# *Information Systems*

## *Transaction Management*

Nikolaj Popov

Research Institute for Symbolic Computation
Johannes Kepler University of Linz, Austria
popov@risc.uni-linz.ac.at

# Outline

# Transactions

- Transaction: Logical unit of work.
- Begins with the execution of a BEGIN TRANSACTION operation.
- Ends with the execution of a COMMIT or ROLLBACK operation.

# Transactions

## Example

```
BEGIN TRANSACTION

UPDATE ACC 123 { BAL := BAL - $100 };
IF error THEN GO TO UNDO ; END IF ;

UPDATE ACC 456 { BAL := BAL + $100 };
IF error THEN GO TO UNDO ; END IF ;

COMMIT ;                              /* successful termination */
GO TO FINISH ;

UNDO :
ROLLBACK ;                            /* unsuccessful termination */

FINISH :
RETURN ;
```

# Transactions

- The purpose of the transaction in the example: To transfer money from one account to another.
- Single atomic operation “transfer money from one account to another”.
- Two separate updates of the database.
- The database is in incorrect state in BETWEEN the updates: $100 is temporarily missing.
- Transaction: sequence of operations that transforms a correct state of the database into another correct state.
- In intermediate states correctness is not guaranteed.

# Transactions

- It must not be allowed one of the updates to be executed and the other not.
- However, things may go wrong at the worst possible moment: system crash between two updates, arithmetic overflow on the second update, etc.
- A special component of DBMS guarantees that in case of such failures the updates already performed will be undone.
- The component is called the transaction manager.
- It guarantees that the transaction is either completed or totally canceled.
- Nested transactions are not allowed. (To be revisited later.)

# COMMIT and ROLLBACK

- COMMIT and ROLLBACK: Operations that are key to the way how the transaction manager works.
- COMMIT signals successful end-of-transaction:
  - A logical unit of work has been successfully completed.
  - The database is in a correct state again.
  - All the updates made by the unit must be “committed”.
- ROLLBACK signals unsuccessful end-of-transaction:
  - Something went wrong.
  - The database might be in an incorrect state.
  - All the updates made by the unit must be “rolled back”.

# Recovery Log

How an update can be undone?

- The system maintains a log.
- The log records the values of updated objects before and after each update.
- When it becomes necessary to undo a particular update, the system can use the corresponding log record to restore the updated object to its previous value.

# Transaction Recovery

- COMMIT establishes a commit point.
- The first BEGIN TRANSACTION statement establishes the first commit point.
- The database is supposed to be in a correct state at any commit point.

# Transaction Recovery

When a commit point is established:

- All database updates made by the executing program since the previous commit point are committed.
- Committed updates become permanent: They are recorded in the database and can not be undone.
- Prior to the commit point, all such updates are tentative: they might subsequently be undone.
- The log must be physically written before COMMIT processing can complete.
- All tuple locks are released.

# Transaction Recovery

Few implementation issues:

- Database updates are kept in buffers in main memory and not physically written to disk until the transaction commits. No need to undo disk updates.
- Database updates are physically written to disk after committing. If the system subsequently crashes, there will be no need to redo any disk updates.

However, in practice these might not hold in general.

# Transaction Recovery

More precise write-ahead log behavior:

- The log record for a given database update must be physically written to the log before that update is physically written to the database.
- All other log records for a given transaction must be physically written to the log before the COMMIT log record for that transaction is physically written to the log.
- COMMIT processing for a given transaction must not complete until the COMMIT log record for that transaction is physically written to the log.

# ACID Properties

Transaction possess ACID properties: atomicity, correctness, isolation, durability.

- Atomicity: Transaction are atomic (all or nothing).
- Correctness: Transaction transform correct states into correct states, without necessarily preserving correctness at all intermediate points.
- Isolation: Transactions are isolated from one another. Even if many transactions are running concurrently, any given transaction's updates are hidden from the rest until that transaction commits.
- Durability: Once a transaction commits, its updates persist in the database.

# Failure Categories

- Local failures affect only the transaction in which the failure occurred.
- Global failures affect all the transactions in progress.
- Two categories of global failures:
  - System failures (e.g., power outage): Affect all transactions in progress but do not physically damage the database.
  - Media failures: (e.g., head crash on the disk): Cause damage to the database and affect all the transactions currently using the damaged portion of the database.

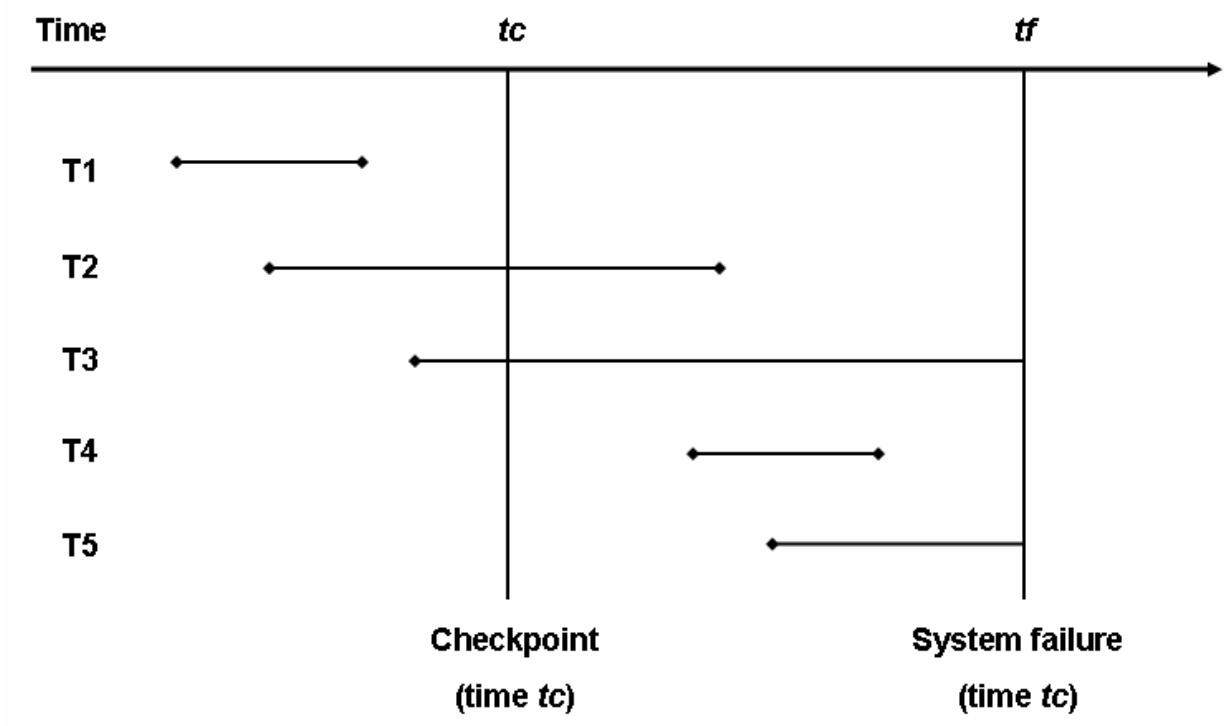
# System Recovery

- System failure causes the contents of the main memory to be lost.
- The precise state of the transaction active at the moment of failure is no longer known.
- Such a transaction must be undone when the system restarts.
- It might be necessary to redo some of the transactions that did successfully complete prior to the crash but did not manage to get their updates transferred from the main memory to the physical database.

# System Recovery

- How does the system know at restart time which transactions to undo and which redo?
- Taking checkpoints at certain prescribed intervals:
  - Forcing the contents of the main memory buffers out to the physical database.
  - Forcing the special checkpoint record out to the physical log.
- The checkpoint record contains the list of all active transactions at the checkpoint time.

# System Recovery




- Transactions of types T3 and T5 must be undone.
- Transactions of types T2 and T4 must be redone.
- T1 does not enter the restart process.

# Media Recovery

- Recovery from media failure.
- First reload or restore the database from a backup copy.
- Then use the log to redo all transactions since the backup copy was taken.
- Nothing to be undone: Transaction been in progress at the time of failure are lost anyway.

# Concurrency

- DBMSs typically allow many transactions to access the same database at the same time.
- Ensure that concurrent transactions do not interfere with each other.

# Three Concurrency Problems

- The lost update problem.
- The uncommitted dependency problem.
- The inconsistent analysis problem.

# The Lost Update Problem

| Transaction A | Time | Transaction B |
|---|---|---|
| RETRIEVE t | $t_1$ | — |
| — | $t_2$ | RETRIEVE t |
| UPDATE t | $t_3$ | — |
| — | $t_4$ | UPDATE t |

Transaction A loses an update at time $t_4$, because B overwrites it without even looking at it.

# The Uncommitted Dependency Problem

| Transaction A | Time | Transaction B |
|---|---|---|
| — | $t_1$ | UPDATE t |
| RETRIEVE t | $t_2$ | — |
| — | $t_3$ | ROLLBACK |

- Transaction A becomes dependent on an uncommitted change at time $t_2$.
- A is operating a false assumption that tuple t has the value seen at time $t_2$.
- In fact t has whatever value it had prior to time $t_1$.

# The Uncommitted Dependency Problem

| Transaction A | Time | Transaction B |
|---|---|---|
| — | $t_1$ | UPDATE t |
| UPDATE t | $t_2$ | — |
| — | $t_3$ | ROLLBACK |

- Transaction A updates an uncommitted change at time $t_2$, and loses that update at time $t_3$.
- Rollback at time $t_3$ causes tuple t to be restored to its value prior to time $t_1$.

# The Inconsistent Analysis Problem

- Transactions A and B operate on account (ACC) tuples.
- Transaction A is running account balances, transaction B is transferring an amount 10 from account 3 to account 1.
- Amounts on the accounts at the beginning:
  - ACC1: 40
  - ACC2: 50
  - ACC3: 30

# The Inconsistent Analysis Problem

ACC1: 40, ACC2: 50, ACC3: 30

| Transaction A | Time | Transaction B |
|---|---|---|
| RETRIEVE ACC1 : sum=40 | $t_1$ | — |
| RETRIEVE ACC2 : sum=90 | $t_2$ | — |
| — | $t_3$ | RETRIEVE ACC3 |
| — | $t_4$ | UPDATE ACC3 : 30 $\mapsto$ 20 |
| — | $t_5$ | RETRIEVE ACC1 |
| — | $t_6$ | UPDATE ACC1 : 40 $\mapsto$ 50 |
| — | $t_7$ | COMMIT |
| RETRIEVE ACC3 : sum=110, not 120 | $t_8$ | — |

# Locking

- The three concurrency problems can be solved by a concurrency control mechanism called locking.
- Idea behind locking:
  - A transaction needs an assurance that some object it is interested in will not change at certain moment.
  - To guarantee this, the transaction acquires a lock on that object.
  - Locking an object prevents other transactions from changing it.

# Locking

- Two kinds of locks: exclusive (X) and shared (S).
- If a transaction A holds an X lock on tuple t, then a request from some other transaction B for a lock of either type on t cannot be immediately granted.
- If transaction A holds an S lock on tuple t, then:
  - A request from a transaction B for an X lock on t cannot be immediately granted.
  - A request from a transaction B for an S lock on t will be immediately granted.

# Locking

| | X | S | — |
|---|---|---|---|
| X | N | N | Y |
| S | N | Y | Y |
| — | Y | Y | Y |

Compatibility matrix for lock types X and S.

# Locking Protocol

Strict two-phase locking:

1. A transaction that wishes to retrieve a tuple first acquires an S lock on that tuple.
2. A transaction that wishes to update a tuple first acquires an X lock (or promotes an existing S lock to an X lock) on that tuple.
3. If a lock requested by transaction B cannot be immediately granted because of lock conflict with transaction A, B goes into a wait state.
4. B will acquire the lock requested after A's lock is released (and there are no other transactions waiting in the queue before B).
5. The locks are released at end-of-transaction (COMMIT or ROLLBACK).

Makes use of X and S locks to guarantee that the concurrency problems do not occur.

# The Lost Update Problem Revisited

| Transaction A | Time | Transaction B |
|---|---|---|
| RETRIEVE t | $t_1$ | — |
| (acquire S lock on t) | | |
| — | $t_2$ | RETRIEVE t |
| — | | (acquire S lock on t) |
| UPDATE t | $t_3$ | — |
| (request X lock on t) | | |
| wait | $t_4$ | UPDATE t |
| wait | | (request X lock on t) |
| wait | | wait |

- No update is lost, but deadlock occurs at time $t_4$.
- We reduced the original problem to a new problem: deadlock.

# The Uncommitted Dependency Problem Revisited

| Transaction A | Time | Transaction B |
|---|---|---|
| — | $t_1$ | UPDATE t |
| — | | (acquire X lock on t) |
| RETRIEVE t | $t_2$ | — |
| (request S lock on t) | | |
| wait | $t_3$ | COMMIT / ROLLBACK |
| wait | | (release X lock on t) |
| resume: RETRIEVE t | $t_4$ | |
| (acquire S lock on t) | | |

- Transaction A is prevented from seeing an uncommitted change at time $t_2$.
- We have solved the original problem.

# The Uncommitted Dependency Problem Revisited

| Transaction A | Time | Transaction B |
|---|---|---|
| — | $t_1$ | UPDATE t |
| — | | (acquire X lock on t) |
| UPDATE t | $t_2$ | — |
| (request X lock on t) | | – |
| wait | $t_3$ | COMMIT / ROLLBACK |
| wait | | (release X lock on t) |
| resume: UPDATE t | | |
| (acquire X lock on t) | | |

- Transaction A is prevented from updating an uncommitted change at time $t_2$.
- We have solved the original problem.

# The Inconsistent Analysis Problem

| Transaction A | Time | Transaction B |
|---|---|---|
| RETRIEVE ACC1 : | $t_1$ | — |
| (acquire S lock on ACC1) | | — |
| sum=40 | | — |
| RETRIEVE ACC2 : | $t_2$ | — |
| (acquire S lock on ACC2) | | — |
| sum=90 | | — |
| — | $t_3$ | RETRIEVE ACC3 |
| — | | (acquire S lock on ACC3) |
| — | $t_4$ | UPDATE ACC3 |
| — | | (acquire X lock on ACC3) |
| — | | $30 \mapsto 20$ |
| — | $t_5$ | RETRIEVE ACC1 |
| — | | (acquire S lock on ACC1) |
| — | $t_6$ | UPDATE ACC1 |
| — | | (request X lock on ACC1) |
| — | | wait |
| RETRIEVE ACC3 | $t_7$ | wait |
| (request S lock on ACC3) | | wait |
| wait | | wait |

# The Inconsistent Analysis Problem

- Inconsistent analysis is prevented, but deadlock occurs at time $t_7$.
- We reduced the original problem to a new problem: deadlock.

# Deadlock

- Deadlock is a situation in which two or more transactions are in a simultaneous wait state.
- Each of these transactions is waiting for one of the others to release a lock before it can proceed
- An example of deadlock:

| Transaction A | Time | Transaction B |
|---|---|---|
| LOCK r1 EXCLUSIVE | $t_1$ | — |
| — | $t_2$ | LOCK r2 EXCLUSIVE |
| LOCK r2 EXCLUSIVE | $t_3$ | — |
| wait | $t_4$ | LOCK r1 EXCLUSIVE |
| wait | | wait |

# Deadlock Resolution

- To predict and avoid deadlocks can be difficult and time consuming (finding circles in graphs).
- In practice if a transaction does not make any progress within a given time limit it is assumed to be deadlocked.
- If there are presumably deadlocked transaction, the transaction manager chooses a “victim” and requests an implicit rollback on it.
- When this happens all the locks the “victim” held get released.

# Serialization

- A (possibly interleaved) execution of a set of transactions is called schedule.
- Serial execution: transactions are run one at a time in some sequence.
- A schedule of a set of transactions, which are individually correct by assumption, is called serializable if and only if it produces the same result as some serial execution of the same set of transactions; and this does not depend on the (a priori correct) initial state we start from.
- Serializable schedules transfer the database from a correct state into a correct state.
- Serializability can be difficult to test for an arbitrary schedule.

# Serialization

- The two-phase locking protocol requires that
  - a transaction must acquire a lock (of the appropriate kind) on a tuple before it operates with it
  - after releasing a lock, the transaction must never acquire a lock again.
- In other terms, the transaction must acquire locks and it is expected to separate the lock acquisition and the lock release phases.

## Theorem (Two-Phase Locking Theorem)

*If all transactions obey the two-phase locking protocol, then all possible interleaved schedules are serializable.*

# Summary

- Transactions are initiated by BEGIN TRANSACTION and are terminated by either COMMIT (successful termination) or ROLLBACK (unsuccessful termination).
- COMMIT establishes a commit point (updates are recorded in the database);
- ROLLBACK rolls the database back to the previous commit point (updates are undone).
- If a transaction does not reach its planned termination, the system automatically executes a ROLLBACK for it (transaction recovery).

# Summary

- In order to be able to undo and redo updates, the system maintains a recovery log.
- The log records for a given transaction must be written to the physical log before COMMIT processing for that transaction can complete.
- If a system crash occurs, the system must (a) redo all work done by transactions completed successfully prior to the crash and (b) undo all work done by transactions that started but did not complete prior to the crash.
- This system recovery activity is carried out as part of the system's restart procedure.

# Summary

- Three problems can arise in an interleaved execution of concurrent transactions if no control is in place: the lost update problem, the uncommitted dependency problem, and the inconsistent analysis problem.
- The most widespread technique for dealing with such problems is locking.