These lecture notes include some material from Professors Guagliardo, Bertossi, Kolaitis, Libkin, Vardi, Barland, McMahan

Transactions Management

Lecture Handout

Dr Eugenia Ternovska
Associate Professor

Simon Fraser University

Transactions

Transaction: a sequence of operations (reads or writes) on database objects such that

- all operations together form a **single logical unit of work**

Examples:

- transform money between accounts
- purchase a group of products
- register for a class

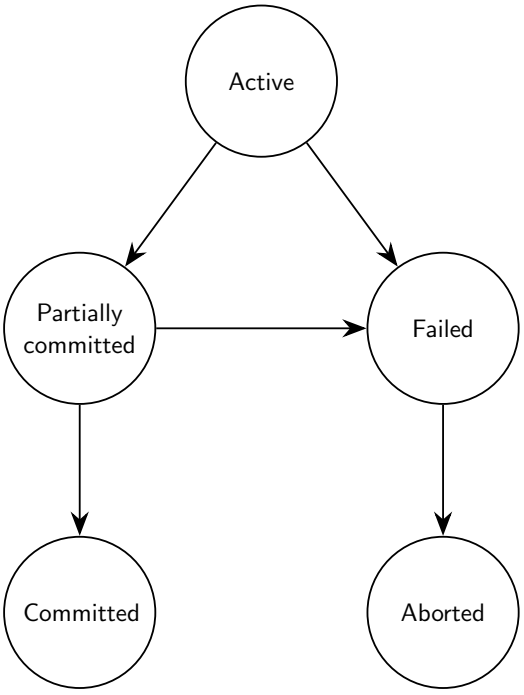
Example of a Transaction

Example

Transfer \$100 from account A to account B

1. Read balance from A into local buffer x
2. $x := x - 100$
3. Write new balance x to A
4. Read balance from B into local buffer y
5. $y := y + 100$
6. Write new balance y to B

Life-cycle of a transaction



Active
Normal execution state

Partially Committed
Last statement executed

Failed
Normal execution cannot proceed

Aborted
Rolled-back
Previous database state restored

Committed
Successful completion
Changes are permanent

The ACID Properties of Transactions

Atomicity
Either all operations are carried out or none are

Consistency
Successful execution of a transaction
leaves the database in a coherent state

Isolation
Each transaction is protected from the effects
of other transactions executed concurrently

Durability
On successful completion, changes persist

ACID: Atomic

Transaction activities are atomic (**all or nothing**):
would either occur completely or not at all

Two possible outcomes for a transaction:
- It commits: All the changes are made
- It aborts: No changes are made

Example

Transfer $100 from the account number 123 to the account 456

```
Step(1)     UPDATE AccountsSET balance = balance + 100
            WHERE accNumber = 456;

Step(2)    UPDATE AccountsSET balance = balance - 100
           WHERE accNumber = 123;
```

What happens if we stop after Step(1) but before Step(2)?

ACID: Consistent

The tables must always satisfy (user-specified) **integrity constraints**

Examples

- Account number is unique
- Stock amount cannot be negative
- Sum of debits and credits is 0

Programmer must make sure that a transaction takes a consistent state to a consistent state

ACID: Isolated

A transaction executes concurrently with other transactions, so

the effect should be as if each transaction executes **in isolation of the others**, i.e., one should not be able to observe changes from other transactions during the run

Example

```
SELECT seatNoFROM Flights
WHERE flightNo = 123
      AND flightDate = '2008-12-25'
      AND seatStatus = 'available';
UPDATE FlightsSET seatStatus = 'occupied'
WHERE flightNo = 123
      AND flightDate = '2008-12-25'
      AND seatNo = '22A';
```

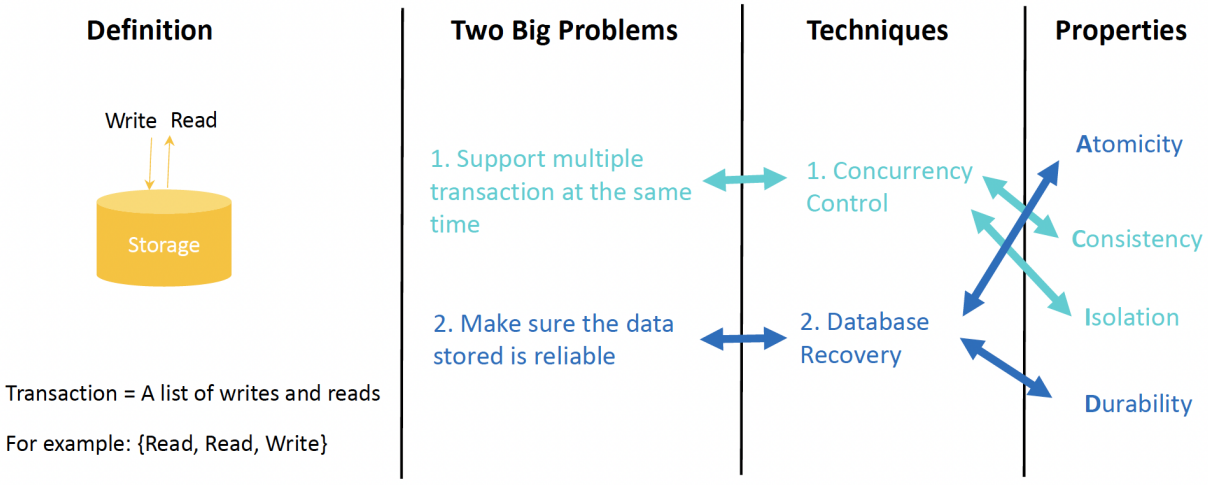
What happens if more than one person has the same request at the same time?

ACID: Durable

- The effect of a transaction must **continue to exist** (persist) after the transaction,
- and after the whole program has terminated,
- and even if there are power failures, crashes, etc.

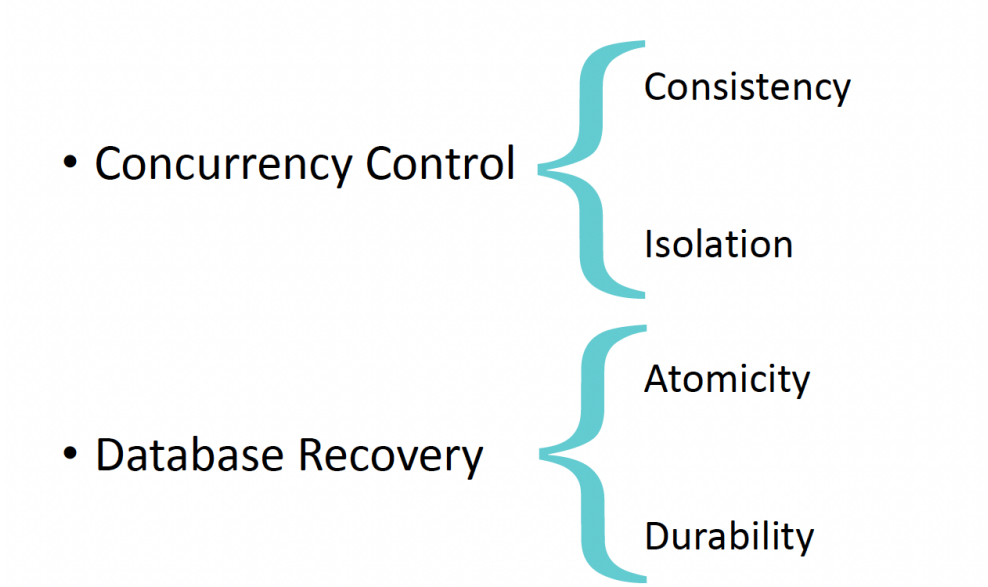
This Means: Write data to disk

Transaction Management



Reference: Slides, CMPT 354: Database Systems I (Jiannan Wang - SFU), Fall 2018

Transaction Management



Reference: Slides, CMPT 354: Database Systems I (Jiannan Wang - SFU), Fall 2018

Concurrency Control

Multiple transactions are allowed to run concurrently in the system

Advantages:

• Increased processor and disk utilization leading to better transaction throughput

• Reduced average response time for transactions
– short transactions need not wait behind long ones

Concurrency Control

The DBMS must handle concurrency so that:

- Isolation is maintained
  - Users must be able to execute each transaction **as if they were the only user**
  - DBMS handles the details of interleaving various transactions
- Consistency is maintained
  - Transactions must leave the DB in a **consistent state**
  - DBMS handles the details of enforcing integrity constraints

Schedules

Schedule: a sequence S of operations taken from the transactions T_i in a set of transactions $\{T_1, \ldots, T_n\}$
where the order of operations of each T_i is **the same** as in S

• Need to swap the control of transaction execution between multiple simultaneous transactions

Example:

• Execute a few instructions from transaction T_1, and a few instructions from transaction T_2

• Again, a few more instructions from transaction T_1 and transaction T_2, and so on

This action is called **interleaving** of transactions

Serial and Concurrent Schedules

A schedule is serial if all operations of each transaction are executed before or after all operations of another

Example

T_1 : op1, op2, op3
T_2 : op1, op2

Concurrent schedule

	T_1	T_2
1		op1
2	op1	
3	op2	
4		op2
5	op3	

Serial schedule

	T_1	T_2
1		op1
2		op2
3	op1	
4	op2	
5	op3	

Concurrency is implemented through **interleaving**

Motivation for Concurrency

- Typically more than one transaction runs on a system
- Each transation consists of many I/O and CPU operations
- We don't want to wait for a transaction to completely finish before executing another

Concurrent execution (through interleaving):

- increases throughput
- reduces response time

Motivating example

T_1 : transfer \$100 from account A to account B

T_2 : transfer 10 % of account A to account B

T_1
1. $x :=$ read(A)
2. $x := x - 100$
3. write(x, A)
4. $y :=$ read(B)
5. $y := y + 100$
6. write(y, B)

T_2
1. $x :=$ read(A)
2. $y := 0{,}1 * x$
3. $x := x - y$
4. write(x, A)
5. $z :=$ read(B)
6. $z := z + y$
7. write(z, B)

$A + B$ should not change:
Money is not created and does not disappear

Motivating example: Serial execution 1

	T_1	T_2	Database	
1	$x :=$ read(A)		$A = 1000$	$B = 1000$
2	$x := x - 100$		$A = 1000$	$B = 1000$
3	write(x, A)		$A = 900$	$B = 1000$
4	$y :=$ read(B)		$A = 900$	$B = 1000$
5	$y := y + 100$		$A = 900$	$B = 1000$
6	write(y, B)		$A = 900$	$B = 1100$
7		$x :=$ read(A)	$A = 900$	$B = 1100$
8		$y := 0{,}1 * x$	$A = 900$	$B = 1100$
9		$x := x - y$	$A = 900$	$B = 1100$
10		write(x, A)	$A = 810$	$B = 1100$
11		$z :=$ read(B)	$A = 810$	$B = 1100$
12		$z := z + y$	$A = 810$	$B = 1100$
13		write(z, B)	$A = 810$	$B = 1190$

Motivating example: Serial execution 2

	T_1	T_2	Database	
1		$x := \mathsf{read}(A)$	$A = 1000$	$B = 1000$
2		$y := 0{,}1 * x$	$A = 1000$	$B = 1000$
3		$x := x - y$	$A = 1000$	$B = 1000$
4		$\mathsf{write}(x, A)$	$A = 900$	$B = 1000$
5		$z := \mathsf{read}(B)$	$A = 900$	$B = 1000$
6		$z := z + y$	$A = 900$	$B = 1000$
7		$\mathsf{write}(z, B)$	$A = 900$	$B = 1100$
8	$x := \mathsf{read}(A)$		$A = 900$	$B = 1100$
9	$x := x - 100$		$A = 900$	$B = 1100$
10	$\mathsf{write}(x, A)$		$A = 800$	$B = 1100$
11	$y := \mathsf{read}(B)$		$A = 800$	$B = 1100$
12	$y := y + 100$		$A = 800$	$B = 1100$
13	$\mathsf{write}(y, B)$		$A = 800$	$B = 1200$

Motivating example: Concurrent execution 1

	T_1	T_2	Database	
1	$x := \mathsf{read}(A)$		$A = 1000$	$B = 1000$
2	$x := x - 100$		$A = 1000$	$B = 1000$
3	$\mathsf{write}(x, A)$		$A = 900$	$B = 1000$
4		$x := \mathsf{read}(A)$	$A = 900$	$B = 1000$
5		$y := 0{,}1 * x$	$A = 900$	$B = 1000$
6		$x := x - y$	$A = 900$	$B = 1000$
7		$\mathsf{write}(x, A)$	$A = 810$	$B = 1000$
8	$y := \mathsf{read}(B)$		$A = 810$	$B = 1000$
9	$y := y + 100$		$A = 810$	$B = 1000$
10	$\mathsf{write}(y, B)$		$A = 810$	$B = 1100$
11		$z := \mathsf{read}(B)$	$A = 810$	$B = 1100$
12		$z := z + y$	$A = 810$	$B = 1100$
13		$\mathsf{write}(z, B)$	$A = 810$	$B = 1190$

Motivating example: Concurrent execution 2

	T_1	T_2	Database	
1	$x :=$ read(A)		$A = 1000$	$B = 1000$
2	$x := x - 100$		$A = 1000$	$B = 1000$
3		$x :=$ read(A)	$A = 1000$	$B = 1000$
4		$y := 0{,}1 * x$	$A = 1000$	$B = 1000$
5		$x := x - y$	$A = 1000$	$B = 1000$
6		write(x, A)	$A = 900$	$B = 1000$
7	write(x, A)		$A = 900$	$B = 1000$
8	$y :=$ read(B)		$A = 900$	$B = 1000$
9	$y := y + 100$		$A = 900$	$B = 1000$
10	write(y, B)		$A = 900$	$B = 1100$
11		$z :=$ read(B)	$A = 900$	$B = 1100$
12		$z := z + y$	$A = 900$	$B = 1100$
13		write(z, B)	$A = 900$	$B = 1200$

We created $100 !!!

Interleaving

Interleaving transactions might lead to anomalous outcomes
Why do we do it then?

Several important reasons:
All concern large differences in **performance**

- Individual transactions might be slow
  - Avoid blocking other users during slow transactions
- Transactions waiting for locks
  - Avoid blocking other users while this transaction is also waiting
- Disk access may be slow
  - Let some transactions use CPUs while others accessing disk

Serializability

DBMS must maintain isolation and consistency (which, together, are called the **serializability** property)

• Serial schedule: a transaction only starts when the other transaction has finished execution

• A non-serial schedule of n transactions is said to be **serializable** if it is equivalent to a serial schedule of those n transactions

A serializable schedule never violates the serializability property

Every serial schedule is of course serializable

If a schedule is different from any serial order: **not serializable**

• Two schedules are **equivalent** if, for any database state, the effect of executing them on the database is identical

Transaction model

The only important operations in scheduling are **read** and **write**

r(A) read data item A

w(A) write data item A

Other operations do not affect the schedule

We represent transactions by a sequence of read/write operations

The transactions in the motivating example are represented as:

T_1 : r(A), w(A), r(B), w(B)

T_2 : r(A), w(A), r(B), w(B)

Transaction model: Schedules

The schedules in the motivating example are represented as:

Schedule 1

T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
r(B)	
w(B)	
	r(B)
	w(B)

Schedule 2

T_1	T_2
r(A)	
	r(A)
	w(A)
w(A)	
r(B)	
w(B)	
	r(B)
	w(B)

Schedule 1 is **equivalent to a serial execution**, Schedule 2 is not

Conflict Serializable Schedules

Two operations are **conflicting** if

- they refer to the same data item, and
- at least one of them is a write

Two **consecutive** non-conflicting operations in a schedule can be swapped

Three types of conflicts:

- Read-Write conflicts (RW)
- Write-Read conflicts (WR)
- Write-Write conflicts (WW)

Anomalies

- Dirty read (Occurring with / because of a WR conflict)

$w_1(A)$; $r_2(A)$; $w_2(A)$; $r_1(A)$;

Changed A
Data written by a transaction that has not yet committed
Dirty read is a read of dirty data written by another transaction
Risk: the transaction that wrote it might **abort** at a later time

- Unrepeatable read (Occurring with / because of a RW conflict)

$r_1(A)$; $r_2(A)$; $w_2(A)$; $r_1(A)$;

Changed A

- Lost update (Occurring because of a WW conflict)

$w_1(A)$; $w_2(A)$; $w_2(B)$; $w_1(B)$;

Changed B

Changed B

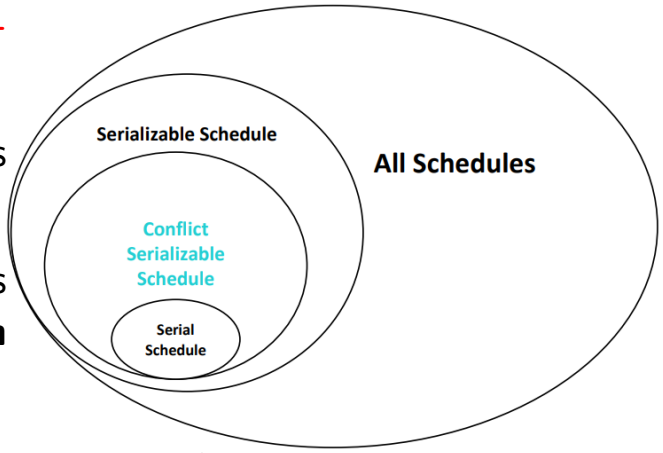
Reference: Slides, CMPT 354: Database Systems I (Jiannan Wang - SFU), Fall 2018

Conflict Serializing

Two schedules are **conflict equivalent** if

• They involve the same actions of the same transactions, and

• Every pair of conflicting actions of two transactions are **ordered in the same way**



Schedule S is called **conflict serializable** if S is conflict equivalent to some serial schedule

Any conflict serializable schedule can be transformed into a serial schedule by a sequence of swaps of non-conflicting operations

Figure from: Slides, CMPT 354: Database Systems I (Jiannan Wang - SFU), Fall 2018

Precedence graph

Captures all potential conflicts between transactions in a schedule

- Each node is a transaction
- There is an edge from T_i to T_j (for $T_i \neq T_j$) if an action of T_i **precedes** and **conflicts** with one of T_j's actions (= refer to the same data item and at least one of the actions is a Write)

A schedule is conflict serializable

if and only if

its precedence graph is acyclic

An equivalent serial schedule is given by any **topological sort** over the precedence graph

Precedence graph: Example

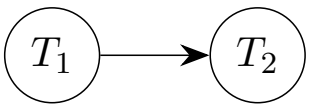
Schedule 1

T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
r(B)	
w(B)	
	r(B)
	w(B)

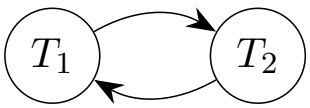
Schedule 2

T_1	T_2
r(A)	
	r(A)
	w(A)
w(A)	
r(B)	
w(B)	
	r(B)
	w(B)

Precedence graph



Precedence graph



	T_1	T_2	T_3	T_4
1			r(A)	
2		w(A)		
3	r(A)			
4			r(B)	
5			r(C)	
6		w(B)		
7	w(C)			
8				r(C)
9				w(C)

Schedules with aborted transactions (1)

We assumed transactions commit successfully after the last operation

But abort and commit must be taken explicitly into account

	T_1	T_2
1	r(A)	
2	w(A)	
3		r(A)
4		w(A)
5		r(B)
6		w(B)
7	Abort	

- T_2 read uncommited changes made by T_1
- But T_2 has not yet committed
- We can recover by aborting also T_2 (**cascading abort**)

Schedules with aborted transactions (2)

	T_1	T_2
1	r(A)	
2	w(A)	
3		r(A)
4		w(A)
5		r(B)
6		w(B)
7		Commit
8	Abort	

- T_2 read uncommited changes made by T_1
- But T_2 has already committed
- The schedule is **unrecoverable**

Recoverable schedules without cascading aborts

Transactions commit only after, and if,
all transactions whose changes they read commit

Lock-based concurrency control

Lock

- Bookkeeping object associated with a data item
- Tells whether the data item is available for read and/or write
- Owner: Transaction currently operating on the data item

Shared lock Data item is available for read to owner
Can be acquired by more than one transaction

Exclusive lock Data item is available for read/write to owner
Cannot be acquired by other transactions

Two locks on the same data item are **conflicting**
if one of them is exclusive

Transaction model with locks

Operations:

s(A)	**shared lock** on A is acquired
x(A)	**exclusive lock** on A is acquired
u(A)	lock on A is released
Abort	transaction aborts
Commit	transaction commits

In a schedule:

- A transaction cannot acquire a lock on A before all exclusive locks on A have been released
- A transaction cannot acquire an exclusive lock on A before all locks on A have been released

Examples of schedules with locking

Schedule 1

T_1	T_2
x(A)	
u(A)	
	x(A)
	u(A)
x(B)	
u(B)	
Commit	
	x(B)
	u(B)
	Commit

Schedule 2

T_1	T_2
s(A)	
	s(A)
	u(A)
u(A)	
	x(A)
	u(A)
x(A)	
x(B)	
u(B)	
	x(B)
	u(B)
	Commit
u(A)	
Commit	

	T_1	T_2	T_3	T_4
1	$\mathsf{s}(A)$			
2		$\mathsf{x}(C)$		
3	$\mathsf{x}(B)$			
4	$\mathsf{u}(A)$			
5		$\mathsf{x}(A)$		
6				$\mathsf{s}(B)$
7				$\mathsf{u}(B)$
8	$\mathsf{u}(B)$			
9		$\mathsf{u}(C)$		
10			$\mathsf{s}(C)$	
11			$\mathsf{u}(C)$	
12		$\mathsf{u}(A)$		

	T_1	T_2	T_3	T_4
1	$\mathsf{s}(A)$			
2		$\mathsf{x}(C)$		
3	$\mathsf{x}(B)$			
4	$\mathsf{u}(A)$			
5	$\mathsf{u}(B)$			
6		$\mathsf{x}(A)$		
7				$\mathsf{s}(B)$
8				$\mathsf{u}(B)$
9		$\mathsf{u}(C)$		
10			$\mathsf{s}(C)$	
11			$\mathsf{u}(C)$	
12		$\mathsf{u}(A)$		

Two-Phase Locking (2PL)

1. Before reading/writing a data item
   a transaction must acquire a shared/exclusive lock on it
2. A transaction cannot request additional locks
   once it releases **any** lock

Each transaction has

Growing phase when locks are acquired

Shrinking phase when locks are released

Every completed schedule of committed transactions
that follow the 2PL protocol is conflict serializable

2PL and aborted transactions

	T_1	T_2
1	x(A)	
2	u(A)	
3		x(A)
4		x(B)
5		u(A)
6		u(B)
7		Commit
8	Abort	

- T_1 and T_2 follow 2PL
- But T_1 cannot be undone
- The schedule is **unrecoverable**

Strict 2PL

1. Before reading/writing a data item
   a transaction must acquire a shared/exclusive lock on it
2. **All locks held by a transaction are released**
   after the transaction is completed (aborts or commits)

Ensures that

- The schedule is always recoverable
- All aborted transactions can be rolled back
  without cascading aborts
- The schedule consisting of the committed transactions
  is conflict serializable

Deadlocks

A transaction requesting a lock must wait
until all conflicting locks are released

We may get a cycle of "waits"

	T_1	T_2	T_3
1	s(A)		
2		x(B)	
3	req s(B)		
4			s(C)
5		req x(C)	
6			req x(A)

T_1 waits for T_2 , T_2 waits for T_3 , T_3 waits for T_1

Deadlock prevention

Each transaction is assigned a **priority** using a timestamp:
The older a transaction is, the higher priority it has

Suppose T_i requests a lock and T_j holds a conflicting lock

Two policies to prevent deadlocks:

Wait-die

Wound-wait

In both schemes, the higher priority transaction is never aborted

Starvation: a transaction keeps being aborted
because it never has sufficiently high priority

Solution: restart aborted transactions with their initial timestamp

Use of Timestamps for Deadlock Prevention(1)

Assume T_i requests a data item currently held by T_j

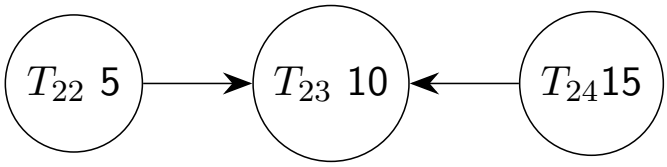
Wait-Die scheme
non-preemptive technique for deadlock prevention

if $TS(T_i) < TS(T_j)$ [T_i is older than T_j]
then T_i is allowed to **wait** until data item is available

otherwise, if $TS(T_i) > TS(T_j)$ [T_i is younger than T_j] then
abort T_i (T_i **dies**)
and restart it later with its initial timestamp but random delay

Wait Die scheme example

Transactions: T_{22}, T_{23}, T_{24}
Timestampts: 5, 10, 15



If T_{22} requests a data item held by T_{23} then T_{22} will **wait**

If T_{24} requests a data item held by T_{23} then T_{24} will be rolled back **(dies)**

Use of Timestamps for Deadlock Prevention (2)

Assume T_i requests a data item currently held by T_j

Wound Wait scheme

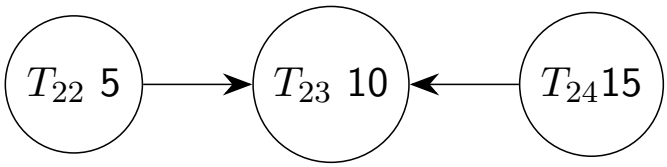
preemptive technique for deadlock prevention

if $TS(T_i) < TS(T_j)$ [T_i is older than T_j]
then **abort** T_j (T_i **wounds** T_j) and restart it (T_j) with with its initial timestamp

otherwise, if $TS(T_i) > TS(T_j)$ [T_i is younger than T_j] then T_i is allowed to **wait**

Wound Wait scheme example

Transactions: T_{22}, T_{23}, T_{24}
Timestampts: 5, 10, 15

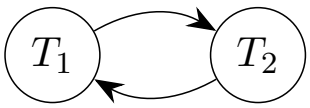


If T_{22} requests a data item held by T_{23} then the data item will be preempted from T_{23} and T_{23} will be rolled back **(wounded)**

If T_{24} requests a data item held by T_{23} then T_{24} will **wait**

Deadlock detection

(2) Waits-for graph

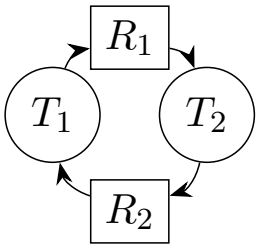


Nodes are active transactions

There is an edge from T_i to T_j (with $T_i \neq T_j$) if T_i waits for T_j to release a conflicting resource

is obtained from:

(1) Resource Allocation graph



T_1 is waiting for R_1
R_1 is allocated to T_2
T_2 is waiting for R_2
R_2 is allocated to T_1

Each cycle represents a deadlock

Recovering from deadlocks

Choose a minimal set of transactions such that rolling them back will make the waits-for graph acyclic

Crash recovery

The log (a.k.a. **trail** or **journal**)

Records every action executed on the database

Each log record has a unique ID called log sequence number (LSN)

Fields in a log record:

LSN	ID of the record
prevLSN	LSN of previous log record
transID	ID of the transaction
type	of action recorded
before	value before the change
after	value after the change

The state of the database is periodically recorded as a **checkpoint**

ARIES

Recovery algorithm used in major DBMSs
"Algorithm for Recovery and Isolation Exploiting Semantics"

- supports the needs of industrial strength transaction processing
- uses logs to record the progress of transactions and their actions

Works in three phases

1. Analysis
   - identify changes that have not been written to disk
   - identify active transactions at the time of crash
2. Redo
   - repeat all actions starting from latest checkpoint
   - restore the database to the state at the time of crash
3. Undo
   - undo actions of transactions that did not commit
   - the database reflects only actions of committed transactions

Principles behind ARIES

Write-Ahead Logging

Before writing a change to disk, a corresponding log record must be inserted and the log forced to stable storable condition

Repeating history during Redo

Actions before the crash are retraced to bring the database to the state it was when the system crashed

Logging changes during Undo

Changes made while undoing transactions are also logged (protection from further crashes)

Summary

Transactions

- ACID properties
- Concurrency Control

Schedules

- Interleaving (Performance)
- Serializability
- Conflict-Serializability

Transaction model with locks, 2PL, strict 2PL

Deadlock prevention:

- Use of timestampts for prevention: wait-die, wound wait schemas

Deadlock detection:

- Resource allocation graph
- Waits-for graph

Crash recovery in DBMSs

- The log (journal), log record
- ARIES recovery algorithm

Acknowledgements

[1] Database Systems: The Complete Book, 2nd Edition Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer WidomPrentice Hall, 2009

[2] Database System Concepts, Seventh EditionAvi Silberschatz, Henry F. Korth, S. SudarshanMcGraw-Hill, March 2019www.db-book.com

Additional references and resources used in preparation of this course are listed on the course webpage or mentioned in slides.