6.1 Foundations

Foundations of TXN Management

A Transaction is a sequence of operations which represent a semantic unit and transfer a database from one consistent state to another consistent state adhering to the ACID-principle.

Aspects:

• Semantic integrity: consistent state must be reached after transaction, no matter if it succeeded or failed
• Process integrity: avoid failures due to concurrent parallel access by multiple users/transactions

Transactions: ACID Properties

• Atomicity means that a transaction can not be interrupted or performed only partially
  – TXN is performed in its entirety or not at all
• Consistency to preserve data integrity
  – A TXN starts from a consistent database state and ends with a consistent database state
• Isolation
  – Result of a TXN must be independent of other possibly running parallel TXNs
• Durability or persistence
  – After a TXN finished successfully (from the user’s view) its results must be in the database and the effect can not be reversed

Commands of a TXN Language

• Begin of Transaction **BOT** (in SQL implicated by first statement)
• **commit**: TXN ends successfully
• **abort**: TXN must be aborted during processing
Problems with Processing Integrity

- Parallel accesses in multi-user DBMS can lead to the following problems
  - Non-repeatable reads
  - Dirty reads
  - The phantom problem
  - Lost updates

Non-repeatable Read

Example:

- Assertion: \( X = A + B + C \) at the end of txn \( T_1 \)
- \( X \) and \( Y \) are local variables
- \( T_i \) is txn \( i \)
- Integrity constraint on persistent data \( A + B + C = 0 \)

Non-repeatable Read /2

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X := A; )</td>
<td>( Y := A/2; )</td>
</tr>
<tr>
<td></td>
<td>( A := Y; )</td>
</tr>
<tr>
<td></td>
<td>( C := C + Y; )</td>
</tr>
<tr>
<td></td>
<td><strong>commit:</strong></td>
</tr>
<tr>
<td>( X := X + B; )</td>
<td></td>
</tr>
<tr>
<td>( X := X + C; )</td>
<td></td>
</tr>
<tr>
<td><strong>commit:</strong></td>
<td></td>
</tr>
</tbody>
</table>

Dirty Read

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{read}(X);</td>
<td>\textbf{read}(X);</td>
</tr>
<tr>
<td>( X := X + 100; )</td>
<td>( Y := Y + X; )</td>
</tr>
<tr>
<td>\textbf{write}(X);</td>
<td>\textbf{write}(Y);</td>
</tr>
<tr>
<td>\textbf{abort;}</td>
<td><strong>commit:</strong></td>
</tr>
</tbody>
</table>
The Phantom Problem

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>select count(*) into $X$ from Employee;</td>
<td>insert into Employee values (Meier, 50000, ...); commit;</td>
</tr>
<tr>
<td>update Employee set Salary = Salary + 10000/$X$; commit;</td>
<td></td>
</tr>
</tbody>
</table>

Lost Update

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($X$);</td>
<td>read($X$);</td>
<td>10</td>
</tr>
<tr>
<td>$X := X + 1$;</td>
<td>$X := X + 1$;</td>
<td>10</td>
</tr>
<tr>
<td>write($X$);</td>
<td>write($X$);</td>
<td>11</td>
</tr>
</tbody>
</table>
Simplified TXN Model
Representation of (abstract) data object (values, tuples, pages) access

- **read**($A, x$): assign value of DB object $A$ to variable $x$
- **write**($x, A$): assign value of $x$ to DB object $A$

Example of a txn $T$:

```plaintext
read(A, x); x := x - 200; write(x, A); read(B, y); y := y + 100;
write(y, B); commit
```

Schedules: possible processing of two txns $T_1, T_2$:

- Serial schedule: $T_1$ before $T_2$ or $T_2$ before $T_1$
- Intertwined schedule: mixed execution of operations from both txns

**Serializability**
An intertwined schedule of a number of transactions is called **serializable**, if the effect of the intertwined schedule is identical to the effect of any of the possible serial schedules. The intertwined schedule is then called correct and **equivalent** to the serial schedule.

- Practical approaches for deciding about serializability most often only considering read/write operations and their conflicts → **Conflict Serializability**
- Considering other operations requires analysis of operations semantics
- Rules out subset of serializable schedules which are hard to detect

### Conflicting Operations

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read $A$</td>
<td>read $A$</td>
<td>read $A$</td>
<td>write $A$</td>
</tr>
<tr>
<td>independent of order</td>
<td>dependent on order</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read $A$</td>
<td>write $A$</td>
<td>write $A$</td>
<td></td>
</tr>
<tr>
<td>dependent on order</td>
<td>dependent on order</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conflict Serializability

A schedule \( s \) is called conflict-serializable, if the order of all pairs of conflicting operations is equivalent to the order of any serial schedule \( s' \) for the same transactions. Tested using conflict graph \( G(s) = (V, E) \) of schedule \( s \):

1. Vertex set \( V \) contains all txns of \( s \)
2. Edge set \( E \) contains an edge for each pair of conflicting operations

In serial schedules \( s' \) there can no cycles, i.e. if a cycle exists the schedule \( s \) can not be equivalent to a serial schedule and must be rejected.

Conflict Serializability: Example /1

\[
\begin{array}{ccc}
T_1 & T_2 & T_3 \\
r(y) & r(y) & r(u) \\
w(y) & w(x) & w(z) \\
w(x) & w(x) & \\
\end{array}
\]

\( s = r_1(y)r_3(u)r_2(y)w_1(y)w_1(x)w_2(x)w_2(z)w_3(x) \)

Conflict Serializability: Example /2

\[
G(s): \\
T_1 \rightarrow T_2 \rightarrow T_3 \\
\]
Transaction Synchronization

1. Most common practical solution: Locking Protocols
   - TXNs get temporarily exclusive access to DB object (tuple, page, etc.)
   - DBMS manages temporary locks
   - Locking protocol grants conflict serializability without further tests

2. In distributed DBMS also: timestamp-based protocols

Locking Protocols
Read and write locks using the following notation:

- $rl(x)$: read lock on object $x$
- $wl(x)$: write lock on object $x$

Unlock $ru(x)$ and $wu(x)$, often combined $u(x)$ unlock object $x$

Locks: Compatibility Matrix

- For basic locks

<table>
<thead>
<tr>
<th></th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_j(x)$</td>
<td>$\sqrt{\text{Yes}}$</td>
<td>—</td>
</tr>
<tr>
<td>$wl_j(x)$</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
2-Phase-Locking Protocol

2-Phase-Locking: Example

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u(x)$</td>
<td>$wl(x)$</td>
</tr>
<tr>
<td>$wl(y)$</td>
<td>$wl(y)$</td>
</tr>
<tr>
<td>$..$</td>
<td>$..$</td>
</tr>
<tr>
<td>$u(x)$</td>
<td>$u(x)$</td>
</tr>
<tr>
<td>$u(y)$</td>
<td>$u(y)$</td>
</tr>
<tr>
<td>$wl(y)$</td>
<td>$wl(y)$</td>
</tr>
<tr>
<td>$..$</td>
<td>$..$</td>
</tr>
</tbody>
</table>
**Strict 2-Phase-Locking**

Current practice in most DBMS:

Avoids cascading aborts!

**Conservative 2-Phase-Locking**

To avoid Deadlocks:

Most often not practical!
6.2 Distributed TXN Processing

**Distributed TXN Processing**

- In DDBS one TXN can run on multiple nodes
- Distributed synchronization required for parallel TXNs required
- Commit as atomic event → same result on all nodes
- Deadlocks (blocking/blocked TXNs) harder to detect

**Structure of Distributed TXNs**

![Diagram of distributed transactions]

**Distributed Synchronization with Locking**

- One central node for lock management
  - Dedicated node becomes bottleneck
  - Low node autonomy
  - High number of messages for lock acquisition/release
- Distributed lock management on all nodes
  - Possible, if data (relations, fragments) is stored non-redundantly
  - Special strategies for replicated data
  - Disadvantage: deadlocks are hard to detect
- Latter, i.e. distributed S2PL, is state-of-the-art
- Alternative: Timestamp-based Synchronisation
Timestamp-based Synchronization

- Unique timestamps are sequentially assigned to TXNs
- For each data object (tuple, page, etc.) \( x \) two values are stored:
  - \( \text{max}-r\text{-scheduled}[x] \): timestamp of last TXN performing a read operation on \( x \)
  - \( \text{max}-w\text{-scheduled}[x] \): timestamp of last TXN performing a write operation on \( x \)

Timestamp-Ordering

An operation \( p_i[x] \) can be performed before a conflicting operation \( q_k[x] \) iff \( ts(T_i) < ts(T_k) \). Otherwise, \( q_k[x] \) must be rejected.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( ts = 200 )</td>
<td>( ts = 150 )</td>
<td>( ts = 175 )</td>
<td>mrs</td>
<td>mws</td>
</tr>
<tr>
<td>read ( B )</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>write ( B )</td>
<td></td>
<td></td>
<td></td>
<td>150</td>
<td>0</td>
</tr>
<tr>
<td>write ( A )</td>
<td></td>
<td></td>
<td></td>
<td>150</td>
<td>0</td>
</tr>
<tr>
<td>read ( C )</td>
<td></td>
<td></td>
<td></td>
<td>150</td>
<td>200</td>
</tr>
<tr>
<td>write ( C ) ( \downarrow )</td>
<td></td>
<td></td>
<td></td>
<td>150</td>
<td>200</td>
</tr>
<tr>
<td>abort</td>
<td></td>
<td></td>
<td></td>
<td>150</td>
<td>200</td>
</tr>
</tbody>
</table>
Distributed Commit

- Synchronization provides Consistency and Isolation
- Commit Protocol provides Atomicity and Durability
- Requirements in DDBS:
  - All participating nodes of one TXN with same result \((\text{Commit, Abort})\)
  - Commit only if all nodes vote "yes"
  - Abort if at least one node votes "no"
- X/open XA standard for 2-Phase-Commit Protocol (used in DBMS, request brokers, application servers, etc.)

2-Phase-Commit Protocol (2PC)

- Roles: 1 coordinator, several other participants
- Procedure:
  1. Commit Request Phase
     (a) Coordinator queries all participants, if Commit can be executed
     (b) Participants send their local reply message, if they agree with the commit
  2. Commit Phase
     (a) Coordinator decides globally: (all messages \(\text{Commit} \rightarrow \text{Global-Commit}\);
        at least one \(\text{Abort} \rightarrow \text{Global-Abort}\)
     (b) Participants which voted "yes" have to wait for final result
2PC: Procedure /1

Coordinator

INITIAL

Write begin_commit to Log

WAIT

All agreed?

Yes

No

Write abort to Log

Prepare-To-Commit

Vote-Commit

Vote-Abort

Participant

INITIAL

Write abort to Log

Ready to commit?

Yes

No

Write ready to Log

Global-Abort

READY

2PC: Procedure /2

Commit

Abort

Global decision?

Global-Commit

Global-Abort

Unilateral Abort

ACK

ACK

EOT

agreed?

All

No

Write abort to Log

Write commit to Log

Write abort to Log

Write commit to Log

ABORT

COMMIT

ABORT

ABORT

COMMIT

Yes

No

Write begin_commit to Log

Write abort to Log

Write abort to Log

Write commit to Log
3-Phase-Commit Protocol (3PC)

- Possible problem with 2PC: if coordinator fails while other participants are in READY state, these may block indefinitely

- Solution:
  - Intermediate PRE-COMMIT phase added, so that a certain number $K$ (system parameter) of other participants know about possible positive result
  - Timeout values to avoid blocking: if communication timed out an no other node is in PRE-COMMIT state, TXN must abort

- Disadvantages
  - 3PC has increased message number
  - Still problematic in case of network partitioning
6.3 Transaction Deadlocks

**Transaction Deadlocks**

![Diagram of transaction deadlocks](image)

**Dealing with TXN Deadlocks**

- **Deadlock Prevention:**
  - Implement TXN management in a way that makes deadlocks impossible, e.g. lock pre-claiming in Conservative 2PL
  - Most often not practical or efficient

- **Deadlock Avoidance:**
  - TXN management reacts to possible deadlock situations, e.g. timeout for lock requests
  - Easy to implement, but overly restrictive and not always efficient

- **Deadlock Detection and Resolution:**
  - TXM management detects deadlocks, e.g. using TXN Wait graphs
  - Efficient, but harder to implement - especially for DDBMS

**Deadlock Avoidance: Timeouts**

- Reset TXN if waiting time for lock acquisition exceeds pre-defined threshold

- Problem: setting the timeout threshold
  - Too short: unnecessary aborts
  - Too long: system throughput declines

- Timeout threshold specific for certain applications (typical TXN running times must be considered)

- Implemented in most commercial systems
Deadlock Avoidance: Timestamp-based

- Alternative: consider unique TXN timestamps assigned to each TXN with \textbf{BOT}
- Avoid deadlocks in case of lock conflict considering timestamp:
  - In case of conflict only the
    - younger TXN (Wound/Wait)
    - older TXN (Wait/Die)
  will continue
- Can be combined with timeout (before timestamp check)

Deadlock Detection

- Common approach
  - Protocol each lock conflict in wait graph (nodes are TXNs, directed edge $T_1 \rightarrow T_2$ means that $T_1$ waits for a lock held by $T_2$
  - Deadlocks exist iff there is a cycle in the wait graph
  - Lock request leads to lock conflict $\rightarrow$ insert new edge in wait graph $\rightarrow$ check for cycles containing new edge

Deadlock Resolution

- Wait graph:

  ![Wait Graph Diagram]

  - Resolution choosing one TXN to abort upon following criteria
    - Number of resolved cycles
    - Age of TXN
    - Effort to rollback TXN
    - Priority of TXN, . . .
Centralized Deadlock Detection

- Wait graph stored on one dedicated node
- Decrease number of messages by sending frequent bundles of collected lock waits
- Problems:
  - Late detection of deadlocks
  - Phantom-Deadlocks
  - Limited availability and node autonomy

Distributed Deadlock Detection

- Avoids dependency on global node
- Hard to realize: cycles can exist across different nodes → wait graph information needs to be exchanged between nodes
- Obermarck-Verfahren (System R*)
  - Each node with its own deadlock detector
  - Detector manages local wait graph + dependencies on locks of external transactions: one special node EX represents external transactions
  - Local deadlocks (not involving EX node) can be detected locally
  - Cycle including EX node may hint at distributed deadlock: cycle sub-graph is sent to node which created local cycle and possibly further to other involved nodes, until EX is resolved and decision can be made

Distributed Wait Graphs
6.4 Transactional Replication

Motivation for Replication

- Replication transparency → DDBMS manages updates on redundant data

Transactional Replication

- Data integrity must be granted across replicas → 1-Copy-Equivalence and 1-Copy-Serializability
- Problems to be solved:
  - How to efficiently update the copies?
  - How to handle failures (single nodes, network fragmentation)?
  - How to synchronize concurrent updates?
- Approaches
  - ROWA
  - Primary Copy
  - Consensus approaches
  - Others: Snapshot-, Epidemic, and Lazy Replication

ROWA Synchronization

- “‘Read One Write All’”
  - one logical read operation → one physical read operation on any copy (“‘read one’”), where read access can be performed most efficiently (local or closest copy)
  - one logical write operation → physical write operations on all copies (“‘write all’”)
  - Equivalent to “‘normal’” transaction synchronization: all updates within one transaction context
ROWA: Evaluation

- Advantages
  - Approach is part of normal TXN processing
  - Easy to implement
  - Grants full consistency (1-Copy-Serializability)
  - Efficient local read operations

- Disadvantages
  - Updates dependent on availability of all nodes storing replicated data
  - Longer run-time of update TXNs \(\rightarrow\) decreased throughput and availability
  - Deadlocks more likely

Primary-Copy Replication

- Also: Master-Slave Replication

- First implemented in Distributed INGRES (Stonebraker 1979)

- Basic principle
  - Designation of a primary (master) copy (“original version”); all secondary (slave) copies are derived from that original
  - All updates must first be performed on primary copy (lock relevant data objects, perform update, release locks)
  - In case of successful update, primary copy forwards updates \textit{asynchronously} to all secondary copies in separate TXNs
  - Read operations are performed locally

Primary-Copy: Example

- Secondary copies get changes from update queue (FIFO) \(\rightarrow\) consistency with primary copy

- In case of failure of secondary node, queue is processed when node is restarted
Primary-Copy: Example

- Secondary copies get changes from update queue (FIFO) -> consistency with primary copy
- In case of failure of secondary node, queue is processed when node is restarted

T1: Update Tuple #1 + Commit
T2: Tuple #2 locked

Copy #1  Primary Copy  Copy #2
  
Copy #3

Primary-Copy: Example

- Secondary copies get changes from update queue (FIFO) -> consistency with primary copy
- In case of failure of secondary node, queue is processed when node is restarted

T2: Update + Commit executed

Copy #1  Primary Copy  Copy #2
  
Copy #3
Primary-Copy: Evaluation

- Advantages compared to ROWA
  - No dependence on availability of all secondary nodes
  - Better throughput, less conflicts
  - Failure of primary copy $\Rightarrow$ one of the secondary copies can become primary copy

- Disadvantages
  - Read consistency not granted
    * Often acceptable, because state of secondary copies is consistent regarding previous point in time ("snapshot semantics")
    * Read-consistency can be implemented by requesting read locks from primary copy $\Rightarrow$ decreases advantage of local reads
  - Network fragmentation: cut-off subnet without primary copy can not continue, though it may have greater number of nodes

Consensus Approaches

- To avoid problems in case of network fragmentation and dependence on one centralized node
- Basic idea: update can be processed, if a node gets necessary if majority of nodes with copies agree
- Based on voting - overall number of possible votes: quorum $Q$
  - Required number of votes for read operations: read quorum $Q_R$
  - Required number of votes for update operations: update quorum $Q_U$
- Quorum-Overlap-Rules: grants 1-Copy-Serializablity
  1. $Q_R + Q_U > Q$
  2. $Q_U + Q_U > Q$

Consensus Approaches: Alternatives

- Weighted vs. equal votes
  - Equal votes: each node has a vote with $weight = 1$
  - Weighted votes: nodes get assigned different weight, allows decreased message number by first asking nodes great weight

- Static vs. dynamic voting
- Static: $Q_R$ and $Q_U$ do not change
- Dynamic: $Q_R$ and $Q_U$ are adjusted to new $Q$ in case of node failures or network fragmentation

- **Tree Quorum**
  - Nodes are hierarchically structured and separate quorums are defined for each level

**Majority Consensus**

- “‘Origin’” of all consensus approaches: equal votes and static
- Works in case of node failures and network fragmentation
- Supports synchronization based on timestamps (without locks)
- Basic principles:
  - Communication along logical ring of nodes $\rightarrow$ request and decisions are passed on from node to node
  - Quorum-Rules grant consistency in case of concurrent conflicting operations
  - With equal votes: $Q_U > Q/2$, i.e. majority of nodes in ring has to agree to operation

**Majority Consensus: Procedure**

1. **Origin node** of operation/TXN performs update locally and passes on changed objects with update timestamp to next node in ring

2. **Other nodes** vote
   - **reject**, if there is a timestamp conflict $\rightarrow$ abort message is sent back to previous nodes, TXN is restarted at origin node
   - **okay**, if there is no conflict, request is marked *pending* until final decision
   - **pass**, if there is no timestamp conflict but a conflict with a concurrent update which is also *pending*

3. **Last node** which votes okay and by that satisfies quorum rule passes on commit to all nodes (backward and forward), update is propagated to all further nodes, pending updates are finalized on all previous nodes
Data Patches

• Problem: resolution of inconsistencies after node failures/ network fragmentation

• Idea: application-specific rules designed during database design → individual rules for each relation

• Rules:
  – Tuple insert rules: to add new tuples (keep, remove, notify, program)
  – Tuple integration rules: merge values of independently changed tuples with the same key (latest, primary, arithmetic, notify, program)

Snapshot Replication

• Snapshots: define (remote) materialized view on master-table

• Similar to primary copy, but also allows operations (filters, projection, aggregation, etc.) as part of view definition

• Synchronization of view by explicit refresh (manual or frequent)

• Handling of updates:
  – Read-only snapshots: updates only on master table
  – Updateable views: requires conflict resolution with master table (e.g. by rules, triggers)

Epidemic Replication

• Updates possible on every node

• Asynchronous forwarding of updates to "neighbours" using version information (timestamps)

• E.g. possible with Lotus Notes

Lazy Replication

• Updates on all nodes possible

• Each node can start separate TXNs to perform updates on other nodes asynchronously

• Requires (application-specific) conflict resolution strategies

• Works in mobile scenarios, where node can connect/disconnect to/from the network