IN3020&4020 – Database Systems (2024)
Part 2: Transaction management
Lectures 16–18: Transaction processing concepts

11, 12, 18 March

Egor V. Kostylev and Martin Giese
IFI, University of Oslo
egork@ifi.uio.no



Big Picture (reminder)

Syllabus of the course:

- (Extended) Intro and SQL recap ✓

Part 1. Query processing in relational databases ✓

Part 2. Transaction management in relational databases

- Transaction processing concepts (3 lectures)
- Concurrency control techniques (3 lectures)
- Database recovery techniques (2 lectures)

Part 3. NoSQL DBMS

Materials to read about transaction management

Part 2. Transaction management in relational databases

Fundamentals of Database Systems by R. Elmasri and S.B. Navathe (7th edition): Part 9 (close to my presentation)

- Transaction processing concepts (3 lectures)
  Chapter 20
- Concurrency control techniques (3 lectures)
  Chapter 21
- Database recovery techniques (2 lectures)
  Chapter 22

Database Systems: the Complete Book by H. Garcia-Molina, J. Ullman & J. Widom (2nd edition): Sections 6.6, 19

etc.

Lecture Plan

1. Motivation and introduction
2. Desirable properties of transactions (ACID)
3. Transactions, schedules, and conflicts
4. Characterising schedules based on serialisability
5. Characterising schedules based on recoverability
6. Isolation levels

1. Motivation and introduction

Introduction

- Transactions (informally):

  mechanism for managing logical units of data processing: independent on others, all or nothing

- Examples:
  - a single data retrieval query
  - a sequence of data manipulation queries that should be executed together

- Transaction management (or processing) systems:

  systems with large databases and many concurrent users require high availability and fast response time

- Examples:
  - banking, airline booking, online retail, stocks

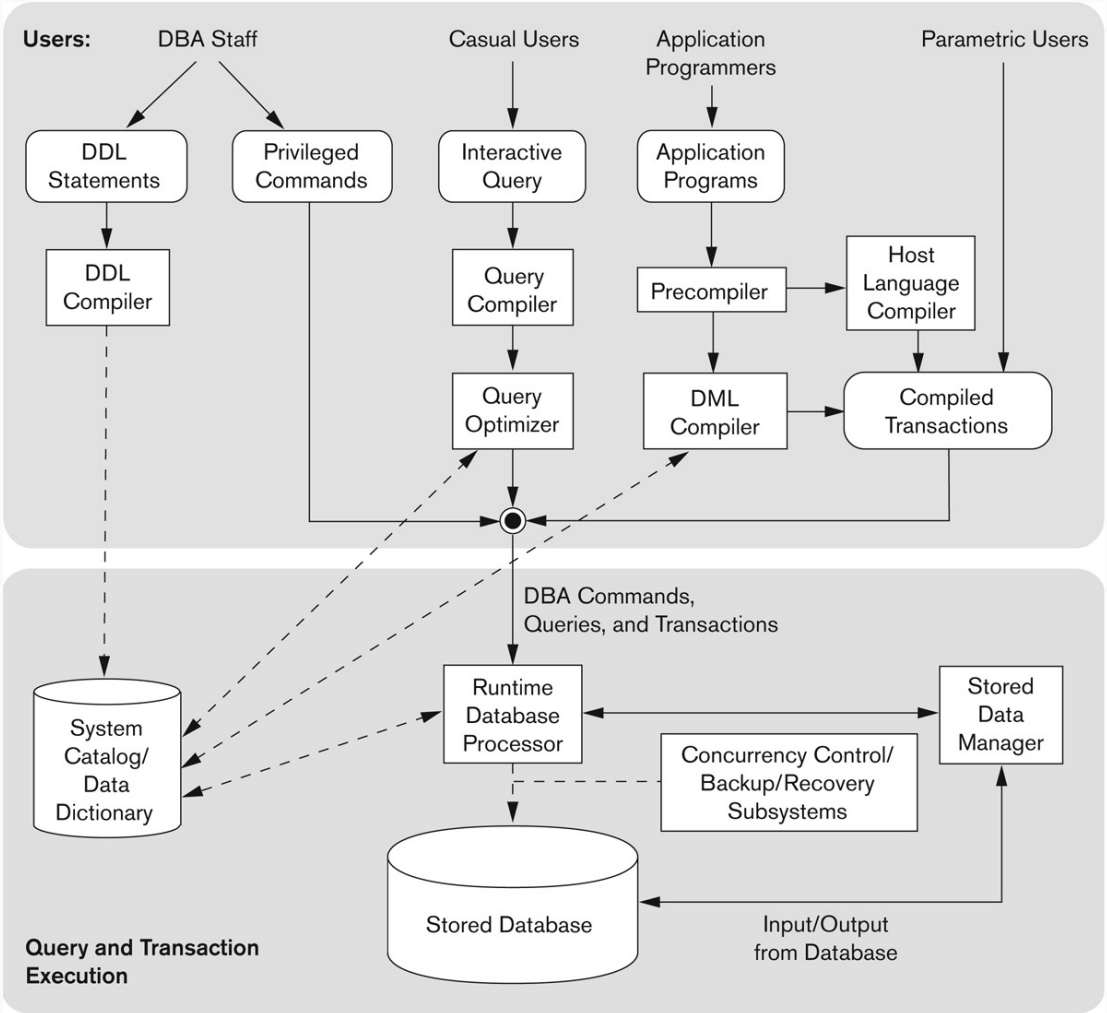
Single-user vs multi-user, challenges

- Single-user DBMS:
  at most one user at a time
  (usually, personal computer)
- Multi-user DBMS:
  many users (processes with own computation)
  with concurrent access to the same data
  (usually, servers with many CPUs,
  but may be handled by one CPU
  with interleaving concurrency)

Both need transaction management, with two challenges:

- concurrency control (transactions independent of each other)
- fail recovery (each transaction executed all or nothing)

The place of transaction management



Transaction idea

Transaction (informally):

- program that forms a logical unit of database processing
- has its own memory and computation ability
- includes one or more access operations to (shared) database
  (e.g., retrieval, insertion, deletion)
- for us, usually a sequence of operations
  (no branching, no cycles, etc.)
- boundaries can be specified by begin and end statements
  (if something goes wrong on the way, effects roll back)
- an application program may have several transactions
  (even parallel)
- may be read-only (only data retrieval) and read-write

Data items and granularity

In Part 2, we use a simple data model:

a database is a set of named data items

- can be a tuple (record), block, attribute value
- can be uniquely identified by name (address, tuple ID, etc.)
- can be read and written by name

Item granularity:

- the size of a data item

Transaction processing concepts are

independent of item granularity

Main transaction operations

`read`(X)

- reads an item named X from the global database into a local program variable named X
- includes finding the address of the block on the disk (or in cache) with X, and copying to a main memory buffer

`write`(X)

- writes the value of local program variable named X into the global database item named X
- includes finding the address of the block on the disk (or in cache) with X, read it to the local memory buffer, modify it, and write it back (to the disk or cache)

Program local operations (for example, update $X := X + 50$)

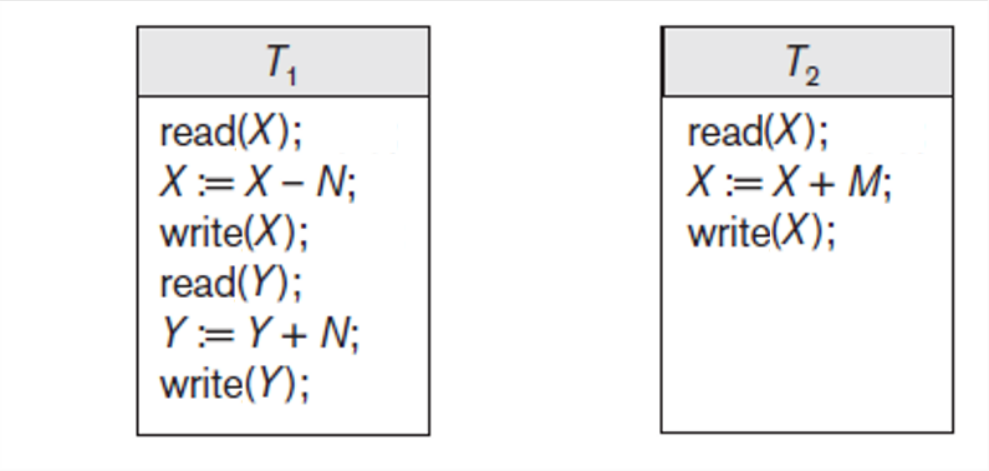
We will see other (transaction management) operations later

Example transactions

Let X and Y be the numbers of reserved seats in two flights (stored in a database)

Transaction T_1 transfers N reservations from X to Y

Transaction T_2 reserves M seats in X



Important:

- updates are local, the (only) database is not updated until the new value is written
- some commands are often omitted if they are not relevant (both local and transaction management)

Why concurrency control is needed?

Potential concurrency control problems (or phenomena):

1. Dirty write (or lost update)
2. Dirty read (or temporary update)
3. Non-repeatable read
4. Incorrect summary (or phantom phenomena)
5. ...

(Other variations of these problems can be found in the literature)

Example transactions

Let X and Y be the numbers of reserved seats in two flights

Transaction T_1 transfers N reservations from X to Y

Transaction T_2 reserves M seats in X

T_1

read(X);
$X := X - N$;
write(X);
read(Y);
$Y := Y + N$;
write(Y);

T_2

read(X);
$X := X + M$;
write(X);

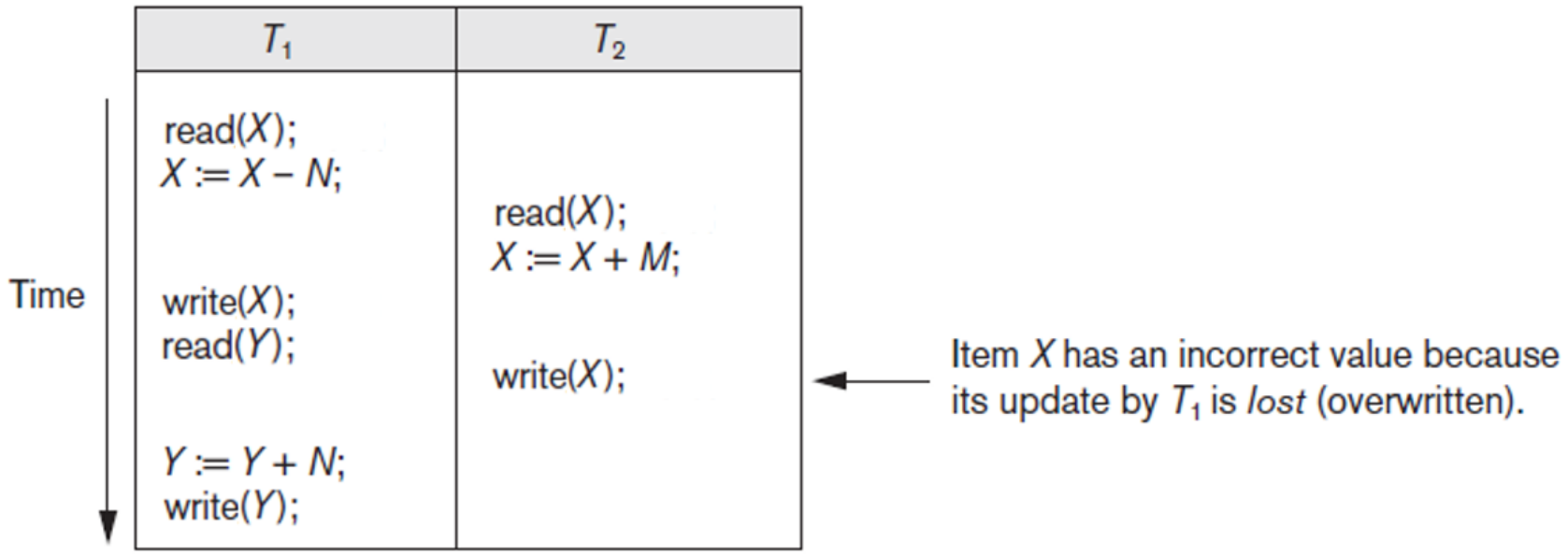
What can go wrong when interleave these transactions?

Dirty write (lost update)

Two transactions update (read and write) the same item, second update starts before the first is complete

(updates are interleaved)

Result: incorrect value (update is lost)



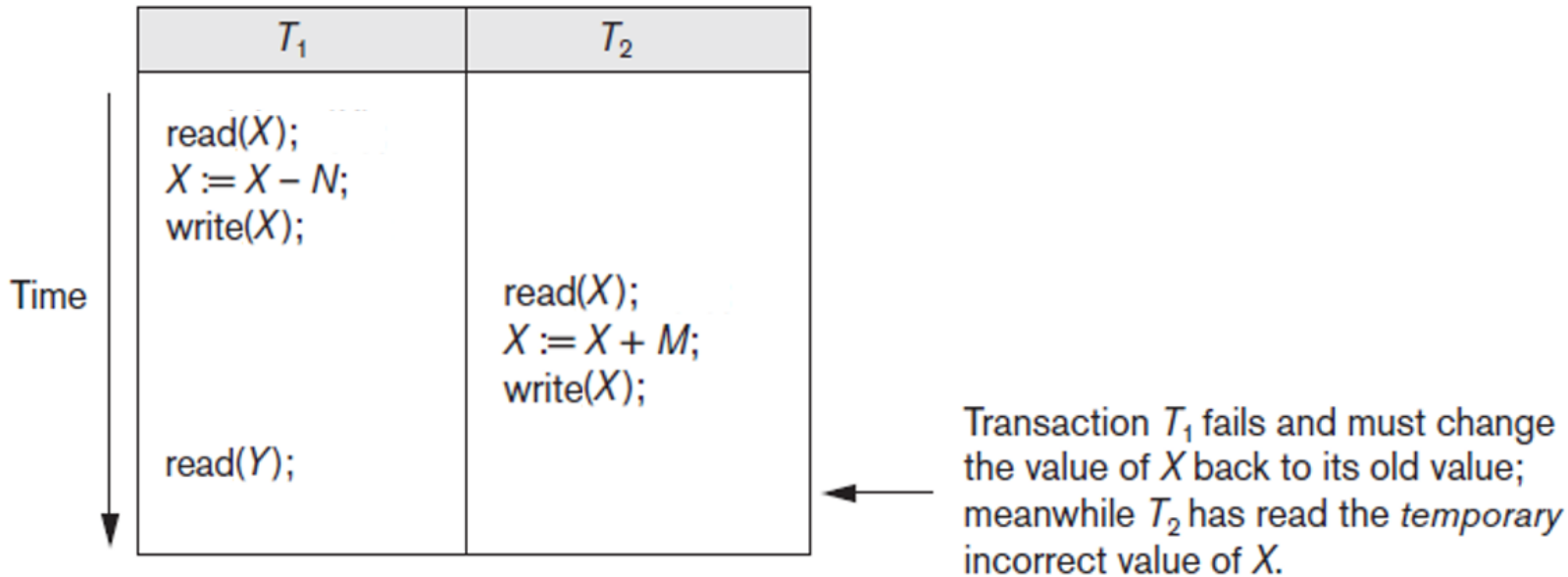
Dirty read (temporary update)

A transaction updates an item,

new value is used by another transaction,

the first transaction fails (see below) and its update is rolled back

Result: the second transaction relies on an incorrect value

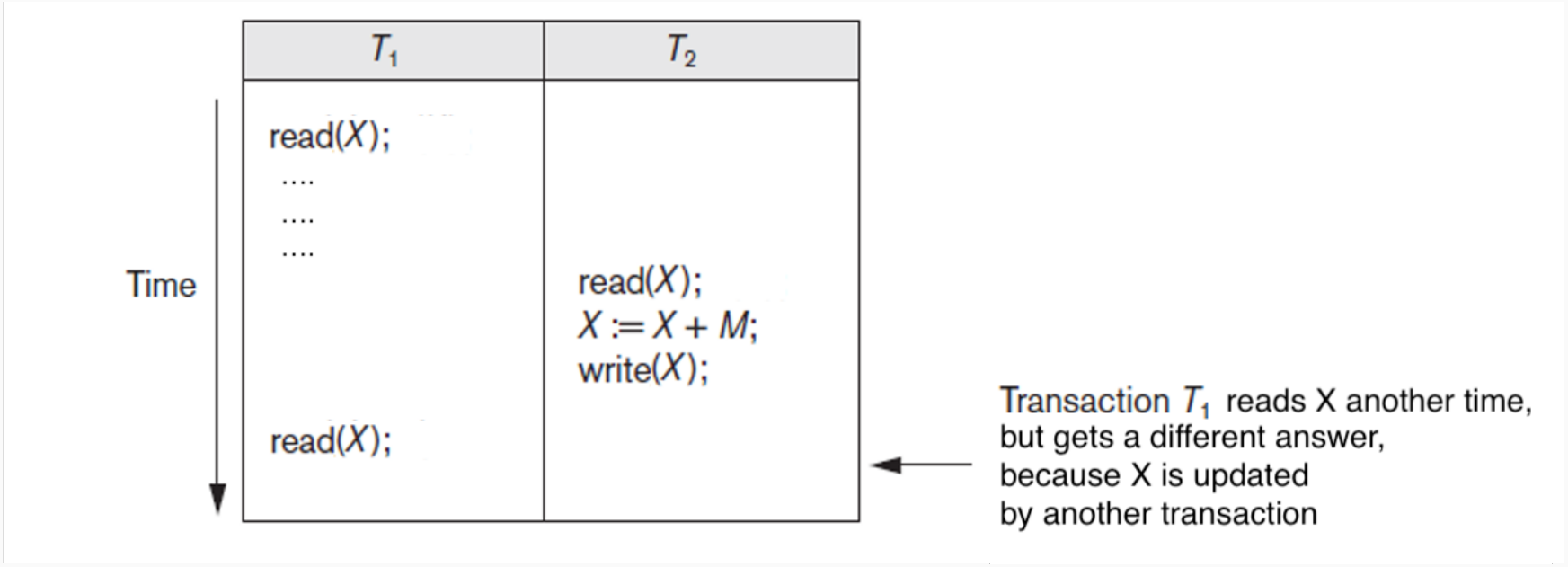


Non-repeatable read

One transaction reads the same item twice,

another transaction changes its value in between

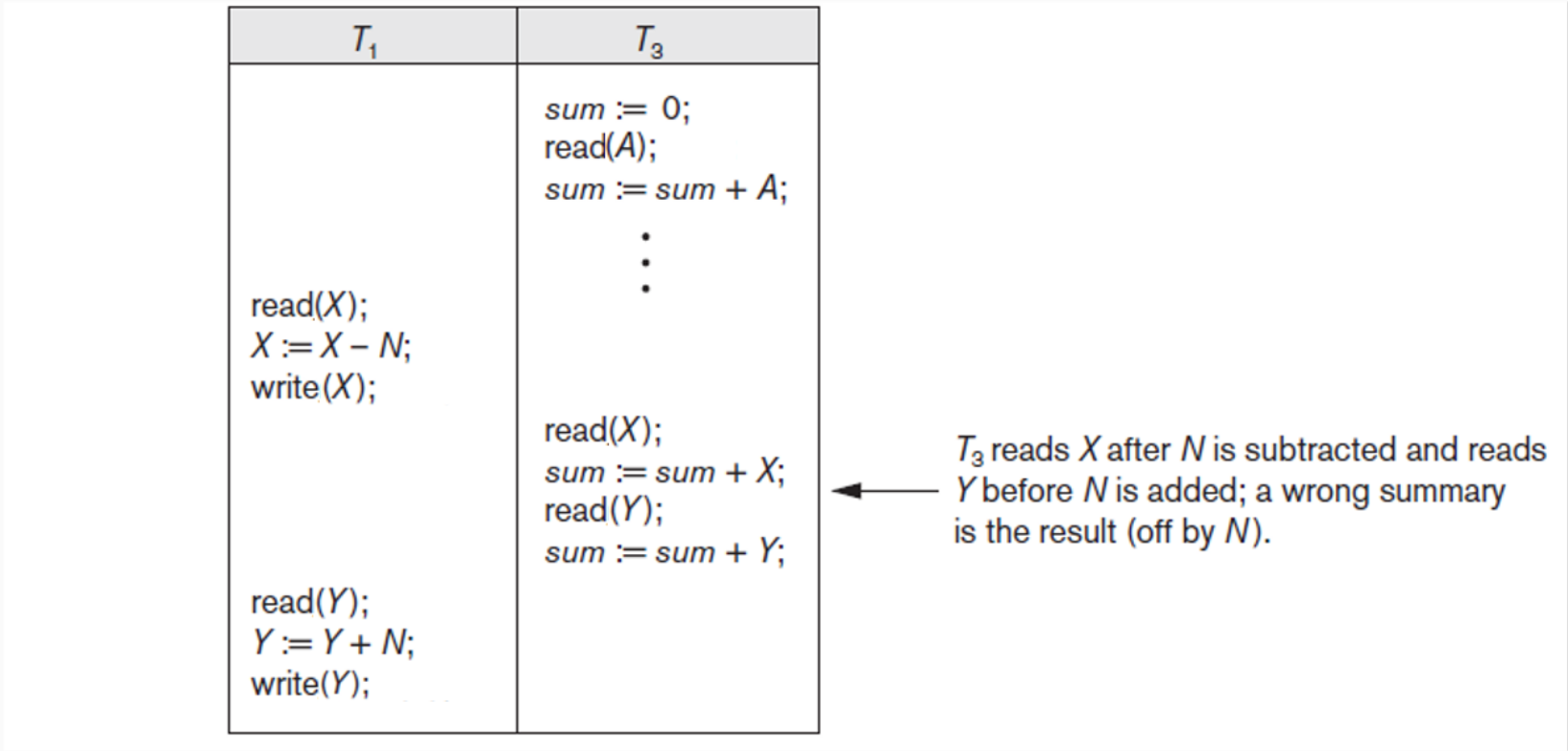
Result: the first transaction gets different values of the same item



Incorrect summary (Phantom phenomena)

One transaction calculates an aggregate summary,
while other transactions update some involved items

Result: the aggregate is inconsistent



Why recovery is needed?

Transaction idea (reminder):

- program that forms a logical unit of database processing
- includes one or more database access operations
  (e.g., retrieval, insertion, deletion)
- boundaries can be specified by begin and end statements
  (if something goes wrong, effects roll back)

Should be either

- Committed:
  evaluated in full and all effects permanently recorded
- Aborted:
  no effect on the database (everything done rolled back)

Should take into account and use cache and log
(which we will introduce later)

Why recovery is needed?

Types of failures:

- Computer failure (system crash):

  hardware, software, network errors

- Transaction failure:

  division by zero, integrity constraint violation, user interrupt, etc.

- Local transaction errors:

  no data found, programmed exception, etc.

- Concurrency control enforcement:

  serialisability violation, deadlock resolving, etc.

- Disk failure:

  errors with disk reads or writes

- Physical problems:

  power cut, fire, catastrophe, etc.

Main goal of Part 2

To develop a principled machinery (formalise concepts, develop algorithms, etc.) for transaction management system:

- concurrency control subsystem
- fail recovery subsystem

2. Desirable properties: ACID principles

ACID properties

ACID:

- Atomicity

  Transaction performed in its entirety or not at all

- Consistency

  The database should always remain consistent

- Isolation

  Transaction should not interfere with other transactions

- Durability

  Changes of committed transactions must persist

Atomicity

Atomicity: Transaction performed in its entirety or not at all

- ensured by transaction recovery subsystem of a DBMS
- the recovery technique must undo any effects
  of a failed (aborted) transaction
- done by using the log and writing back the old values
  (we will discuss the log and techniques)
- cooperation with concurrency control subsystem
  is also necessary

Consistency

Consistency: The database should always remain consistent

- each transaction should bring database from one consistent state to another consistent state
- consistent state (reminder): the database satisfies the integrity constraints specified in the schema as well as any other constraints on the database that should hold
- the responsibility of the database program
  (i.e., the programmer)
- transactions can use `abort` instructions, leading to rollbacks
  (handled by recovery subsystem)

Isolation

Isolation: Transaction should not interfere with other transactions

- ensured by the concurrency control subsystem
- ideally, we should avoid all problems (dirty writes and reads, non-repeatable reads, phantoms, etc.)
- this may be too expensive (in terms of efficiency),
  so often may be relaxed in one way or another
  (isolation levels, we will see)
- one of the most difficult and interesting issues
  for transaction management

Durability

Durability (or permanency): Changes of committed transactions must persist

- ensured by the recovery subsystem
- after a crash, the log is used
  to restore committed transactions and
  roll back uncommitted ones
- cache complicates restoration

3. Transactions and schedules

Transactions

Transaction is a sequence of operations, an atomic unit of work

Structure of a committed (successful) transaction:

1. `begin` ← marks the beginning of transaction execution
2. one or several `read`(X), `write`(X), local operations (e.g., var updates), transaction control operations (e.g., locks), etc.
3. `end` ← mark the end of transaction execution
4. one or several operations checking consistency, serialisability, etc.
5. `commit` ← successful completion, changes cannot be undone

Structure of an aborted (unsuccessful) transaction:

same beginning, but ends at any point with

N. `abort` ← unsuccessful completion, all changes must be undone

A partial transaction:

a beginning of one above ('waiting' for next operation)

Transaction notation

Transaction is a sequence of operations (not arbitrary)

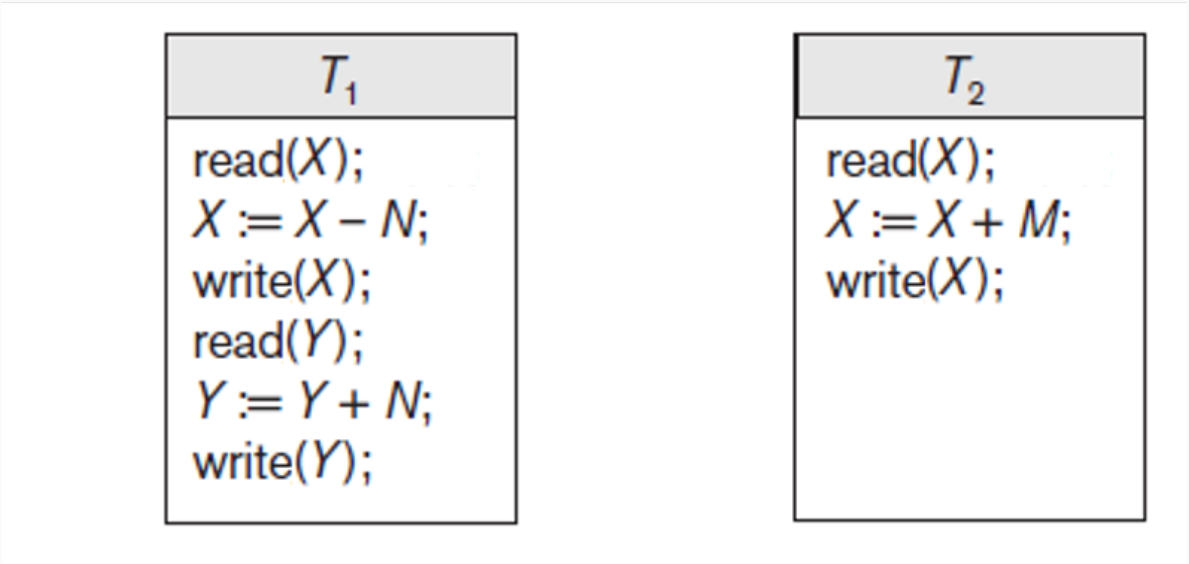
- `begin`, `end`, `commit`,
  `abort`, `read`(X), `write`(X), variable updates, etc.
- can be committed, aborted, or partial
  (depending on the last operation)

Often we

- use abbreviations:
  `b`, `e`, `c`, `a`, `r`(X), `w`(X) (var updates the same)
- omit operations irrelevant in the context
  (in most cases, only `r`(X) and `w`(X) are relevant)
- when considering several transactions,
  append transaction ids to operations:
  $\mathtt{b}_1$, $\mathtt{r}_1(X)$, $\mathtt{r}_2(X)$, $\mathtt{w}_3(X)$

Example transactions

Let X and Y be the numbers of reserved seats in two flights



(Some operations are already omitted here)

May be written as

T_1: $r_1(X)$, $w_1(X)$, $r_1(Y)$, $w_1(Y)$

T_2: $r_2(X)$, $w_2(X)$

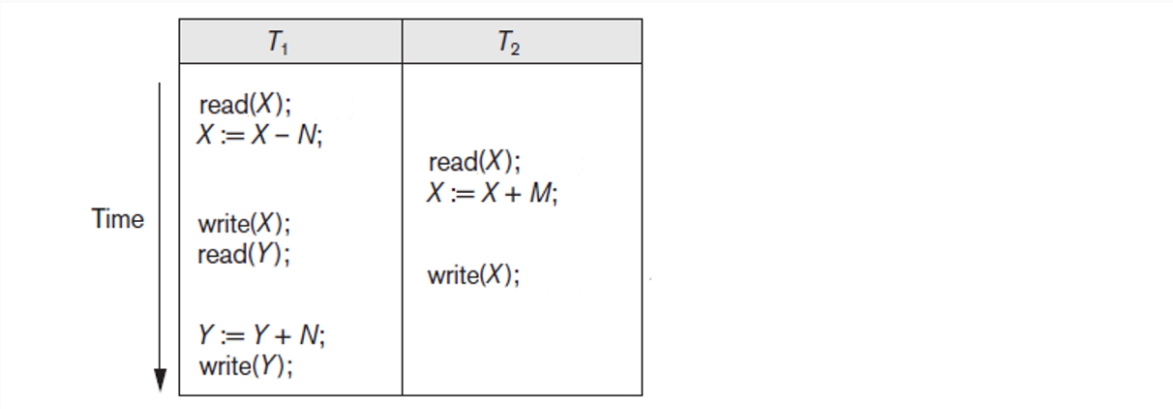
Schedules

Schedule (or history, execution plan) S of transactions $T_1, \ldots, T_n$: a (total) ordering of the operations of $T_1, \ldots, T_n$

- operations of different transitions can interleave
- operations of each T_i are in the same order as in T_i

Complete schedule: all $T_1, \ldots, T_n$ are committed or aborted

Example: S_a: $\mathtt{r}_1(X)$, $\mathtt{r}_2(X)$, $\mathtt{w}_1(X)$, $\mathtt{r}_1(Y)$, $\mathtt{w}_2(X)$, $\mathtt{w}_1(Y)$



Example (complete): S_b: $\mathtt{r}_1(X)$, $\mathtt{w}_1(X)$, $\mathtt{r}_2(X)$, $\mathtt{w}_2(X)$, $\mathtt{r}_1(Y)$, $\mathtt{a}_1$, $\mathtt{c}_2$

4. Characterising schedules based on serialisability

Serialisability informally

For isolation (transaction should not interfere with other transactions), we need to be able to ensure that all transactions of a schedule are executed correctly

We will characterise the types of schedules that are considered to be correct:

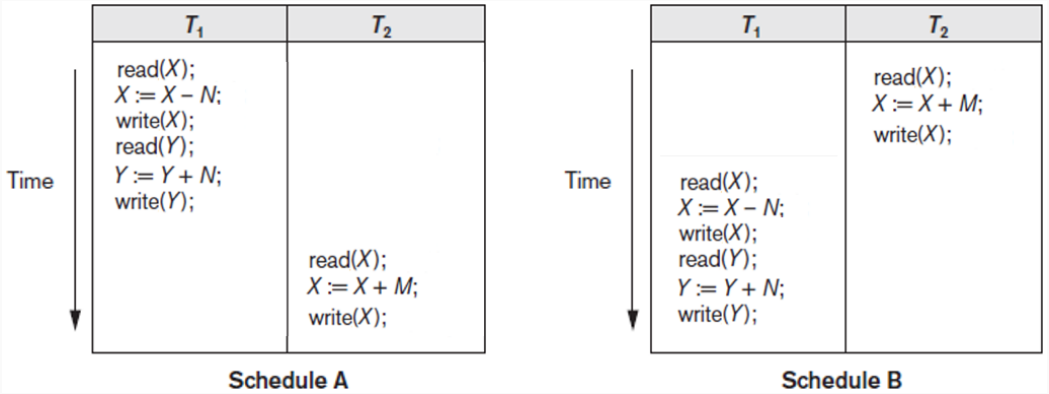
serial, (conflict-)serialisable, view-serialisable

Serial schedules

Serial schedule intuitively: no interleaving

Serial schedule: no transaction begins when there is an active (non-committed and non-aborted) transaction

(i.e., all the operations of every transaction are executed consecutively, and only one transition at a time is active)



A: $r_1(X)$, $w_1(X)$, $r_1(Y)$, $w_1(Y)$, c_1, $r_2(X)$, $w_2(X)$, c_2

B: $r_2(X)$, $w_2(X)$, c_2, $r_1(X)$, $w_1(X)$, $r_1(Y)$, $w_1(Y)$, c_1

Every serial is considered correct

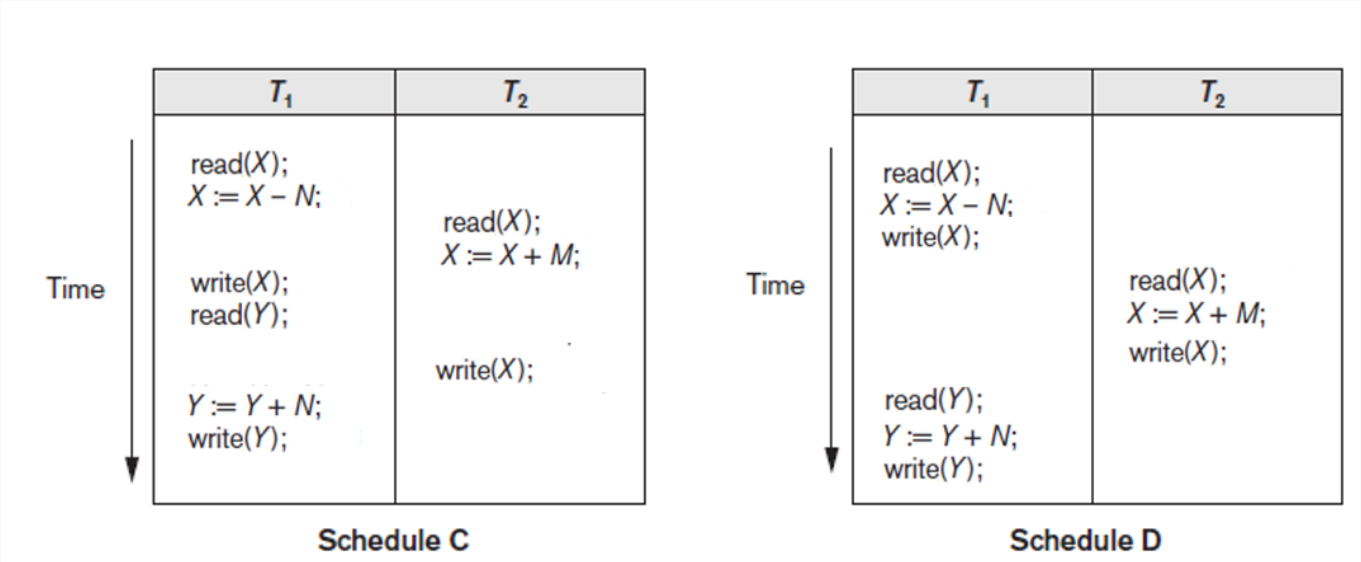
Limit (essentially no) concurrency: unacceptable

Serialisable schedules

Serialisable schedule: equivalent to a serial schedule

We need to define transaction equivalence

Examples:



C should not be serialisable, *D* should be

Most common is conflict equivalence (in two slides),
but there are other possibilities

Naive approaches to equivalence

Naive approaches to equivalence:

- result equivalence:
  two schedules give the same result on the current database
- semantic equivalence:
  two schedules give the same result on all possible databases

Question: What are the problems with these approaches?

We will develop syntactic notions of equivalence, which guarantee semantic (and hence result) equivalence. The first and the main is based on the notion of a conflict.

Conflicts

Intuitively: a conflict is a pair of operations in a schedule such that

- they are from different transactions
- flipping them can result in a different outcome

Formally: two operations in a schedule is

- a read-write conflict if they are $r_i(X)$ and $w_j(X)$ with $i \neq j$ (in any order)
- a write-write conflict if they are $w_i(X)$ and $w_j(X)$ with $i \neq j$

Examples:

- $\mathtt{r}_1(X)$, $\mathtt{r}_2(X)$, $\mathtt{w}_1(X)$, $\mathtt{r}_1(Y)$, $\mathtt{w}_2(X)$, $\mathtt{w}_1(Y)$
- $\mathtt{r}_1(X)$, $\mathtt{w}_1(X)$, $\mathtt{r}_2(X)$, $\mathtt{w}_2(X)$, $\mathtt{r}_1(Y)$, $\mathtt{a}_1$
- $\mathtt{r}_1(X)$, $\mathtt{w}_1(X)$, $\mathtt{r}_2(X)$, $\mathtt{w}_2(X)$, $\mathtt{r}_1(Y)$, $\mathtt{a}_1$

(You may sometimes see intra-transaction conflicts in literature,

but we do not need them in our formalisation)

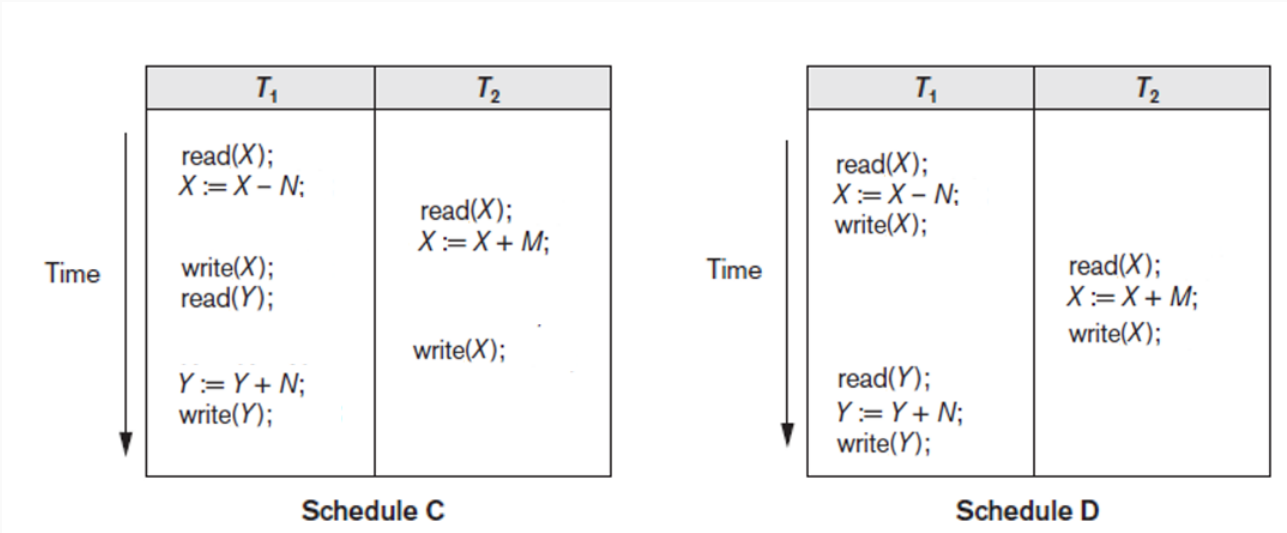
Misleading name: Nothing wrong per se!

Conflict equivalence

Two schedules are conflict equivalent if

- they are schedules of the same transactions
- if the relative order of every two conflicts (read-write, write-write) is the same in both schedules

(Conflict-)serialisability: serialisability based on conflict equivalence



Schedule C is not (conflict-)serialisable

Schedule D is serialisable (equivalent to $T_1; T_2$)

Testing for serialisability

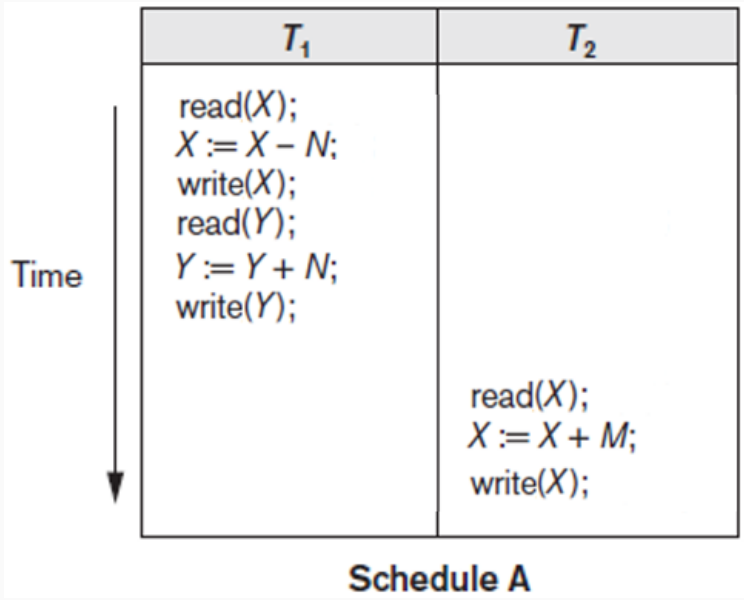
Simple algorithm for testing serialisability of a schedule S

1. Construct precedence graph of a S:
   - nodes: transactions in S
   - (directed) edges: conflict pairs of operations
     (from the earlier to the later)

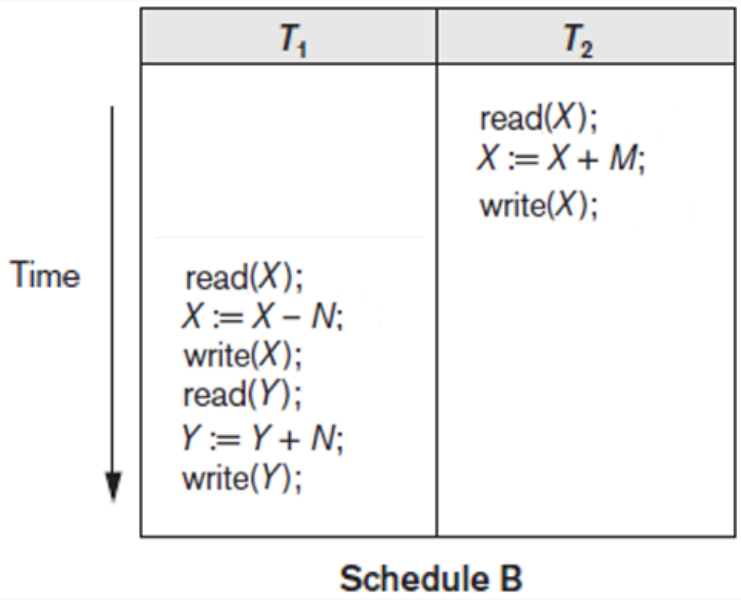
     (i.e., operations from different transactions
     one of which is write)
2. Return serialisable if and only if
   the precedence graph has no cycles

Sanity check examples

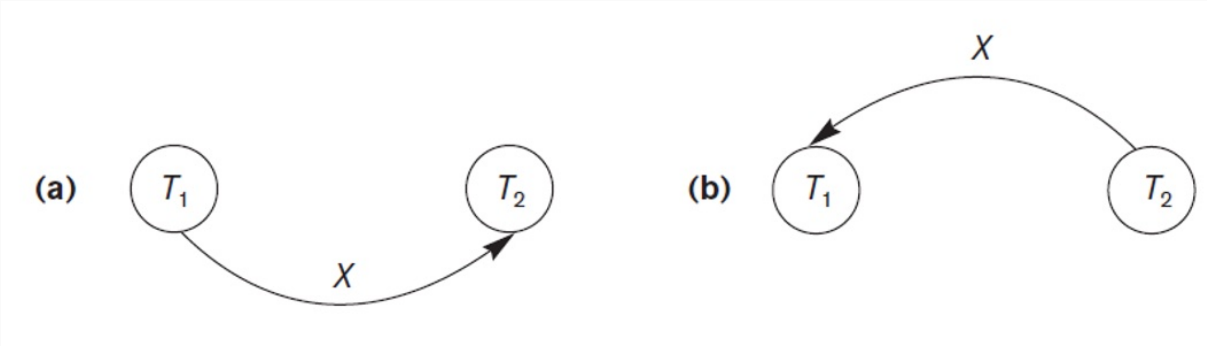
Serial schedules (obviously) have acyclic precedence graph



Schedule A



Schedule B



(Labels X are not essential, just for reference)

More examples – 1

Non-serialisable and serialisable schedules:

Time

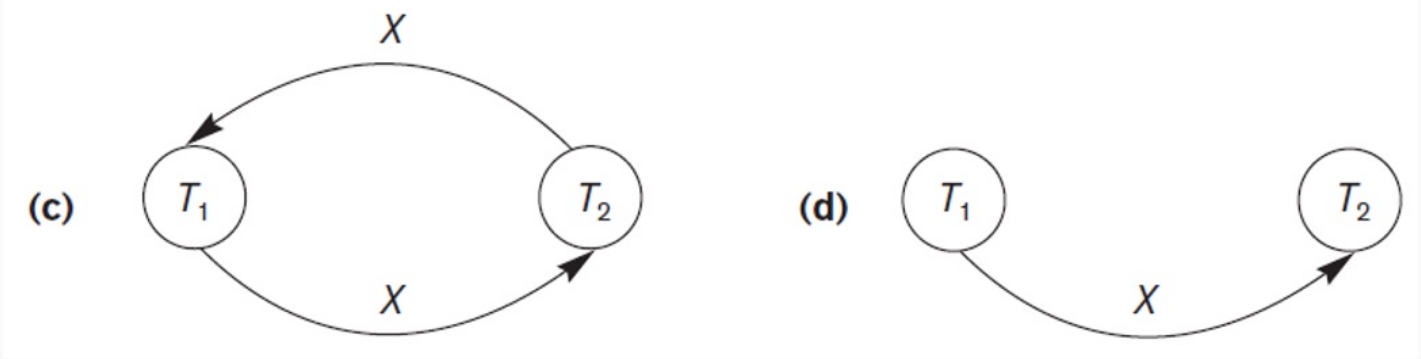
T_1	T_2
read(X);	
$X := X - N$;	
	read(X);
	$X := X + M$;
write(X);	
read(Y);	
	write(X);
$Y := Y + N$;	
write(Y);	

Schedule C

Time

T_1	T_2
read(X);	
$X := X - N$;	
write(X);	
	read(X);
	$X := X + M$;
	write(X);
read(Y);	
$Y := Y + N$;	
write(Y);	

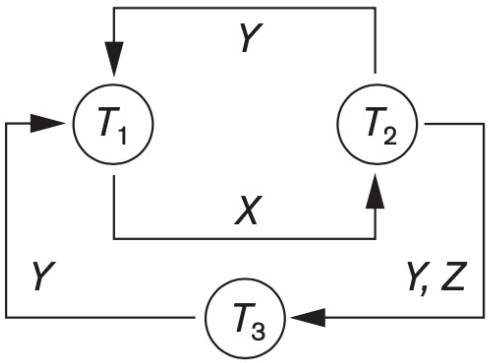
Schedule D



More examples – 2

Time

Transaction T_1	Transaction T_2	Transaction T_3
	read(Z);	
	read(Y);	
	write(Y);	
		read(Y);
		read(Z);
read(X);		
write(X);		
		write(Y);
		write(Z);
	read(X);	
read(Y);		
write(Y);		
	write(X);	



Equivalent serial schedules

None

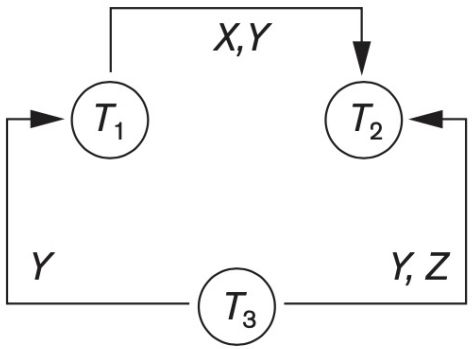
Reason

Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
Cycle $X(T_1 \rightarrow T_2), YZ\,(T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

More examples – 3

Time

Transaction T_1	Transaction T_2	Transaction T_3
		read(Y);
		read(Z);
read(X);		
write(X);		
		write(Y);
		write(Z);
	read(Z);	
read(Y);		
write(Y);		
	read(Y);	
	write(Y);	
	read(X);	
	write(X);	



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

View equivalence: Motivation

Serialisability based on conflict-equivalence is rather strong,
and can be relaxed

Example: $\mathtt{r}_1(X), \mathtt{w}_2(X), \mathtt{w}_1(X), \mathtt{w}_3(X), \mathtt{c}_1, \mathtt{c}_2, \mathtt{c}_3$

Not conflict-serialisable, but the effect is equivalent to serial
$\mathtt{r}_1(X), \mathtt{w}_1(X), \mathtt{c}_1, \mathtt{w}_2(X), \mathtt{c}_2, \mathtt{w}_3(X), \mathtt{c}_3$
(due to blind write $\mathtt{w}_3(X)$)

View equivalence

Schedules S and S' are view equivalent if

- they are schedules of the same transactions
- each $\mathtt{r}(X)$ reads from the same place in S and S'
  (i.e., in both schedules either X has been previously written by the same $\mathtt{w}(X)$ or not written at all)
- the last write of each X is also the same

View-serialisability: serialisability based on view equivalence

Example: $\mathtt{r}_1(X), \mathtt{w}_2(X), \mathtt{w}_1(X), \mathtt{w}_3(X), \mathtt{c}_1, \mathtt{c}_2, \mathtt{c}_3$

Each conflict-serialisable is view-serialisable

Each view-serialisable is semantic-serialisable (so result-serialisable)

Difficult to check view-equivalence (NP-complete)

By default, we rely on conflict-equivalence,
and 'serialisability' below means 'conflict-serialisability'

Serialisability in concurrency control

Every serial schedule is serialisable, but not other way round

Serialisable schedules give benefit of concurrent execution without giving up any correctness

Difficult to test for serialisability in practice (even conflict-serialisability)

- system load, time of transaction submission, and process priority affect ordering of operations
- often we need to ensure serialisability before the transactions complete

DBMSs enforce concurrency control protocols that ensure serialisability

5. Characterising schedules based on recoverability

Recoverability informally

For durability (changes of committed transactions must persist), we need to be able to recover schedules from transaction and system failures

We will see techniques for this (in a couple of weeks), but before this it is important to classify schedules with respect to recoverability:

- impossible to recover
  (we will see examples)
- possible to recover
  recoverable (we will define formally and see examples)
- easy to recover
  cascadeless, strict (we will define and see examples)

Recoverable schedules

Transaction T reads from (different) transaction T' in a schedule if an item X is first written by T' and then read by T (and no transactions write X in between)

Schedule S is recoverable if no transaction T in S commits until all transactions T' such that T reads from T' have committed

Examples:

$\mathtt{r}_1(X), \mathtt{r}_2(X), \mathtt{w}_1(X), \mathtt{r}_1(Y), \mathtt{w}_2(X), \mathtt{c}_2, \mathtt{w}_1(Y), \mathtt{c}_1$

recoverable

$\mathtt{r}_1(X), \mathtt{w}_1(X), \mathtt{r}_2(X), \mathtt{r}_1(Y), \mathtt{w}_2(X), \mathtt{w}_1(Y), \mathtt{c}_2, \mathtt{a}_1$

non-recoverable

$\mathtt{r}_1(X), \mathtt{w}_1(X), \mathtt{r}_2(X), \mathtt{r}_1(Y), \mathtt{w}_2(X), \mathtt{w}_1(Y), \mathtt{a}_1, \mathtt{c}_2$

still non-recoverable

$\mathtt{r}_1(X), \mathtt{w}_1(X), \mathtt{r}_2(X), \mathtt{r}_1(Y), \mathtt{w}_2(X), \mathtt{w}_1(Y), \mathtt{a}_1, \mathtt{a}_2$

recoverable

Cascadeless schedules

Recoverable schedules may need cascading aborts (cascading rollbacks) to recover—that is, an (non-committed) transaction has to abort since it has read from another failed transaction

Example: $\mathtt{r}_1(X), \mathtt{w}_1(X), \mathtt{r}_2(X), \mathtt{r}_1(Y), \mathtt{w}_2(X), \mathtt{w}_1(Y), \mathtt{a}_1, \mathtt{a}_2$

This may affect many transactions, and rollbacks may be expensive

Cascadeless schedule idea: no cascade rollbacks

Cascadeless schedule: every transaction in the schedule reads only from committed transactions

(i.e., for each $\mathtt{r}(X)$, X can be previously written by the same transaction, written by another committed transaction or have not been written before)

Example:

$\mathtt{r}_1(X), \mathtt{w}_1(X), \mathtt{r}_1(Y), \mathtt{r}_2(Z), \mathtt{w}_1(Y), \mathtt{c}_1, \mathtt{w}_2(Z), \mathtt{r}_2(X), \mathtt{w}_2(X), \mathtt{c}_2$

Strict schedules

Cascadeless schedules may still have fairy complicated recovery protocols, because we cannot just restore the values of all writes of an aborted transaction

Example: $\mathtt{w}_1(X), \mathtt{w}_2(X), \mathtt{a}_1$

This may all affect many transactions (no cascades though)

Strict schedule: no transaction can read or write an item X until the previous write of X is committed or aborted

Examples:

$\mathtt{w}_1(X), \mathtt{c}_1, \mathtt{w}_2(X)$ $\mathtt{w}_1(X), \mathtt{a}_1, \mathtt{w}_2(X)$

Recovery hierarchy

Every strict schedule is cascadeless

Every cascadeless is recoverable

We will study recovery techniques for different types of schedules
in a couple of lectures

6. Isolation levels

Back to problems (phenomena)

Potential concurrency control problems (or phenomena):

1. Dirty write (or lost update): $\ldots\ \mathtt{r}_1(X), \ldots, \mathtt{w}_2(X), \ldots, \mathtt{w}_1(X)\ \ldots$
   not conflict-serialisable

2. Dirty read (or temporary update): $\ldots\ \mathtt{w}_1(X), \ldots, \mathtt{r}_2(X), \ldots, \mathtt{a}_1\ \ldots$
   not recoverable

3. Non-repeatable read: $\ldots\ \mathtt{r}_1(X), \ldots, \mathtt{w}_2(X), \ldots, \mathtt{r}_1(X),\ \ldots$
   not conflict-serialisable

4. Phantoms (incl. incorrect summary): $\ldots \mathtt{r}_1(X) \ldots \mathtt{w}_2(Y) \ldots \mathtt{r}_1(Y) \ldots$
   where X and Y contribute to query of T_1 in the same way
   need to treat this by other mechanisms

Isolation levels

It is not always necessary to ensure no problems of all types

- in some applications we can tolerate some problems
- for the sake of more concurrency (i.e., efficiency)

Simple isolation level hierarchy of schedules
(and also transaction management protocols):

Level 0: no dirty reads
Level 1: no dirty writes
Level 2: no dirty reads, no dirty writes
Level 3 (true isolation): no dirty reads, no dirty writes,
no non-repeatable reads

A protocol that ensure
recoverable and serialisable schedules is Level 3

Isolation levels in SQL

SQL standard concentrate on

dirty reads, non-repeatable reads, and phantoms

Isolation levels in SQL ('yes' is allowed):

	Type of Violation		
Isolation Level	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Note: SERIALIZABLE here is different from conflict-serialisable

(Real DBMSs do not always follow the standard:

for example, no dirty writes are usually also ensured)

SQL transaction language

SQL has a language for transactions

(systems may not follow the standard in all details)

Example:

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
    READ WRITE                          <— access mode
    DIAGNOSTIC SIZE 5                   <— diagnostic area size
    ISOLATION LEVEL SERIALIZABLE;       <— isolation level
EXEC SQL INSERT INTO EMPLOYEE (Fname, Lname, Ssn, Dno, Salary)
    VALUES ('Robert', 'Smith', '991004321', 2, 35000);
EXEC SQL UPDATE EMPLOYEE
    SET Salary = Salary * 1.1 WHERE Dno = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ... ;
```

Access mode: READ ONLY (only SELECT queries) or READ WRITE

Diagnostic size: number of statements with diagnostic messages

Note: need to handle schedules with transaction of different levels

What have we learned?

Basics of transaction management:

- ACID principles (Atomicity, Concurrency, Isolation, Durability)
- main concepts (transactions, schedules, conflicts)
- recoverable schedules (including cascadeless and strict)
- serialisable schedules (including conflict- and view-serialisable)
- isolation levels (several types)

In the rest of Part 2:

1. techniques for concurrency control
2. techniques for recovery