Transaction management

Lecture 9
2ID35, Spring 2015

George Fletcher

Faculteit Wiskunde & Informatica
Technische Universiteit Eindhoven

3 June 2015

Agenda

Today's outline:

- overview of transactions
- overview of recovery management
- overview of concurrency control

The story so far ...

Last lecture, we relaxed distribution of data and query processing...

However, we are still (implicitly) positioning a DBMS as:

1. guardian of a precious commodity
2. a read-only store
3. servicing only one client

The story so far ...

Last lecture, we relaxed distribution of data and query processing...

However, we are still (implicitly) positioning a DBMS as:

1. guardian of a precious commodity
2. a read-only store
3. servicing only one client

of course, 2 and 3 are gross simplifications

how do we relax 2 and 3, while still fulfilling the obligations of 1?

Transactions

this motivates the study of “transaction” management, i.e., the management of database interactions

- which have side effects (i.e., updates), and
- which are executing concurrently (for increased throughput, and decreased response times)

Transactions

this motivates the study of “transaction” management, i.e., the management of database interactions

- which have side effects (i.e., updates), and
- which are executing concurrently (for increased throughput, and decreased response times)

a key concept in OLTP (Online Transaction Processing) systems.

Transactions

Application	**Example Transaction**
banking	withdraw money from an account
securities trading	purchase 100 shares of a stock
insurance	pay a premium
inventory control	record fulfillment of an order
manufacturing	log a step of an assembly process
retail	record a sale
government	register an automobile
online shopping	place an order
transportation	track and log a shipment
social	follow a friend; like a post
telecom	connect a phone call

Transactions

A transaction is a logical unit of work

- a finite list of *reads* and *writes* on a fixed collection of independent data objects
- terminated by an *abort* or *commit*

Transactions

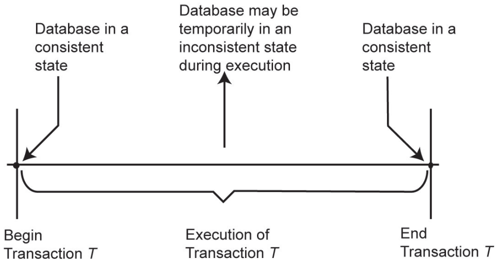
A transaction is a logical unit of work

- a finite list of *reads* and *writes* on a fixed collection of independent data objects
- terminated by an *abort* or *commit*
- only interacts with other transactions via read and write operations on DB objects

Transactions

A transaction is a logical unit of work

- a finite list of *reads* and *writes* on a fixed collection of independent data objects
- terminated by an *abort* or *commit*
- only interacts with other transactions via read and write operations on DB objects
- assumed to be consistent
  - i.e., leaves a consistent DB in a consistent state



Transactions

The SQL standard specifies that a transaction implicitly begins when an SQL statement is executed

- must be eventually followed by a `COMMIT` or `ROLLBACK` statement
- note, however, that most DBMSs auto-commit by default after each statement
- in SQL:1999, there is a `BEGIN ATOMIC` ... `END` construct for longer transactions, but this syntax has still not been widely adopted

Transactions

For our purposes, we will reason about a transaction as a finite list of reads and writes

Transactions

For our purposes, we will reason about a transaction as a finite list of reads and writes

Example of two different transactions:

- T_1: $R_1(A), W_1(A), R_1(B), W_1(B), C_1$
- T_2: $R_2(B), R_2(A), W_2(A), W_2(B), C_2$

Transactions

A schedule (also called a *history*) for a set of transactions $T_1, \ldots, T_n$ is a list of the (read, write, commit, abort) actions of the transactions which respects the order of actions of each T_i

Transactions

A schedule (also called a *history*) for a set of transactions $T_1, \ldots, T_n$ is a list of the (read, write, commit, abort) actions of the transactions which respects the order of actions of each T_i

A schedule is *complete* if it contains either an abort or a commit for each T_i

Transactions

Transaction example:

- T_1: $R_1(A), W_1(A), R_1(B), W_1(B), C_1$
- T_2: $R_2(B), R_2(A), W_2(A), W_2(B), C_2$

Transactions

Transaction example:

- T_1: $R_1(A), W_1(A), R_1(B), W_1(B), C_1$
- T_2: $R_2(B), R_2(A), W_2(A), W_2(B), C_2$

Non-schedule:

$R_1(A), W_1(A), R_1(B), W_2(A), R_2(B), W_1(B), R_2(A), W_2(B)$

Transactions

Transaction example:

- T_1: $R_1(A), W_1(A), R_1(B), W_1(B), C_1$
- T_2: $R_2(B), R_2(A), W_2(A), W_2(B), C_2$

Non-schedule:

$R_1(A), W_1(A), R_1(B), W_2(A), R_2(B), W_1(B), R_2(A), W_2(B)$

Incomplete schedule:

$R_2(B), R_1(A), W_1(A), R_2(A), R_1(B), W_2(A), W_1(B), C_1$

Transactions

Transaction example:

- T_1: $R_1(A), W_1(A), R_1(B), W_1(B), C_1$
- T_2: $R_2(B), R_2(A), W_2(A), W_2(B), C_2$

Non-schedule:

$R_1(A), W_1(A), R_1(B), W_2(A), R_2(B), W_1(B), R_2(A), W_2(B)$

Incomplete schedule:

$R_2(B), R_1(A), W_1(A), R_2(A), R_1(B), W_2(A), W_1(B), C_1$

Complete schedule:

$R_1(A), W_1(A), R_2(B), R_1(B), R_2(A), W_1(B), W_2(A), W_2(B), C_1, C_2$

ACID properties

Four desirable properties for the DBMS to enforce

ACID properties

Four desirable properties for the DBMS to enforce

- atomicity: all or nothing
  - i.e., if a transaction commits, then all of its effects are made permanent; else, it has no effect at all

ACID properties

Four desirable properties for the DBMS to enforce

- atomicity: all or nothing
  - i.e., if a transaction commits, then all of its effects are made permanent; else, it has no effect at all
- consistency: transactions map between consistent DB states
  - responsibility of programmer/client, and integrity enforcement mechanisms of the DBMS
  - transaction manager assumes all transactions are consistent

ACID properties

Four desirable properties for the DBMS to enforce

- isolation: each transaction is independent of all other (concurrent) transactions
  - i.e., each transaction sees a consistent database at all times
  - i.e., an executing transaction cannot reveal its results to other concurrent transactions before its commitment

ACID properties

Four desirable properties for the DBMS to enforce

- isolation: each transaction is independent of all other (concurrent) transactions
  - i.e., each transaction sees a consistent database at all times
  - i.e., an executing transaction cannot reveal its results to other concurrent transactions before its commitment
- durability: committed transactions should persist on stable storage, even if system fails

Architecture

In stable storage (i.e., on disk):

- stable database
- stable log, of before/after images for each update to the database

Architecture

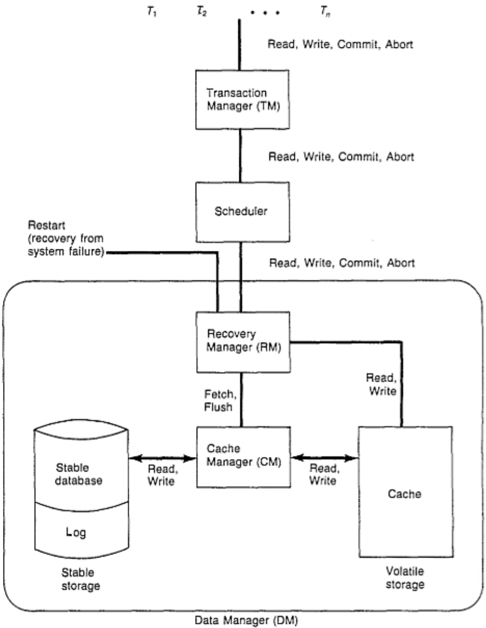
In stable storage (i.e., on disk):

- stable database
- stable log, of before/after images for each update to the database

In volatile storage (i.e., in main memory):

- a working cache (i.e., buffer) of some of the DB pages

Architecture



Architecture

Write-ahead log rule

- "a committed transaction is a completely logged transaction"
- i.e., each update must be logged in the stable log before the change itself is recorded in the stable database

Recovery management

Let's first consider how to ensure *atomicity* and *durability*, which is the responsibility of the recovery manager

Recovery management

Let's first consider how to ensure *atomicity* and *durability*, which is the responsibility of the recovery manager

Three types of failure

- *transaction:* (self) abort
    - responsibility of the transaction manager

Recovery management

Let's first consider how to ensure *atomicity* and *durability*, which is the responsibility of the recovery manager

Three types of failure

- *transaction:* (self) abort
    - responsibility of the transaction manager
- *media:* loss or corruption of stable storage
    - responsibility of the sys admin

Recovery management

Let's first consider how to ensure *atomicity* and *durability*, which is the responsibility of the recovery manager

Three types of failure

- *transaction:* (self) abort
  - responsibility of the transaction manager
- *media:* loss or corruption of stable storage
  - responsibility of the sys admin
- *system:* loss or corruption of volatile storage
  - responsibility of the recovery manager

Recovery management

Upon restart from a system failure, the recovery manager is responsible for

- undo-ing those transactions which were incomplete at the time of failure (atomicity)
- redo-ing committed transactions that didn't make it to stable storage (durability)

i.e., returning the stable database to a consistent state, reflecting all committed transactions at time of failure

Recovery management

Buffer management policy recap

- A steal policy: buffer manager can write buffer to disk before a transaction commits
- Alternative policy: no-steal

Recovery management

Buffer management policy recap

- A steal policy: buffer manager can write buffer to disk before a transaction commits
- Alternative policy: no-steal
- A force policy: all pages updated by a transaction are immediately written to disk when the transaction commits
- Alternative policy: no-force

Recovery management

interaction with cache management

- a recovery manager requires undo if "steals" in cache/buffer are possible
  - i.e., buffer pages can be forced to the stable database before a commit

Recovery management

interaction with cache management

- a recovery manager requires undo if "steals" in cache/buffer are possible
  - i.e., buffer pages can be forced to the stable database before a commit
- a recovery manager requires redo if "no-force" of cache/buffer is required upon commits
  - i.e., buffer pages aren't necessarily flushed to the stable database upon commits

Recovery management

interaction with cache management

- a recovery manager requires undo if “steals” in cache/buffer are possible
  - i.e., buffer pages can be forced to the stable database before a commit
- a recovery manager requires redo if “no-force” of cache/buffer is required upon commits
  - i.e., buffer pages aren’t necessarily flushed to the stable database upon commits

no-steal/force is ideal combination, but impractical

steal/no-force is realistic, so let’s focus on this situation

Recovery management

So, the system fails (e.g., due to a power outage). How does RM determine which transactions to undo and which to redo (without replaying the whole log)?

Recovery management

So, the system fails (e.g., due to a power outage). How does RM determine which transactions to undo and which to redo (without replaying the whole log)?

Via periodically performing a checkpoint, when

1. the buffer is flushed to stable storage, and
2. a checkpoint record is written to the stable log, indicating transactions in progress.

Recovery management

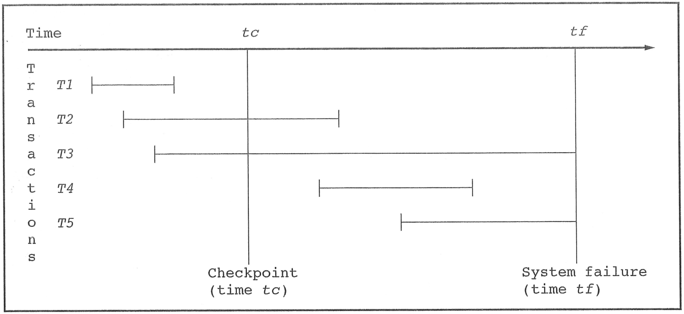
So, the system fails (e.g., due to a power outage). How does RM determine which transactions to undo and which to redo (without replaying the whole log)?

Via periodically performing a checkpoint, when

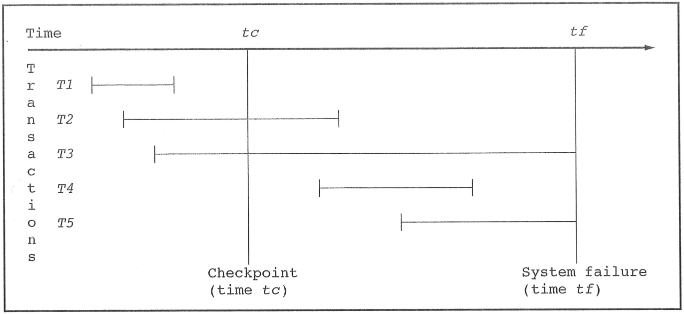
1. the buffer is flushed to stable storage, and
2. a checkpoint record is written to the stable log, indicating transactions in progress.

Tuning the frequency of checkpoints is a critical issue in practice

Recovery management

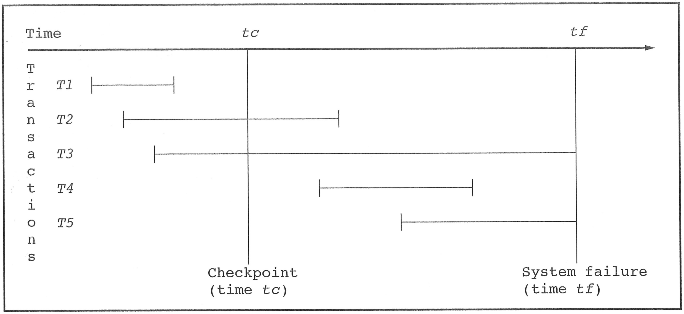


Recovery management



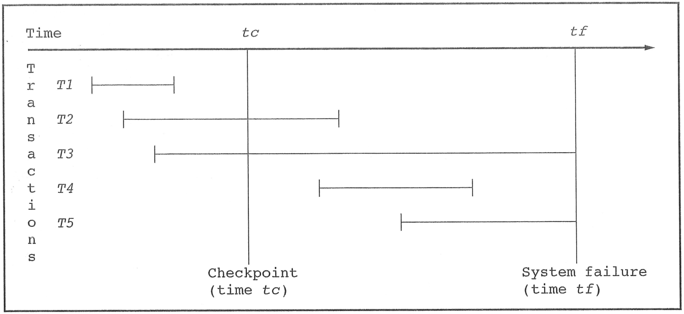
- T_1:

Recovery management



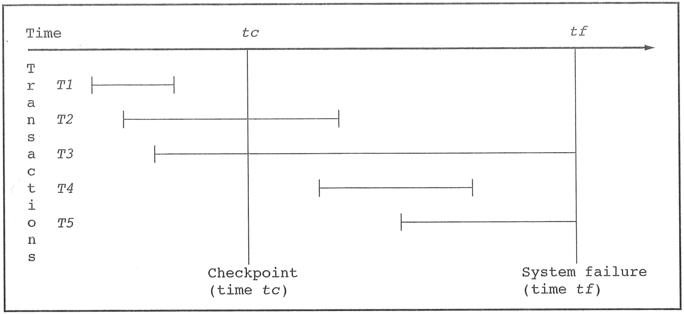
- T_1: ignore
- T_2:

Recovery management



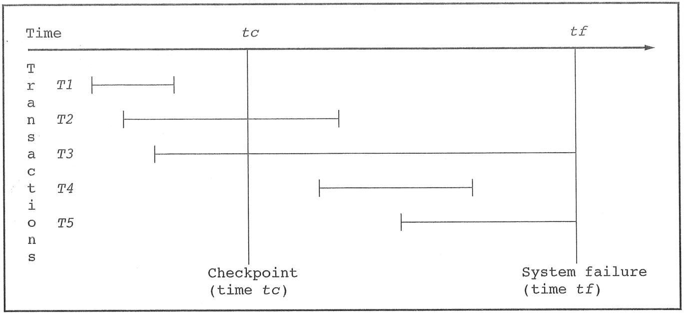
- T_1: ignore
- T_2: redo
- T_3:

Recovery management



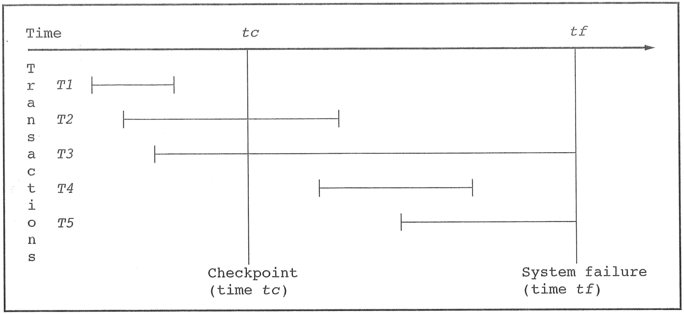
- T_1: ignore
- T_2: redo
- T_3: undo
- T_4:

Recovery management



- T_1: ignore
- T_2: redo
- T_3: undo
- T_4: redo
- T_5:

Recovery management



- T_1: ignore
- T_2: redo
- T_3: undo
- T_4: redo
- T_5: undo

Recovery management: restart procedure

1. $REDO \leftarrow \emptyset$
2. locate most recent checkpoint C in log
3. $UNDO \leftarrow \{T \mid T \text{ in progress at } C\}$

Recovery management: restart procedure

1. $REDO \leftarrow \emptyset$
2. locate most recent checkpoint C in log
3. $UNDO \leftarrow \{T \mid T \text{ in progress at } C\}$
4. while scanning forward from C to end of log
    4.1 if `BEGIN TRANSACTION` found for transaction T
    - $UNDO \leftarrow UNDO \cup \{T\}$

    4.2 if `COMMIT` found for transaction T,
    - $UNDO \leftarrow UNDO - \{T\}$
    - $REDO \leftarrow REDO \cup \{T\}$

Recovery management: restart procedure

1. $REDO \leftarrow \emptyset$
2. locate most recent checkpoint C in log
3. $UNDO \leftarrow \{T \mid T \text{ in progress at } C\}$
4. while scanning forward from C to end of log
   4.1 if `BEGIN TRANSACTION` found for transaction T
      - $UNDO \leftarrow UNDO \cup \{T\}$

   4.2 if `COMMIT` found for transaction T,
      - $UNDO \leftarrow UNDO - \{T\}$
      - $REDO \leftarrow REDO \cup \{T\}$
5. working backwards in the log, undo all actions of each $T \in UNDO$

Recovery management: restart procedure

1. $REDO \leftarrow \emptyset$
2. locate most recent checkpoint C in log
3. $UNDO \leftarrow \{T \mid T \text{ in progress at } C\}$
4. while scanning forward from C to end of log
   4.1 if `BEGIN TRANSACTION` found for transaction T
   - $UNDO \leftarrow UNDO \cup \{T\}$

   4.2 if `COMMIT` found for transaction T,
   - $UNDO \leftarrow UNDO - \{T\}$
   - $REDO \leftarrow REDO \cup \{T\}$
5. working backwards in the log, undo all actions of each $T \in UNDO$
6. working forwards in the log, redo all actions of each $T \in REDO$

Recovery management: restart procedure

1. $REDO \leftarrow \emptyset$
2. locate most recent checkpoint C in log
3. $UNDO \leftarrow \{T \mid T \text{ in progress at } C\}$
4. while scanning forward from C to end of log
   4.1 if `BEGIN TRANSACTION` found for transaction T
      - $UNDO \leftarrow UNDO \cup \{T\}$

   4.2 if `COMMIT` found for transaction T,
      - $UNDO \leftarrow UNDO - \{T\}$
      - $REDO \leftarrow REDO \cup \{T\}$
5. working backwards in the log, undo all actions of each $T \in UNDO$
6. working forwards in the log, redo all actions of each $T \in REDO$

note that this procedure itself is a transaction ...

Concurrency control

Let's now look at ensuring *isolation* and *consistency* of transactions, the responsibility of the transaction manager/scheduler

Concurrency control

Let's now look at ensuring *isolation* and *consistency* of transactions, the responsibility of the transaction manager/scheduler

Consider the ways in which transactions can interfere with each other, via shared data objects

Obviously, two transactions only reading the same object can't interfere with each other.

Concurrency control

Let's now look at ensuring *isolation* and *consistency* of transactions, the responsibility of the transaction manager/scheduler

Consider the ways in which transactions can interfere with each other, via shared data objects

Obviously, two transactions only reading the same object can't interfere with each other.

Two actions are said to conflict on the same data object if at least one of them is a write: write-read (WR), read-write (RW), write-write (WW)

Dirty reads (WR)

Consider

- A and B are both initially 200 €
- T_1 which transfers 100 € from A to B
- T_2 which increases both A and B by 6%

Dirty reads (WR)

Consider

- A and B are both initially 200 €
- T_1 which transfers 100 € from A to B
- T_2 which increases both A and B by 6%

with the following execution history

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
	$W_2(A)$
	$R_2(B)$
	$W_2(B)$
	commit_2
$R_1(B)$	
$W_1(B)$	
commit_1	

Dirty reads (WR)

Consider

- A and B are both initially 200 €
- T_1 which transfers 100 € from A to B
- T_2 which increases both A and B by 6%

with the following execution history

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
	$W_2(A)$
	$R_2(B)$
	$W_2(B)$
	commit_2
$R_1(B)$	
$W_1(B)$	
commit_1	

what are the final values of A and B?

Dirty reads (WR)

Final values: $A = 106$ and $B = 312$

does this correspond to the isolated execution of T_1 and T_2?

- if T_1 executed in isolation, followed by T_2:
  - final values would be $A = 106$ and $B = 318$
- if T_2 executed in isolation, followed by T_1:
  - final values would be $A = 112$ and $B = 312$

Dirty reads (WR)

Final values: $A = 106$ and $B = 312$

does this correspond to the isolated execution of T_1 and T_2?

- if T_1 executed in isolation, followed by T_2:
  - final values would be $A = 106$ and $B = 318$
- if T_2 executed in isolation, followed by T_1:
  - final values would be $A = 112$ and $B = 312$

No! This interference was caused by T_2 reading uncommitted data, also known as a dirty read

Unrepeatable reads (RW)

Consider

- A is initially 5
- T_1 increments A by 1
- T_2 decrements A by 1

Unrepeatable reads (RW)

Consider

- A is initially 5
- T_1 increments A by 1
- T_2 decrements A by 1

with the following execution history

$$
\begin{array}{r|l}
R_1(A) & \\
 & R_2(A) \\
W_1(A) & \\
 & W_2(A) \\
 & \mathrm{commit}_2 \\
\mathrm{commit}_1 &
\end{array}
$$

Unrepeatable reads (RW)

Consider

- A is initially 5
- T_1 increments A by 1
- T_2 decrements A by 1

with the following execution history

$R_1(A)$	
	$R_2(A)$
$W_1(A)$	
	$W_2(A)$
	commit_2
commit_1	

what is the final value of A?

Unreadable reads (RW)

Consider

- A is initially 5
- T_1 increments A by 1
- T_2 decrements A by 1

with the following execution history

$$
\begin{array}{r|l}
R_1(A) & \\
 & R_2(A) \\
W_1(A) & \\
 & W_2(A) \\
 & \text{commit}_2 \\
\text{commit}_1 &
\end{array}
$$

what is the final value of A?

how does this compare with isolated execution?

Dirty writes (WW)

Harry and Larry must always have equal salaries.
Consider

- T_1 sets H and L's salaries to 1000 €
- T_2 sets H and L's salaries to 2000 €

Dirty writes (WW)

Harry and Larry must always have equal salaries.
Consider

- T_1 sets H and L's salaries to 1000 €
- T_2 sets H and L's salaries to 2000 €

with the following execution history

$W_1(H)$	
	$W_2(L)$
$W_1(L)$	
	$W_2(H)$
	commit_2
commit_1	

Dirty writes (WW)

Harry and Larry must always have equal salaries.
Consider

- T_1 sets H and L's salaries to 1000 €
- T_2 sets H and L's salaries to 2000 €

with the following execution history

$W_1(H)$	
	$W_2(L)$
$W_1(L)$	
	$W_2(H)$
	commit_2
commit_1	

what are the final salaries?

Dirty writes (WW)

Harry and Larry must always have equal salaries.
Consider

- T_1 sets H and L's salaries to 1000 €
- T_2 sets H and L's salaries to 2000 €

with the following execution history

$W_1(H)$	
	$W_2(L)$
$W_1(L)$	
	$W_2(H)$
	commit_2
commit_1	

what are the final salaries?
how does this compare with isolated execution?

Concurrency control

How do we avoid these conflicts?

- we could only allow serial schedules,
  - i.e., for every pair of transactions, all of the actions of one transaction execute before any of the actions of the other
  - i.e., no two transactions are interleaved

but this is too restrictive.

Concurrency control

How do we avoid these conflicts?

- we could only allow serial schedules,
  - i.e., for every pair of transactions, all of the actions of one transaction execute before any of the actions of the other
  - i.e., no two transactions are interleaved

  but this is too restrictive.
- what we'd like instead is to just enforce *serial behavior/outcomes* for schedules
  - i.e., those which produce the same output and have the same effect on the stable DB as some complete serial schedule of the same transactions

  also known as serializable schedules

Concurrency control

We will say two schedules are conflict equivalent if

1. they involve the same set of actions of the same set of transactions, and
2. they order every pair of conflicting actions of two committed transactions in the same way.

Concurrency control

We will say two schedules are conflict equivalent if

1. they involve the same set of actions of the same set of transactions, and
2. they order every pair of conflicting actions of two committed transactions in the same way.

example. consider $T_1 = \langle R(A), R(B), W(B), C \rangle$ and $T_2 = \langle W(A), C \rangle$. Then the following schedules are conflict equivalent.

$$
\begin{aligned}
S_1 &= \langle T_1 : R(A), T_2 : W(A), T_2 : C, T_1 : R(B), T_1 : W(B), T_1 : C \rangle \\
S_2 &= \langle T_1 : R(A), T_1 : R(B), T_2 : W(A), T_2 : C, T_1 : W(B), T_1 : C \rangle
\end{aligned}
$$

Concurrency control

We will say a schedule S is conflict serializable if it is conflict equivalent to some serial schedule S'.

Concurrency control

We will say a schedule S is conflict serializable if it is conflict equivalent to some serial schedule S'. Note: every conflict serializable schedule is serializable.

Concurrency control

We will say a schedule S is conflict serializable if it is conflict equivalent to some serial schedule S'. Note: every conflict serializable schedule is serializable.

- seems impractical to check S, as there are $\mathcal{O}(n!)$ serial schedules over n transactions ...

Concurrency control

We will say a schedule S is conflict serializable if it is conflict equivalent to some serial schedule S'. Note: every conflict serializable schedule is serializable.

- seems impractical to check S, as there are $\mathcal{O}(n!)$ serial schedules over n transactions ...

A nice way to capture potential conflicts between transactions in a schedule S is the precedence graph $PG(S)$ for S, which has

- one node for each committed transaction of S, and
- an edge from node i to node j iff an action of transaction i precedes and conflicts with one of the actions of transaction j.

Concurrency control

example. recall

$$S_1 \quad = \quad \langle T_1 : R(A), T_2 : W(A), T_2 : C, T_1 : R(B), T_1 : W(B), T_1 : C \rangle$$

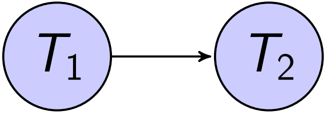
which has $PG(S_1)$ as

Concurrency control

example. recall

$$S_1 \quad = \quad \langle T_1 : R(A), T_2 : W(A), T_2 : C, T_1 : R(B), T_1 : W(B), T_1 : C \rangle$$

which has $PG(S_1)$ as



Concurrency control

example. recall Harry and Larry's salary updates

$$S_{HL} \quad = \quad \langle T_1 : W(H), T_2 : W(L), T_1 : W(L), T_2 : W(H), T_2 : C, T_1 : C \rangle$$

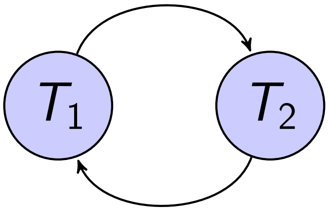
which has $PG(S_{HL})$ as

Concurrency control

example. recall Harry and Larry's salary updates

$$S_{HL} = \langle T_1 : W(H), T_2 : W(L), T_1 : W(L), T_2 : W(H), T_2 : C, T_1 : C \rangle$$

which has $PG(S_{HL})$ as



Concurrency control

This generalizes nicely as follows.

Serializability Theorem. A schedule S is conflict serializable if and only if $PG(S)$ is acyclic.

Concurrency control

exercise. Is the following schedule, over three transactions, conflict serializable?

$$T_1 : R(A), T_1 : W(B), T_2 : R(B), T_2 : R(C), T_3 : R(A), T_3 : W(C),$$
$$T_3 : W(E), T_1 : R(E), T_2 : W(D), T_3 : W(F),$$
$$T_2 : C, T_1 : C, T_3 : C$$

Concurrency control

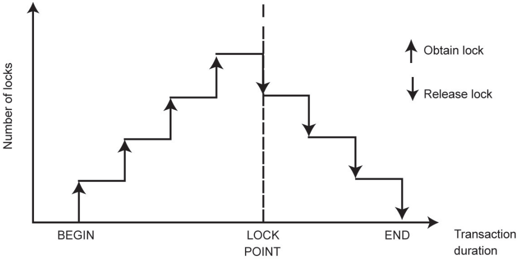
How can the transaction manager implement the Serializability Theorem, since it is still too time consuming to (statically) check and enforce such schedules?

Concurrency control: 2PL

… with two phase locking protocol, which allows concurrency while enforcing serial behavior/effect

A schedule satisfies the 2PL protocol if

1. each transaction first requests and waits for a shared (resp., exclusive) lock if it wants to read (resp., write) a DB object, and
2. a transaction cannot request additional locks once it releases any lock.



Concurrency control: 2PL

fact. If schedule S satisfies 2PL, then S is conflict serializable.

Concurrency control: 2PL

fact. If schedule S satisfies 2PL, then S is conflict serializable.

intuitively, an equivalent serial order of transactions is given by the order in which they enter their "shrinking" phases

- e.g., if $T_1 \rightarrow T_2$, then T_1 must release its lock first.

Concurrency control: 2PL - lock mgmt

We of course will need a lock manager to queue and grant lock requests according to the following compatibility matrix

Concurrency control: 2PL - lock mgmt

We of course will need a lock manager to queue and grant lock requests according to the following compatibility matrix

	read request	write request
read request	compatible	not compatible
write request	not compatible	not compatible

A read request is a request for a shared read-only lock

A write request is a request for an exclusive read/write lock

Concurrency control: 2PL - lock mgmt

We of course will need a lock manager to queue and grant lock requests according to the following compatibility matrix

	read request	write request
read request	compatible	not compatible
write request	not compatible	not compatible

A read request is a request for a shared read-only lock

A write request is a request for an exclusive read/write lock

Compatible means two transactions that request these locks to access the same data item can obtain these locks on that data item at the same time

Concurrency control: 2PL - lock mgmt

The lock manager must ensure that no transactions are starved, i.e., made to queue indefinitely due to transaction load handling

Concurrency control: 2PL - lock mgmt

The lock manager must ensure that no transactions are starved, i.e., made to queue indefinitely due to transaction load handling

- this can be addressed by ensuring that earlier requests take precedence over later requests

Concurrency control: 2PL - lock mgmt

The lock manager must ensure that no transactions are starved, i.e., made to queue indefinitely due to transaction load handling

- this can be addressed by ensuring that earlier requests take precedence over later requests

and that deadlocks are resolved

- for example, if transaction T_1 holds an exclusive lock on A and then requests a shared lock on B, while transaction T_2 holds an exclusive lock on B and then requests a shared lock on A

Concurrency control: 2PL - lock mgmt

The lock manager must ensure that no transactions are starved, i.e., made to queue indefinitely due to transaction load handling

- this can be addressed by ensuring that earlier requests take precedence over later requests

and that deadlocks are resolved

- for example, if transaction T_1 holds an exclusive lock on A and then requests a shared lock on B, while transaction T_2 holds an exclusive lock on B and then requests a shared lock on A

Once present, deadlocks are permanent and must be addressed by outside intervention

Concurrency control: 2PL - lock mgmt

Two basic approaches for handling deadlocks

- prevention
- detection and recovery

Concurrency control: 2PL - lock mgmt

Deadlock prevention. Allows transactions to be forcefully aborted (i.e., rolled back). Suppose T_i requests a lock held by T_j.

- Under a **wait-die scheme** if T_i is older than T_j, then T_i waits. Otherwise T_i is aborted and restarted with the same timestamp.
- Under a **wound-wait scheme** if T_i is older than T_j, then T_j is aborted and restarted with the same timestamp. Otherwise T_i waits.

Concurrency control: 2PL - lock mgmt

Deadlock prevention. Allows transactions to be forcefully aborted (i.e., rolled back). Suppose T_i requests a lock held by T_j.

- Under a **wait-die scheme** if T_i is older than T_j, then T_i waits. Otherwise T_i is aborted and restarted with the same timestamp.
- Under a **wound-wait scheme** if T_i is older than T_j, then T_j is aborted and restarted with the same timestamp. Otherwise T_i waits.
  In other words, granted locks can be preempted, unlike under wait-die.

Concurrency control: 2PL - lock mgmt

Deadlock prevention. Allows transactions to be forcefully aborted (i.e., rolled back). Suppose T_i requests a lock held by T_j.

- Under a **wait-die scheme** if T_i is older than T_j, then T_i waits. Otherwise T_i is aborted and restarted with the same timestamp.
- Under a **wound-wait scheme** if T_i is older than T_j, then T_j is aborted and restarted with the same timestamp. Otherwise T_i waits.
  In other words, granted locks can be preempted, unlike under wait-die.

Deadlocks can also be handled by timeouts on lock requests, but it is difficult in practice to tune timeout length

Concurrency control: 2PL - lock mgmt

Deadlock detection and recovery. A deadlock can be detected by a cycle in the wait-for graph for the transactions, having a node for each transaction and an edge from T_i to T_j iff T_i is waiting for T_j to release a lock on some object

Concurrency control: 2PL - lock mgmt

Deadlock detection and recovery. A deadlock can be detected by a cycle in the wait-for graph for the transactions, having a node for each transaction and an edge from T_i to T_j iff T_i is waiting for T_j to release a lock on some object

- in our example above, there is an edge from T_1 to T_2 and an edge from T_2 to T_1

Concurrency control: 2PL - lock mgmt

Deadlock detection and recovery. A deadlock can be detected by a cycle in the wait-for graph for the transactions, having a node for each transaction and an edge from T_i to T_j iff T_i is waiting for T_j to release a lock on some object

- in our example above, there is an edge from T_1 to T_2 and an edge from T_2 to T_1

So, with some tuned frequency, we perform cycle detection in the WFG, and then break cycles by selecting a victim transaction(s) to abort

- based on some cost function of transaction duration, transaction size and expected remaining duration, number of times already aborted, cycle size, ...

Concurrency control: recoverability

the RM deals with *system* failures. We still need to deal with *transaction* failures (i.e., aborts)

Concurrency control: recoverability

the RM deals with *system* failures. We still need to deal with *transaction* failures (i.e., aborts)

to recover, the TM must undo all of an aborted transactions writes, replacing updated values with before-images from the log.

Concurrency control: recoverability

the RM deals with *system* failures. We still need to deal with *transaction* failures (i.e., aborts)

to recover, the TM must undo all of an aborted transactions writes, replacing updated values with before-images from the log.

unfortunately, this isn't always possible ...

Concurrency control: recoverability

Consider

- A and B are both initially 1
- T_1 sets A to 2
- T_2 sets B to $A + B$

Concurrency control: recoverability

Consider

- A and B are both initially 1
- T_1 sets A to 2
- T_2 sets B to $A + B$

with the following execution history

$$
\begin{array}{r|l}
W_1(A) & \\
 & R_2(A) \\
 & R_2(B) \\
 & W_2(B) \\
 & \text{commit}_2 \\
\text{abort}_1 &
\end{array}
$$

Concurrency control: recoverability

Consider

- A and B are both initially 1
- T_1 sets A to 2
- T_2 sets B to $A + B$

with the following execution history

$$\begin{array}{r|l} W_1(A) & \\ & R_2(A) \\ & R_2(B) \\ & W_2(B) \\ & \mathrm{commit}_2 \\ \mathrm{abort}_1 & \end{array}$$

it is impossible to recover from this history!

T_2 shouldn't commit before T_1 ...

Concurrency control: recoverability

A schedule is recoverable if, for every transaction T that commits, T's commit follows the commit of every transaction whose changes T read.

Concurrency control: recoverability

A schedule is recoverable if, for every transaction T that commits, T's commit follows the commit of every transaction whose changes T read.

this is the bare minimum we require for isolation and consistency, to deal with aborts.

Concurrency control: recoverability

A schedule is recoverable if, for every transaction T that commits, T's commit follows the commit of every transaction whose changes T read.

this is the bare minimum we require for isolation and consistency, to deal with aborts.

however, in practice this is still not enough ....

Concurrency control: recoverability

Consider

- A and B are both initially 1
- T_1 sets A to 2
- T_2 sets B to $A + B$

Concurrency control: recoverability

Consider

- A and B are both initially 1
- T_1 sets A to 2
- T_2 sets B to $A + B$

with the following execution history

$$
\begin{array}{r|l}
W_1(A) & \\
 & R_2(A) \\
 & R_2(B) \\
 & W_2(B) \\
\text{abort}_1 &
\end{array}
$$

Concurrency control: recoverability

Consider

- A and B are both initially 1
- T_1 sets A to 2
- T_2 sets B to $A + B$

with the following execution history

$$
\begin{array}{r|l}
W_1(A) & \\
 & R_2(A) \\
 & R_2(B) \\
 & W_2(B) \\
\mathrm{abort}_1 &
\end{array}
$$

to recover, kill T_1, and restore $A = 1$ ...

Concurrency control: recoverability

Consider

- A and B are both initially 1
- T_1 sets A to 2
- T_2 sets B to $A + B$

with the following execution history

$$\begin{array}{r|l} W_1(A) & \\ & R_2(A) \\ & R_2(B) \\ & W_2(B) \\ \text{abort}_1 & \end{array}$$

to recover, kill T_1, and restore $A = 1$ ...
and, kill T_2 and restore $B = 1$

Concurrency control: recoverability

Consider

- A and B are both initially 1
- T_1 sets A to 2
- T_2 sets B to $A + B$

with the following execution history

$$\begin{array}{r|l} W_1(A) & \\ & R_2(A) \\ & R_2(B) \\ & W_2(B) \\ \text{abort}_1 & \end{array}$$

to recover, kill T_1, and restore $A = 1$ ...
and, kill T_2 and restore $B = 1$
i.e., a cascading abort

Concurrency control: recoverability

in general, uncontrollably many aborts are possible, which is unacceptable in practice

- hence, TM should disallow cascading aborts

Concurrency control: recoverability

in general, uncontrollably many aborts are possible, which is unacceptable in practice

- hence, TM should disallow cascading aborts

A schedule avoids cascading aborts (ACA) if it ensures that every transaction reads only those values that were written by committed transactions.

- also ensures recoverability

Concurrency control: recoverability

in general, uncontrollably many aborts are possible, which is unacceptable in practice

- hence, TM should disallow cascading aborts

A schedule avoids cascading aborts (ACA) if it ensures that every transaction reads only those values that were written by committed transactions.

- also ensures recoverability

Note that "recoverable" is a semantic notion, whereas ACA is a practical notion

Concurrency control: recoverability

Finally, consider

- A is initially 1
- T_1 sets A to 2
- T_2 sets A to 3

Concurrency control: recoverability

Finally, consider

- A is initially 1
- T_1 sets A to 2
- T_2 sets A to 3

with the following execution history

$$\begin{array}{r|l} W_1(A) & \\ & W_2(A) \\ \mathrm{abort}_1 & \end{array}$$

Concurrency control: recoverability

Finally, consider

- A is initially 1
- T_1 sets A to 2
- T_2 sets A to 3

with the following execution history

$$\begin{array}{l|l} W_1(A) & \\ & W_2(A) \\ \mathrm{abort}_1 & \end{array}$$

what is the value of A after undoing T_1?

Concurrency control: recoverability

Finally, consider

- A is initially 1
- T_1 sets A to 2
- T_2 sets A to 3

with the following execution history

$$\begin{array}{r|l} W_1(A) & \\ & W_2(A) \\ \mathrm{abort}_1 & \end{array}$$

what is the value of A after undoing T_1?
$A = 1$, whereas it should be 3!

Concurrency control: recoverability

Finally, consider

- A is initially 1
- T_1 sets A to 2
- T_2 sets A to 3

with the following execution history

$$\begin{array}{r|l} W_1(A) & \\ & W_2(A) \\ \text{abort}_1 & \end{array}$$

what is the value of A after undoing T_1?
$A = 1$, whereas it should be 3!
now suppose T_2 aborts.
then $A = 2$ (using pre-image), but it should be 1!

Concurrency control: recoverability

So, in order to handle aborts just using pre/post-images from the stable log, the TM should delay writes on an object until after all transactions previously issuing writes on that object have aborted/committed.

Concurrency control: recoverability

So, in order to handle aborts just using pre/post-images from the stable log, the TM should delay writes on an object until after all transactions previously issuing writes on that object have aborted/committed.

A schedule is strict if it ensures that every transaction reads and writes only those data objects that were written to by committed transactions.

- also ensures ACA, and hence recoverability

Concurrency control: S2PL

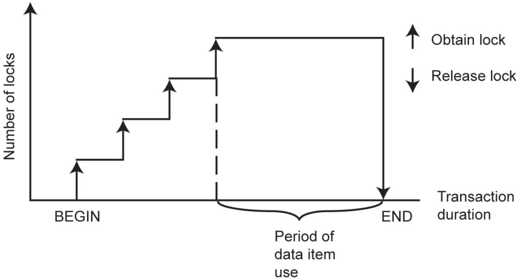
In practice, the strict two phase locking protocol is commonly followed to permit concurrency while enforcing serial and practically recoverable behavior/effect

- i.e., ensures strict and conflict serializable schedules

Concurrency control: S2PL

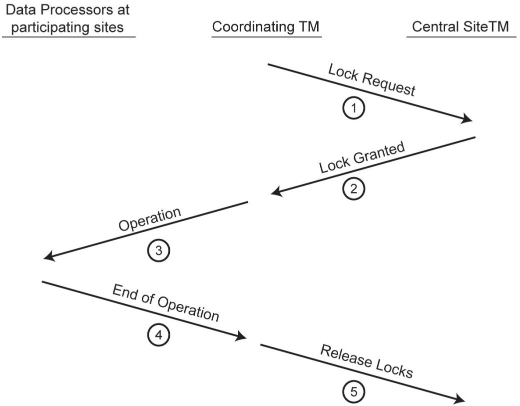
A schedule satisfies the S2PL protocol if

1. each transaction first requests and waits for a shared (resp., exclusive) lock if it wants to read (resp., write) a DB object, and
2. all locks held by a transaction are released when the transaction is complete (i.e., after commit/abort is acknowledged)



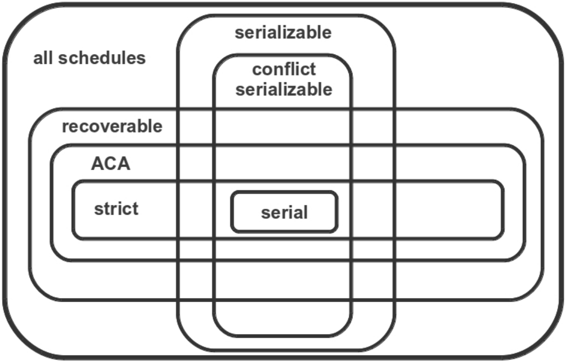
Concurrency control

Note that (S)2PL can be easily extended to distributed DBMSs

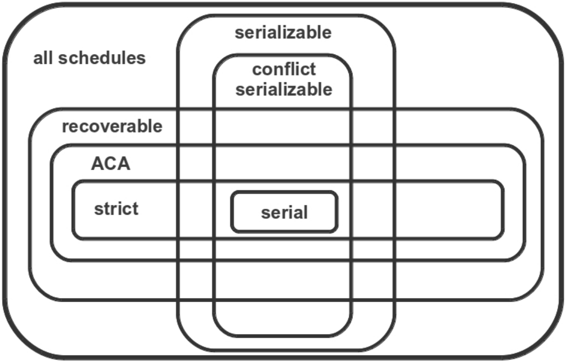


... and many more variants have been developed

Concurrency control

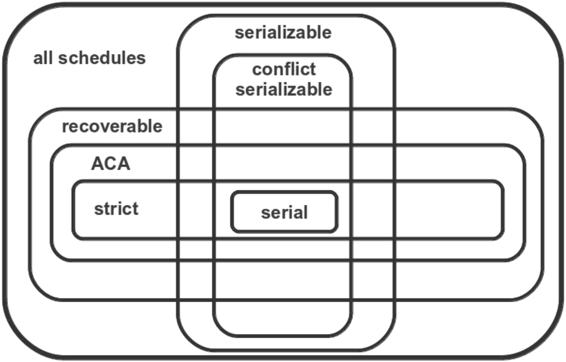


Concurrency control



Note that all classes above are distinct.

Concurrency control



Note that all classes above are distinct.

Exercise. demonstrate separation of the classes of serializable and conflict serializable transactions

Recap

overview of transactions

- concurrent units of consistent work
- ACID properties

overview of recovery management

- UNDO/REDO

overview of concurrency control

- serializable schedules, 2PL
- recoverable schedules, S2PL

Looking ahead ...

This Friday

- NoSQL and Graph databases

Next week

- Performance tuning, course summary, and exam review (Wednesday)
- First batch of project presentations: teams 1-8 (Friday)
- Physical hand-in of written assignment (Friday)

Credits

- our textbook
- Özsu & Valduriez, 2011
- Bernstein et al, 1987
- Bernstein et al, 2009
- Date, 2004