Transactions: Overview

CSE 4/562: Database Systems | Lecture 18

DB. Sys.: T.C.B.: Ch. 18.1-18.2, 19.1

```
INSERT INTO Trees (location, species)
  VALUES ('123 A Street', 'pin oak');

DELETE FROM Trees WHERE date > now();

UPDATE Trees SET species = 'pin oak'
  WHERE species = 'pinoak';
```

What is a “Correct” update?

What is a “Correct” update?

- My command(s) get(s) executed fully, or not at all

What is a “Correct” update?

- My command(s) get(s) executed fully, or not at all
- The final state of the data is sane

What is a “Correct” update?

- My command(s) get(s) executed fully, or not at all
- The final state of the data is sane
- I shouldn’t have to worry about other commands executing at the same time

What is a “Correct” update?

- My command(s) get(s) executed fully, or not at all
- The final state of the data is sane
- I shouldn’t have to worry about other commands executing at the same time
- If things explode, my data is safe

My command(s) get(s) executed fully, or not at all

Trees	#	...	species
	1	...	pinoak
	2	...	pinoak
	3	...	pinoak
	4	...	pinoak

My command(s) get(s) executed fully, or not at all

Trees	#	...	species
	1	...	pinoak
	2	...	pinoak
	3	...	pinoak
	4	...	pinoak
		...	

Atomicity

```
UPDATE Trees SET species = 'pin oak'
  WHERE species = 'pinoak';
```

Atomicity


```
UPDATE Trees SET species = 'pin oak'
  WHERE species = 'pinoak';
```

- Segmentation Fault: Core Dumped

Atomicity


```
UPDATE Trees SET species = 'pin oak'
  WHERE species = 'pinoak';
```

- Segmentation Fault: Core Dumped
- 

Atomicity


```sql
UPDATE Trees SET species = 'pin oak'
  WHERE species = 'pinoak';
```

- Segmentation Fault: Core Dumped
- 
- Ctrl-C

Atomicity

Trees	#	...	species
	1	...	pin oak
	2	...	pin oak
	3	...	pinoak
	4	...	pin oak
		...	

Atomicity

Trees	#	...	species
	1	...	pin oak
	2	...	pin oak
	3	...	pinoak
	4	...	pin oak
		...	

Atomicity

Trees	#	...	species
	1	...	pin oak
	2	...	pin oak
	3	...	pinoak
	4	...	pin oak
		...	

My command(s) get(s) executed fully, or not at all

The final state of the data is “sane”

```
Bob.Balance -= 20
Alice.Balance += 20
```

The final state of the data is “sane”

```
Bob.Balance -= 20
Alice.Balance += 20
```

It is an error for a user’s balance to be negative.

Constraints

Primary Key / Unique

- Every record has a unique value for the primary key column(s)

Foreign Key

- An attribute references a key from another table that must exist.

Domain

- Restrictions on allowable values like `Balance >= 0` or `NOT NULL`

Check

- An arbitrary existential query that must return true

Bob pays Alice

```
Bob.Balance -= 20
Alice.Balance += 20
```

Alice pays Carol

```
Alice.Balance -= 10
Carol.Balance += 10
```

Bob pays Alice

```
Bob.Balance -= 20
Alice.Balance += 20
```

Alice pays Carol

```
Alice.Balance -= 10
Carol.Balance += 10
```

Ok Outcomes

Everything Succeeds

```
Alice.Balance += 10
Bob.Balance -= 20
Carol.Balance += 10
```

Bob pays Alice

```
Bob.Balance -= 20
Alice.Balance += 20
```

Alice pays Carol

```
Alice.Balance -= 10
Carol.Balance += 10
```

Ok Outcomes

Everything Succeeds

```
Alice.Balance += 10
Bob.Balance -= 20
Carol.Balance += 10
```

Operation 2 fails

```
Alice.Balance += 20
Bob.Balance -= 20
```

Everything Fails

Bob pays Alice

```
Bob.Balance -= 20
Alice.Balance += 20
```

Alice pays Carol

```
Alice.Balance -= 10
Carol.Balance += 10
```

Not Ok Outcomes

```
Bob.Balance -= 20
Alice.Balance += 20
Carol.Balance += 10
```

Ok Outcomes

Everything Succeeds

```
Alice.Balance += 10
Bob.Balance -= 20
Carol.Balance += 10
```

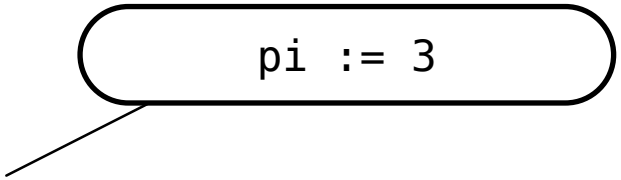
Operation 2 fails

```
Alice.Balance += 20
Bob.Balance -= 20
```

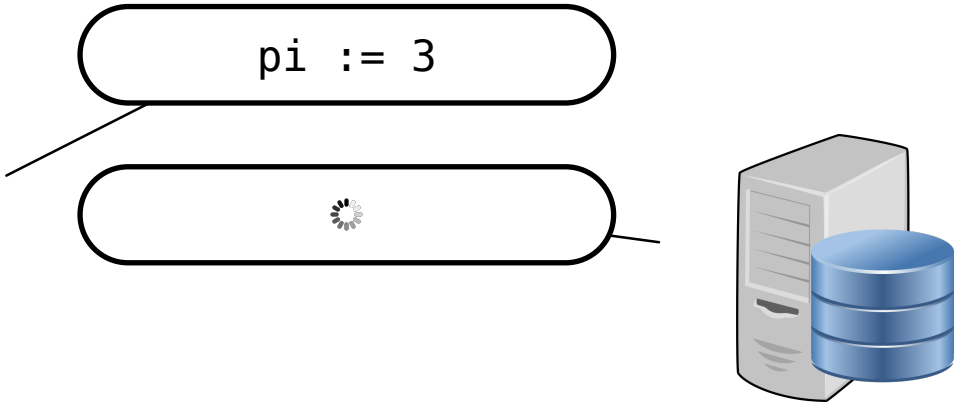
Everything Fails

If things explode, my data is safe

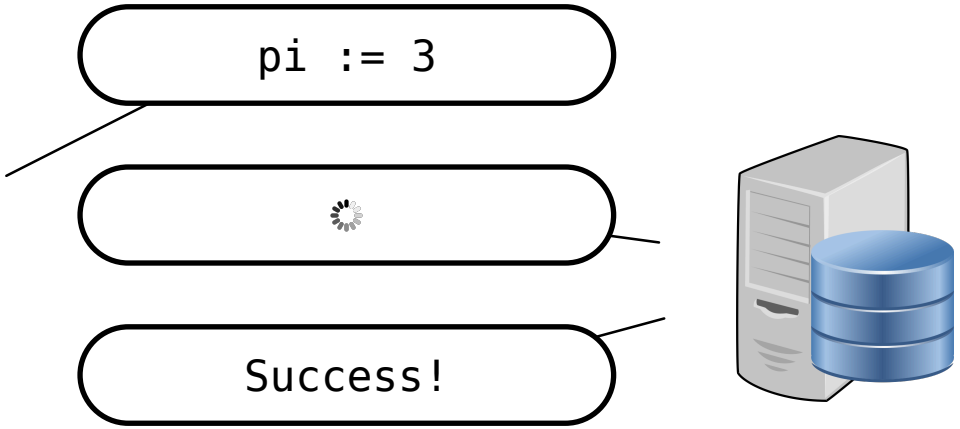
If things explode, my data is safe



If things explode, my data is safe



If things explode, my data is safe



If things explode, my data is safe

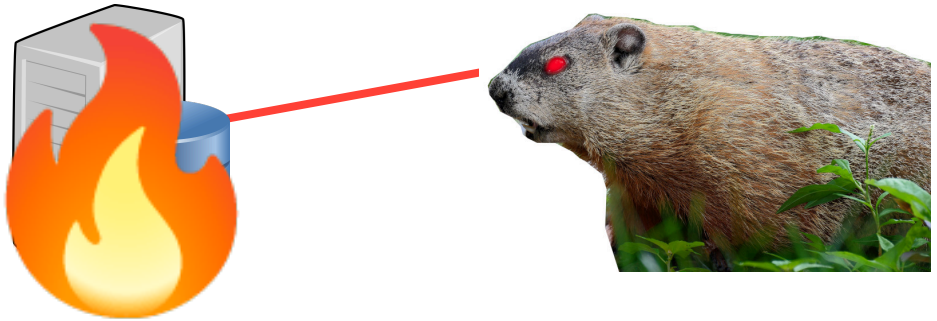


Image credit: Adapted from 'Marmota, Montreal, 2024', by Pierre5018; CC BY 4.0

If things explode, my data is safe

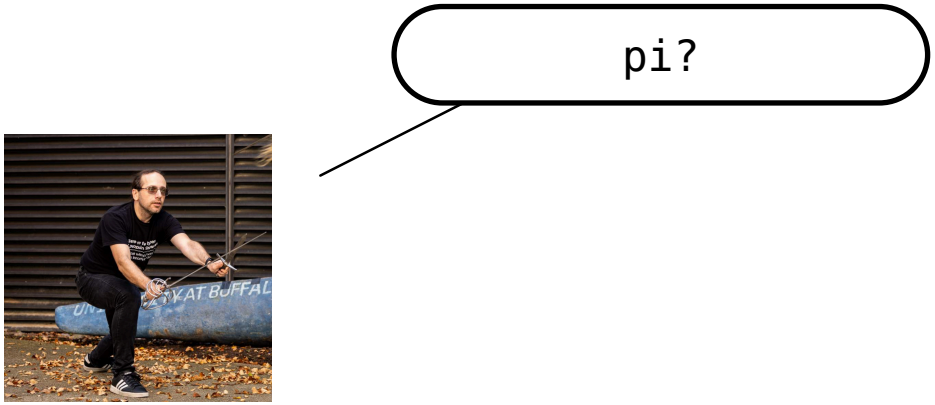


Image credit: Adapted from 'Marmota, Montreal, 2024', by Pierre5018; CC BY 4.0

If things explode, my data is safe

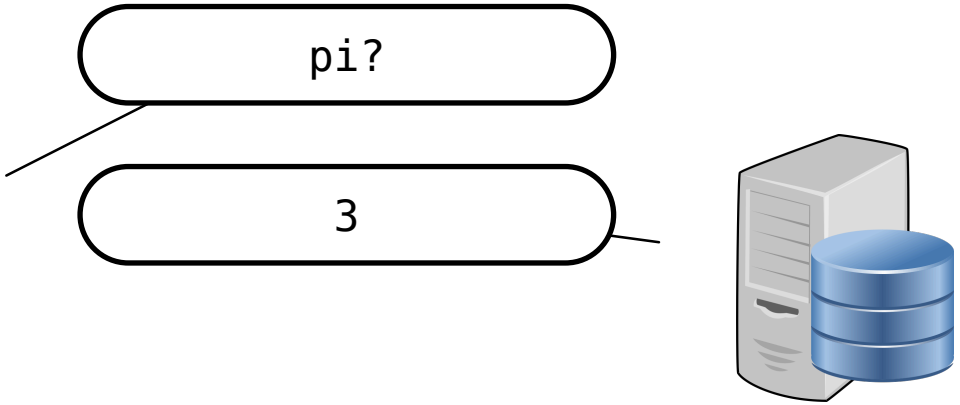


Image credit: Adapted from 'Marmota, Montreal, 2024', by Pierre5018; CC BY 4.0

The ACID Properties

- My command(s) get(s) executed fully, or not at all
- The final state of the data is sane
- I shouldn’t have to worry about other commands executing at the same time
- If things explode, my data is safe

Atomicity

- My command(s) get(s) executed fully, or not at all
- The final state of the data is sane
- I shouldn’t have to worry about other commands executing at the same time
- If things explode, my data is safe

Atomicity

- My command(s) get(s) executed fully, or not at all

Consistency

- The final state of the data is sane
- I shouldn’t have to worry about other commands executing at the same time
- If things explode, my data is safe

Atomicity

- My command(s) get(s) executed fully, or not at all

Consistency

- The final state of the data is sane

Isolation

- I shouldn't have to worry about other commands executing at the same time

- If things explode, my data is safe

Atomicity

- My command(s) get(s) executed fully, or not at all

Consistency

- The final state of the data is sane

Isolation

- I shouldn’t have to worry about other commands executing at the same time

Durability

- If things explode, my data is safe

The ACID Properties

Atomicity

- My command(s) get(s) executed fully, or not at all

Consistency

- The final state of the data is sane

Isolation

- I shouldn't have to worry about other commands executing at the same time

Durability

- If things explode, my data is safe

These are known as the A.C.I.D. Properties

Each use case may require different properties

Relational DBMSes (Postgres, DB2, SQLServer, Oracle, etc...)

- Full ACID, but configurable

Caches (e.g., Redis, Memcached)

- No need for durability

Key-Value Stores (e.g., Riak, Aerospike, LevelDB/Rocks)

- Only trivial Atomicity required

ACID for more than one operation

```
START TRANSACTION;
UPDATE Ledger SET Balance = Balance - 20
  WHERE name = 'Bob';
UPDATE Ledger SET Balance = Balance + 20
  WHERE name = 'Alice';
COMMIT;
```

Transactions

A batch of operations that should execute atomically

START TRANSACTION

- Start the batch

COMMIT

- Conclude the transaction, apply final checks, and apply changes if successful
- If this operation succeeds, the data is durable

ABORT/ROLLBACK

- Undo all changes applied as part of the transaction

Atomicity

Consistency

Isolation

Durability

Atomicity

A transaction is either applied fully (committed) or not applied at all (aborted).

Consistency

Isolation

Durability

Atomicity
A transaction is either applied fully (committed) or not applied at all (aborted).

Consistency
A transaction is only allowed to commit if its final state satisfies the database's constraints.

Isolation

Durability

Atomicity
A transaction is either applied fully (committed) or not applied at all (aborted).

Consistency
A transaction is only allowed to commit if its final state satisfies the database's constraints.

Isolation
Transactions are executed "as if" only one were running at a time.

Durability

Atomicity
A transaction is either applied fully (committed) or not applied at all (aborted).

Consistency
A transaction is only allowed to commit if its final state satisfies the database's constraints.

Isolation
Transactions are executed "as if" only one were running at a time.

Durability
Once a transaction returns successfully from a commit, the data is safe from marmots.

Atomicity

A transaction is either applied fully (committed) or not applied at all (aborted).

Consistency

A transaction is only allowed to commit if its final state satisfies the database's constraints.

Isolation

Transactions are executed "as if" only one were running at a time.

Durability

Once a transaction returns successfully from a commit, the data is safe from marmots.

Enforcing Isolation

A little abstraction

I'm going to use the word "object" to mean some database "thing".

A little abstraction

I'm going to use the word "object" to mean some database "thing".

- ... a table
- ... a record
- ... a collection of records (e.g., a page)
- ... a column of data
- ... a specific field in a specific record

A little abstraction

I'm going to use the word "object" to mean some database "thing".

- ... a table
- ... a record
- ... a collection of records (e.g., a page)
- ... a column of data
- ... a specific field in a specific record

What level of granularity we're talking about affects performance, but the techniques we're talking about work at all levels.

A little abstraction

I'm going to use the word "object" to mean some database "thing".

- ... a table
- ... a record
- ... a collection of records (e.g., a page)
- ... a column of data
- ... a specific field in a specific record

What level of granularity we're talking about affects performance, but the techniques we're talking about work at all levels.

Independence

Observation: Two transactions interacting with different objects can't possibly interfere with each other.

Transaction (the 50k view)

- Reads from a set of objects (the “read set”)
- Writes to a set of objects (the “write set”)

Transaction (the 50k view)

- Reads from a set of objects (the “read set”)
- Writes to a set of objects (the “write set”)
- Either a commit or an abort

Transaction (the 50k view)

- Reads from a set of objects (the “read set”)
- Writes to a set of objects (the “write set”)
- Either a commit or an abort

What does it mean for a transaction to be isolated?

1. Bob pays Alice

```
Bob.Balance -= 20
Alice.Balance += 20
```

2. Alice pays Carol

```
Alice.Balance -= 10
Carol.Balance += 10
```

1. Bob pays Alice

```
Bob.Balance -= 20
Alice.Balance += 20
```

2. Alice pays Carol

```
Alice.Balance -= 10
Carol.Balance += 10
```

```
# Transaction 1
Bob.Balance -= 20
Alice.Balance += 20

# Transaction 2
Alice.Balance -= 10
Carol.Balance += 10
```

1. Bob pays Alice

```
Bob.Balance -= 20
Alice.Balance += 20
```

2. Alice pays Carol

```
Alice.Balance -= 10
Carol.Balance += 10
```

```
# Transaction 1
Bob.Balance -= 20
Alice.Balance += 20

# Transaction 2
Alice.Balance -= 10
Carol.Balance += 10
```

1. Bob pays Alice

```
Bob.Balance -= 20
Alice.Balance += 20
```

2. Alice pays Carol

```
Alice.Balance -= 10
Carol.Balance += 10
```

```
# Transaction 2
Alice.Balance -= 10
Carol.Balance += 10

# Transaction 1
Bob.Balance -= 20
Alice.Balance += 20
```

1. Bob pays Alice

```
Bob.Balance -= 20
Alice.Balance += 20
```

2. Alice pays Carol

```
Alice.Balance -= 10
Carol.Balance += 10
```

```
# Transaction 2
Alice.Balance -= 10
Carol.Balance += 10

# Transaction 1
Bob.Balance -= 20
Alice.Balance += 20
```

1. Bob pays Alice

```
Bob.Balance -= 20
Alice.Balance += 20
```

2. Alice pays Carol

```
Alice.Balance -= 10
Carol.Balance += 10
```

```
#Transaction 2
Alice.Balance -= 10
#Transaction 1
Bob.Balance -= 20
#Transaction 2
Carol.Balance += 10
#Transaction 1
Alice.Balance += 20
```

1. Bob pays Alice

```
Bob.Balance -= 20
Alice.Balance += 20
```

2. Alice pays Carol

```
Alice.Balance -= 10
Carol.Balance += 10
```

```
#Transaction 2
Alice.Balance -= 10
#Transaction 1
Bob.Balance -= 20
#Transaction 2
Carol.Balance += 10
#Transaction 1
Alice.Balance += 20
```

???

1. Bob pays Alice

```
Bob.Balance -= 20
Alice.Balance += 20
```

2. Alice pays Carol

```
Alice.Balance -= 10
Carol.Balance += 10
```

```
#Transaction 1
Bob.Balance -= 20
#Transaction 2
Alice.Balance -= 10
#Transaction 1
Alice.Balance += 20
#Transaction 2
Carol.Balance += 10
```

???

Idea: Run transactions serially!

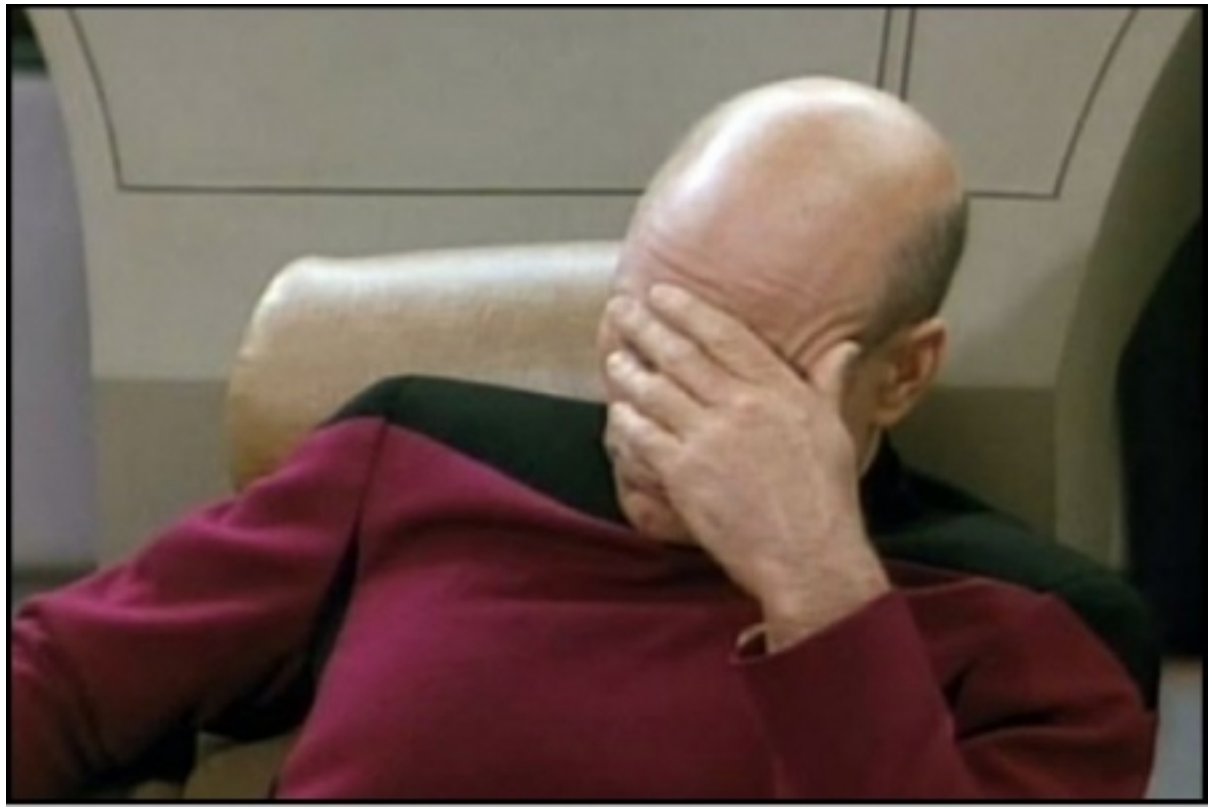


Image credit: Paramount Pictures: Star Trek: The Next Generation

Less Bad Idea: Create the illusion that each transaction is running serially.

Transactions

Transaction T1

T1-R(A), T1-W(A), T1-R(B), T1-W(B)

Transaction T2

T2-R(A), T2-W(A), T2-R(B), T2-W(B)

1. Do some compute
2. Read A
3. Do some compute
4. Write A

...

Schedules

A schedule is a sequence of operations.

Schedules

A schedule is a sequence of operations. For a schedule to be valid

- Every read and write in every transaction must be present.
- Every read and write in every transaction must appear in the same order.

Schedules

A schedule is a sequence of operations. For a schedule to be valid

- Every read and write in every transaction must be present.
- Every read and write in every transaction must appear in the same order.

... but a *valid* schedule can interleave operations however we like.

Transaction T1

T1-R(A), T1-W(A), T1-R(B), T1-W(B)

Transaction T2

T2-R(A), T2-W(A), T2-R(B), T2-W(B)

—

Transaction T1

T1-R(A), T1-W(A), T1-R(B), T1-W(B)

Transaction T2

T2-R(A), T2-W(A), T2-R(B), T2-W(B)

—

These are all valid schedules:

- T1-R(A), T1-W(A), T1-R(B), T1-W(B), T2-R(A), T2-W(A), T2-R(B), T2-W(B)

Transaction T1

T1-R(A), T1-W(A), T1-R(B), T1-W(B)

Transaction T2

T2-R(A), T2-W(A), T2-R(B), T2-W(B)

—

These are all valid schedules:

- T1-R(A), T1-W(A), T1-R(B), T1-W(B), T2-R(A), T2-W(A), T2-R(B), T2-W(B)
- T2-R(A), T2-W(A), T2-R(B), T2-W(B), T1-R(A), T1-W(A), T1-R(B), T1-W(B)

Transaction T1

T1-R(A), T1-W(A), T1-R(B), T1-W(B)

Transaction T2

T2-R(A), T2-W(A), T2-R(B), T2-W(B)

—

These are all valid schedules:

- T1-R(A), T1-W(A), T1-R(B), T1-W(B), T2-R(A), T2-W(A), T2-R(B), T2-W(B)
- T2-R(A), T2-W(A), T2-R(B), T2-W(B), T1-R(A), T1-W(A), T1-R(B), T1-W(B)
- T1-R(A), T2-R(A), T1-W(A), T2-W(A), T1-R(B), T2-R(B), T1-W(B), T2-W(B)

Transaction T1

T1-R(A), T1-W(A), T1-R(B), T1-W(B)

Transaction T2

T2-R(A), T2-W(A), T2-R(B), T2-W(B)

—

These are all valid schedules:

- T1-R(A), T1-W(A), T1-R(B), T1-W(B), T2-R(A), T2-W(A), T2-R(B), T2-W(B)
- T2-R(A), T2-W(A), T2-R(B), T2-W(B), T1-R(A), T1-W(A), T1-R(B), T1-W(B)
- T1-R(A), T2-R(A), T1-W(A), T2-W(A), T1-R(B), T2-R(B), T1-W(B), T2-W(B)

These are **not** valid schedules:

- T1-W(B), T1-R(B), T1-W(A), T1-R(A), T2-R(A), T2-W(A), T2-R(B), T2-W(B)

Transaction T1

T1-R(A), T1-W(A), T1-R(B), T1-W(B)

Transaction T2

T2-R(A), T2-W(A), T2-R(B), T2-W(B)

—

These are all valid schedules:

- T1-R(A), T1-W(A), T1-R(B), T1-W(B), T2-R(A), T2-W(A), T2-R(B), T2-W(B)
- T2-R(A), T2-W(A), T2-R(B), T2-W(B), T1-R(A), T1-W(A), T1-R(B), T1-W(B)
- T1-R(A), T2-R(A), T1-W(A), T2-W(A), T1-R(B), T2-R(B), T1-W(B), T2-W(B)

These are **not** valid schedules:

- T1-W(B), T1-R(B), T1-W(A), T1-R(A), T2-R(A), T2-W(A), T2-R(B), T2-W(B)
- T1-R(A), T1-W(A), T2-R(A), T2-W(A), T2-R(B), T2-W(B)

Time	T1	T2
↓		R(A)
↓		W(A)
↓		R(B)
↓		W(B)
↓	R(A)	
↓	W(A)	
↓	R(B)	
↓	W(B)	

Time	T1	T2
↓		R(A)
↓		W(A) A = A - 20
↓		R(B)
↓		W(B) B = B + 20
↓	R(A)	
↓	W(A) A = A + 20	
↓	R(B)	
↓	W(B) B = B - 20	

What does it mean for a schedule to be “Correct”?

Idea: Run transactions serially!

Serial Schedule

A schedule with no interleaving

Serial Schedule

A schedule with no interleaving

Serializable Schedule

A schedule guaranteed to produce the same output as a serial schedule

Question: How do we guarantee that transactions are only ever executed with serializable schedules?

Challenge: We can't know if a schedule will be serializable until the transaction is done.

Allow concurrency, but...

Allow concurrency, but...

Locking (Pessimistic)

... pause a transaction before it ever risks non-serializable behavior.

Allow concurrency, but...

Locking (Pessimistic)

... pause a transaction before it ever risks non-serializable behavior.

Snapshot Isolation (Optimistic)

... check whether a transaction had a serializable schedule prior to committing and abort it if not.

Allow concurrency, but...

Locking (Pessimistic)
... pause a transaction before it ever risks non-serializable behavior.

Snapshot Isolation (Optimistic)
... check whether a transaction had a serializable schedule prior to committing and abort it if not.

Timestamp Concurrency Control (Optimistic)
... detect serializability violations while the transaction is running and abort.

How do we convince ourselves that these concurrency strategies are guaranteed to produce serializable schedules?

Serializability requires us to reason about the actual logic of the transaction.

Serializability requires us to reason about the actual logic of the transaction.

Let's define with a weaker property

Conflict Serializability

What can we reason about?

What can we reason about?

Rule 1: Reads never affect other reads

You can safely reverse the order of two reads without changing the serializability of a schedule.

What can we reason about?

Rule 1: Reads never affect other reads

You can safely reverse the order of two reads without changing the serializability of a schedule.

—

Rule 2: Operations on different objects don’t affect each other

You can safely reverse the order of operations on different objects without changing the serializability of a schedule.

Conflict Equivalence

Two schedules are conflict equivalent if there is a sequence of pairwise “flips” of reads or operations on different objects that transforms one schedule into the other.

Example

Time	T1	T2
↓		W(B)
↓	R(B)	
↓		W(A)
↓	W(A)	

Example

Time	T1	T2
↓		W(B)
↓		W(A)
↓	R(B)	
↓	W(A)	

Example

Time	T1	T2
↓		W(B)
↓		W(A)
↓	R(B)	
↓	W(A)	

The original schedule is conflict equivalent to a serial schedule

Example

Time	T1	T2
↓		W(B)
↓	R(B)	
↓	W(A)	
↓		W(A)

Example

Time	T1	T2
↓		W(B)
↓	R(B)	
↓	W(A)	
↓		W(A)

No way to rewrite this into a conflict equivalent serial schedule

Conflict Equivalence

Two schedules are conflict equivalent if there is a sequence of pairwise “flips” of reads or operations on different objects that transforms one schedule into the other.

Conflict Serializability

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

How do we decide whether a schedule is conflict serializable?

Conflicts

write/write, read/write, and write/read operation pairs can't be flipped.

Let's give these types of operation pairs a name: **Conflicts**[1].

[1] A note on language: A conflict does **not** indicate a problem.

Detecting Conflict Serializability

Time	T1	T2
↓	W(A)	
↓		W(A)

Detecting Conflict Serializability

Time	T1	T2
↓	W(A)	
↓		W(A)

This conflict requires that T1 **happens before** T2

Detecting Conflict Serializability

Conflicts create a partial order over the conflict-equivalent serial schedules.

Detecting Conflict Serializability

Time	T1	T2
↓		W(B)
↓	R(B)	
↓		W(A)
↓	W(A)	

In this schedule...

Detecting Conflict Serializability

Time	T1	T2
↓		W(B)
↓	R(B)	
↓		W(A)
↓	W(A)	

In this schedule...

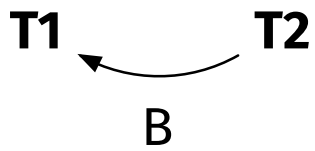
- **T2**′s write to B “happens before” **T1**′s read.

Detecting Conflict Serializability

Time	T1	T2
↓		W(B)
↓	R(B)	
↓		W(A)
↓	W(A)	

In this schedule...

- **T2**′s write to B “happens before” **T1**′s read.

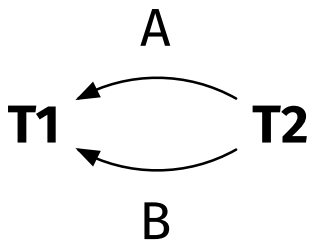


Detecting Conflict Serializability

Time	T1	T2
↓		W(B)
↓	R(B)	
↓		W(A)
↓	W(A)	

In this schedule...

- **T2**'s write to B "happens before" **T1**'s read.
- **T2**'s write to A "happens before" **T1**'s write.

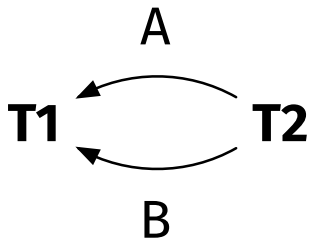


Detecting Conflict Serializability

Time	T1	T2
↓		W(B)
↓	R(B)	
↓		W(A)
↓	W(A)	

In this schedule...

- **T2**'s write to B "happens before" **T1**'s read.
- **T2**'s write to A "happens before" **T1**'s write.



No cycles in the "conflict graph"
Schedule is Conflict Serializable

Detecting Conflict Serializability

Time	T1	T2
↓		W(B)
↓	R(B)	
↓	W(A)	
↓		W(A)

In this schedule...

Detecting Conflict Serializability

Time	T1	T2
↓		W(B)
↓	R(B)	
↓	W(A)	
↓		W(A)

In this schedule...

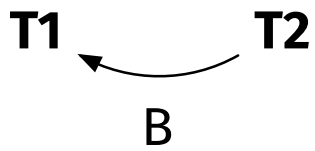
- **T2**′s write to B “happens before” **T1**′s read.

Detecting Conflict Serializability

Time	T1	T2
↓		W(B)
↓	R(B)	
↓	W(A)	
↓		W(A)

In this schedule...

- **T2**′s write to B “happens before” **T1**′s read.

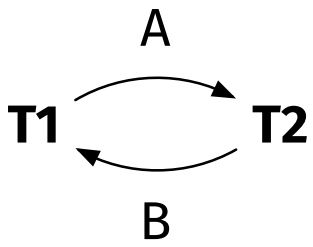


Detecting Conflict Serializability

Time	T1	T2
↓		W(B)
↓	R(B)	
↓	W(A)	
↓		W(A)

In this schedule...

- **T2**′s write to B “happens before” **T1**′s read.
- **T1**′s write to A “happens before” **T2**′s write.

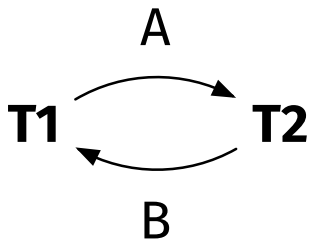


Detecting Conflict Serializability

Time	T1	T2
↓		W(B)
↓	R(B)	
↓	W(A)	
↓		W(A)

In this schedule...

- **T2**'s write to B "happens before" **T1**'s read.
- **T1**'s write to A "happens before" **T2**'s write.



Cycle = Schedule is **not** Conflict Serializable

Side note...

If the schedule is conflict serializable, it is serializable.

If the schedule is not conflict serializable, it may still be serializable...
... but we don't have the tools to prove it

Side note 2...

You will be expected to know whether a schedule is (conflict) serializable...

Side note 2...

You will be expected to know whether a schedule is (conflict) serializable...

... but this is usually not the point.

Side note 2...

You will be expected to know whether a schedule is (conflict) serializable...

... but this is usually not the point.

By learning to work with serializability, we can prove to ourselves that a new concurrency control scheme can **guarantee** that it will only ever permit transactions to execute according to a (conflict) serializable schedule.