual data item, while still allowing for concurrent reads.

- This is especially interesting for long running modifications.
- The analysis lock has to be followed by an exclusive write lock before the actual update on the database can be committed.
- Trade-off between read concurrency and the work lost in case of rollbacks.

### 9.5 Lock Modes

<table>
<thead>
<tr>
<th>Lock held</th>
<th>read lock</th>
<th>analyze lock</th>
<th>write lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>read lock</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>analyze lock</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>write lock</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
9.5 Lock Conversion

• If a transaction starts with a read operation and later also writes a data record
  – The scheduler may first obtain a **shared lock** to allow concurrent read of other transactions
  – Then upgrade the shared lock to an **exclusive lock** (respecting the compatibility with other transactions)
  • Concurrent shared locks have to be unlocked before the conversion can take place
9.5 Two-Phase Locking Protocol

• How can we actually **generate** a conflict-serializable schedule?

• Prominent technique: **Two-Phase Locking (2PL)**
  – Locks are granted in a **growing phase**
  – Locks are released in a **shrinking phase**
  • This means, that for each transaction all necessary locks are acquired **before** the first lock is released.
9.5 Two-Phase Locking Protocol

- Two-phase locking protocols are a simple way to generate only serializable schedules
  
  \[ S := \text{lock}(x) \text{r}(x) \text{lock}(y) \text{r}(y) \text{lock}(p) \text{r}(p) \text{w}(p) \text{unlock}(p) \text{w}(x) \text{unlock}(x) \text{unlock}(y) \]

  - Each legal schedule 2PL schedule is serializable

- Actually, the output schedules are a proper subset of conflict serializable schedules

  \[ S := \text{r}(x) \text{w}(x) \text{r}(y) \text{r}(x) \text{w}(y) \]

  - \( S \) is conflict serializable, but not in 2PL
    - Lock on \( x \) must be released before acquiring lock on \( y \)
• **2-Phase-Locking** schedules are a proper subset of conflict serializable schedules
9.5 Two-Phase Locking Protocol

• In two-phase locking, data is usually only written physically to the database during shrinking
  – All necessary locks have been acquired, the transaction can commit
  – Usually, transactions are aborted during the first phase while acquiring locks
    • Transactions that have only read data are easy to roll back
    • The amount of work lost is usually rather small
• 2-Phase-Locking has two major weak points

  – Conflicts during growing phase
    • Two transactions compete for the same lock
      – i.e. $T_1 := w(x) \ w(y); T_2 := w(y) \ w(x);$ 
  
  – Aborts during shrinking phase
    • $T_1$ already released a lock which then is immediately acquired by $T_2$
    • $T_1$ is rolled backed
      – Also, $T_2$ must be rolled back (cascading rollback)
9.5 Variants of Two-Phase Locking

• There are two main variants of two-phase locking protocols to avoid these weaknesses

  – **Conservative locking**
    • A pessimistic approach that immediately claims all necessary read and write locks prior to execution

  – **Strict two-phase locking**
    • All locks are held until commit
    • Used in most practical DBMS
9.5 Variants of Two-Phase Locking

• **Conservative locking** (also called static locking or preclaiming) acquires all necessary locks before its first read or write
  – Restricts concurrency
  – Improves chances of successful commit
  – Only possible in restricted scenario, since read/write sets must be declared to the scheduler **in advance**
9.5 Variants of Two-Phase Locking

- **Strong Strict two-phase locking (SS2PL)** holds all locks until the transaction terminates
  - Output are only **strict** and **cascadeless** schedules
    - Easy to recover
    - No cascading rollbacks
  - Most commonly used in DBMS
• SS2PL locking also avoids **dirty reads**

• Consider a dirty read situation
  – After T1 is aborted, also T2 has to be rolled back

  • This example cannot happen in SS2PL
  • This contradicts the idea of **durability** specified by the ACID properties
  • Even worse, it can result in a **cascade of rollbacks** and thus waste a lot of efforts!
Basic database transactions
The ACID principle
Transaction schedules
Conflict serializability
Locking schedulers
Outlook: Transaction Processing

Locking schedulers
Altruistic locking
Predicate-oriented locking
Non-locking schedulers
Implementation details
Isolation levels