Module 11: Transaction and Recovery Management

Fall 2026

Cheriton School of Computer Science

CS 338: Intro to Database Management

Outline

Basics of Transaction Processing

DML processing converts interaction with *sets of tuples* to a sequence of operations that read and write *physical objects* in the database.

Database objects can correspond to:

- individual record fields,
- records,
- physical pages,
- entire files (typically for concurrency control purposes), and
- predicates qualifying one or more records, parts of records, etc.

Transaction and recovery management entails:

- ensuring correct and concurrent execution of operations for reading and writing physical objects originating from DML commands of client transactions; and
- guaranteeing that *acknowledged client transaction commits* are reliably recorded, and that *acknowledged client transaction aborts* are reliably undone.

Transactions (from Module 1)

Transaction

A sequence of *indivisible* DML requests. Applications access a database via transactions.

An application programmer may assume exclusive access to the database within a transaction. The DBMS schedules DML requests from *all* transactions in such a way that guarantees data integrity.

ACID Properties of a Transaction

Atomicity	A transaction occurs entirely, or not at all.
Consistency	Each transaction preserves the consistency of the database.
Isolation	Concurrent transactions do not interfere with each other.
Durability	Once completed, a transaction's changes are permanent.

Transactions (cont'd)

Transaction and recovery management is implemented in a DBMS with two subsystems:

Concurrency Control Management

Responsible for scheduling the execution of reads and writes on physical objects in a way that ensures transaction isolation.

Recovery Management

Responsible for ensuring transaction atomicity and durability.

NOTE: DML processing is responsible for ensuring the part of transaction consistency defined by integrity constraints.

Concurrency Control: Assumptions

- We consider a database to be a finite set of independent physical objects x_j that are read and written by transactions T_i, and write:
  1. $r_i[x_j]$ to say that transaction T_i reads object x_j, and
  2. $w_i[x_j]$ to say that transaction T_i writes (modifies) object x_j.

- A transaction T_i is a sequence of read and write operations on a database

  $$T_i = r_i[x_1], r_i[x_2], w_i[x_1], \ldots, r_i[x_4], w_i[x_2], c_i$$

  where c_i is the *commit request* of T_i.

- For a *set of transactions* $\{T_1, \ldots, T_k\}$, we want to produce an execution order of operations S, called a *schedule*, such that
  1. every operation $o_i \in T_i$ appears also in S, and
  2. T_i's operations in S are ordered the same way as in T_i.

Goal: Produce a *correct schedule* with maximal parallelism.

Transactions and Schedules

If T_i and T_j are concurrent transactions, then it is always correct to schedule the operations in such a way that:

- T_i will appear to precede T_j meaning that T_j will "see" all updates made by T_i, and T_i will not see any updates made by T_j, or
- T_i will appear to follow T_j, meaning that T_i will see T_j's updates and T_j will not see T_i's.

Idea

Define a schedule S to be *correct* if it appears to clients that the transactions are executed sequentially, that is, in some strictly serial order.

Serializable Schedule

A schedule S is said to be serializable if it is equivalent to some serial execution of the same transactions.

Serializable Schedules

Examples:

- An interleaved execution of two transactions:

$$S_a = w_1[x], r_2[x], w_1[y], c_1, r_2[y], c_2$$

- An equivalent serial execution (T_1 , T_2):

$$S_b = w_1[x], w_1[y], c_1, r_2[x], r_2[y], c_2$$

- An interleaved execution with no equivalent serial execution:

$$S_c = w_1[x], r_2[x], r_2[y], c_2, w_1[y], c_1$$

Serializable Schedules (cont'd)

How do we determine if two schedules are *equivalent*?

⇒ *cannot be based on any particular database instance*

Conflict Equivalence

Two operations conflict if they

1. belong to different transactions
2. access the same data item x, and
3. at least one of them is a write operation $w[x]$.

Two schedules S_1 and S_2 are conflict equivalent when all *conflicting operations* are ordered the same way.

A schedule S_1 is a conflict serializable schedule if it is conflict equivalent to some serial schedule S_2.

View Equivalence: Preserves initial reads and final writes; allows more schedules, but is harder (NP-hard) to diagnose.

Serializable Schedules (cont'd)

How does one diagnose if a schedule S is a conflict serializable schedule?

Serialization Graph

A *serialization graph* $SG(N, E)$ *for a schedule* S is a directed graph with

1. nodes $T_i \in N$, corresponding to the transactions of S, and
2. edges $T_i \rightarrow T_j \in E$ whenever an operation $o_i[x]$ for transaction T_i occurs prior to an operation $o_j[x]$ for transaction T_j in S, where $o_i[x]$ and $o_j[x]$ are conflicting operations.

Theorem

A schedule S is serializable if and only if the serialization graph SG for S is acyclic.

Other Properties of Schedules

Serializability guarantees correctness. However, we'd like to avoid other *undesirable* situations.

Recoverable Schedules (RC)

The situation: Transaction T_j *reads* a value T_i has written, then T_j commits, then T_i attempts to abort.

$\Rightarrow$ *aborting T_i makes it necessary to undo effects of a committed transaction T_j*

The repair: Allow committing only in order of the read-from dependency between transactions.

Cascadeless Schedules (ACA)

The situation: Transaction T_j *reads* a value T_i has written, then T_i attempts to abort .

$\Rightarrow$ *aborting T_i makes it necessary to abort T_j, which may lead to a cascade of further aborts of other transactions*

The repair: Do not allow transactions to read uncommitted objects.

Outline

Two Phase Locking

How does a DBMS admit ultimate schedules for execution of the operations of transactions that are conflict serializable and cascade free?

The DBMS scheduler is responsible for this.

1. It receives requests to execute operations from the query processor(s), and
2. for each operation it chooses one of the following actions:
   - execute it (by sending to a lower module),
   - delay it (by inserting in some queue),
   - reject it (thereby causing abort of the transaction), or
   - ignore it (since it has no effect).

There are two kinds of schedulers:

1. conservative schedules favour delaying operations, and
2. aggressive schedulers favour rejecting operations.

Two Phase Locking

Conservative schedulers are lock based.

A simple lock based scheduler requires transactions to have a lock on objects before access:

- a *shared lock* is required to read an object, and
- an *exclusive lock* is required to write an object,

where an exclusive lock on an object x by a transaction disallows any other lock at all to be held on x by any other transaction.

It is insufficient for a scheduler just to acquire a lock, access the data item, and then release it immediately.

Exercise: *Show why this is the case.*

Two Phase Locking (cont'd)

Two Phase Locking

A scheduler follows the *two phase locking protocol* (2PL) if it allows no lock to be acquired by a transaction T on some object after the first lock by T on some other (not necessarily distinct) object is released by T.

Theorem

Any execution order of operations admitted by a scheduler following the two phase locking protocol will be a prefix of an ultimate schedule that is a conflict serializable schedule.

Strict Two Phase Locking (cont'd)

The 2PL locking protocol can still allow operations to proceed that lead to:

1. cascading aborts, and
2. deadlock.

A refinement of 2PL addresses the first of these problems.

Strict Two Phase Locking

A scheduler follows the *strict two phase locking protocol* (strict 2PL) if it allows no lock for a transaction T to be released until T requests either a commit or abort operation.

Theorem

Any execution order of operations admitted by a scheduler following the strict two phase locking protocol will be a prefix of an ultimate schedule that is a conflict serializable schedule and will also have the ACA property ensuring there are no cascading aborts.

Deadlocks

Either 2PL or strict 2PL may lead to a *deadlock* involving a sequence of transactions $(T_1, \ldots, T_n)$ where T_i must wait for T_{i+1} to release locks, $1 \leq i < n$, and where T_n must wait for T_1 to release locks.

Example: *Consider where a strict 2PL scheduler has received the following sequence of operations leading to deadlock with the transaction sequence* (T_2, T_1):

$$r_1[x], r_2[y], w_2[x] \text{ (blocked by } T_1\text{)}, w_1[y] \text{ (blocked by } T_2\text{)}$$

There are two general ways to resolve deadlocking:

deadlock prevention

$\Rightarrow$ *locks granted only if they can't lead to a deadlock*
$\Rightarrow$ *all objects are ordered, and locks granted in this order*

deadlock detection

$\Rightarrow$ *use wait for graphs and cycle detection*
$\Rightarrow$ *resolution: the system aborts one of the offending transactions; called an involuntary abort*

In practice, some schedulers detect a lock request timeout instead of a deadlock and resolve by aborting.

Variations on Locking

- Multi-granularity Locking

  ⇒ *not all locked objects have the same size*
  ⇒ *advantageous in presence of bulk versus tiny updates*

- Predicate Locking

  ⇒ *locks based on selection predicate rather than on a value*

- Tree Locking

  ⇒ *can be used to avoid congestion in roots of Btrees*
  ⇒ *allows relaxation of 2PL due to tree structure of data*

- Lock Upgrade protocols

- ...

Inserts and Deletes

We have been assuming a database consists of a *fixed set of physical objects*.

Consider where DML processing requires creating or deleting physical objects, such as records in an index.

Neither 2PL nor strict 2PL correctly handles such cases.

Example: *Consider the following pair of transactions:*

1. *Based on a sufficient fixed budget, transaction T_1 gives each employee a bonus.*
2. *As a consequence of a new hire, transaction T_2 adds a new employee (who should not receive the bonus).*

This situation is called the phantom tuple problem.

Resolution: DML requests that modify the database according to the results of a query need larger scale locks, such as

- locks on entire indices, or
- predicate locks on tables

together with appropriate *lock compatibility* rules.

Isolation Levels in SQL

Scheduling that admits only conflict serializable schedules can lead to severe performance issues: transaction throughput or response time can become unacceptable as a consequence of blocked transactions and deadlocks.

The SQL standard addresses by allowing four isolation levels for a transaction that can be set by a client SQL command:

- Level 3: (*serializability*) Realized by strict 2PL with table level locks.
- Level 2: (*repeatable read*) Realized by strict 2PL with tuple level locks only; phantom tuples may occur.
- Level 1: (*cursor stability*) Realized by strict 2PL with tuple level exclusive locks only.

  $\Rightarrow$ *the same query executed twice in a transaction can produce different answers*
- Level 0: Realized by simply not requiring any locking.

  $\Rightarrow$ *transaction queries may read uncommitted updates*

Exercise: *Consider cases where each of these isolations levels might suffice.*

Outline

Unit 1: Concurrency Control

Unit 2: Two Phase Locking

Unit 3: **Failure Recovery**

Failure Recovery

The DBMS module responsible for recovery management must satisfy two requirements:

1. (regarding atomicity) enable a transaction to be
   - committed (with a guarantee database changes are permanent), or
   - aborted (with a guarantee that there are no database changes)
2. (regarding reliability) enable a database to be recovered to a consistent state in case of hardware or software failure.

Interaction with recovery management:

Input: a 2PL and ACA schedule of operations produced by the transaction manager, and

Output: a schedule of object reads, object writes, and object forced writes.

Approaches to Recovery

Two essential approaches:

1. Shadowing

   ⇒ *copy-on-write and merge-on-commit approaches*
   ⇒ *poor clustering*
   ⇒ *used in system R, but not in modern systems*

2. Logging

   ⇒ *uses log files on (separate) stable media*
   ⇒ *good utilization of buffers*
   ⇒ *preserves original clusters*

Logging Approaches to Recovery

A log is a read/append only data structure on stable media.

⇒ *recovery manager appends transaction log records*

⇒ *aborting and failure recovery reads transaction log records*

Log records record several types of information:

- UNDO information: old versions of objects that have been modified by a transaction. UNDO information can be used to undo database changes made by a transaction that aborts.
- REDO information: new versions of objects that have been modified by a transaction. REDO records can be used to redo the work done by a transaction that commits.
- BEGIN/COMMIT/ABORT information: record when transactions begin, commit, or abort.

Example of a LOG

Five transactions: $\{T_0, T_1, T_2, T_3, T_4\}$

Transaction status:

$\{T_1, T_3\}$ committed,

$\{T_2\}$ aborted, and

$\{T_0, T_4\}$ active.

Exercises:

1. *Explain how to abort a translation using a log.*
2. *Explain how to recover from a failure using a log, assuming active transactions should be aborted.*

log head	$\rightarrow$	T_0, BEGIN
(oldest part)		T_0, UNDO, x_1, 99
		T_0, REDO, x_1, 100
		T_1, BEGIN
		T_1, UNDO, x_2, 199
		T_1, REDO, x_2, 200
		T_2, BEGIN
		T_2, UNDO, x_3, 51
		T_2, REDO, x_3, 50
		T_1, UNDO, x_4, 1000
		T_1, REDO, x_4, 10
		T_1, COMMIT
		T_3, BEGIN
		T_2, ABORT
		T_3, UNDO, x_2, 200
		T_3, REDO, x_2, 50
		T_4, BEGIN
		T_4, UNDO, x_4, 10
(newest part)		T_4, REDO, x_4, 100
log tail	$\rightarrow$	T_3, COMMIT

Write-Ahead Logging

Allowing indices on stable store to be updated prior to a LOG can lead to inconsistency.

A *write-ahead logging* (WAL) protocol avoids this by following:

1. an undo rule: a log record for an update is appended to the LOG file *before* the corresponding data page is written to stable store.

   $\Rightarrow$ *guarantees atomicity*

2. a redo rule: all log records for a transaction are appended to the LOG file *before* acknowledging a commit.

   $\Rightarrow$ *guarantees durability*

Summary

ACID properties of transactions guarantee correctness of concurrent access to the database and of data storage.

- Consistency and isolation based on serializability
  1. leads to definition of correct schedulers, and
  2. is the responsibility of the transaction manager.
- Durability and atomicity are the responsibility of the recovery manager.
- Synchronous writing is too inefficient, and replaced by synchronous writes to a LOG file according to WAL.