

# SQL – Transactions

---

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom slides

# Agenda

---

Introduction

Properties

Isolation levels

# Motivation for transactions

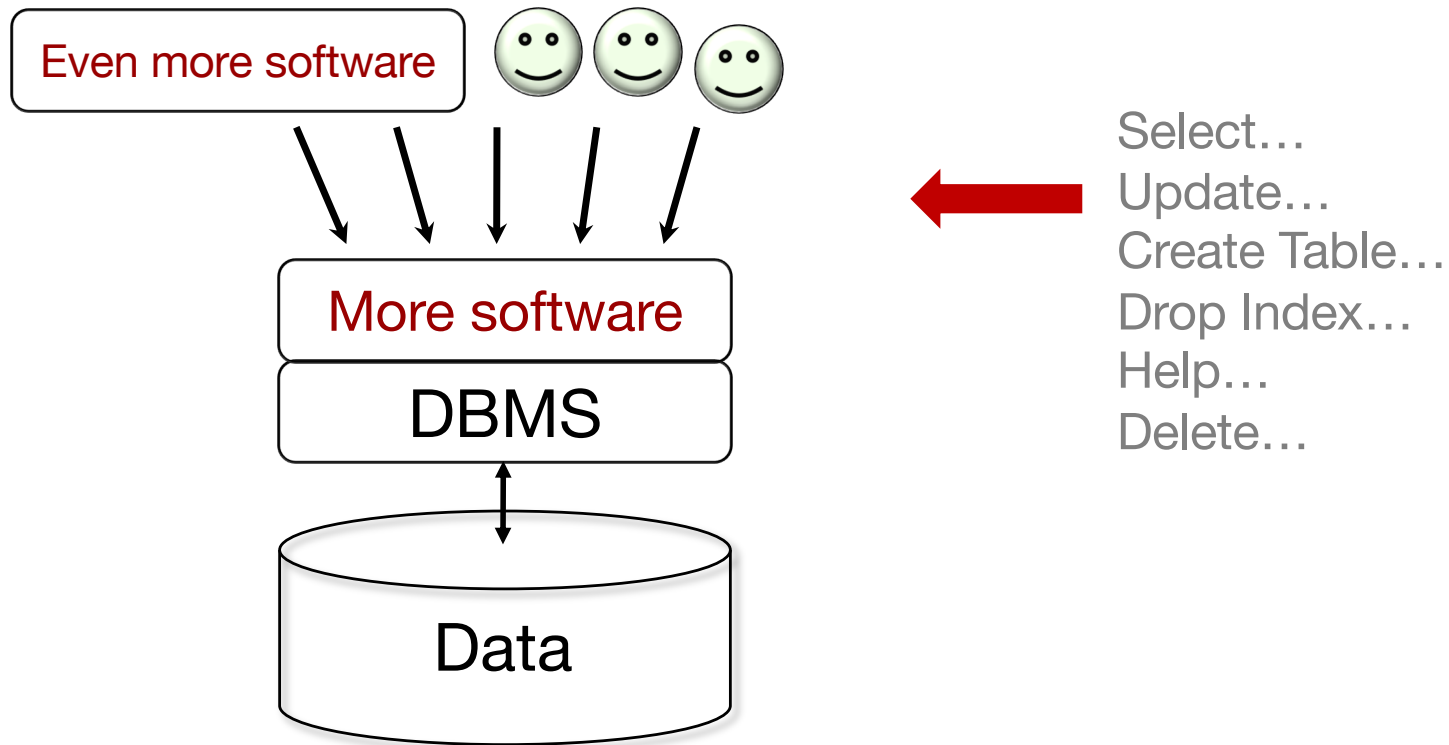
---

Concurrent database access

Resilience to system failures

# Concurrent Database Access

---



# Attribute-level Inconsistency

---

**Update** College **Set** enr = enr + 1000 **Where** cName = 'Stanford'

concurrent with ...

**Update** College **Set** enr = enr + 1500 **Where** cName = 'Stanford'

	15000	

get; modify; put

$$15\ 000 + 2\ 500 = 17\ 500$$

$$15\ 000 + 1\ 000 = 16\ 000$$

$$15\ 000 + 1\ 500 = 16\ 500$$


# Tuple-level Inconsistency

---

Update Apply Set major = 'CS' Where sID = 123

concurrent with ...

Update Apply Set dec = 'Y' Where sID = 123



sID	major	dec
123		

get; modify; put

both changes

One of the two changes

# Table-level Inconsistency

---

Update Apply Set decision = 'Y'

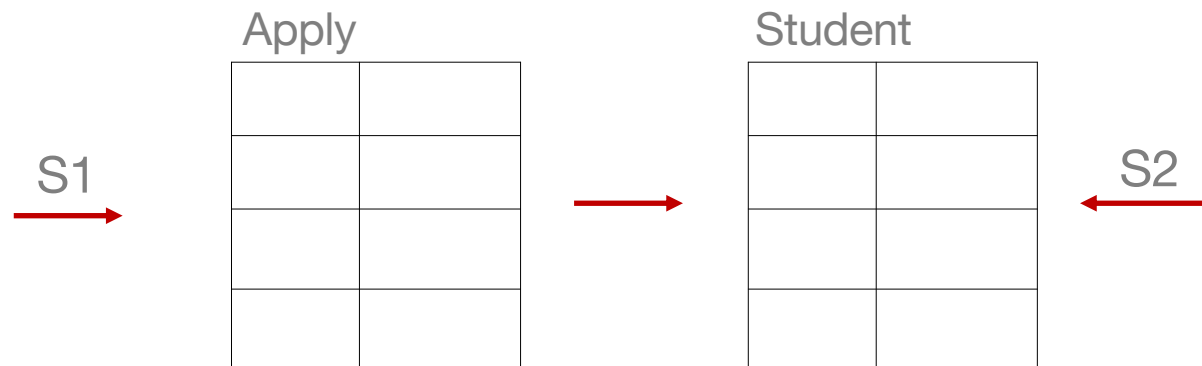
Where sID In (Select sID From Student Where GPA > 3.9)

} S1

concurrent with ...

Update Student Set GPA = (1.1) \* GPA Where sizeHS > 2500

} S2



# Multi-statement Inconsistency

---

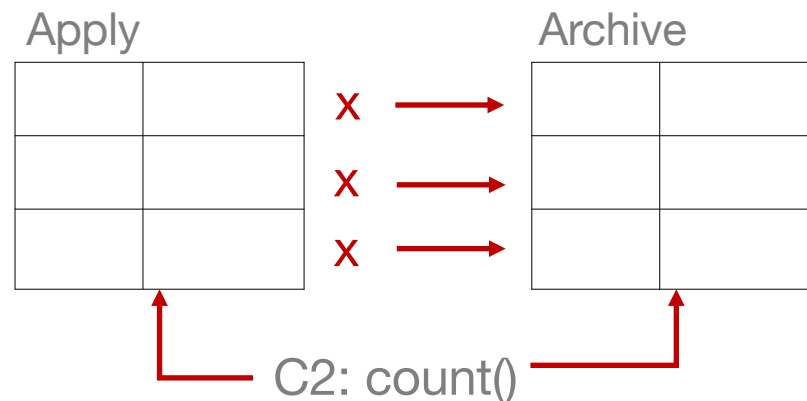
Insert Into Archive Select \* From Apply Where decision = 'N';  
Delete From Apply Where decision = 'N';

C1

concurrent with ...

Select Count(\*) From Apply;  
Select Count(\*) From Archive;

C2





# Concurrency goal

---

Execute sequence of SQL statements so they appear to be running in isolation

Simple solution: execute them in isolation

But want to enable concurrency whenever safe to do so

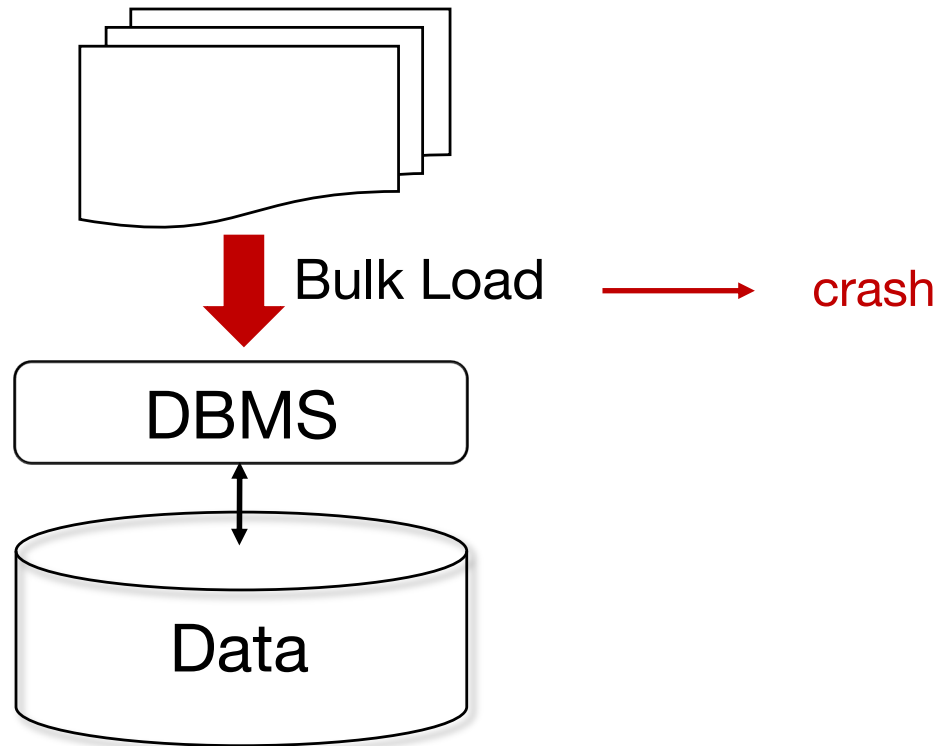
Multiprocessor system

Multithreaded system

Asynchronous I/O

# Resilience to System Failures

---



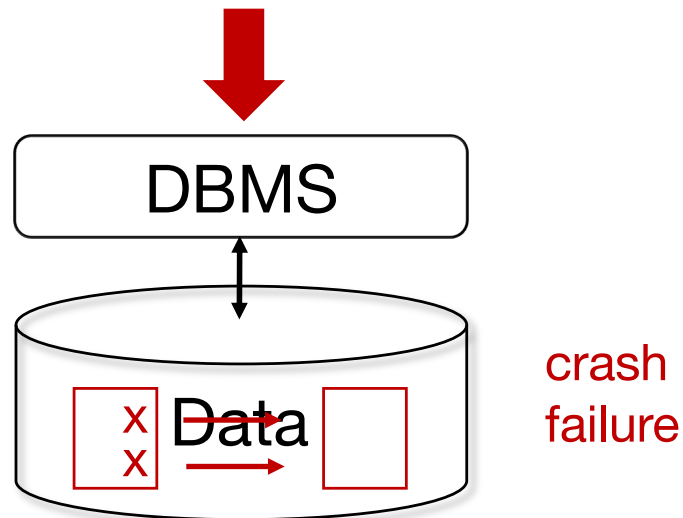
# Resilience to System Failures

---

Insert Into Archive

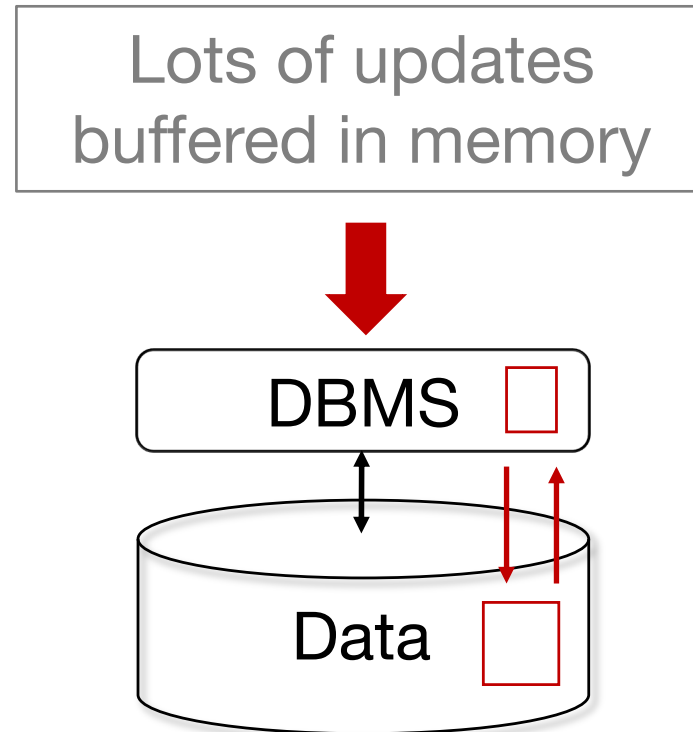
Select \* From Apply Where decision = 'N';

Delete From Apply Where decision = 'N';



# Resilience to System Failures

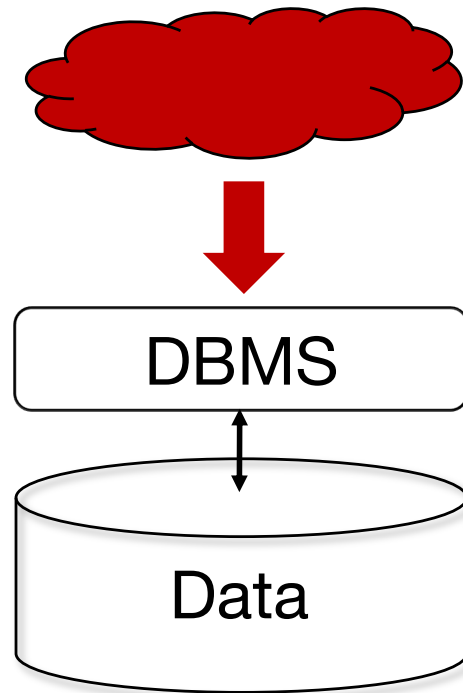
---



# System-Failure Goal

---

Guarantee all-or-nothing execution, regardless of failures



# Transactions

---

Solution for both concurrency and failures

A transaction is a sequence of one or more SQL operations treated as a unit

Transactions appear to run in isolation

If the system fails, each transaction's changes are reflected either entirely or not at all

# Transactions: SQL standard

---

Transaction begins automatically on first SQL statement

On “**commit**” transaction ends and new one begins

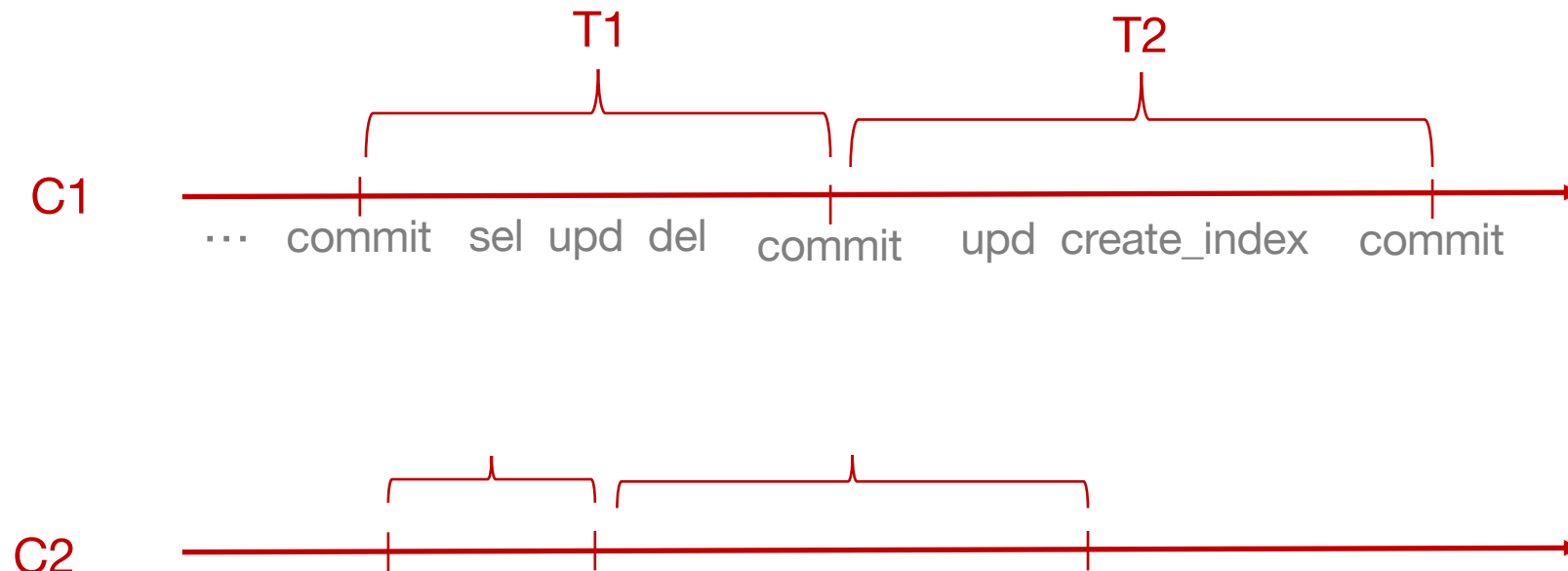
Current transaction ends on session termination

“**Autocommit**” turns each statement into transaction

# Transactions

---

A transaction is a sequence of one or more SQL operations treated as a unit





# Agenda

---

Introduction

Properties

Isolation levels

# ACID Properties

---

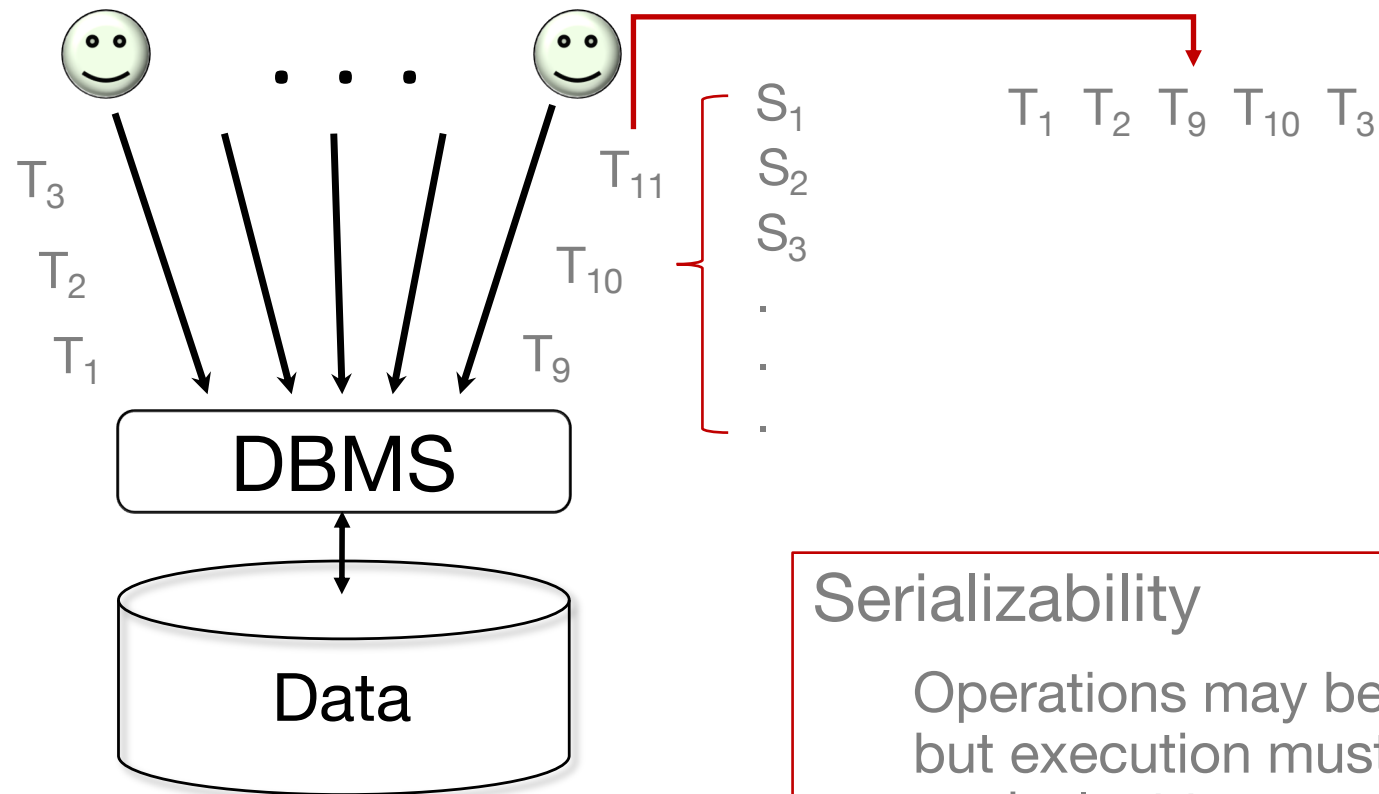
**A**tomicity                      3

**C**onsistency                      4

**I**solation                      1

**D**urability                      2

# ACID Properties: Isolation



## Serializability

Operations may be interleaved, but execution must be equivalent to some sequential (serial) order of all transactions

# Attribute-level Inconsistency

---

Update College Set enr = enr + 1000 Where cName = 'Stanford'  $T_1$

concurrent with ...

Update College Set enr = enr + 1500 Where cName = 'Stanford'  $T_2$

If serializability is guaranteed

$T_1; T_2$   
 $T_2; T_1$   $\longrightarrow$  15 000  $\rightarrow$  17 500

# Tuple-level Inconsistency

---

Update Apply Set major = 'CS' Where sID = 123

$T_1$

concurrent with ...

Update Apply Set dec = 'Y' Where sID = 123

$T_2$

If serializability is guaranteed

$T_1; T_2$   
 $T_2; T_1$   Both changes

# Table-level Inconsistency

---

Update Apply Set decision = 'Y'

Where sID In (Select sID From Student Where GPA > 3.9)

} T<sub>1</sub>

concurrent with ...

Update Student Set GPA = (1.1) \* GPA Where sizeHS > 2500

} T<sub>2</sub>

If serializability is guaranteed

T<sub>1</sub>; T<sub>2</sub>  
T<sub>2</sub>; T<sub>1</sub>



Order  
matters



DBMS don't guarantee the exact sequential order if the transactions are being issued at the same time

# Multi-statement Inconsistency

---

Insert Into Archive Select \* From Apply Where decision = 'N';  
Delete From Apply Where decision = 'N';

$T_1$

concurrent with ...

Select Count(\*) From Apply;  
Select Count(\*) From Archive;

$T_2$

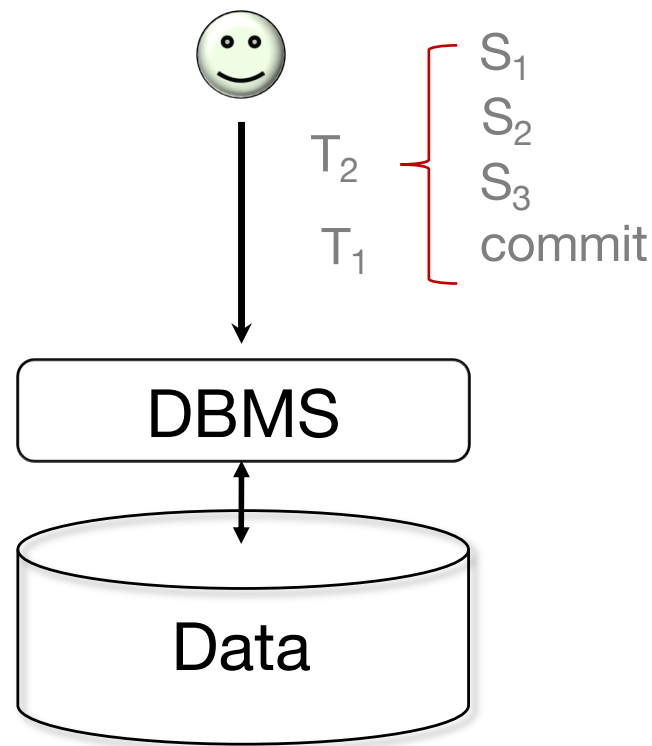
If serializability is guaranteed

$T_1; T_2$   
 $T_2; T_1$        $\longrightarrow$       Order matters

# ACID Properties: Durability

---

If system crashes after transaction commits, all effects of transaction remain in database



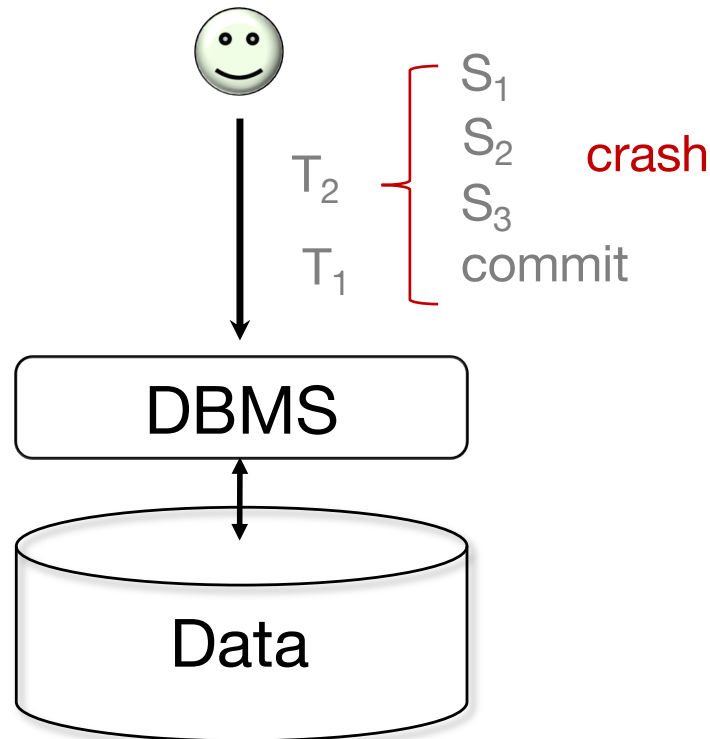


# ACID Properties: Atomicity

---

Each transaction is “all-or-nothing”, never left half done

Using a logging mechanism, partial effects of transactions at the time of crash are undone



# Transaction Rollback (= Abort)

---

Undoes partial effects of transaction

Can be system- or client-initiated

```
Begin Transaction;  
<get input from user>  
SQL commands based on input  
<confirm results with user>  
If ans='ok' Then Commit; Else Rollback;
```

Transactions should  
be constructed to run  
quickly



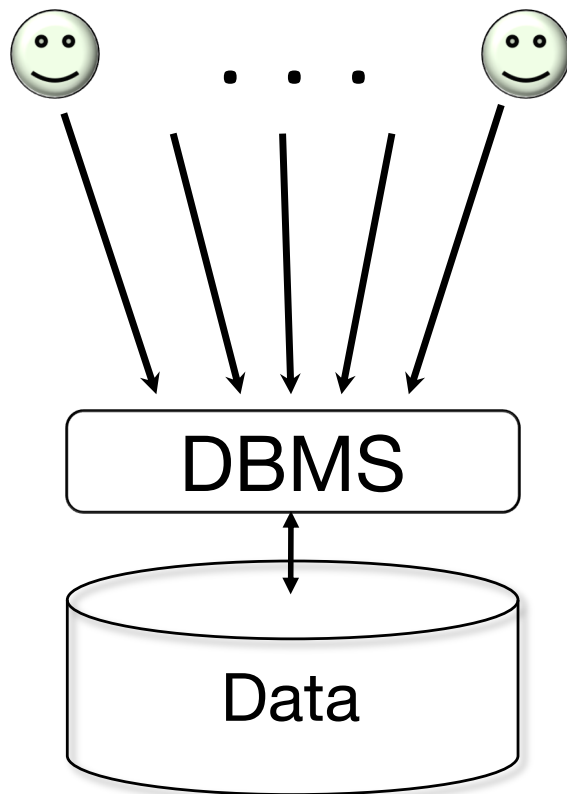
Not wait arbitrary  
amounts of time

Locking

Only undoes effects on the data itself

# ACID Properties: Consistency

---



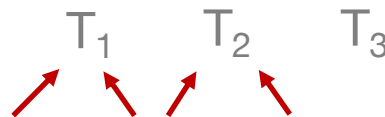
Each client, each transaction:

Can assume all constraints hold when transaction begins

Must guarantee all constraints hold when transaction ends

Serializability

Constraints always hold



# Agenda

---

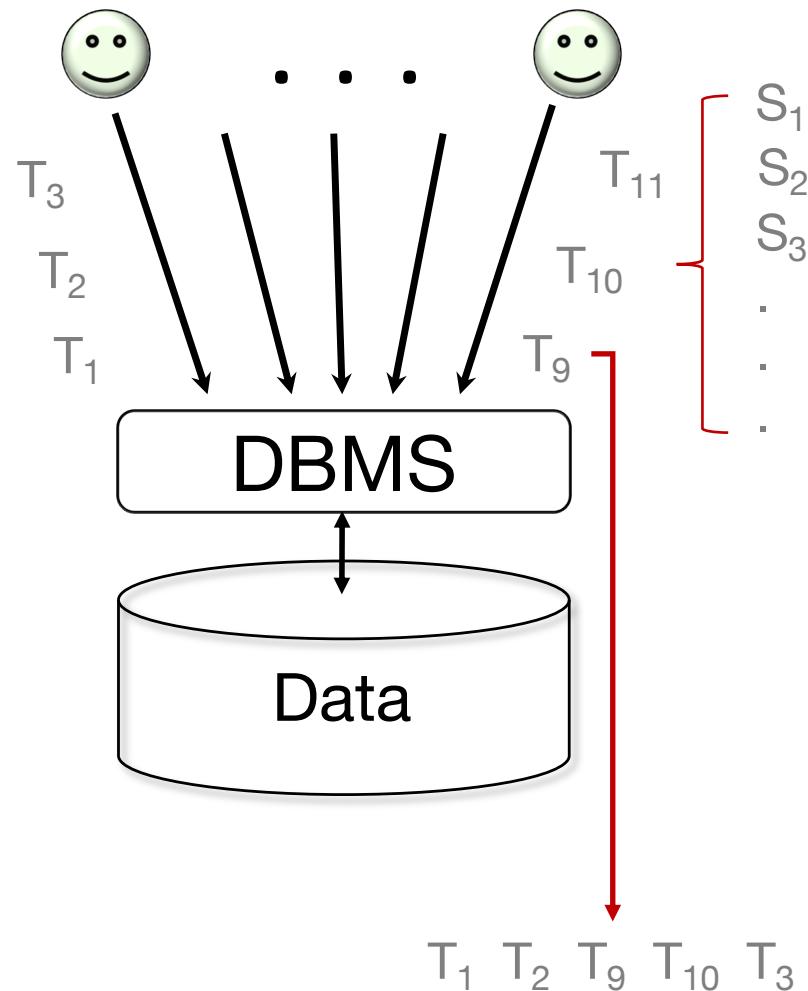
Introduction

Properties

Isolation levels

# ACID Properties: Isolation

---



## Serializability

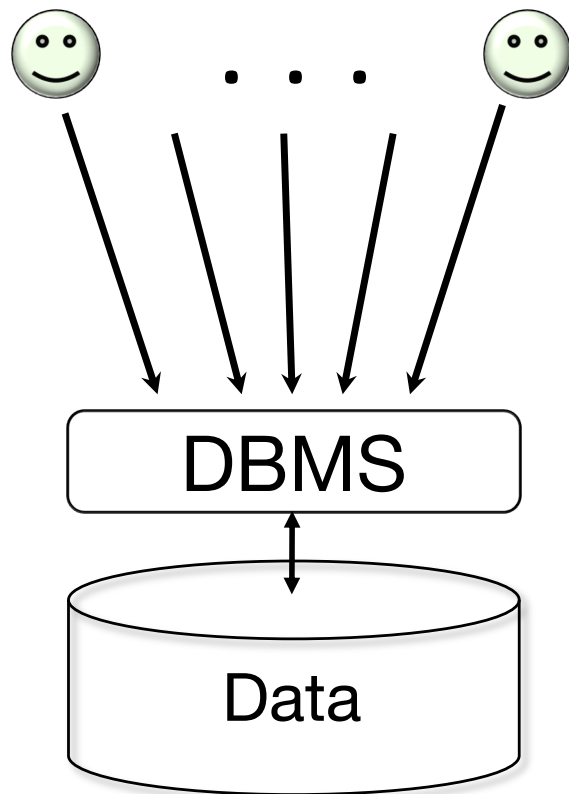
Operations may be interleaved, but execution must be equivalent to some sequential (serial) order of all transactions

## Disadvantages

- Overhead in locking
- Reduction in concurrency

# ACID Properties: Isolation

---



## Weaker “Isolation Levels”

Read Uncommitted

Read Committed

Repeatable Read

**Serializable**

Weak  
↓  
Strong

↓ Overhead in locking

↑ Concurrency

↓ Consistency Guarantees

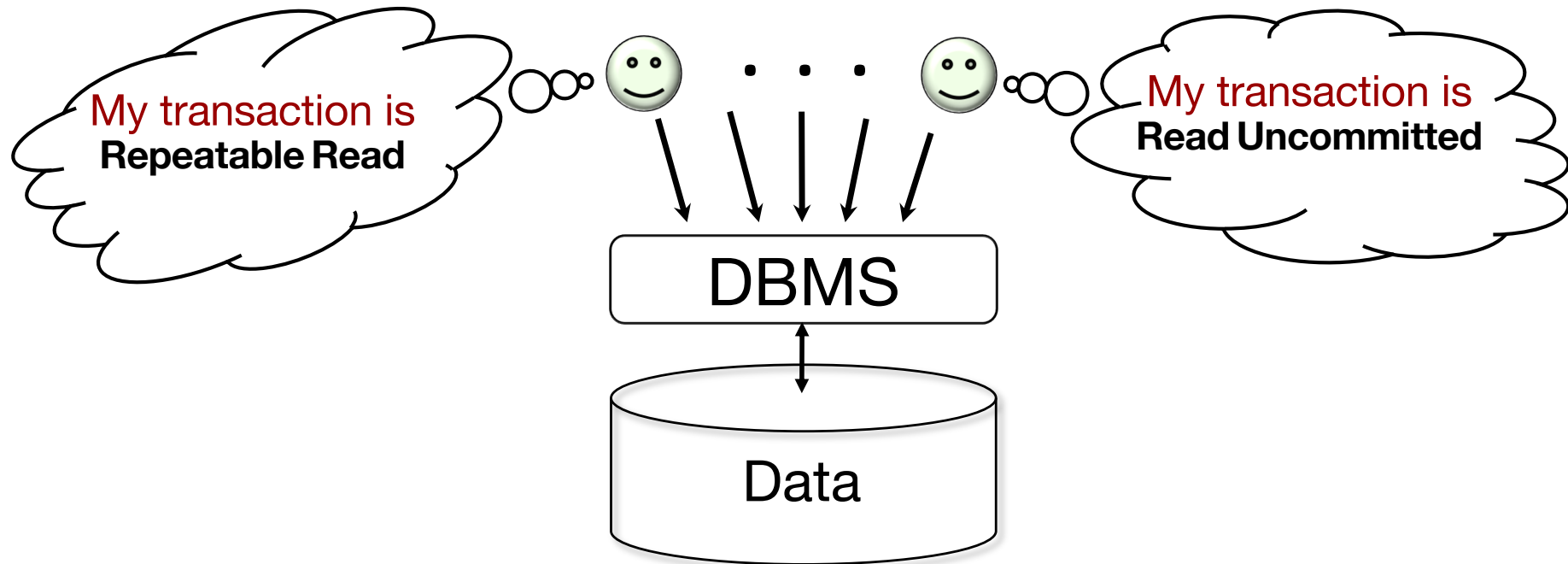
# Isolation Levels

---

Per transaction

It does not affect the behaviour of any other transaction

Specific to Reads



# Dirty Reads

---

“Dirty” data item: written by an uncommitted transaction

Update College Set enr = enr + 1000 Where cName = 'Stanford'

T<sub>1</sub>

concurrent with ...

Select avg(enr) From College

T<sub>2</sub>



If read before T<sub>1</sub> commits, this value is known as dirty

Assume there is a commit at the end of each box



## Dirty Reads – Example 2

---

Update Student Set GPA = (1.1) \* GPA Where sizeHS > 2500

T<sub>1</sub>

concurrent with ...

Select GPA From Student Where sID=123

T<sub>2</sub>

concurrent with ...

Update Student Set sizeHS=2600 Where sID=234

T<sub>3</sub>

Where can we have dirty data items?

There are no  
dirty reads  
within the same  
transaction

# Read Uncommitted

---

A transaction may perform dirty reads

Update Student Set GPA =  $(1.1) * \text{GPA}$  Where sizeHS > 2500

T<sub>1</sub>

concurrent with ...

Select avg(GPA) From Student

T<sub>2</sub>

If transactions are serializable

T1; T2 or

T2; T1

# Read Uncommitted

---

Update Student Set GPA = (1.1) \* GPA Where sizeHS > 2500

T<sub>1</sub>

concurrent with ...

Set Transaction Isolation Level Read Uncommitted;  
Select avg(GPA) From Student;

T<sub>2</sub>

We don't have serializable behaviour

We might don't care that much about consistency

# Read Committed

---

A transaction may **not** perform dirty reads

Still does not guarantee global serializability

Update Student Set GPA = (1.1) \* GPA Where sizeHS > 2500

T<sub>1</sub>

concurrent with ...

Set Transaction Isolation Level Read Committed;

Select avg(GPA) From Student

Select max(GPA) From Student

T<sub>2</sub>

# Repeatable Read

---

A transaction may **not** perform dirty reads

An item read multiple times cannot change value

Still does not guarantee global serializability

Update Student Set GPA = (1.1) \* GPA Where sizeHS > 2500;  
Update Student Set sizeHS=1500 Where sID = 123;

T<sub>1</sub>

concurrent with ...

Set Transaction Isolation Level Repeatable Read;  
Select avg(GPA) From Student  
Select avg(sizeHS) From Student

T<sub>2</sub>

# Repeatable Read

---

A transaction may **not** perform dirty reads

An item read multiple times cannot change value

But a relation *can* change: “phantom” tuples

Insert into Student [100 new tuples]

$T_1$

Phantom tuples

concurrent with ...

Set Transaction Isolation Level Repeatable Read;

Select avg(GPA) From Student

Select max(GPA) From Student

$T_2$

# Repeatable Read

---

A transaction may **not** perform dirty reads

An item read multiple times cannot change value

But a relation *can* change: “phantom” tuples

Delete from Student [100 new tuples]

$T_1$

concurrent with ...

Set Transaction Isolation Level Repeatable Read;

Select avg(GPA) From Student

Select max(GPA) From Student

$T_2$

Once read, values get locked and deletion is not possible in the middle of  $T_2$

# Read Only Transactions

---

Helps system optimize performance

Independent of isolation level

Not going to perform modifications to the database within the transaction



Set Transaction Read Only;

Set Transaction Isolation Level Repeatable Read;

Select avg(GPA) From Student

Select max(GPA) From Student



weak		dirty reads	nonrepeatable reads	phantoms
	Read Uncommitted	Y	Y	Y
	Read Committed	N	Y	Y
	Repeatable Read	N	N	Y
	Serializable	N	N	N
strong				

# Isolation Levels: Summary

---

Standard default: Serializable

## Weaker isolation levels

Increased concurrency + decreased overhead = increased performance

Weaker consistency guarantees

Some systems have default Repeatable Read

## Isolation level per transaction

Each transaction's reads must conform to its isolation level

# Kahoot time!

---

Any doubts?

# Readings

---

Jeffrey Ullman, Jennifer Widom, A first course in  
Database Systems 3<sup>rd</sup> Edition

Section 6.6 – Transactions in SQL