Database Systems

Transactions

Christian S. Jensen

Department of Computer Science
Aalborg University
csj@cs.aau.dk

Spring 2020

Learning goals: transactions and schedules

Learning goals

- Understanding the transaction concept
- Understanding the ACID properties
- Understanding the schedule concept
- Understanding serializability
- Understanding recoverable and cascadeless schedules

Motivation

- Users think in transactions
- Transaction boundaries are an important part of system design
- Offers a foundation for database tuning
- Enables assessment of system capabilities

Outline I

Outline II

- Deadlock prevention

4. Recovery
   - Failure classification
   - Data storage
   - Log entries
   - Log-based recovery

Introduction

An example bank transfer

1. Read the account balance of A into variable a: $read(A, a);$
2. Reduce account balance by 500 kr.: $a := a - 500;$
3. Write the new account balance into the database: $write(A, a);$
4. Read account balance of B into variable b: $read(B, b);$
5. Increase account balance by 500 kr.: $b := b + 500;$
6. Write new balance into the database: $write(B, b);$

Introduction

An example bank transfer

1. Read the account balance of A into variable a: $read(A, a);$
2. Reduce account balance by 500 kr.: $a := a - 500;$
3. Write the new account balance into the database: $write(A, a);$
4. Read account balance of B into variable b: $read(B, b);$
5. Increase account balance by 500 kr.: $b := b + 500;$
6. Write new balance into the database: $write(B, b);$

What could cause a problem?

Introduction

An example bank transfer

1. Read the account balance of A into variable a: $read(A, a);$
2. Reduce account balance by 500 kr.: $a := a - 500;$
3. Write the new account balance into the database: $write(A, a);$
4. Read account balance of B into variable b: $read(B, b);$
5. Increase account balance by 500 kr.: $b := b + 500;$
6. Write new balance into the database: $write(B, b);$

- All steps must be treated as a unit: "All or nothing."
- Once completed, the changes should be stored permanently.

What is a transaction?

A **transaction** is a collection of operations that forms a **logical unit** of work, during which various data items are accessed and possibly updated.

Transaction boundaries are user-defined!

Characteristics of transactions: ACID properties

Atomicity

- Either all operations of the transaction are properly reflected in the database or none are.
- Often implemented via logs

Characteristics of transactions: ACID properties

Atomicity

- Either all operations of the transaction are properly reflected in the database or none are.
- Often implemented via logs

Consistency

- Execution of a transaction in isolation preserves the consistency of the database.
- According to constraints, checks, assertions
- In addition, consistency is defined by the application, e.g., fund transfers should not generate or destroy money – the overall sum is the same before and afterwards

Characteristics of transactions: ACID properties

Isolation

- Each transaction appears to have the DB exclusively on its own.
- Intermediate results must be hidden for other transactions.
- Often implemented via locks

Characteristics of transactions: ACID properties

Isolation

- Each transaction appears to have the DB exclusively on its own.
- Intermediate results must be hidden for other transactions.
- Often implemented via locks

Durability

- Updates of successfully completed transactions must not get lost despite system failures.
- Often implemented via logs

Outline

Operations on transactions

begin of transaction (BOT)

Represents the beginning of a transaction, i.e., all following statements together form a transaction.

In SQL `BEGIN;`

Operations on transactions

begin of transaction (BOT)

Represents the beginning of a transaction, i.e., all following statements together form a transaction.
In SQL `BEGIN;`

commit

Represents the end of a transaction, i.e., all changes are made persistent and visible to others.
In SQL `COMMIT;`

Operations on transactions

begin of transaction (BOT)

Represents the beginning of a transaction, i.e., all following statements together form a transaction.
In SQL `BEGIN;`

commit

Represents the end of a transaction, i.e., all changes are made persistent and visible to others.
In SQL `COMMIT;`

rollback or abort

Causes a transaction to roll back, i.e., all changes are undone/discarded.
In SQL `ROLLBACK;`

Operations on transactions

“autocommit” mode

Each statement is executed in its own transaction

Basic consistency checks

```
CREATE TABLE emp(
eid    INT          PRIMARY KEY,
ename  VARCHAR(30)  NOT NULL,
salary INT          NOT NULL CHECK (salary > 0)
);
```

```
-- primary key violation
insert into emp values (11, 'Kim', 200);
-- Not null constraint violation
insert into emp values (44, NULL, 200);
-- Check statement violation
insert into emp values (44, 'Kim', -200);
```

Basic consistency checks

```
CREATE TABLE emp(
eid     INT          PRIMARY KEY,
ename   VARCHAR(30)  NOT NULL,
salary  INT          NOT NULL CHECK (salary > 0)
);
```

```
-- primary key violation
insert into emp values (11, 'Kim', 200);
-- Not null constraint violation
insert into emp values (44, NULL, 200);
-- Check statement violation
insert into emp values (44, 'Kim', -200);
```

- Many errors can be caught by the DBMS—Use it!

Savepoints

Long running transactions can specify savepoints.

SAVEPOINT savepoint_name;

Defines a point/state within a transaction
A transaction can be **rolled back partially** back up to the savepoint.

Savepoints

Long running transactions can specify savepoints.

SAVEPOINT savepoint_name;

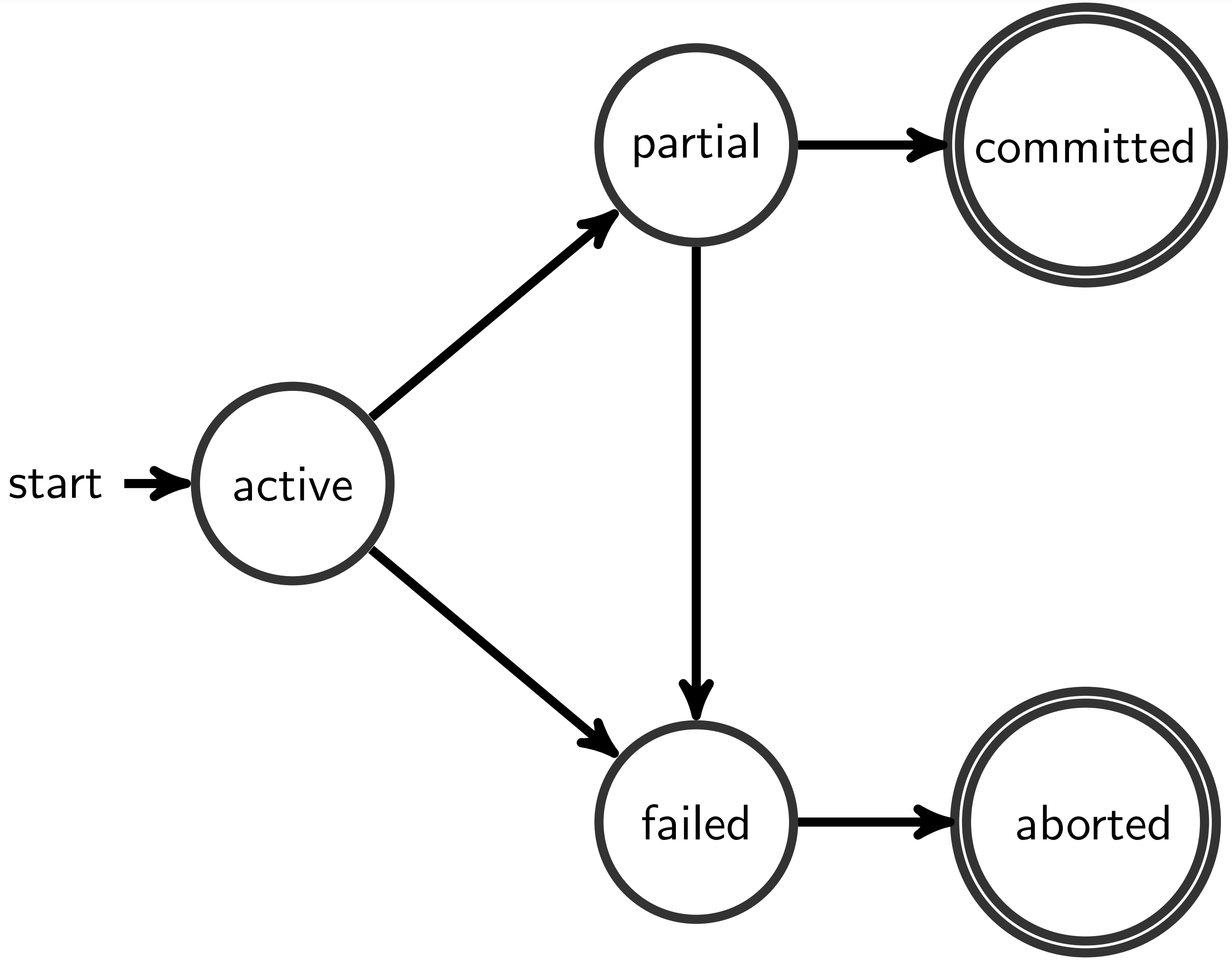
Defines a point/state within a transaction
A transaction can be **rolled back partially** back up to the savepoint.

ROLLBACK TO <savepoint_name>:
rolls the active transaction back to the savepoint <savepoint_name>

Example

```
BEGIN;
INSERT INTO tab VALUES...
SAVEPOINT A;
INSERT INTO tab VALUES...
SAVEPOINT B;
SELECT * FROM tab;
ROLLBACK TO A;
SELECT * FROM tab;
...
```

Transaction states



How do DBMSs support transactions?

The two most important components of transaction management are

Multi-user synchronization (isolation)

- Semantic correctness despite concurrency
  Concurrency allows for high throughput
- Serializability
- Weaker isolation levels

How do DBMSs support transactions?

The two most important components of transaction management are

Multi-user synchronization (isolation)

- Semantic correctness despite concurrency
  Concurrency allows for high throughput
- Serializability
- Weaker isolation levels

Recovery (atomicity and durability)

- Roll back partially executed transactions
- Re-executing transactions after failures
- Guaranteeing persistence of transactional updates

Outline

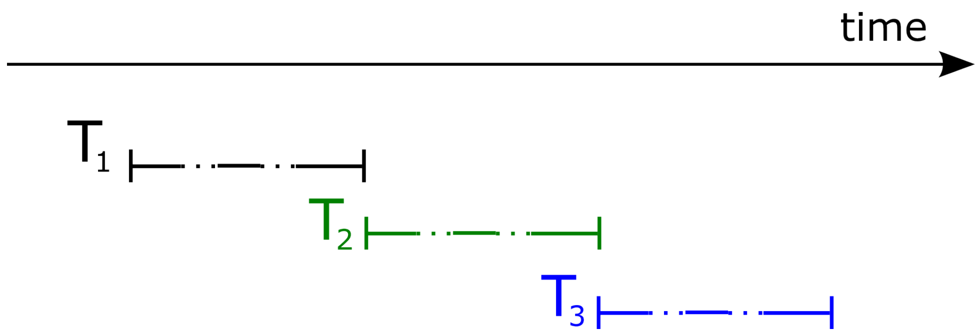
2. Schedules and serializability
   - Schedules
   - Conflict serializability
   - Conflict graphs (precedence graphs)
   - Recoverable and cascadeless schedules

Concurrency

Affects the „I“ in ACID.

The execution of multiple transactions T_1, T_2, and T_3

(a) in a single-user environment

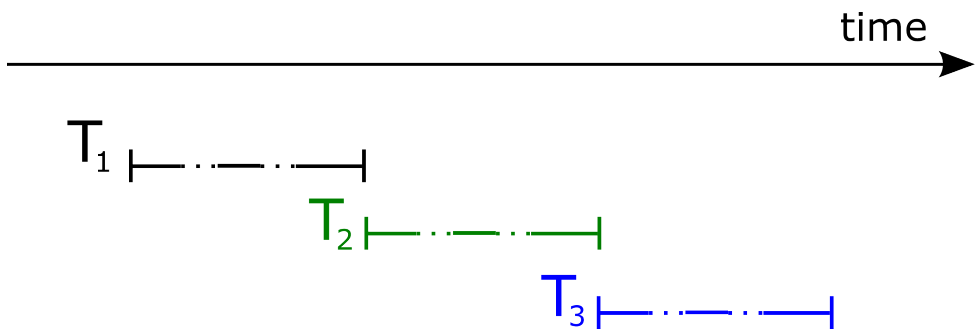


Concurrency

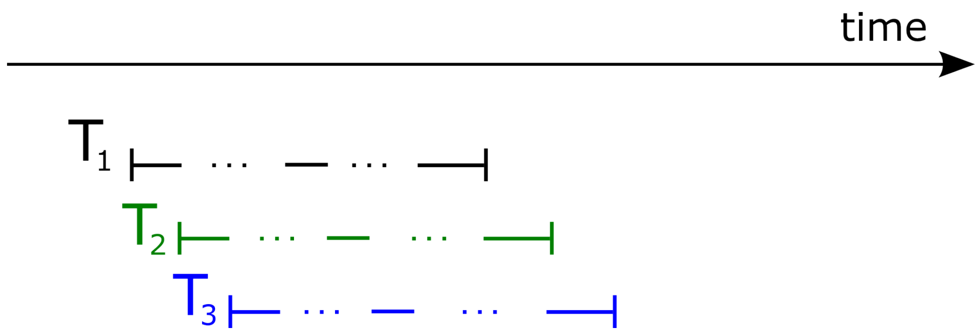
Affects the „I“ in ACID.

The execution of multiple transactions T_1, T_2, and T_3

(a) in a single-user environment



(b) in a (concurrent) multi-user environment with interleaved execution



Potential problems during concurrent execution

What's the problem?

Steps	T_1	T_2
1.	read(A,$a1$)	
2.	$a1 := a1 - 300$	
3.		read(A,a_2)
4.		$a_2 := a_2 * 1.03$
5.		**write(A,a_2)**
6.	**write(A,a_1)**	
7.	read(B,b_1)	
8.	$b_1 := b_1 + 300$	
9.	write(B,b_1)	

Potential problems during concurrent execution

Lost updates (overwriting updates)

Steps	T_1	T_2
1.	read(A,$a1$)	
2.	$a1 := a1 - 300$	
3.		read(A,a_2)
4.		$a_2 := a_2 * 1.03$
5.		**write(A,a_2)**
6.	**write(A,a_1)**	
7.	read(B,b_1)	
8.	$b_1 := b_1 + 300$	
9.	write(B,b_1)	

Potential problems during concurrent execution

What's the problem?

Steps	T_1	T_2
1.	read(A,a_1)	
2.	$a_1 := a_1 - 300$	
3.	**write(A,a_1)**	
4.		**read(A,a_2)**
5.		$a_2 := a_2 * 1.03$
6.		write(A,a_2)
7.	read(B, b_1)	
8.	...	
9.	abort	

Potential problems during concurrent execution

Dirty read (dependency on non-committed updates)

Steps	T_1	T_2
1.	read(A,a_1)	
2.	$a_1 := a_1 - 300$	
3.	**write(A,a_1)**	
4.		**read(A,a_2)**
5.		$a_2 := a_2 * 1.03$
6.		write(A,a_2)
7.	read(B, b_1)	
8.	...	
9.	abort	

Potential problems during concurrent execution

What's the problem?

T_1	T_2
	select sum(balance) **from** account
update account **set** balance=42000 **where** accountID=12345	
	select sum(balance) **from** account

Potential problems during concurrent execution

Non-repeatable read (dependency on other updates)

T_1	T_2
	select sum(balance) **from** account
update account **set** balance=42000 **where** accountID=12345	
	select sum(balance) **from** account

Potential problems during concurrent execution

What's the problem?

T_1	T_2
	select sum(balance) **from** account
insert into account **values** (C,1000,...)	
	select sum(balance) **from** account

Potential problems during concurrent execution

Phantom problem (dependency on new/deleted tuples)

T_1	T_2
	select sum(balance) **from** account
insert into account **values** (C,1000,...)	
	select sum(balance) **from** account

Outline

2. Schedules and serializability
   - Schedules
   - Conflict serializability
   - Conflict graphs (precedence graphs)
   - Recoverable and cascadeless schedules

Concurrency and correctness

Centralized system with concurrent access by multiple users

- Database consisting of two data items: X and Y
- Only criterion for correctness: X = Y
- The following transactions

$$T_1 \quad X \leftarrow X + 1 \qquad T_2 \quad X \leftarrow 2 * X$$
$$\quad\;\; Y \leftarrow Y + 1 \qquad\quad\;\; Y \leftarrow 2 * Y$$

- Initially: X=10 and Y=10.
- T_1 followed by $T_2 \Rightarrow$ X = 22 and Y = 22
- T_2 followed by $T_1 \Rightarrow$ X = 21 and Y = 21

An example

schedule S_0	
T_1	T_2
	read(X, x)

Value of X: 10
Value of Y: 10

An example

schedule S_0	
T_1	T_2
	read(X, x)
	x $\leftarrow$ 2x

Value of X: 10
Value of Y: 10

An example

schedule S_0	
T_1	T_2
	read(X, x)
	x $\leftarrow$ 2x
	write(X, x)

Value of X: 20
Value of Y: 10

An example

Value of X: 20
Value of Y: 10

schedule S_0	
T_1	T_2
	read(X, x)
	x $\leftarrow$ 2x
	write(X, x)
	read(Y, y)

An example

Value of X: 20
Value of Y: 10

schedule S_0	
T_1	T_2
	read(X, x)
	x $\leftarrow$ 2x
	write(X, x)
	read(Y, y)
	y $\leftarrow$ 2y

An example

Value of X: 20
Value of Y: 20

schedule S_0	
T_1	T_2
	read(X, x)
	x $\leftarrow$ 2x
	write(X, x)
	read(Y, y)
	y $\leftarrow$ 2y
	write(Y, y)

An example

Value of X: 20
Value of Y: 20

schedule S_0	
T_1	T_2
	read(X, x)
	x $\leftarrow$ 2x
	write(X, x)
	read(Y, y)
	y $\leftarrow$ 2y
	write(Y, y)
read(X, x)	

An example

Value of X: 20
Value of Y: 20

schedule S_0	
T_1	T_2
	read(X, x)
	x ← 2x
	write(X, x)
	read(Y, y)
	y ← 2y
	write(Y, y)
read(X, x)	
x ← x+1	

An example

Value of X: 21
Value of Y: 20

schedule S_0	
T_1	T_2
	read(X, x)
	x $\leftarrow$ 2x
	write(X, x)
	read(Y, y)
	y $\leftarrow$ 2y
	write(Y, y)
read(X, x)	
x $\leftarrow$ x+1	
write(X, x)	

An example

Value of X: 21
Value of Y: 20

schedule S_0	
T_1	T_2
	read(X, x)
	x $\leftarrow$ 2x
	write(X, x)
	read(Y, y)
	y $\leftarrow$ 2y
	write(Y, y)
read(X, x)	
x $\leftarrow$ x+1	
write(X, x)	
read(Y, y)	

An example

Value of X: 21
Value of Y: 20

schedule S_0	
T_1	T_2
	read(X, x)
	x ← 2x
	write(X, x)
	read(Y, y)
	y ← 2y
	write(Y, y)
read(X, x)	
x ← x+1	
write(X, x)	
read(Y, y)	
y ← y+1	

An example

Value of X: 21
Value of Y: 21

schedule S_0	
T_1	T_2
	read(X, x)
	x ← 2x
	write(X, x)
	read(Y, y)
	y ← 2y
	write(Y, y)
read(X, x)	
x ← x+1	
write(X, x)	
read(Y, y)	
y ← y+1	
write(Y, y)	

Formal definition of a schedules

A **schedule** is a **sequence of operations** from one or more transactions. For concurrent transactions, the operations are interleaved.

Operations

- read(Q, q)
  Read the value of database item Q and store it in the local variable q.
- write(Q, q)
  Store the value of the local variable q in database item Q
- Arithmetic operations
- commit
- abort

Formal definition of a schedules

A **schedule** is a **sequence of operations** from one or more transactions. For concurrent transactions, the operations are interleaved.

serial schedule

The operations of the transactions are executed sequentially with no overlap in time.

concurrent schedule

The operations of the transactions are executed with overlap in time.

Formal definition of a schedules

A **schedule** is a **sequence of operations** from one or more transactions. For concurrent transactions, the operations are interleaved.

serial schedule

The operations of the transactions are executed sequentially with no overlap in time.

concurrent schedule

The operations of the transactions are executed with overlap in time.

valid schedule

A schedule is valid if the result of its execution is "correct".

Example schedules

schedule S_0

T_1	T_2
	read(X, x)
	x ← 2x
	write(X, x)
	read(Y, y)
	y ← 2y
	write(Y, y)
read(X, x)	
x ← x+1	
write(X, x)	
read(Y, y)	
y ← y+1	
write(Y, y)	

schedule $S_{0'}$

T_1	T_2
	read(X, x)
	x ← 2x
	write(X, x)
read(X, x)	
x ← x+1	
write(X, x)	
	read(Y, y)
	y ← 2y
	write(Y, y)
read(Y, y)	
y ← y+1	
write(Y, y)	

schedule S_1

T_1	T_2
read(X, x)	
x ← x+1	
write(X, x)	
	read(X, x)
	x ← 2x
	write(X, x)
	read(Y, y)
	y ← 2y
	write(Y, y)
read(Y, y)	
y ← y+1	
write(Y, y)	

Are these schedules valid concurrent schedules, invalid concurrent schedules, or serial schedules? Initially: X=Y=10, correctness criterion: X=Y

Example schedules

schedule S_0

T_1	T_2
	read(X, x)
	x ← 2x
	write(X, x)
	read(Y, y)
	y ← 2y
	write(Y, y)
read(X, x)	
x ← x+1	
write(X, x)	
read(Y, y)	
y ← y+1	
write(Y, y)	

- X = 21, Y = 21
- **serial schedule**

schedule $S_{0'}$

T_1	T_2
	read(X, x)
	x ← 2x
	write(X, x)
read(X, x)	
x ← x+1	
write(X, x)	
	read(Y, y)
	y ← 2y
	write(Y, y)
read(Y, y)	
y ← y+1	
write(Y, y)	

- X = 21, Y = 21
- **concurrent schedule**

schedule S_1

T_1	T_2
read(X, x)	
x ← x+1	
write(X, x)	
	read(X, x)
	x ← 2x
	write(X, x)
	read(Y, y)
	y ← 2y
	write(Y, y)
read(Y, y)	
y ← y+1	
write(Y, y)	

- X = 22, Y = 21
- **an invalid schedule**

Notion of correctness

Definition D1

A concurrent execution of transactions must leave the database in a consistent state.

Notion of correctness

Definition D1

A concurrent execution of transactions must leave the database in a consistent state.

Definition D2

Concurrent execution of transactions must be (result) equivalent to some serial execution of the transactions.

Example

schedule S_2	
T_3	T_4
read(X, x)	
x ← x+1	
	read(X, x)
write(X, x)	
	x ← 2x
	write(X, x)
	read(Y, y)
	y ← 2y
read(Y, y)	
y ← y+1	
write(Y, y)	
	write(Y, y)

Initially: X = 10 and Y = 10
⇒ X = 20 and Y = 20

- S_2 is not result equivalent to a serial execution of T_3, T_4
- But the final database state is consistent.

Example

schedule S_2	
T_3	T_4
read(X, x)	
x ← x+1	
	read(X, x)
write(X, x)	
	x ← 2x
	write(X, x)
	read(Y, y)
	y ← 2y
read(Y, y)	
y ← y+1	
write(Y, y)	
	write(Y, y)

Initially: X = 10 and Y = 10
⇒ X = 20 and Y = 20

- S_2 is not result equivalent to a serial execution of T_3, T_4
- But the final database state is consistent.

Correctness of a schedule

The choice is definition D2:
An execution sequence is **correct** if it is **result equivalent** to a **serial execution**.

Correctness of a schedule

The choice is definition D2:
An execution sequence is **correct** if it is **result equivalent** to a **serial execution**.

Given a set of n transactions running concurrently. How do we efficiently check for correctness?

Correctness of a schedule

The choice is definition D2:
An execution sequence is **correct** if it is **result equivalent** to a **serial execution**.

Given a set of n transactions running concurrently. How do we efficiently check for correctness?

In the following: simplifying assumptions

- **Only reads and writes** are used to determine correctness.
- This assumption is stronger than definition D2, as even fewer schedules are considered correct.

Outline

2. Schedules and serializability
   - Schedules
   - Conflict serializability
   - Conflict graphs (precedence graphs)
   - Recoverable and cascadeless schedules

A fourth[1] notion of correctness: conflict serializability

Definition (D4[1])

A schedule is **conflict serializable** if it is **conflict equivalent** to a serial schedule.

[1] The third notion/definition (D3) is view serializability.

Possible conflicts between transactions

Conflicts between pairs of transactions (T_1 and T_2) and their instructions.

schedule S_A	
T_1	T_2
write(X, x)	
	read(X, x)

Conflict

schedule S_C	
T_1	T_2
	read(X, x)
write(X, x)	

Conflict

schedule S_B	
T_1	T_2
write(X, x)	
	write(X, x)

Conflict

schedule S_D	
T_1	T_2
read(X, x)	
	read(X, x)

No conflict

A fourth notion of correctness: conflict serializability

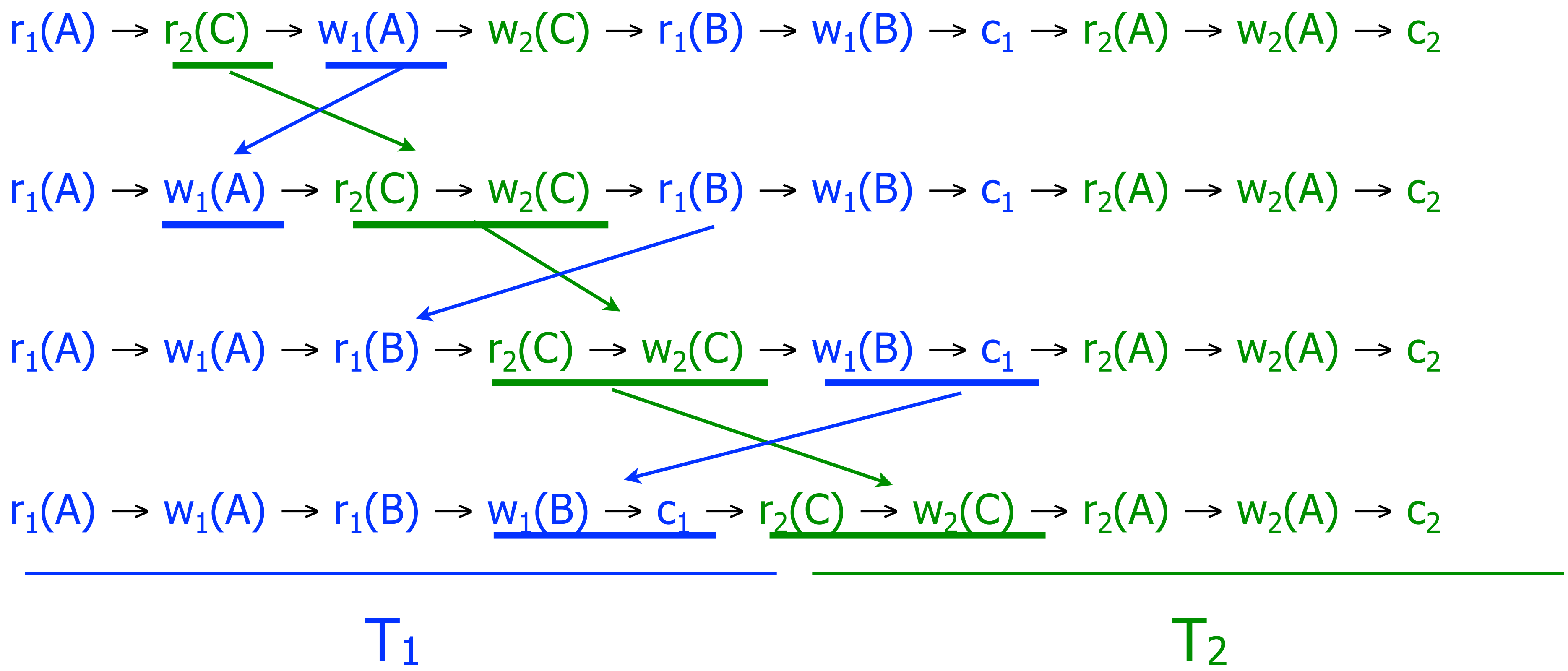
Definition (D4)

A schedule is **conflict serializable** if it is **conflict equivalent** to a serial schedule.

- Let I and J be consecutive instructions of a schedule S of multiple transactions.
- If I and J do not conflict, we can swap their order to produce a new schedule S'.
- The instructions appear in the same order in S and S', except for I and J, whose order does not matter.
- S and S' are termed **conflict equivalent schedules**.

Conflict equivalence of two schedules

As the transformation shows, the initial concurrent schedule is conflict equivalent to a serial schedule and is therefore conflict serializable.



c is short for commit, r (read), w (write)

Conflict serializable or not?

schedule S_A	
T_1	T_2
read(Y, y)	
	read(X, x)
	write(X, x)
write(Y, y)	

Conflict serializable or not?

schedule S_A	
T_1	T_2
read(Y, y)	
	read(X, x)
	write(X, x)
write(Y, y)	

conflict serializable

Conflict serializable or not?

schedule S_A	
T_1	T_2
read(Y, y)	
	read(X, x)
	write(X, x)
write(Y, y)	

schedule S_B	
T_3	T_4
read(X, x)	
	read(X, x)
	write(X, x)
write(X, x)	

conflict serializable

Conflict serializable or not?

schedule S_A	
T_1	T_2
read(Y, y)	
	read(X, x)
	write(X, x)
write(Y, y)	

conflict serializable

schedule S_B	
T_3	T_4
read(X, x)	
	read(X, x)
	write(X, x)
write(X, x)	

not conflict serializable

Conflict serializable or not?

schedule S_A	
T_1	T_2
read(Y, y)	
	read(X, x)
	write(X, x)
write(Y, y)	

conflict serializable

schedule S_B	
T_3	T_4
read(X, x)	
	read(X, x)
	write(X, x)
write(X, x)	

not conflict serializable

schedule S_C	
T_5	T_6
read(X, x)	
	read(X, x)
write(X, x)	

Conflict serializable or not?

schedule S_A	
T_1	T_2
read(Y, y)	
	read(X, x)
	write(X, x)
write(Y, y)	

conflict serializable

schedule S_B	
T_3	T_4
read(X, x)	
	read(X, x)
	write(X, x)
write(X, x)	

not conflict serializable

schedule S_C	
T_5	T_6
read(X, x)	
	read(X, x)
write(X, x)	

conflict serializable

Conflict serializable or not?

schedule S_A	
T_1	T_2
read(Y, y)	
	read(X, x)
	write(X, x)
write(Y, y)	

conflict serializable

schedule S_B	
T_3	T_4
read(X, x)	
	read(X, x)
	write(X, x)
write(X, x)	

not conflict serializable

schedule S_C	
T_5	T_6
read(X, x)	
	read(X, x)
write(X, x)	

conflict serializable

schedule S_D	
T_7	T_8
read(X, x)	
	write(X, x)
write(X, x)	

Conflict serializable or not?

schedule S_A	
T_1	T_2
read(Y, y)	
	read(X, x)
	write(X, x)
write(Y, y)	

conflict serializable

schedule S_B	
T_3	T_4
read(X, x)	
	read(X, x)
	write(X, x)
write(X, x)	

not conflict serializable

schedule S_C	
T_5	T_6
read(X, x)	
	read(X, x)
write(X, x)	

conflict serializable

schedule S_D	
T_7	T_8
read(X, x)	
	write(X, x)
write(X, x)	

not conflict serializable

Outline

Conflict graph

We construct a directed graph (conflict/precedence graph) for a schedule involving a set of transactions.

Assumption:
a transaction will always read an item before it writes that item.

Conflict graph

We construct a directed graph (conflict/precedence graph) for a schedule involving a set of transactions.

Assumption:
a transaction will always read an item before it writes that item.

Given a schedule for a set of transactions T_1, T_2, …, T_n

- The vertices of the conflict graph are the transaction identifiers.
- An edge from T_i to T_j denotes that the two transactions are conflicting, with T_i making the relevant access earlier.
- Sometimes the edge is labeled with the item involved in the conflict.

Determining serializability

Given a schedule S and a conflict graph how can we determine if the schedule is conflict serializable?

Determining serializability

Given a schedule S and a conflict graph how can we determine if the schedule is conflict serializable?

- A schedule is **conflict serializable** if its conflict graph is **acyclic**.
- Intuitively, a conflict between two transactions forces an execution order between them (topological sorting)

Determining serializability

Given a schedule S and a conflict graph how can we determine if the schedule is conflict serializable?

- A schedule is **conflict serializable** if its conflict graph is **acyclic**.
- Intuitively, a conflict between two transactions forces an execution order between them (topological sorting)

We use conflict serializability (not any other definition of serializability) because it has a practical implementation.

Conflict graph example

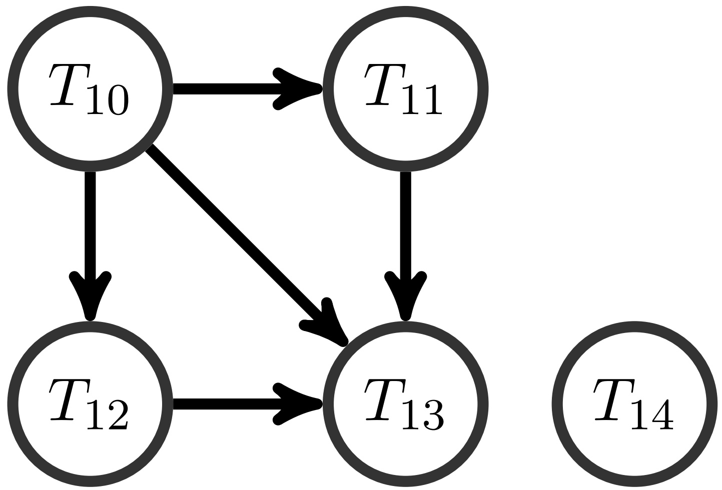
schedule S_6

T_{10}	T_{11}	T_{12}	T_{13}	T_{14}
	read(X, x)			
read(Y, y)				
read(Z, z)				
				read(V, v)
				read(W, w)
				write(W, w)
	read(Y, y)			
	write(Y, y)			
		read(Z, z)		
		write(Z, z)		
read(T, t)				
			read(Y, y)	
			write(Y, y)	
			read(Z, z)	
			write(Z, z)	
read(U, u)				

Conflict graph example

schedule S_6

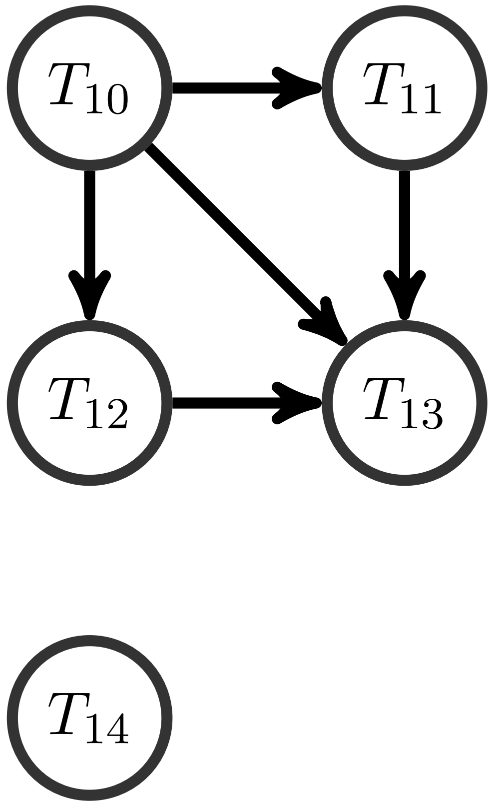
T_{10}	T_{11}	T_{12}	T_{13}	T_{14}
	read(X, x)			
read(Y, y)				
read(Z, z)				
				read(V, v)
				read(W, w)
				write(W, w)
	read(Y, y)			
	write(Y, y)			
		read(Z, z)		
		write(Z, z)		
read(T, t)				
			read(Y, y)	
			write(Y, y)	
			read(Z, z)	
			write(Z, z)	
read(U, u)				



Conflict graph example

schedule S_6

T_{10}	T_{11}	T_{12}	T_{13}	T_{14}
	read(X, x)			
read(Y, y)				
read(Z, z)				
				read(V, v)
				read(W, w)
				write(W, w)
	read(Y, y)			
	write(Y, y)			
		read(Z, z)		
		write(Z, z)		
read(T, t)				
			read(Y, y)	
			write(Y, y)	
			read(Z, z)	
			write(Z, z)	
read(U, u)				



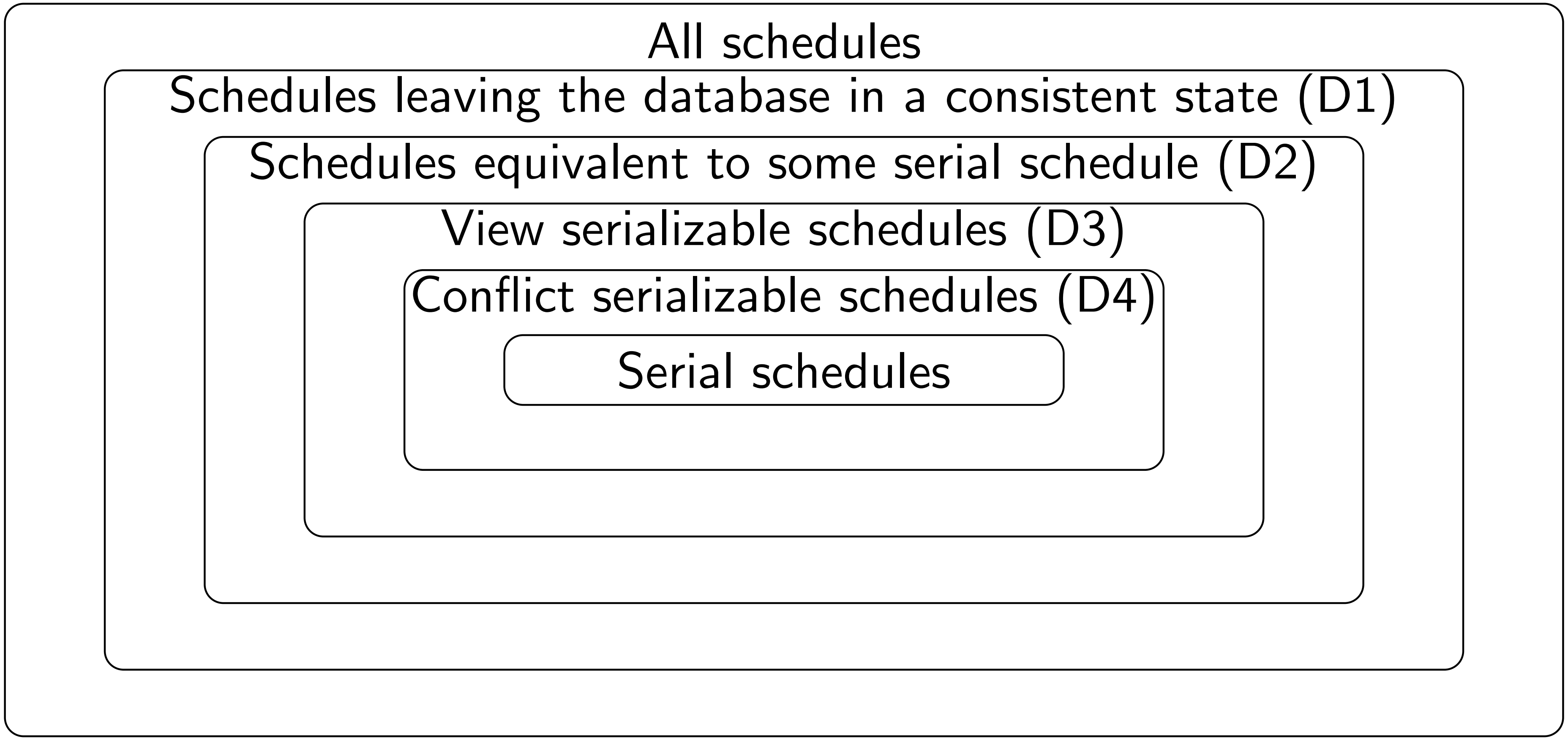
Which of the following are conflict equivalent serial schedules?

T_{10}, T_{11}, T_{12}, T_{13}, and T_{14} Yes

T_{14}, T_{10}, T_{12}, T_{11}, and T_{13} Yes

T_{14}, T_{13}, T_{12}, T_{11}, and T_{10} No

Relationship among schedules



Outline

Transaction Isolation and Atomicity

Transactions can fail

Recoverable schedules

schedule S_A	
T_i	T_j
read(X, x)	
write(X, x)	
	read(X, x)
	write(X, x)
	commit
rollback	

- If T_i fails, it must be rolled back to retain the **atomicity** property of transactions (see recovery).

Recoverable schedules

schedule S_A	
T_i	T_j
read(X, x)	
write(X, x)	
	read(X, x)
	write(X, x)
	commit
rollback	

- If T_i fails, it must be rolled back to retain the **atomicity** property of transactions (see recovery).
- If another transaction T_j has read a data item written by T_i, then T_j must also be rolled back.
  $\Rightarrow$ database systems must ensure that schedules are recoverable

Recoverable schedules

schedule S_A	
T_i	T_j
read(X, x)	
write(X, x)	
	read(X, x)
	write(X, x)
	commit
rollback	

- If T_i fails, it must be rolled back to retain the **atomicity** property of transactions (see recovery).
- If another transaction T_j has read a data item written by T_i, then T_j must also be rolled back.
  $\Rightarrow$ database systems must ensure that schedules are recoverable
- This schedule is not recoverable.

Recoverable schedules

A schedule is **recoverable** if for each pair of transactions T_i and T_j where T_j reads data items written by T_i, then T_i must commit before T_j commits.

schedule S_A	
T_i	T_j
read(X, x)	
write(X, x)	
rollback	
	read(X, x)
	write(X, x)
	commit

schedule S_B	
T_i	T_j
read(Y, y)	
	read(X, x)
write(Y, y)	
	write(X, x)
rollback	
	commit

Recoverable schedules

A schedule is **recoverable** if for each pair of transactions T_i and T_j where T_j reads data items written by T_i, then T_i must commit before T_j commits.

schedule S_A	
T_i	T_j
read(X, x)	
write(X, x)	
rollback	
	read(X, x)
	write(X, x)
	commit

schedule S_B	
T_i	T_j
read(Y, y)	
	read(X, x)
write(Y, y)	
	write(X, x)
rollback	
	commit

Is this schedule recoverable?

Is this schedule recoverable?

Recoverable schedules

A schedule is **recoverable** if for each pair of transactions T_i and T_j where T_j reads data items written by T_i, T_i must commit before T_j commits.

schedule S_A	
T_i	T_j
read(X, x)	
write(X, x)	
rollback	
	read(X, x)
	write(X, x)
	commit

recoverable

schedule S_B	
T_i	T_j
read(Y, y)	
	read(X, x)
write(Y, y)	
	write(X, x)
rollback	
	commit

recoverable

Cascading rollbacks

schedule S_{11}		
T_{22}	T_{23}	T_{24}
read(A, a)		
read(B, b)		
write(A, a)		
write(B, b)		
	read(A, a)	
		read(A, a)
		read(B, b)
rollback		

What happens if we need to rollback T_{22}?
Is this schedule recoverable?

Cascading rollbacks

schedule S_{11}		
T_{22}	T_{23}	T_{24}
read(A, a)		
read(B, b)		
write(A, a)		
write(B, b)		
	read(A, a)	
		read(A, a)
		read(B, b)
rollback		

- T_{22} rollback $\Rightarrow$ we have to rollback T_{23} and T_{24} because they read "dirty" data. (cascading rollbacks)
- This schedule is not cascadeless.
- But this schedule is recoverable.

Cascadeless schedules

A schedule is **cascadeless** if for each pair of transactions T_i and T_j, where T_j reads data items written by T_i, the commit operation of T_i must appear before the read by T_j.

schedule S_A

T_1	T_2	T_3
read(A, a)		
write(A, a)		
commit		
		read(A, a)
	read(A, a)	
	commit	
		commit

Cascadeless schedules

A schedule is **cascadeless** if for each pair of transactions T_i and T_j, where T_j reads data items written by T_i, the commit operation of T_i must appear before the read by T_j.

schedule $S_{11'}$

T_{22}	T_{23}	T_{24}
read(A, a)		
read(B, b)		
write(A, a)		
write(B, b)		
rollback		
	read(A, a)	
	commit	
		read(A, a)
		read(B, b)
		commit

This is also a recoverable schedule

Cascadeless schedules

A schedule is **cascadeless** if for each pair of transactions T_i and T_j, where T_j reads data items written by T_i, the commit operation of T_i must appear before the read by T_j.

schedule $S_{11'}$

T_{22}	T_{23}	T_{24}
read(A, a)		
read(B, b)		
write(A, a)		
write(B, b)		
rollback		
	read(A, a)	
	commit	
		read(A, a)
		read(B, b)
		commit

This is also a recoverable schedule

Cascading rollbacks could be avoided by only reading from committed transactions.

Cascadeless schedules

- Every cascadeless schedule is also recoverable.
- Cascading rollbacks can easily become expensive.
- It is desirable to restrict the schedules to those that are cascadeless.

Summary: transactions and schedules

- Each transaction preserves database consistency
- The serial execution of a set of transactions preserves database consistency
- In a concurrent execution, steps of a set of transactions may be interleaved
- A concurrent schedule is serializable if it is equivalent to a serial schedule
  - Conflict serializability
    Method of choice because it has a practical implementation
  - Conflict graphs
- Schedules must be recoverable and cascadeless

Learning goals

Learning goals: concurrency control

- Understand and use lock-based concurrency control
- Understand and use two-phase locking

Motivation

- Exclusive access to a database used by multiple users comes at the expense of throughput and runtime

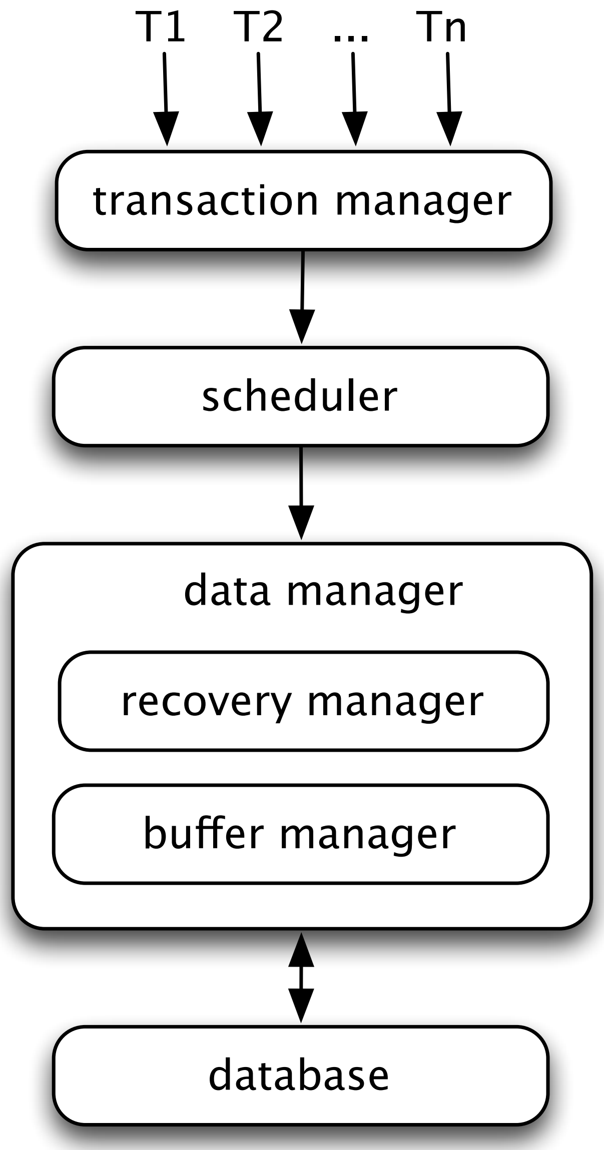
Outline I

Outline II

- Deadlock prevention

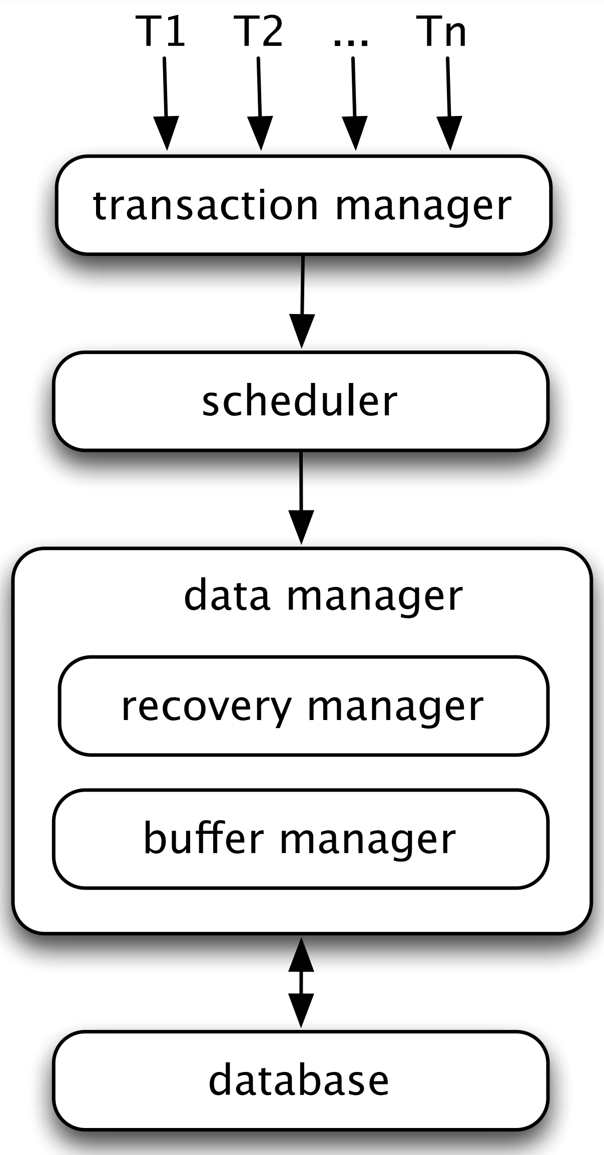
4. Recovery
   - Failure classification
   - Data storage
   - Log entries
   - Log-based recovery

Scheduler



Based on "Datenbanksysteme: Ein Einführung"
by Alfons Kemper and Andre Eickler, Oldenbourg Verlag 2011.

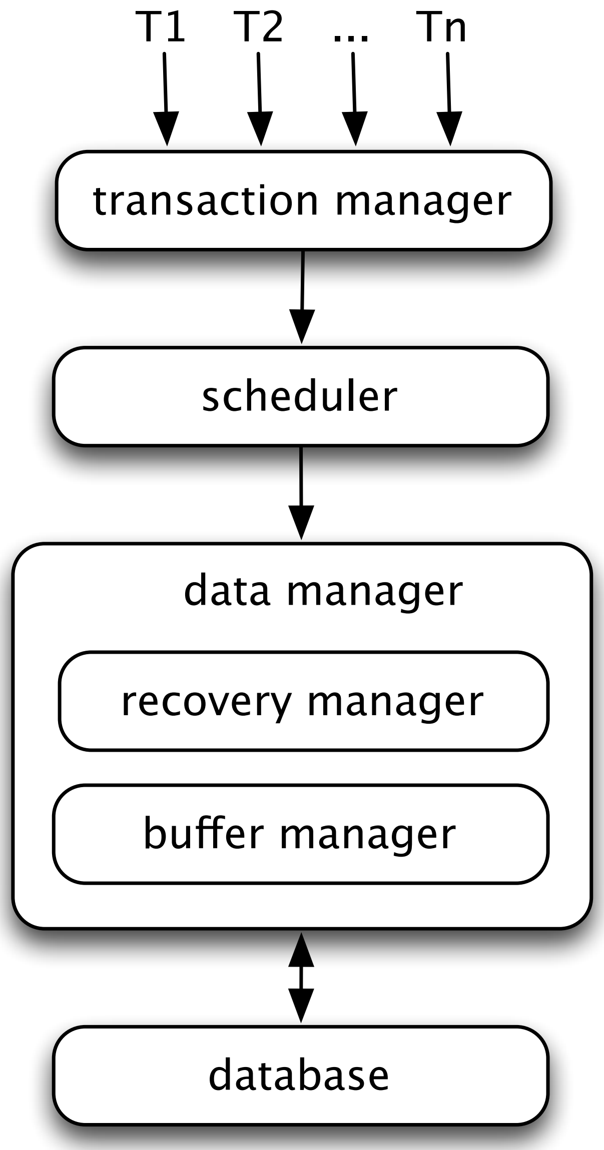
Scheduler



Task of the scheduler:
produce serializable schedules of instructions (transactions $T_1, \ldots, T_n$) that avoid cascading rollbacks

Based on "Datenbanksysteme: Ein Einführung"
by Alfons Kemper and Andre Eickler, Oldenbourg Verlag 2011.

Scheduler



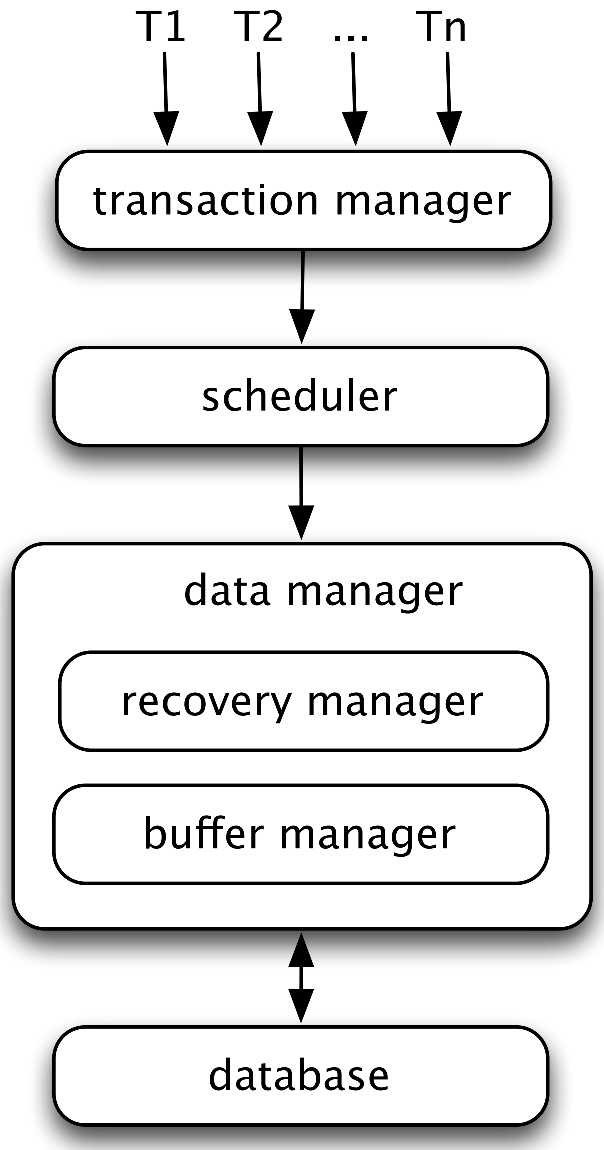
Task of the scheduler:
produce serializable schedules of instructions (transactions $T_1, \ldots, T_n$) that avoid cascading rollbacks

Realized by **synchronization** strategies

- pessimistic
  - lock-based synchronization
  - timestamp-based synchronization
- optimistic

Based on "Datenbanksysteme: Ein Einführung"
by Alfons Kemper and Andre Eickler, Oldenbourg Verlag 2011.

Scheduler



Task of the scheduler:
produce serializable schedules of instructions (transactions $T_1, \ldots, T_n$) that avoid cascading rollbacks

Realized by **synchronization** strategies

- pessimistic
  - **lock-based synchronization**
  - timestamp-based synchronization
- optimistic

Based on "Datenbanksysteme: Ein Einführung"
by Alfons Kemper and Andre Eickler, Oldenbourg Verlag 2011.

Lock-based synchronization

Ensuring (conflict) serializable schedules by **delaying** transactions that could violate serializability.

Lock-based synchronization

Ensuring (conflict) serializable schedules by **delaying** transactions that could violate serializability.

Two types of locks can be held on a data item Q

- S (shared, read lock)
- X (exclusive, write lock)

Operations on locks

- **lock_S(Q)** – set shared lock on data item Q
- **lock_X(Q)** – set exclusive lock on data item Q
- **unlock(Q)** – release lock on data item Q

Lock-based synchronization

Privileges associated with locks

A transaction holding

- an exclusive lock may issue a write or read access request on the item
- a shared lock may issue a read access request on the item

Lock-based synchronization

Privileges associated with locks

A transaction holding

- an exclusive lock may issue a write or read access request on the item
- a shared lock may issue a read access request on the item

Compatibility matrix

	NL	**S**	**X**
S	OK	OK	-
X	OK	-	-

NL – no lock

- Concurrent transactions can only be granted compatible locks
- A transaction might have to wait until a requested lock can be granted!

Problems with early unlocking

schedule S_7	
T_{15}	T_{16}
lock_X(B)	
read(B, b)	
b ← b - 50	
write(B, b)	
unlock(B)	
	lock_S(A)
	read(A, a)
	unlock(A)
	lock_S(B)
	read(B, b)
	unlock(B)
	display(A+B)
lock_X(A)	
read(A, A)	
a ← a + 50	
write(A, a)	
unlock(A)	

- Initially A = 100 and B = 200
- serial schedule T_{15};T_{16} prints 300
- serial schedule T_{16};T_{15} prints 300
- S_7 prints 250

Early unlocking can cause **incorrect** results (non-serializable schedules)

but allows for a higher degree of concurrency.

Problems with late unlocking

Conclusion: Let's delay unlocking until the end of the transaction.

schedule S_8	
T_{17}	T_{18}
lock_X(B)	
read(B, b)	
b ← b - 50	
write(B, b)	
	lock_S(A)
	read(A, a)
...	...
unlock(B)	**unlock(A)**

Problems with late unlocking

Conclusion: Let's delay unlocking until the end of the transaction.

schedule S_8	
T_{17}	T_{18}
lock_X(B)	
read(B, b)	
b ← b - 50	
write(B, b)	
	lock_S(A)
	read(A, a)
...	...
unlock(B)	**unlock(A)**

Is that a good conclusion?

Problems with late unlocking

Conclusion: Let's delay unlocking until the end of the transaction.

schedule S_8	
T_{17}	T_{18}
lock_X(B)	
read(B, b)	
b ← b - 50	
write(B, b)	
	lock_S(A)
	read(A, a)
...	...
unlock(B)	**unlock(A)**

- Late unlocking avoids non-serializable schedules.
  But it increases the chances of **deadlocks**.

Problems with late unlocking

Conclusion: Let's delay unlocking until the end of the transaction.

schedule S_8	
T_{17}	T_{18}
lock_X(B)	
read(B, b)	
b ← b - 50	
write(B, b)	
	lock_S(A)
	read(A, a)
...	...
unlock(B)	**unlock(A)**

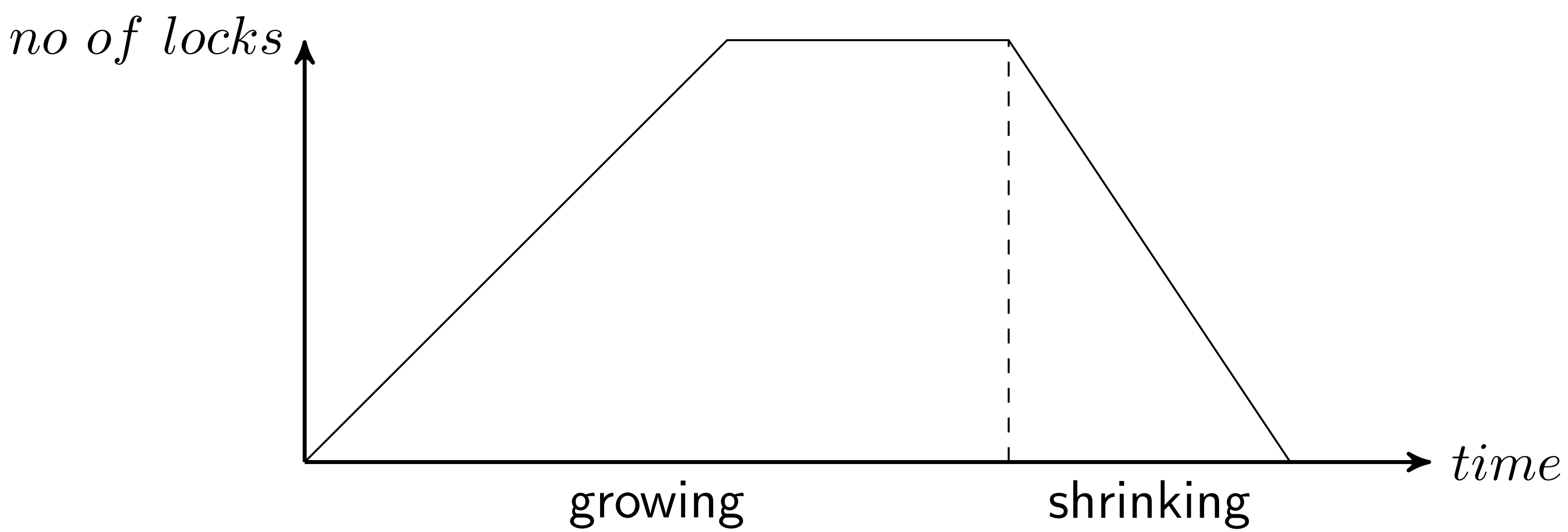
- Late unlocking avoids non-serializable schedules.
  But it increases the chances of **deadlocks**.
- Learn to live with it!

Outline

3. Concurrency control
   - Lock-based synchronization
   - Two-phase locking (2PL)
   - Lock conversion
   - Deadlock detection
   - Deadlock prevention

The Two-Phase Locking (2PL) protocol

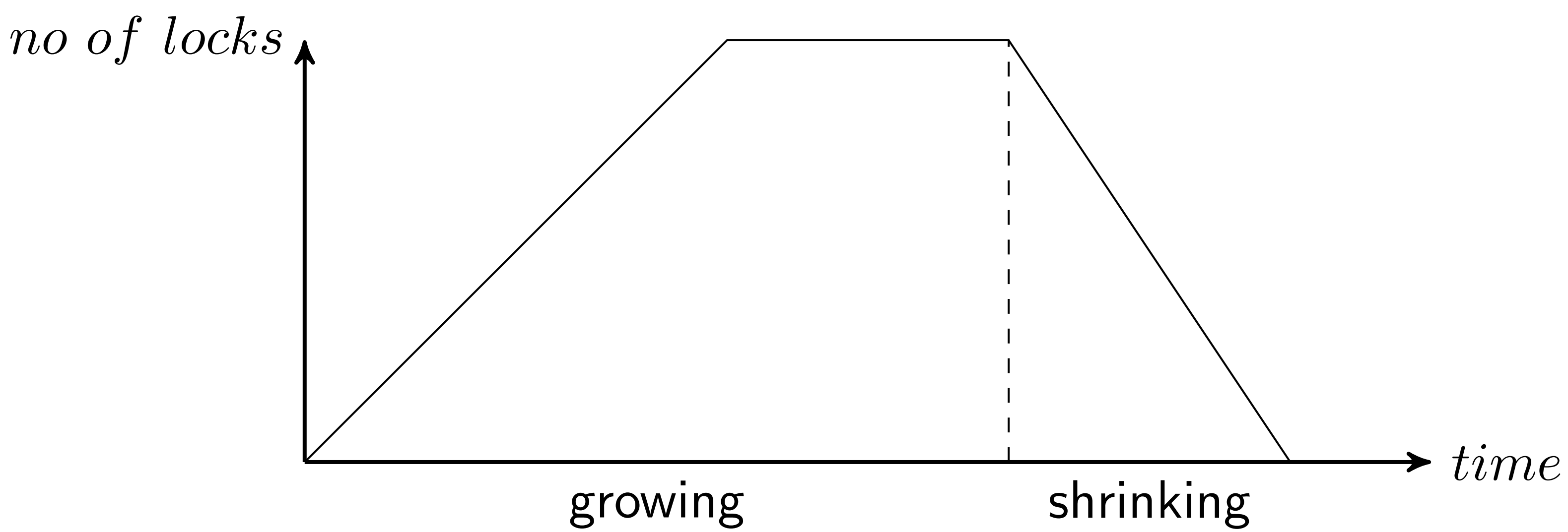
- First phase (growing phase):
  - Transaction may request locks.
  - Transaction may not release locks.
- Second phase (shrinking phase):
  - Transaction may not request locks.
  - Transaction may release locks.



The Two-Phase Locking (2PL) protocol

- First phase (growing phase):
  - Transaction may request locks.
  - Transaction may not release locks.
- Second phase (shrinking phase):
  - Transaction may not request locks.
  - Transaction may release locks.

When the first lock is released, the transaction moves from the first phase to the second phase.



2PL: yes or no?

schedule S_A
T_1
lock_X(A)
lock_X(B)
lock_X(C)
unlock(A)
unlock(C)
unlock(B)

2PL: yes or no?

schedule S_A
T_1
lock_X(A)
lock_X(B)
lock_X(C)
unlock(A)
unlock(C)
unlock(B)

yes

2PL: yes or no?

schedule S_A	schedule S_B	
T_1	T_2	T_3
lock_X(A)	**lock_X(A)**	
lock_X(B)	**lock_X(B)**	
lock_X(C)	**lock_X(C)**	
unlock(A)	**unlock(B)**	
unlock(C)		**lock_X(B)**
unlock(B)	**unlock(C)**	
	unlock(A)	
yes		**unlock(B)**

2PL: yes or no?

schedule S_A	schedule S_B	
T_1	T_2	T_3
lock_X(A)	**lock_X(A)**	
lock_X(B)	**lock_X(B)**	
lock_X(C)	**lock_X(C)**	
unlock(A)	**unlock(B)**	
unlock(C)		**lock_X(B)**
unlock(B)	**unlock(C)**	
	unlock(A)	
		unlock(B)

Schedule S_A: yes

Schedule S_B: yes

2PL: yes or no?

schedule S_A	schedule S_B		schedule S_C	
T_1	T_2	T_3	T_4	T_5
lock_X(A)	**lock_X(A)**		**lock_X(A)**	
lock_X(B)	**lock_X(B)**			**lock_X(B)**
lock_X(C)	**lock_X(C)**			**lock_X(C)**
unlock(A)	**unlock(B)**			**unlock(C)**
unlock(C)		**lock_X(B)**		**unlock(B)**
unlock(B)	**unlock(C)**		**unlock(A)**	
	unlock(A)			
yes				
		unlock(B)		
	yes			

2PL: yes or no?

schedule S_A	schedule S_B		schedule S_C	
T_1	T_2	T_3	T_4	T_5
lock_X(A)	**lock_X(A)**		**lock_X(A)**	
lock_X(B)	**lock_X(B)**			**lock_X(B)**
lock_X(C)	**lock_X(C)**			**lock_X(C)**
unlock(A)	**unlock(B)**			**unlock(C)**
unlock(C)		**lock_X(B)**		**unlock(B)**
unlock(B)	**unlock(C)**		**unlock(A)**	
	unlock(A)			
		unlock(B)		
yes	yes		yes	

2PL: yes or no?

schedule S_A	schedule S_B		schedule S_C		schedule S_D
T_1	T_2	T_3	T_4	T_5	T_6
lock_X(A)	**lock_X(A)**		**lock_X(A)**		**lock_X(A)**
lock_X(B)	**lock_X(B)**			**lock_X(B)**	**lock_X(B)**
lock_X(C)	**lock_X(C)**			**lock_X(C)**	**unlock(B)**
unlock(A)	**unlock(B)**			**unlock(C)**	**lock_X(C)**
unlock(C)		**lock_X(B)**		**unlock(B)**	**unlock(A)**
unlock(B)	**unlock(C)**		**unlock(A)**		**unlock(C)**
	unlock(A)				
yes			yes		
		unlock(B)			
	yes				

2PL: yes or no?

schedule S_A	schedule S_B		schedule S_C		schedule S_D
T_1	T_2	T_3	T_4	T_5	T_6
lock_X(A)	**lock_X(A)**		**lock_X(A)**		**lock_X(A)**
lock_X(B)	**lock_X(B)**			**lock_X(B)**	**lock_X(B)**
lock_X(C)	**lock_X(C)**			**lock_X(C)**	**unlock(B)**
unlock(A)	**unlock(B)**			**unlock(C)**	**lock_X(C)**
unlock(C)		**lock_X(B)**		**unlock(B)**	**unlock(A)**
unlock(B)	**unlock(C)**		**unlock(A)**		**unlock(C)**
	unlock(A)				
		unlock(B)			
yes	yes		yes		no

Characteristics of the 2PL protocol

- 2PL produces only serializable schedules
  - It ensures conflict serializability
  - 2PL produces a subset of all possible serializable schedules
- 2PL does not prevent deadlocks
- 2PL does not prevent cascading rollbacks
  - "Dirty" reads are possible (reading from non-committed transactions)

Cascading rollbacks

One aborted transaction can cause other transactions to abort.

schedule S_{11}		
T_{22}	T_{23}	T_{24}
lock_X(A)		
lock_X(B)		
unlock(A)		
	lock_X(A)	
	unlock(A)	
		lock_X(A)
abort		

- These schedules use two-phase locking
- When T_{22} aborts $\Rightarrow$ T_{23} and T_{24} also have to abort

Cascading rollbacks

One aborted transaction can cause other transactions to abort.

schedule S_{11}		
T_{22}	T_{23}	T_{24}
lock_X(A)		
lock_X(B)		
unlock(A)		
	lock_X(A)	
	unlock(A)	
		lock_X(A)
abort		

- These schedules use two-phase locking
- When T_{22} aborts $\Rightarrow$ T_{23} and T_{24} also have to abort

How to eliminate these cascading rollbacks?

Cascading rollbacks

One aborted transaction can cause other transactions to abort.

schedule S_{11}			schedule $S_{11'}$		
T_{22}	T_{23}	T_{24}	$T_{22'}$	$T_{23'}$	$T_{24'}$
lock_X(A)			**lock_X(A)**		
lock_X(B)			**lock_X(B)**		
unlock(A)			**unlock(A)**		
	lock_X(A)		commit		
	unlock(A)			**lock_X(A)**	
		lock_X(A)		**unlock(A)**	
abort				commit	
					lock_X(A)

- These schedules use two-phase locking
- When T_{22} aborts $\Rightarrow$ T_{23} and T_{24} also have to abort

How to eliminate these cascading rollbacks?
Don't let transactions read uncommitted data: problem fixed in $S_{11'}$

Strict and rigorous two phase locking

Strict 2PL

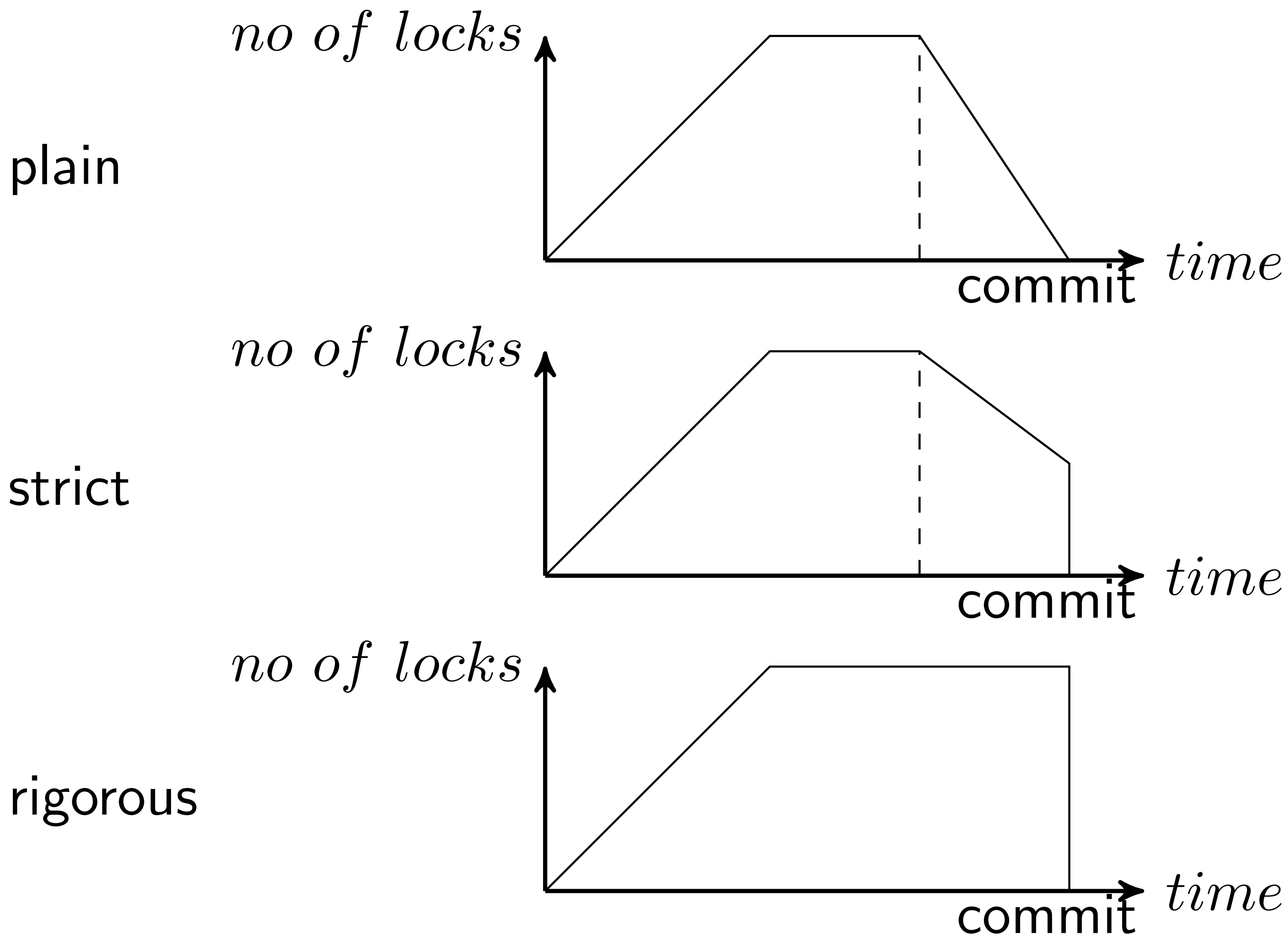
- **Exclusive** locks are not released before the transaction commits
- Prevents “dirty reads“

Rigorous 2PL:

- **All** locks are released after commit time
- Transactions can be serialized in the order they commit

- Advantage:
  no cascading rollbacks
- Disadvantage:
  loss of potential concurrency

Overview: 2PL protocols



Lock conversion

Goal: Apply 2PL but allow for a higher degree of concurrency

- First phase
  - Acquire an S-lock on a data item
  - Acquire an X-lock on a data item
  - Convert (upgrade) an S-lock to an X-lock
- Second phase
  - Release an S-lock
  - Release an X-lock
  - Convert (downgrade) an X-lock to an S-lock

- This protocol still ensures serializability
- It relies on the application programmer to insert the appropriate locks

Plain, strict, or rigorous 2PL?

schedule S_1
T_1
lock_S(A)
lock_S(B)
lock_X(B)
lock_S(C)
unlock(A)
unlock(C)
commit

Plain, strict, or rigorous 2PL?

schedule S_1
T_1
lock_S(A)
lock_S(B)
lock_X(B)
lock_S(C)
unlock(A)
unlock(C)
commit

strict

Plain, strict, or rigorous 2PL?

schedule S_1
T_1
lock_S(A)
lock_S(B)
lock_X(B)
lock_S(C)
unlock(A)
unlock(C)
commit

strict

schedule S_2
T_2
lock_S(A)
lock_S(B)
lock_X(B)
commit

Plain, strict, or rigorous 2PL?

schedule S_1
T_1
lock_S(A)
lock_S(B)
lock_X(B)
lock_S(C)
unlock(A)
unlock(C)
commit

strict

schedule S_2
T_2
lock_S(A)
lock_S(B)
lock_X(B)
commit

rigorous

Plain, strict, or rigorous 2PL?

schedule S_1
T_1
lock_S(A)
lock_S(B)
lock_X(B)
lock_S(C)
unlock(A)
unlock(C)
commit

strict

schedule S_2
T_2
lock_S(A)
lock_S(B)
lock_X(B)
commit

rigorous

schedule S_3
T_3
lock_S(A)
lock_S(B)
lock_X(B)
unlock(B)
lock_S(C)
unlock(A)
commit

Plain, strict, or rigorous 2PL?

schedule S_1
T_1
lock_S(A)
lock_S(B)
lock_X(B)
lock_S(C)
unlock(A)
unlock(C)
commit

strict

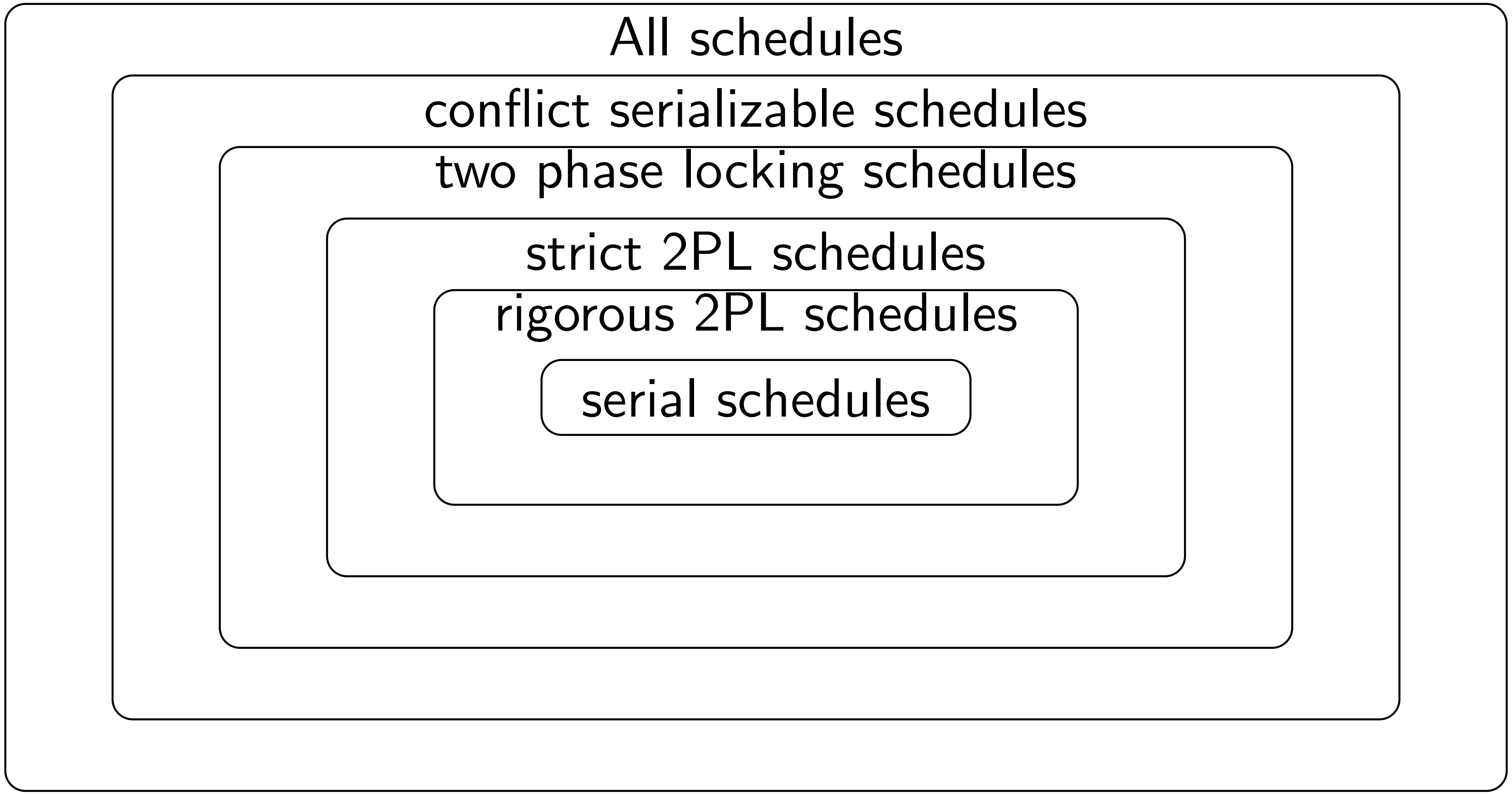
schedule S_2
T_2
lock_S(A)
lock_S(B)
lock_X(B)
commit

rigorous

schedule S_3
T_3
lock_S(A)
lock_S(B)
lock_X(B)
unlock(B)
lock_S(C)
unlock(A)
commit

not two phase

Overview of 2PL schedules



Outline

3 Concurrency control
- Lock-based synchronization
- Two-phase locking (2PL)
- Lock conversion
- Deadlock detection
- Deadlock prevention

Deadlocks

2PL does not prevent deadlocks

T_1	T_2	
lock_X(A)		
	lock_S(B)	
	read(B)	
read(A)		
write(A)		
lock_X(B)		T_1 needs to wait for T_2
	lock_S(A)	T_2 needs to wait for T_1
...	...	$\Rightarrow$ deadlock

Solutions

- detection and recovery
- prevention
- timeout

Deadlocks

2PL does not prevent deadlocks

T_1	T_2	
lock_X(A)		
	lock_S(B)	
	read(B)	
read(A)		
write(A)		
lock_X(B)		T_1 needs to wait for T_2
	lock_S(A)	T_2 needs to wait for T_1
...	...	$\Rightarrow$ deadlock

Solutions

- **detection and recovery**
- **prevention**
- timeout

Deadlock detection

Create a "Wait-for graph" and check for cycles

- One node for each active transaction T_i
- Edge $T_i \rightarrow T_j$ if T_i waits for the release of locks by T_j

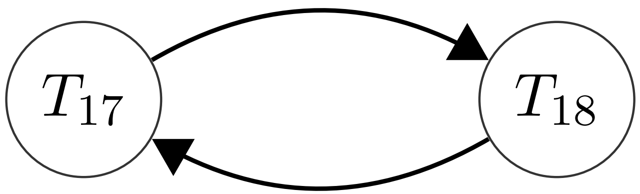
A deadlock exists if the wait-for graph has a cycle

Deadlock detection

If a deadlock is detected

- Select an appropriate victim
- Abort the victim and release its locks

schedule S_8	
T_{17}	T_{18}
lock_X(B)	
read(B, b)	
b $\leftarrow$ b - 50	
write(B, b)	
	lock_S(A)
	read(A, a)
	lock_S(B)
lock_X(A)	



Deadlock detection example

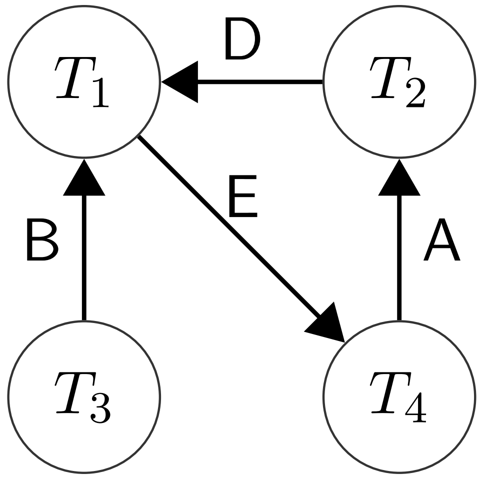
schedule S_A

T_1	T_2	T_3	T_4
	lock_X(A)		
lock_X(B)			
lock_X(C)			
		lock_X(B)	
lock_X(D)			
			lock_X(E)
	lock_X(D)		
			lock_X(A)
lock_X(E)			

Deadlock detection example

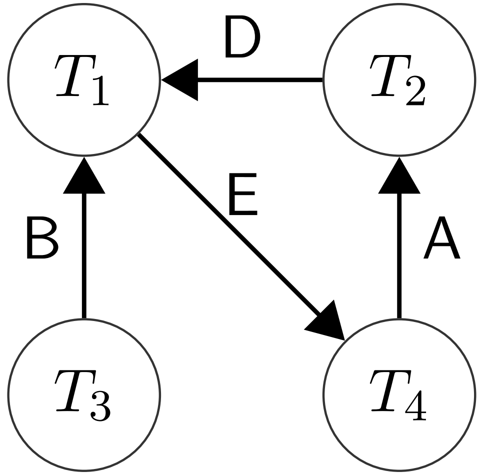
schedule S_A

T_1	T_2	T_3	T_4
	lock_X(A)		
lock_X(B)			
lock_X(C)			
		lock_X(B)	
lock_X(D)			
			lock_X(E)
	lock_X(D)		
			lock_X(A)
lock_X(E)			



Deadlock detection example

schedule S_A			
T_1	T_2	T_3	T_4
	lock_X(A)		
lock_X(B)			
lock_X(C)			
		lock_X(B)	
lock_X(D)			
			lock_X(E)
	lock_X(D)		
			lock_X(A)
lock_X(E)			



Cycle between T_1, T_4, and T_2
$\Rightarrow$ deadlock detected

Rollback of one or multiple involved transactions to release the deadlock

Rollback candidates

Choosing a good victim transaction

Rollback of one or more transactions that are involved in the cycle

- The latest (minimization of rollback effort)
- The one holding the most locks (maximization of released resources)

Rollback candidates

Choosing a good victim transaction

Rollback of one or more transactions that are involved in the cycle

- The latest (minimization of rollback effort)
- The one holding the most locks (maximization of released resources)

Prevent that always the same victim is chosen (starvation)

- "rollback counter"
  $\rightarrow$ above a certain threshold: no more rollbacks to break deadlocks

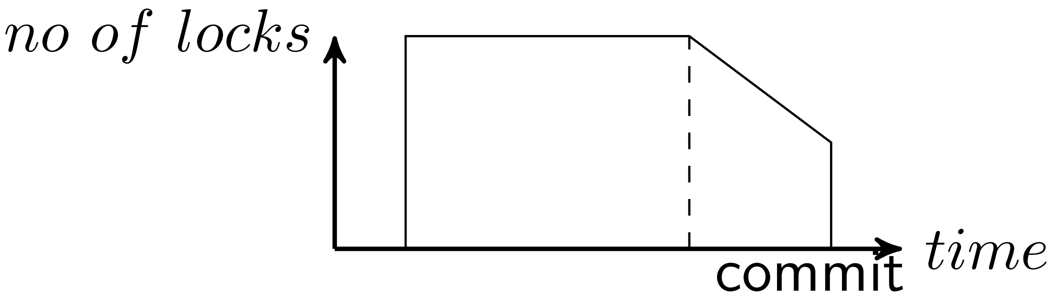
Outline

3. Concurrency control
   - Lock-based synchronization
   - Two-phase locking (2PL)
   - Lock conversion
   - Deadlock detection
   - **Deadlock prevention**

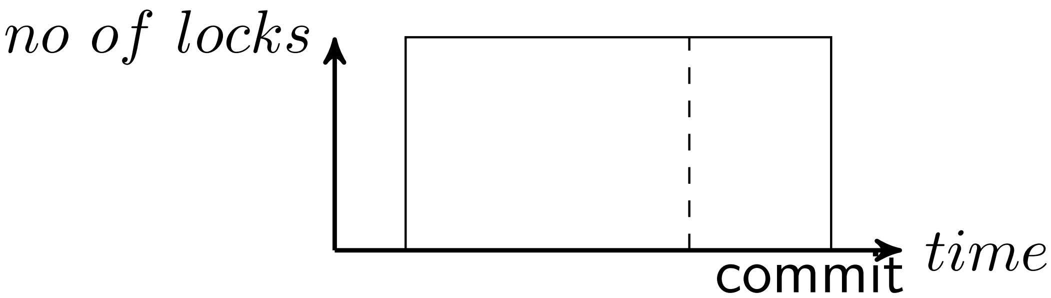
Conservative 2PL protocol

- 2PL as well as strict and rigorous 2PL do not prevent deadlocks
- Additional requirement:
  All locks (shared and exclusive) are obtained right in the beginning of a transaction

conservative strict 2PL



conservative rigorous 2PL



Only applicable for a few applications

Summary: concurrency control

- Many concurrency control protocols have been developed
  - Main goal: allowing only serializable, recoverable, and cascadeless schedules
  - Two-phase locking
    Most relational DBMS's use rigorous two-phase locking
- Deadlock detection (wait-for graph) and prevention (conservative 2PL)
- Serializability vs. concurrency

Learning goals

Learning goals: recovery

- Understanding basic logging algorithms
- Understanding the importance of atomicity and durability

Motivation

- Communicating to the user that a transaction was successful without guaranteeing that the effect is permanent can easily become expensive for commercial applications.
- We want to preserve consistency and availability even in the case of failures.

Outline

4 Recovery
- Failure classification
- Data storage
- Log entries
- Log-based recovery

Recovery

"Problems" with transactions

- Atomicity
  - Transactions may abort (rollback)
- Durability
  - What if a DBMS crashes?

The DBMS ensures that a transaction

- either completes and has a permanent result (committed transaction) or
- has no effect at all on the database (aborted transaction).

Recovery

"Problems" with transactions

- Atomicity
  - Transactions may abort (rollback)
- Durability
  - What if a DBMS crashes?

The DBMS ensures that a transaction

- either completes and has a permanent result (committed transaction) or
- has no effect at all on the database (aborted transaction).

The role of the **recovery** component is to ensure atomicity and durability of transactions in the presence of system failures.

How can durability be guaranteed?

- A transaction changes data in main memory
- Data is **not yet** written to the hard disk
- Transaction commits

User assumes that the transaction was successfully completed and all its changes are persistently stored in the database.

How can durability be guaranteed?

- A transaction changes data in main memory
- Data is **not yet** written to the hard disk
- Transaction commits

User assumes that the transaction was successfully completed and all its changes are persistently stored in the database.

- What happens when there is a blackout?
- What data is in the database?

How can durability be guaranteed?

- A transaction changes data in main memory
- Data is **partially** written to the hard disk
- Transaction commits

User assumes that the transaction was successfully completed and all its changes are persistently stored in the database.

How can durability be guaranteed?

- A transaction changes data in main memory
- Data is **partially** written to the hard disk
- Transaction commits

User assumes that the transaction was successfully completed and all its changes are persistently stored in the database.

- What happens when there is a blackout?
- What data is in the database?

How can durability be guaranteed?

- A transaction changes data in main memory
- Data is **completely** written to the hard disk
- Transaction commits

User assumes that the transaction was successfully completed and all its changes are persistently stored in the database.

How can durability be guaranteed?

- A transaction changes data in main memory
- Data is **completely** written to the hard disk
- Transaction commits

User assumes that the transaction was successfully completed and all its changes are persistently stored in the database.

- What happens if there is a hardware failure
  $\Rightarrow$ loss of a hard disk
- What data is in the database?

How can durability be guaranteed?

- A transaction changes data in main memory
- Data is **completely** written to **multiple** hard disks
- Transaction commits

User assumes that the transaction was successfully completed and all its changes are persistently stored in the database.

How can durability be guaranteed?

- A transaction changes data in main memory
- Data is **completely** written to **multiple** hard disks
- Transaction commits

User assumes that the transaction was successfully completed and all its changes are persistently stored in the database.

- What happens if there is a fire, flood, earthquake, or. . . ?
  $\Rightarrow$ all hard disks are lost
- What data is in the database?

How can durability be guaranteed?

- A transaction changes data in main memory
- Data is **completely** written to **multiple** hard disks and the disks are located at **multiple geographically distributed** computing centers
- Transaction commits

User assumes that the transaction was successfully completed and all its changes are persistently stored in the database.

How can durability be guaranteed?

- A transaction changes data in main memory
- Data is **completely** written to **multiple** hard disks and the disks are located at **multiple geographically distributed** computing centers
- Transaction commits

User assumes that the transaction was successfully completed and all its changes are persistently stored in the database.

- What happens if there is a fire, flood, earthquake, or. . . ? at all computing centers at the same time?
  $\Rightarrow$ all computing centers and all hard disks are lost
- What data is in the database?

Durability

- Durability is **relative** and depends on the number of copies and the geographical location.
- Guarantees only possible if
  - we first update the copies and
  - notify the user afterwards that a transaction's commit was successful

We hence assume that the WAL (Write Ahead Logging) rule is satisfied.

Durability

- Durability is **relative** and depends on the number of copies and the geographical location.
- Guarantees only possible if
  - we first update the copies and
  - notify the user afterwards that a transaction's commit was successful

We hence assume that the WAL (Write Ahead Logging) rule is satisfied.

Variations of applying the WAL rule:

- Log-based recovery
- Full redundancy: mirroring/shadowing all data on multiple computers (disks, computing centers) that redundantly do the same

Durability

- Durability is **relative** and depends on the number of copies and the geographical location.
- Guarantees only possible if
  - we first update the copies and
  - notify the user afterwards that a transaction's commit was successful

We hence assume that the WAL (Write Ahead Logging) rule is satisfied.

Variations of applying the WAL rule:

- **Log-based recovery**
- Full redundancy: mirroring/shadowing all data on multiple computers (disks, computing centers) that redundantly do the same

Failure classification

Transaction failure (failure of a not yet committed transaction)

- Undo the changes of the transaction

Failure classification

Transaction failure (failure of a not yet committed transaction)

- Undo the changes of the transaction

System crash (failure with main memory loss)

- Changes of committed transactions must be preserved
- Changes of all non-committed transactions need to be undone

Failure classification

Transaction failure (failure of a not yet committed transaction)

- Undo the changes of the transaction

System crash (failure with main memory loss)

- Changes of committed transactions must be preserved
- Changes of all non-committed transactions need to be undone

Disk failure (failure with hard disk loss)

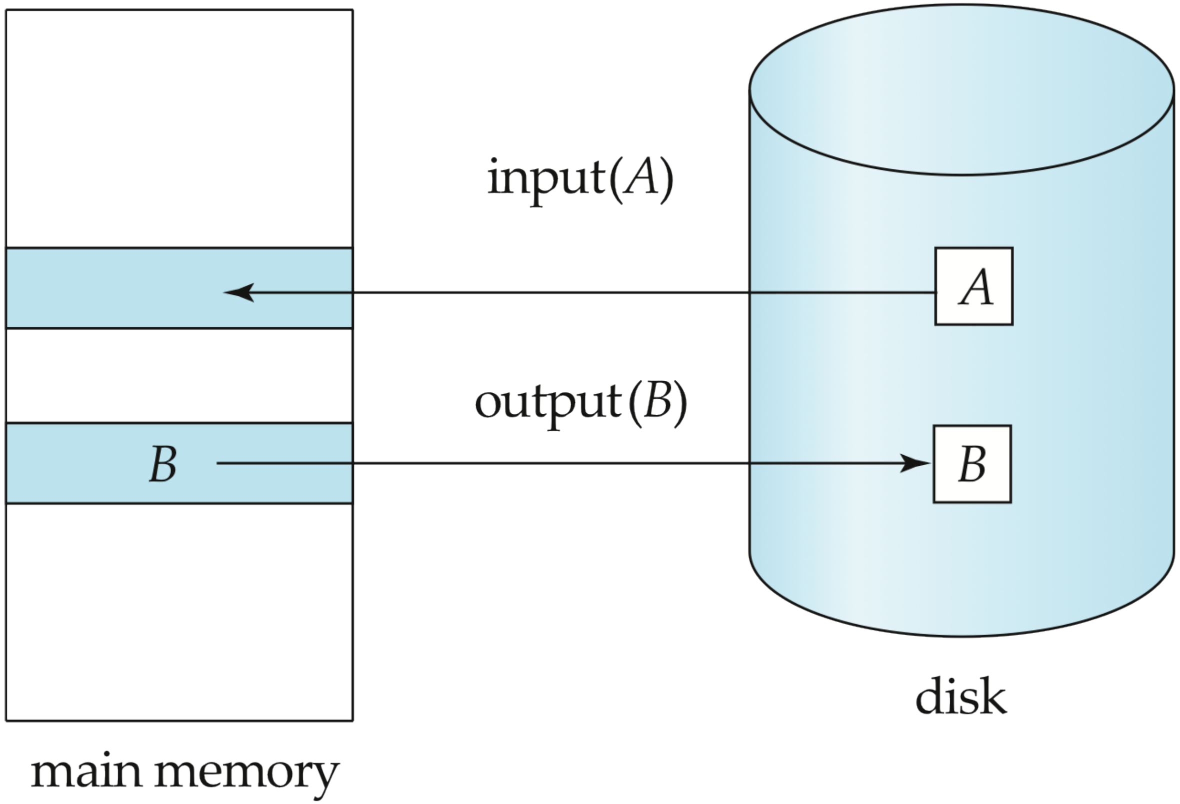
- Recovery based on archives/dumps

Outline

4. Recovery
   - Failure classification
   - Data storage
   - Log entries
   - Log-based recovery

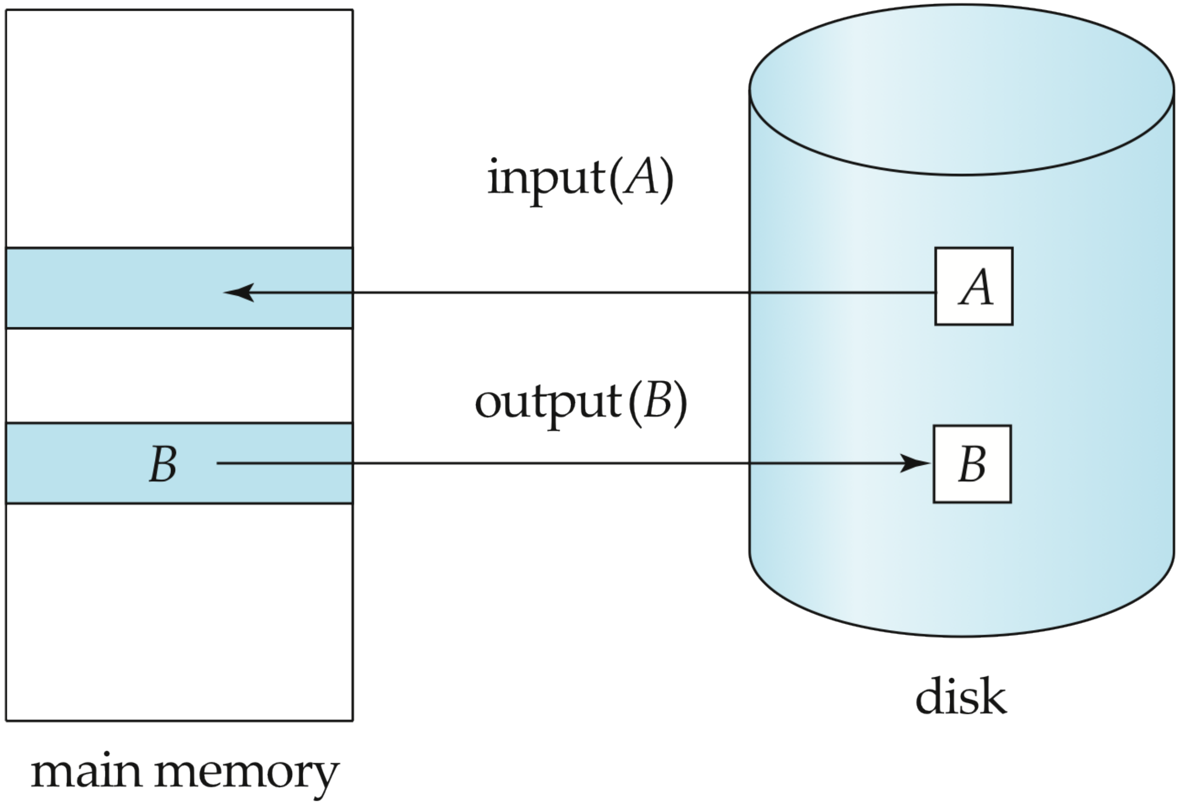
Two-level storage hierarchy

Data is organized in pages and blocks



Two-level storage hierarchy

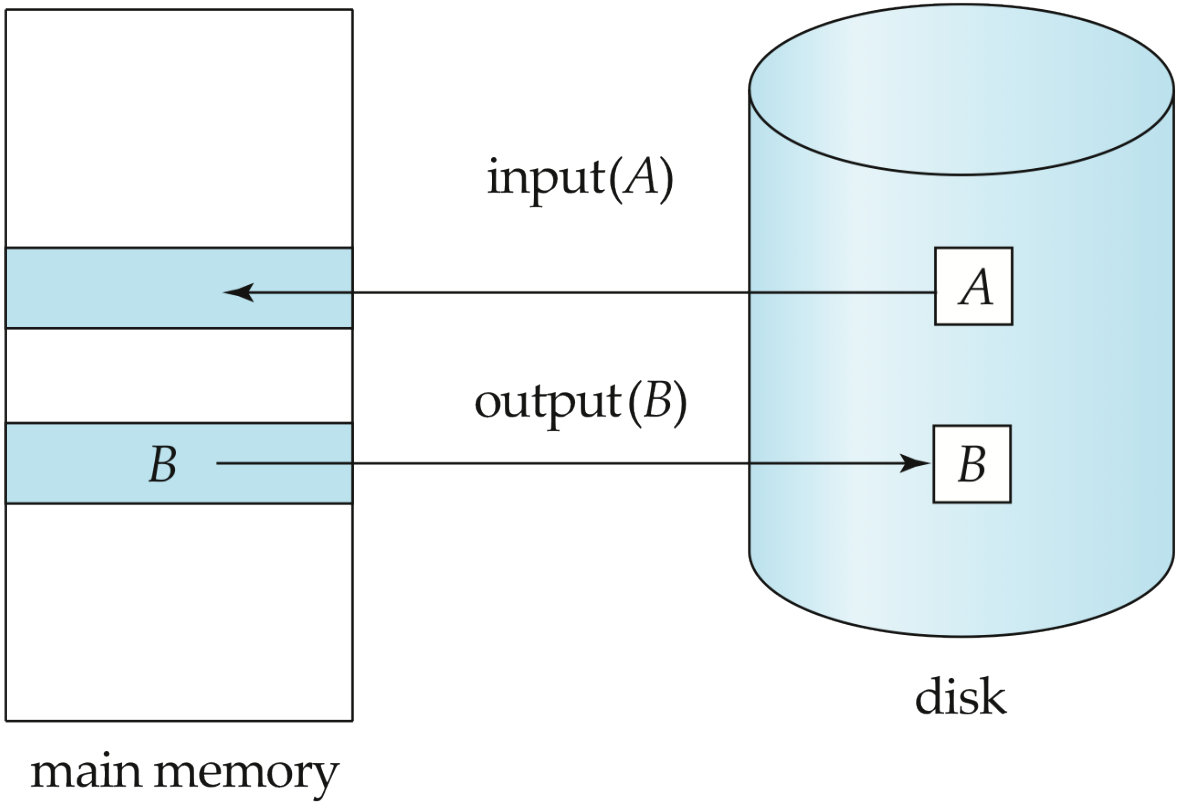
Data is organized in pages and blocks



- Volatile storage (main memory buffer)
- Non-volatile storage (hard disk)
- Stable storage (RAIDS, remote backups,. . .)

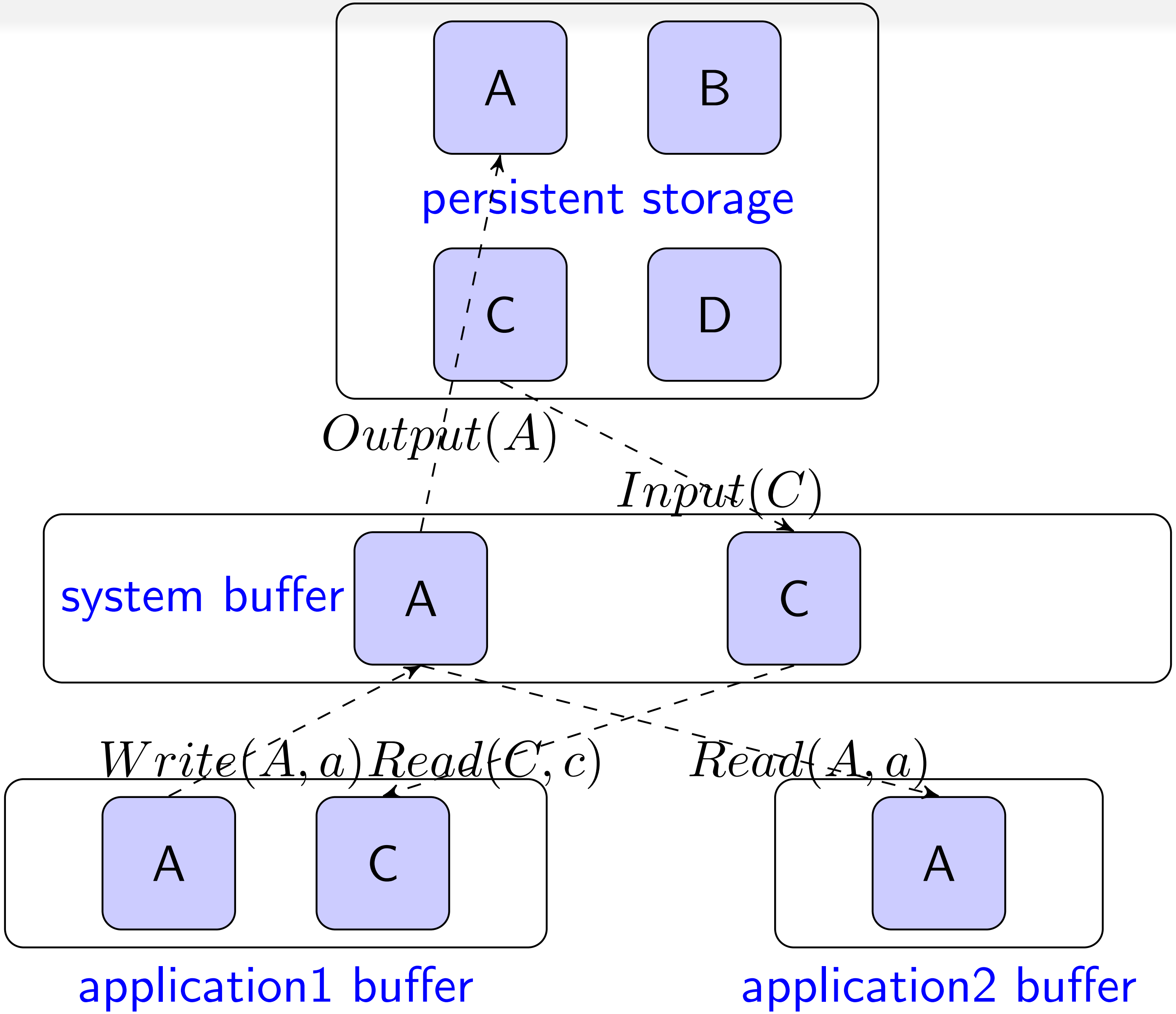
Two-level storage hierarchy

Data is organized in pages and blocks



- **Volatile storage** (main memory buffer)
- **Non-volatile storage** (hard disk)
- Stable storage (RAIDS, remote backups,. . .)

Movement of values



Storage operations

Transactions access and update the database

- Operations for moving blocks with data items between disk and main memory (the system buffer)
  - **Input(Q)**
    transfer block containing data item Q to main memory
  - **Output(Q)**
    transfer block containing Q to disk & replace

Storage operations

Transactions access and update the database

- Operations for moving blocks with data items between disk and main memory (the system buffer)
  - **Input(Q)**
    transfer block containing data item Q to main memory
  - **Output(Q)**
    transfer block containing Q to disk & replace
- Operations for moving values between data items and application variables
  - **read(Q,q)**
    assigns the value of data item Q to variable q
  - **write(Q,q)**
    assigns the value of variable q to data item Q

Outline

The WAL rule for log-based recovery

WAL (Write Ahead Logging)

- Before a transaction enters the **commit** state, “all its” log entries have to be written to stable storage, incl. the commit log entry
- Before a modified page (or block) in main memory can be written to the database (non-volatile storage), “all its” log entries have to be written to stable storage

Logging

During normal operation

- When starting, a transaction T registers itself in the **log**: [T start]

Logging

During normal operation

- When starting, a transaction T registers itself in the **log**: [T start]
- When modifying data item X by write(X, x)
  1. Add log entry with
     - [T, X, V-old, V-new]
     - transaction's ID (i.e., T)
     - data item name (i.e., X)
     - old value of the item
     - new value of the item

Logging

During normal operation

- When starting, a transaction T registers itself in the **log**: [T start]
- When modifying data item X by write(X, x)
  1. Add log entry with
     - [T, X, V-old, V-new]
     - transaction's ID (i.e., T)
     - data item name (i.e., X)
     - old value of the item
     - new value of the item
  2. Write the new value of X

Logging

During normal operation

- When starting, a transaction T registers itself in the **log**: [T start]
- When modifying data item X by write(X, x)
  1. Add log entry with
     - [T, X, V-old, V-new]
     - transaction's ID (i.e., T)
     - data item name (i.e., X)
     - old value of the item
     - new value of the item
  2. Write the new value of X

The buffer manager asynchronously outputs the value to disk later

Logging

During normal operation

- When starting, a transaction T registers itself in the **log**: [T start]
- When modifying data item X by write(X, x)
  1. Add log entry with
     - [T, X, V-old, V-new]
     - transaction's ID (i.e., T)
     - data item name (i.e., X)
     - old value of the item
     - new value of the item
  2. Write the new value of X

  The buffer manager asynchronously outputs the value to disk later
- When finishing, a transaction T appends [T commit] to the log, T then commits

Logging

During normal operation

- When starting, a transaction T registers itself in the **log**: [T start]
- When modifying data item X by write(X, x)
  1. Add log entry with
     - [T, X, V-old, V-new]
     - transaction's ID (i.e., T)
     - data item name (i.e., X)
     - old value of the item
     - new value of the item
  2. Write the new value of X

  The buffer manager asynchronously outputs the value to disk later
- When finishing, a transaction T appends [T commit] to the log, T then commits

  The transaction commits precisely when the commit entry (after all previous entries for this transaction) is output to the log!

Structure of a log entry (log record)

[TID, DID, old, new]

- **TID** identifier of the transaction that caused the update
- **DID** data item identifier
  location on disk (page, block, offset)
- **old** value of the data item before the update
- **new** value of the data item after the update

Structure of a log entry (log record)

[TID, DID, old, new]

TID	identifier of the transaction that caused the update
DID	data item identifier location on disk (page, block, offset)
old	value of the data item before the update
new	value of the data item after the update

Additional entries

start	Transaction TID has started	[TID start]
commit	Transaction TID has committed	[TID commit]
abort	Transaction TID has aborted	[TID abort]

Log entry example

schedule S_1		
T_1	T_2	T_3
begin		
read(B, b)		
b $\leftarrow$ b+100		
write(B, b)		
commit		
	begin	
	read(D, d)	
	d $\leftarrow$ d+470	
	write(D, d)	
	commit	
		begin
		read(D, d)
		read(E, e)
		d $\leftarrow$ d-10
		write(D, d)
		e $\leftarrow$ e-20
		write(E, e)
		commit

[TID, DID, old, new]

[T1 start]
[T1, B, 300, 400]
[T1 commit]
[T2 start]
[T2, D, 60, 530]
[T2 commit]
[T3 start]
[T3, D, 530, 520]
[T3, E, 70, 50]
[T3 commit]

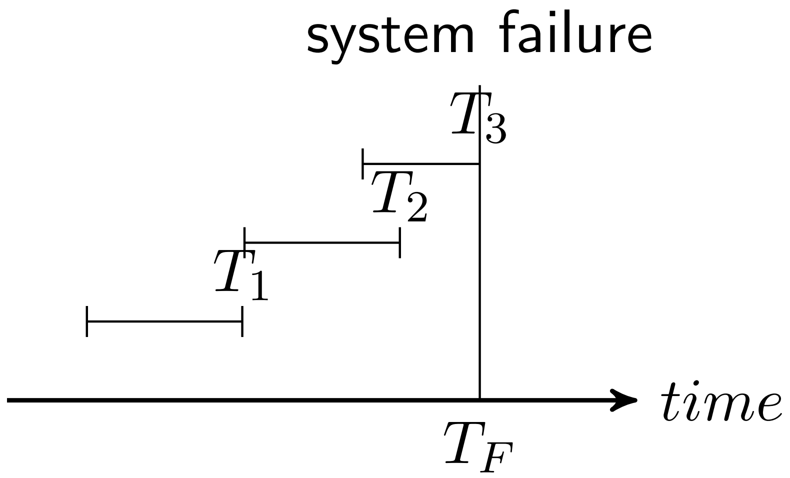
Outline

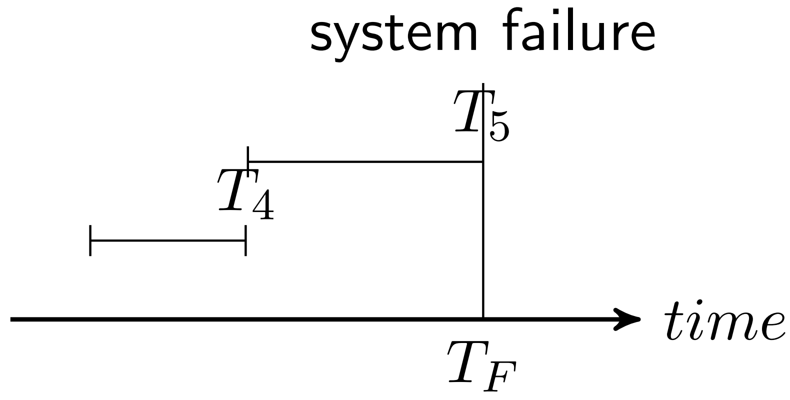
4. Recovery
   - Failure classification
   - Data storage
   - Log entries
   - Log-based recovery

Log-based recovery

Operations to recover from failures

- **Redo**: perform the changes to the database again
- **Undo**: restore database to state prior to execution



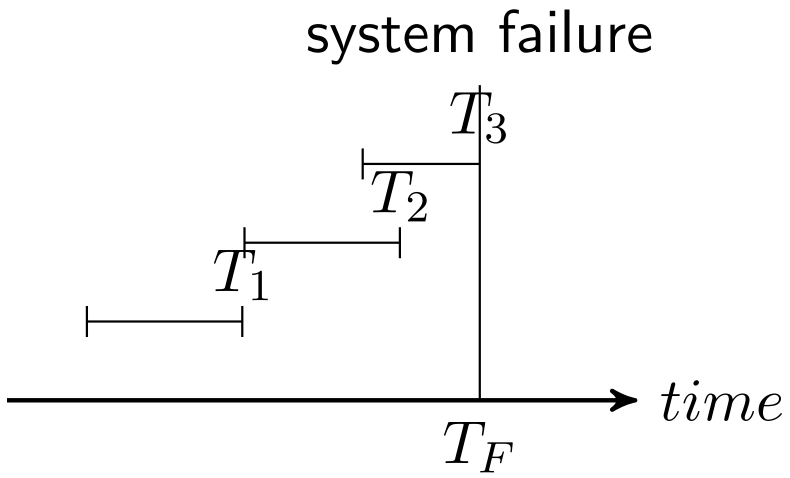


What to do with the transactions?

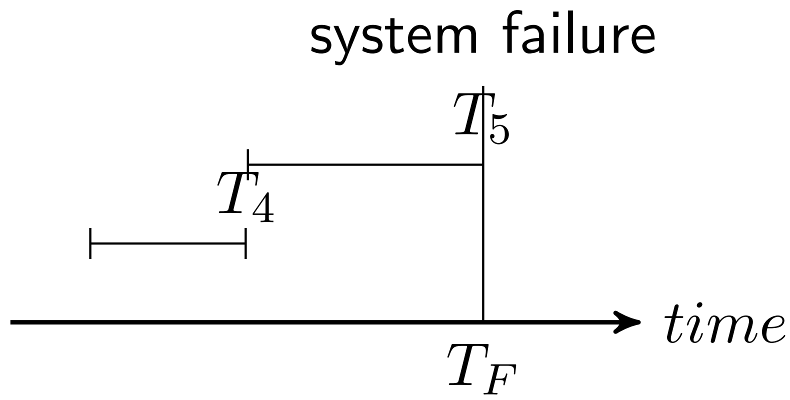
Log-based recovery

Operations to recover from failures

- **Redo**: perform the changes to the database again
- **Undo**: restore database to state prior to execution



- Redo T_1 and T_2
- Undo T_3



- Redo T_4
- Undo T_5

Recovery algorithm

To recover from a failure

- Reproduce (redo) results for committed transactions
- Undo changes of transactions that did not commit

Recovery algorithm

To recover from a failure

- Reproduce (redo) results for committed transactions
- Undo changes of transactions that did not commit

Remarks

- In a multitasking system, more than one transaction may need to be undone.
- If a system crashes during the recovery stage, the new recovery must still give correct results (**idempotence**).

Log-based recovery

database

A	100
B	300
C	5
D	60
E	80

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

How would you use the log (systematically) to recover from the crash?

The phases of recovery

1. Redo (repeat history)
    - Forward scan through the log
    - Repeat **all** updates in the same order as in the log file
    - Determine “undo” transactions
        - $[T_i$ start$]$ add T_i to the “undo list”
        - $[T_i$ abort$]$ or $[T_i$ commit$]$ remove T_i from the “undo list”

The phases of recovery

1. Redo (repeat history)
   - Forward scan through the log
   - Repeat **all** updates in the same order as in the log file
   - Determine "undo" transactions
     - [T_i start] add T_i to the "undo list"
     - [T_i abort] or [T_i commit] remove T_i from the "undo list"
2. Undo (rollback) all transactions in the "undo list"
   - Backward scan through the log
   - Undo all updates of transactions in the "undo list" – create a compensating log record
   - For a [T_i start] record of a transaction T_i in the "undo list", add a [T_i abort] record to the log file, remove T_i from the "undo list"
   - Stop, when "undo list" is empty

Compensation log records

[TID, DID, value]

- Created to undo (compensate) the changes of [TID, DID, value, newValue]
- Redo-only log record
- Can also be used to rollback a transaction during normal operation

Example

Phase 1 (redo)

database

A	100
B	300
C	5
D	60
E	80

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	100
B	300
C	5
D	60
E	80

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	100
B	300
C	5
D	60
E	80

undo list

{ T1 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	100
B	300
C	5
D	60
E	80

undo list
{ T1 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	100
B	400
C	5
D	60
E	80

undo list
{ T1 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	100
B	400
C	5
D	60
E	80

undo list
{ T1 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	100
B	400
C	10
D	60
E	80

undo list
{ T1 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	100
B	400
C	10
D	60
E	80

undo list

{ T1 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	100
B	400
C	10
D	60
E	80

undo list
{ T1, T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	100
B	400
C	10
D	60
E	80

undo list
{ T1, T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	100
B	400
C	10
D	60
E	480

undo list
{ T1, T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	100
B	400
C	10
D	60
E	480

undo list
{ T1, T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	560
B	400
C	10
D	60
E	480

undo list
{ T1, T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	560
B	400
C	10
D	60
E	480

undo list
{ T1, T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	560
B	400
C	10
D	60
E	480

undo list
{ T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	560
B	400
C	10
D	60
E	480

undo list
{ T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	570
B	400
C	10
D	60
E	480

undo list
{ T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	570
B	400
C	10
D	60
E	480

undo list
{ T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 1 (redo)

database

A	570
B	400
C	10
D	530
E	480

undo list
{ T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 2 (undo)

database

A	570
B	400
C	10
D	530
E	480

undo list
{ T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 2 (undo)

database

A	570
B	400
C	10
D	530
E	480

undo list
{ T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]

Example

Phase 2 (undo)

database

A	570
B	400
C	10
D	60
E	480

undo list

{ T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]
[T2, D, 60]

Example

Phase 2 (undo)

database

A	570
B	400
C	10
D	60
E	480

undo list

{ T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]
[T2, D, 60]

Example

Phase 2 (undo)

database

A	560
B	400
C	10
D	60
E	480

undo list
{ T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]
[T2, D, 60]
[T2, A, 560]

Example

Phase 2 (undo)

database

A	560
B	400
C	10
D	60
E	480

undo list
{ T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]
[T2, D, 60]
[T2, A, 560]

Example

Phase 2 (undo)

database

A	560
B	400
C	10
D	60
E	480

undo list

{ T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]
[T2, D, 60]
[T2, A, 560]

Example

Phase 2 (undo)

database

A	560
B	400
C	10
D	60
E	480

undo list

{ T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]
[T2, D, 60]
[T2, A, 560]

Example

Phase 2 (undo)

database

A	560
B	400
C	10
D	60
E	80

undo list

{ T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]
[T2, D, 60]
[T2, A, 560]
[T2, E, 80]

Example

Phase 2 (undo)

database

A	560
B	400
C	10
D	60
E	80

undo list

{ T2 }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]
[T2, D, 60]
[T2, A, 560]
[T2, E, 80]

Example

Phase 2 (undo)

database

A	560
B	400
C	10
D	60
E	80

undo list

{ }

log records

[T1 start]
[T1, B, 300, 400]
[T1, C, 5, 10]
[T2 start]
[T2, E, 80, 480]
[T1, A, 100, 560]
[T1 commit]
[T2, A, 560, 570]
[T2, D, 60, 530]
[T2, D, 60]
[T2, A, 560]
[T2, E, 80]
[T2 abort]

Summary: recovery

- Goal: ensuring atomicity and durability despite failures and crashes
- Durability is relative
- WAL rule
- Log-based recovery
  - All changes need to be written into the log file
  - A transaction commits when the commit entry in the log is written