LECTURE 21

Transactions

November 12, 2024

Data 101, Fall 2024 @ UC Berkeley

Lisa Yan, Michael Ball https://data101.org/fa24/

CC BY NC SA

slido

Join at slido.com #transactions

ⓘ Click **Present with Slido** or install our Chrome extension to display joining instructions for participants while presenting.

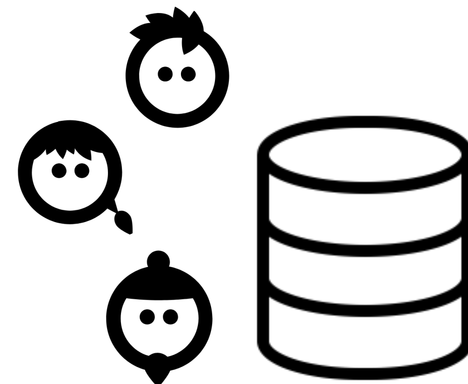
cc BY NC SA

Updates create challenges

#transactions

So far, we've largely focused on data science/read-only workloads.
In many settings, we need to also support **updates**.

- Single-user, one-at-a-time updates are easy.
- But **multi-user, simultaneous updates** are challenging.



When updating data, we want **correctness + speed**, particularly when users are accessing and modifying the *same* relations.

Course/lecture goals:

- Understand **challenges** that updates cause
- Understand **APIs**/related **guarantees**.

Today, a glance at database internals: **transactions**. More in CS186: Database Systems!

CC BY NC SA

Two Main Features We Expect from our Database System for Updates

#transactions

Concurrency Control:

- Many users query and update a database simultaneously.
- How do we avoid confusion / incorrect state?

Recovery:

- What happens when things fail?
- Many such failure modes: Cancel modification partway,
- app failure, DB engine failure, HW failure...

To understand these features, we need to introduce the concept of **transactions**.

cc BY NC SA

Transactions/ TCL

Lecture 21, Data 101, Fall 2024

Transactions/TCL

The ACID Principle

Isolation

Transaction Schedules

Serializability

Details:

- Strict 2-Phase Locking
- Conflicting Actions

[Extra] Additional slides

CC BY NC SA

What is a Transaction?

#transactions

Colloquially, a **transaction** in a database is a unit of work that should appear to "happen together."

Classic example: Debit/credit banking transaction, i.e., moving $1k from one account (1111) to another (9999).

```
BEGIN
-- "debit" one account
UPDATE checking
   SET amount = amount - 1000
 WHERE acctId = 1111;
-- "credit" the other account
UPDATE savings
   SET amount = amount + 1000
 WHERE acctId = 9999;
COMMIT
```

These SQL commands need to "happen together."

BEGIN, **COMMIT** are SQL **TCL (Transaction Control Language)** commands.

CC BY NC SA

What constitutes “happening together”? Observation #1

#transactions

```
BEGIN
-- "debit" one account
UPDATE checking
   SET amount = amount − 1000
 WHERE acctId = 1111;
-- "credit" the other account
UPDATE savings
   SET amount = amount + 1000
 WHERE acctId = 9999;
COMMIT
```

A few observations:

1. We need both debit and credit to happen, i.e., we should **not have partial transactions**.
2. ...?

(to be continued...)

CC BY NC SA

SQL TCL

#transactions

Note added 11/17/24: `SET` is not included in the TCL (as mentioned in Disc 12). The diagram says `SET Constraint` is under the TCL.

CC BY NC SA

SQL TCL, Briefly

Postgres BEGIN, START TRANSACTION, COMMIT, END, ROLLBACK, ABORT, SAVEPOINT

#transactions

A **transaction** in SQL is a list of commands sandwiched by **`BEGIN`** and **`COMMIT`**.

```
BEGIN
    <command 1>
    …
    <command n>
COMMIT
```

If `BEGIN/COMMIT` not specified, then most systems will **autocommit** individual SQL commands, i.e., auto-wrap each command in its own transaction.

Generally (with slight syntax variation across systems):

- **`BEGIN`** equivalent to **`START`**, **`BEGIN WORK`**, **`START TRANSACTION`**, etc.
- **`COMMIT`** equivalent to **`END`**, **`END WORK`**, **`END TRANSACTION`**, etc.

CC BY NC SA

[extra] SQL TCL: Savepoints

Postgres BEGIN, START TRANSACTION, COMMIT, END, ROLLBACK, ABORT, SAVEPOINT

#transactions

Savepoints let you break your transactions up into pieces. You can then "partially **rollback**" to a prior savepoint, or abort altogether.

- ABORT
- SAVEPOINT *save_name*
- ROLLBACK TO SAVEPOINT *save_name*
    - Undo any commands that happened after the specified savepoint; and
    - Implicitly destroy any savepoints created after the specified one.

Use case: beyond the scope of this class, but generally used with SQL conditionals or as part of database constraints.

```
BEGIN
UPDATE checking
   SET amount = amount - 1000
 WHERE acctId = 1234;
SAVEPOINT debit_done;
UPDATE savings
   SET amount = amount + 1000
 WHERE acctId = 9999;
SAVEPOINT credit_done;
ROLLBACK TO SAVEPOINT debit_done;
UPDATE savings
   SET amount = amount + 1000
 WHERE acctId = 4321;
END
```

(arbitrary example; what is this doing?)

CC BY NC SA

The ACID Principle

Lecture 21, Data 101, Fall 2024

#transactions

Transactions/TCL

The ACID Principle

Isolation

Transaction Schedules

Serializability

Details:

- Strict 2-Phase Locking
- Conflicting Actions

[Extra] Additional slides

CC BY NC SA

What constitutes “happening together”? Observations

#transactions

```
BEGIN
-- "debit" one account
UPDATE checking
   SET amount = amount – 1000
 WHERE acctId = 1111;
-- "credit" the other account
UPDATE savings
   SET amount = amount + 1000
 WHERE acctId = 9999;
COMMIT
```

These four properties, known as **ACID**, define how transactions guarantee **(1) concurrency control** and **(2) recovery**.

A few observations:

1. We need both debit and credit to happen, i.e., we should **not have partial transactions**.
2. At the end of transactions, any **database constraints** should still be satisfied.
3. Even if another transaction happens simultaneously, **one should appear to have finished “first.”**
4. A **committed transaction** should appear to have happened, even if there is a **power failure/reboot** later.

CC BY NC SA

ACID: Basic Guarantees

#transactions

ACID defines four properties of transactions that guarantee **concurrency control** and **recovery**.

Atomicity

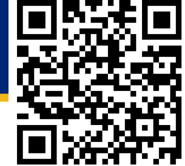
Consistency

Isolation

Durability

CC BY NC SA

ACID: Basic Guarantees



#transactions

ACID defines four properties of transactions that guarantee **concurrency control** and **recovery**.

Atomicity Either all the commands are reflected in the database, or none are.
Ex: Both debit+credit should occur, or both should fail to occur.

Consistency If COMMIT succeeds, all the database integrity checks hold true. (primary key/foreign keys, constraints, etc.)

Isolation Concurrent transactions should externally appear to run sequentially, i.e., 2 concurrent transactions should not "see" each other's intermediate results.

Durability If COMMIT succeeds, all changes from the transaction persist, even if there is a power failure or a reboot, until the transaction is overwritten by a later transaction.

CC BY NC SA

[History] Why ACID? Unknown, but...

Gray, 1981, Hearder and Reuter, 1983, Wikipedia (acid test (disambiguation))

The Transaction Concept: Virtues and Limitations

Jim Gray
Tandem Computers Incorporated
19333 Vallco Parkway, Cupertino CA 95014

June 1981

Jim Gray, PhD, UC Berkeley, Industry/ Academic researcher. 1998 Turing Award Winner "For seminal contributions to database and transaction processing research and technical leadership in system implementation."

Principles of Transaction-Oriented Database Recovery

THEO HAERDER
Fachbereich Informatik, University of Kaiserslautern, West Germany

ANDREAS REUTER[1]
IBM Research Laboratory, San Jose, California 95193

1983

These four properties, atomicity, consistency, isolation, and durability (ACID), describe the major highlights of the transaction paradigm, which has influenced many aspects of development in database systems. We therefore consider the question of whether the transaction is supported by a particular system to be the ACID test of the system's quality.

An **acid test** is any qualitative chemical or metallurgical assay which uses acid; most commonly, and historically, the use of a strong acid to distinguish gold from base metals.

- Acid Tests, parties in San Francisco in the mid-1960s centered on use of the drug LSD

CC BY NC SA

[Exercise] ACID

[#transactions]

```
BEGIN
-- "debit" one account
UPDATE checking
   SET amount = amount – 1000
 WHERE acctId = 1111;
-- "credit" the other account
UPDATE savings
   SET amount = amount + 1000
 WHERE acctId = 9999;
COMMIT
```

A few observations:

1. We need both debit and credit to happen, i.e., we should **not have partial transactions**.
2. At the end of transactions, any **database constraints** should still be satisfied.
3. Even if another transaction happens simultaneously, **one should appear to have finished "first."**
4. A **committed transaction** should appear to have happened, even if there is a **power failure/reboot** later.

A. D, I, C, A
B. I, C, A, D
C. A, C, I, D
D. A, C, D, I
E. Something else

Match 1-4 with A, C, I, and D from the ACID Principle.

CC BY NC SA

slido

Match 1-4 with A, C, I, and D from the ACID Principle.

ⓘ Click **Present with Slido** or install our [Chrome extension] to activate this poll while presenting.

CC BY NC SA

[Solution] ACID

#transactions

```
BEGIN
-- "debit" one account
UPDATE checking
   SET amount = amount – 1000
 WHERE acctId = 1111;
-- "credit" the other account
UPDATE savings
   SET amount = amount + 1000
 WHERE acctId = 9999;
COMMIT
```

A few observations:

A. We need both debit and credit to happen, i.e., we should **not have partial transactions**.

C. At the end of transactions, any **database constraints** should still be satisfied.

I. Even if another transaction happens simultaneously, **one should appear to have finished "first."**

D. A **committed transaction** should appear to have happened, even if there is a **power failure/reboot** later.

CC BY NC SA

How does the database address each ACID property?

#transactions

Atomicity

Consistency If COMMIT succeeds, all the database integrity checks hold true. (primary key/foreign keys, constraints, etc.)

Isolation

Standard database checks (relatively efficient to check for core things like attribute types, keys, constraints, etc.)

Durability

How does the database address each ACID property?

#transactions

Atomicity Either all the commands are reflected in the database, or none are.
Ex: Both debit+credit should occur, or both should fail to occur.

Consistency

Isolation

The database's internal recovery system.

After a crash:

- **Redo** all **committed** work; and
- **Undo** all **uncommitted** work!

See **CS186** for the implementation. Devil is in the details!

Durability If COMMIT succeeds, all changes from the transaction persist, even if there is a power failure or a reboot, until the transaction is overwritten by a later transaction.

CC BY NC SA

How does the database address each ACID property?

#transactions

Atomicity

Consistency

Isolation Concurrent transactions should externally appear to run sequentially, i.e., 2 concurrent transactions should not “see” each other’s intermediate results.

Durability

Provided by **concurrency control**, a component of the database. We’ll grasp the *intuition* today!!

CC BY NC SA

Isolation

Lecture 21, Data 101, Fall 2024

#transactions

Transactions/TCL

The ACID Principle

Isolation

Transaction Schedules

Serializability

Details:

- Strict 2-Phase Locking
- Conflicting Actions

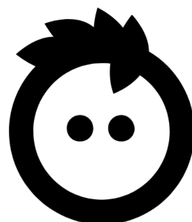
[Extra] Additional slides

CC BY NC SA

Isolation

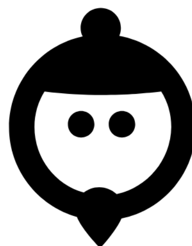
Isolation: Concurrent transactions should externally appear to run sequentially.

- If the database receives these transactions **simultaneously**, we should be able to successfully execute all three as if they happened "**in isolation**."



Hire Mercy as the new VP of Engineering!

Prepare tax projections for the 2nd quarter!

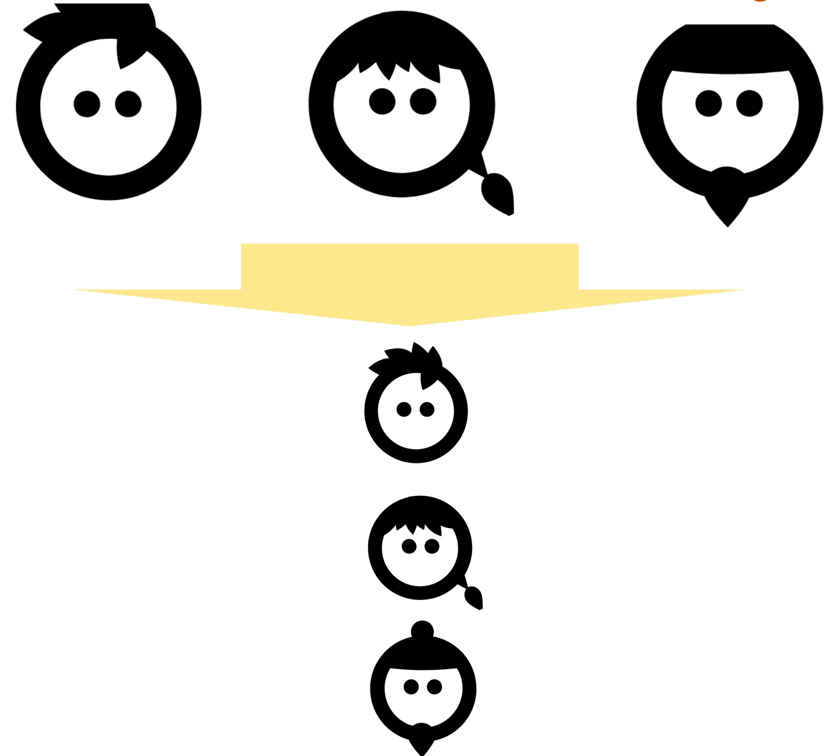


Move the entire payroll of the London Office to the Cambridge Office!

CC BY NC SA

Isolation,

The challenge: How do we execute these transactions "in isolation" but "concurrently"? With one single machine?



Assumption: **the precise order** of these three transactions **doesn't matter**. What matters is that they appeared to have been executed by the DBMS in **some** order.

CC BY NC SA

Isolation,

For simplicity, we will limit our discussion to **reads and writes of individual "objects"**:

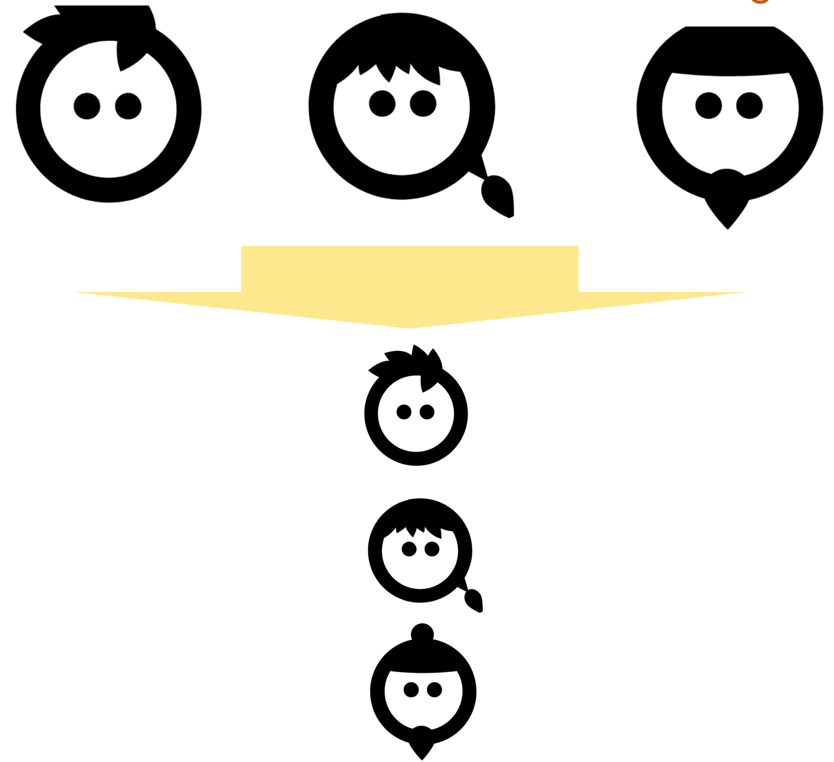
"Objects" := records (for now)

i-th transaction has Read from O:

$$R_i(O)$$

i-th transaction has Write to O:

$$W_i(O\ [= value])$$



CC BY NC SA

Transaction Schedules

Lecture 21, Data 101, Fall 2024

#transactions

Transactions/TCL

The ACID Principle

Isolation

Transaction Schedules

Serializability

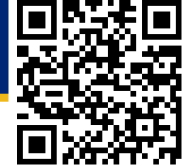
Details:

- Strict 2-Phase Locking
- Conflicting Actions

[Extra] Additional slides

CC BY NC SA

Determining Transaction Schedules that Maintain Isolation



#transactions

Our goal: Understand how multiple transactions can run **concurrently** (for performance) but also **in isolation** (for ACID).

To do so, we'll define the following:

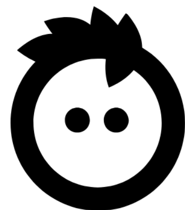
1. Define **transaction schedules** (i.e., list of read/writes).

2. Define **serial schedules**, which satisfy isolation by definition.

3. Define **serializable schedules**, which allow for concurrency while maintaining isolation.

CC BY NC SA

Example: Two Transactions



```
-- set Parth’s salary to 10% more
-- than Jonah’s
UPDATE employee
SET salary = (SELECT salary*1.1
              FROM employee
              WHERE name='Jonah')
WHERE name = 'Parth';
```

T_1:

$R_1(J)$
$W_1(P)$

```
-- set Parth’s salary to 10% more

UPDATE employee
SET salary = (SELECT salary*1.1
              FROM employee
              WHERE name='Parth')
WHERE name = ‘Parth';
```

T_2:

$R_2(P)$
$W_2(P)$

CC BY NC SA

Transaction Schedules

#transactions

A **transaction schedule** is a ordered list of actions from a set of transactions.

While there are many possible transaction schedules, a DBMS will pick one with which to schedule and execute read/write actions.

time ↓

T_1	T_2
$R_1(J)$	
$W_1(P)$	
	$R_2(P)$
	$W_2(P)$

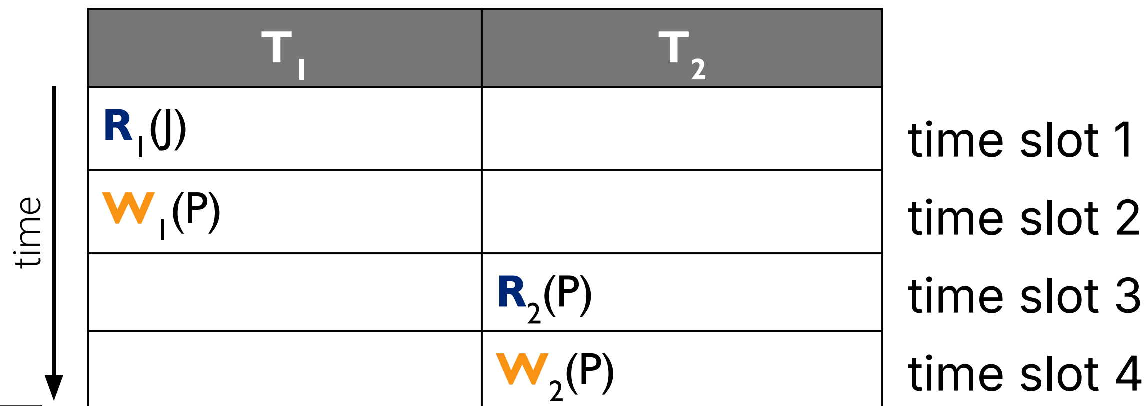
A proposed **Transaction Schedule** of T1 and T2

CC BY NC SA

Transaction Schedules

A **transaction schedule** is a ordered list of actions from a set of transactions.

✅ The **ordered schedule of actions** (reads from/writes to objects) represents the actual/potential **execution sequence** in time, as seen by the DBMS.



Transaction Schedules

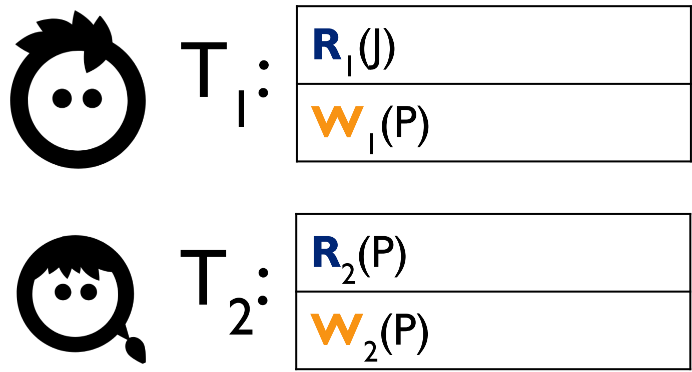
A **transaction schedule** is a ordered list of actions from a set of transactions.

- The **ordered schedule of actions** (reads from/writes to objects) represents the actual/potential **execution sequence** in time, as seen by the DBMS.

✅ The **order** in which two actions from the **same transaction T** are scheduled must reflect the **order in which they appear in T**.

time ↓

T_1	T_2
R_1(J)	
W_1(P)	
	R_2(P)
	W_2(P)



cc BY NC SA

Determining Transaction Schedules that Maintain Isolation

#transactions

Our goal: Understand how multiple transactions can run **concurrently** (for performance) but also **in isolation** (for ACID).

To do so, we'll define the following:

✅ Define **transaction schedules** (i.e., list of read/writes).

2. Define **serial schedules**, which satisfy isolation by definition.

3. Define **serializable schedules**, which allow for concurrency while maintaining isolation.

CC BY NC SA

Serial Schedules

#transactions

A **serial schedule** is a transaction schedule for which actions from different transactions are **not interleaved**.

time ↓

T_1	T_2
$R_1(J)$	
$W_1(P)$	
	$R_2(P)$
	$W_2(P)$

serial schedule

T_1	T_2
	$R_2(P)$
	$W_2(P)$
$R_1(J)$	
$W_1(P)$	

serial schedule

T_1	T_2
	$R_2(P)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	

not a serial schedule; transactions **interleaved**

T_1: $R_1(J)$, $W_1(P)$

T_2: $R_2(P)$, $W_2(P)$

Serial schedules exhibit **no concurrency**, because actions of a transaction are executed together and separate from those in other transactions.

CC BY NC SA

Do these schedules satisfy the isolation property?

#transactions

1.

T_1	T_2
$R_1(J)$	
$W_1(P)$	
	$R_2(P)$
	$W_2(P)$

time ↓

2.

T_1	T_2
	$R_2(P)$
	$W_2(P)$
$R_1(J)$	
$W_1(P)$	

3.

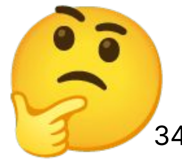
T_1	T_2
	$R_2(P)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	

T_1: $R_1(J)$, $W_1(P)$

T_2: $R_2(P)$, $W_2(P)$

Isolation: Concurrent transactions should externally appear to run sequentially, i.e., 2 concurrent transactions should not (appear to) "see" each other's intermediate results.

Which of these three schedules satisfy the **isolation** property? Select all.



CC BY NC SA

slido

Which of these three schedules satisfy the isolation property? Select all.

ⓘ Click **Present with Slido** or install our Chrome extension to activate this poll while presenting.

CC BY NC SA

Serializability

Lecture 21, Data 101, Fall 2024

#transactions

Transactions/TCL

The ACID Principle

Isolation

Transaction Schedules

Serializability

Details:

- Strict 2-Phase Locking
- Conflicting Actions

[Extra] Additional slides

CC BY NC SA

Do these schedules satisfy the isolation property?

1.

time ↓

T_1	T_2
$R_1(J)$	
$W_1(P)$	
	$R_2(P)$
	$W_2(P)$

Yes

2.

T_1	T_2
	$R_2(P)$
	$W_2(P)$
$R_1(J)$	
$W_1(P)$	

Yes

3.

T_1	T_2
	$R_2(P)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	

Yes!

T_1: $R_1(J)$, $W_1(P)$

T_2: $R_2(P)$, $W_2(P)$

Isolation. If we execute a given schedule, from the DBMS's POV, individual transactions appear to be executed sequentially.

CC BY NC SA

All three schedules satisfy the isolation property!

#transactions

1.

T_1	T_2
$R_1(J)$	
$W_1(P)$	
	$R_2(P)$
	$W_2(P)$

2.

T_1	T_2
	$R_2(P)$
	$W_2(P)$
$R_1(J)$	
$W_1(P)$	

3.

T_1	T_2
	$R_2(P)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	

time

T_1: $R_1(J)$, $W_1(P)$

T_2: $R_2(P)$, $W_2(P)$

It is okay that these two serial schedules produce **non-equivalent database outcome states**!

Isolation. If we execute a given schedule, from the DBMS's POV, individual transactions appear to be executed sequentially.

CC BY NC SA

All three schedules satisfy the isolation property!

time

1.

2.

T_1	T_2
	$R_2(P)$
	$W_2(P)$
$R_1(J)$	
$W_1(P)$	

3.

T_1	T_2
	$R_2(P)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	

T_1:

$R_1(J)$
$W_1(P)$

T_2:

$R_2(P)$
$W_2(P)$

Despite the interleaving, this schedule has an **equivalent** database outcome to one of the serial schedules!

Isolation. If we execute a given schedule, from the DBMS's POV, individual transactions appear to be executed sequentially.

CC BY NC SA

Schedule 3 is a serializable schedule

time

1.

2.

T_1	T_2
	$R_2(P)$
	$W_2(P)$
$R_1(J)$	
$W_1(P)$	

3.

T_1	T_2
	$R_2(P)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	

T_1:

$R_1(J)$
$W_1(P)$

T_2:

$R_2(P)$
$W_2(P)$

Despite the interleaving, this schedule has an **equivalent** database outcome to one of the serial schedules!

Schedule 3 is a **serializable schedule**: a transaction schedule whose database outcome is **equivalent to some serial schedule**.

CC BY NC SA

Unserializable Schedules

2.

T_1	T_2
	$R_2(P)$
	$W_2(P)$
$R_1(J)$	
$W_1(P)$	

serial schedule

3.

T_1	T_2
	$R_2(P)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	

serializable schedule

4.

T_1	T_2
	$R_2(P)$
$R_1(J)$	
$W_1(P)$	
	$W_2(P)$

T_1: $R_1(J)$, $W_1(P)$

T_2: $R_2(P)$, $W_2(P)$

time

Not all schedules are serializable! This is an **unserializable schedule**, because there is no serial equivalent, and therefore transactions do not appear isolated.

CC BY NC SA

A Joke

[#transactions]

CC BY NC SA

Announcements

BY NC SA

Strict 2-Phase Locking

Lecture 21, Data 101, Fall 2024

#transactions

Transactions/TCL

The ACID Principle

Isolation

Transaction Schedules

Serializability

Details:

- Strict 2-Phase Locking
- Conflicting Actions

[Extra] Additional slides

CC BY NC SA

Summary so far

Our goal: Allow multiple transactions to run **concurrently** (for performance) but also **in isolation** (for ACID).

To do so, we've traced the following steps:

1. Define transaction schedules (i.e., list of read/writes).
2. Define **serial schedules**, which satisfy isolation by definition.
3. Define **serializable schedules**, which allow for concurrency while maintaining isolation.

T_1	T_2
	$R_2(P)$
	$W_2(P)$
$R_1(J)$	
$W_1(P)$	

serial schedule

T_1	T_2
	$R_2(P)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	

serializable schedule

T_1	T_2
	$R_2(P)$
$R_1(J)$	
$W_1(P)$	
	$W_2(P)$

unserializable schedule

CC BY NC SA

DBMS goals

#transactions

Under the covers, a database system **can allow serializable schedules** that may not be serial, but after execution have the same outcome as some serial schedule.

- Allows multiple transactions to run at the same time.
- Much better for performance!!

CS186: Build systems that guarantee serializability for all executed schedules.

T_1	T_2
	$R_2(P)$
	$W_2(P)$
$R_1(J)$	
$W_1(P)$	

serial schedule

T_1	T_2
	$R_2(P)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	

serializable schedule

T	T
	(P)
R_1(J	
$W_1(P)$	
	$W_2(P)$

unserializable schedule

BY NC SA

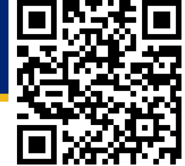
Two final goals of this lecture

#transactions

- Briefly, how do databases build schedule that ensure serializability?
  - Strict Two-Phase Locking
- Conceptually, how do we know a schedule is serializable?
  - Conflicting actions

CC BY NC SA

Database locking



#transactions

One of the most straightforward implementations that databases can use to ensure serializability is called **Strict Two-Phase Locking** (**Strict 2PL**).

- This is a **conservative** method to guarantee serializability.
- It prevents certain serializable schedules and therefore may suffer some performance hits, but overall there is no harm done because **it is always correct / satisfies ACID principle**.
- What theoretical guarantees? See **conflict serializability**

Skipped slides: details on Strict 2PL.

BY NC SA

Determining Serializability: Conflicting Actions

Lecture 21, Data 101, Fall 2024

#transactions

Transactions/TCL

The ACID Principle

Isolation

Transaction Schedules

Serializability

Details:

- Strict 2-Phase Locking
- Conflicting Actions

[Extra] Additional slides

CC BY NC SA

How do we know if a schedule is serializable?

#transactions

We like **serializable schedules**.

- **Isolation**, again: After the dust settles, transactions appear to have happened in some order (which may seem "arbitrary"). However, the order means that:
  - the txns appear to have followed a serial schedule.
  - that txns can be "rolled back" one-by-one.

Conflicting actions between transactions will determine if a schedule is **serializable**.

What does it mean???
Let's dive in!

CC BY NC SA

Conflicting Actions

Def: Two actions **conflict** if:

- They are two different, concurrent transactions.
- They reference the same object.
- At least one is a write.

Alt Def: If T1 and T2 have **conflicting actions**, then every **equivalent serial schedule** (i.e., with the same database outcome) must have T1 and T2 in some **specific order**.

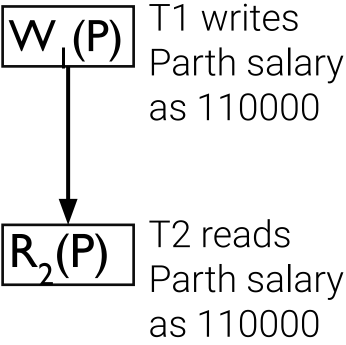
CC BY NC SA

Which of the following are conflicting actions?

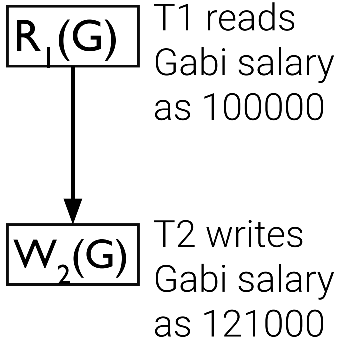
Alt Def: If T1 and T2 have **conflicting actions**, then every **equivalent serial schedule** (i.e., with the same database outcome) must have T1 and T2 in some **specific order**.

Suppose T1 → T2 in a schedule, i.e., T1 comes before T2.

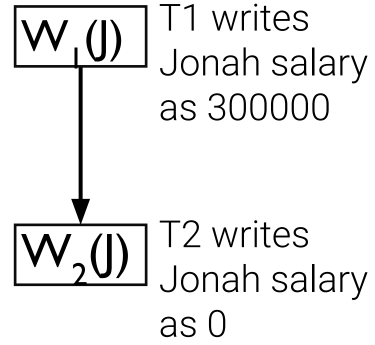
For which of the following would the resulting flip of actions mean that this transaction order would change, i.e., that now T2 → T1? Select all.



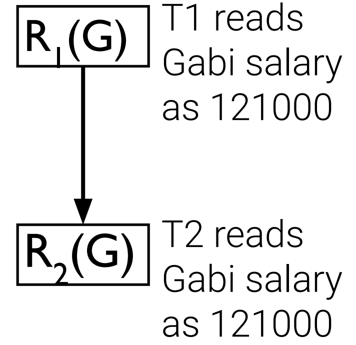
A.



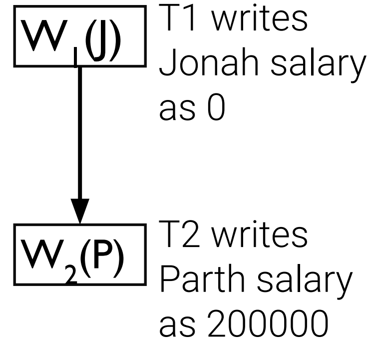
B.



C.



D.



E.

slido

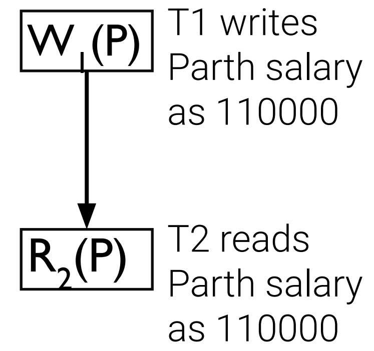
Select all for which the following is true: Flipping the order of the two actions in T1 and T2 would result in a different database outcome state.

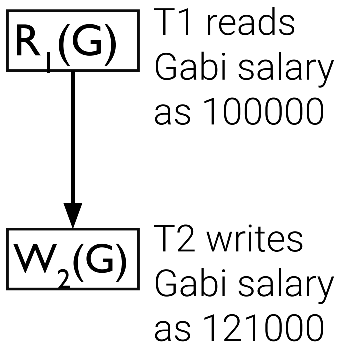
ⓘ Click **Present with Slido** or install our [Chrome extension] to activate this poll while presenting.

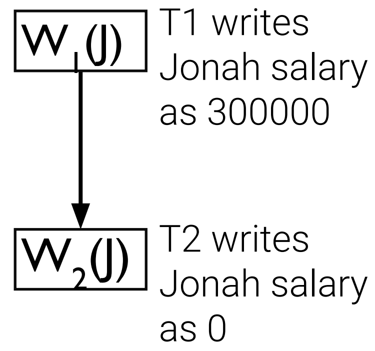
CC BY NC SA

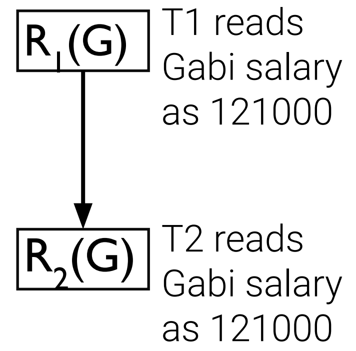
Suppose T1 → T2 in a schedule. For which of the following would the resulting flip of actions mean that this **transaction order** would change, i.e., that now T2 → T1? Select all.

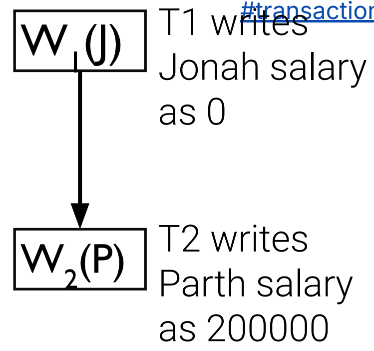
#transactions











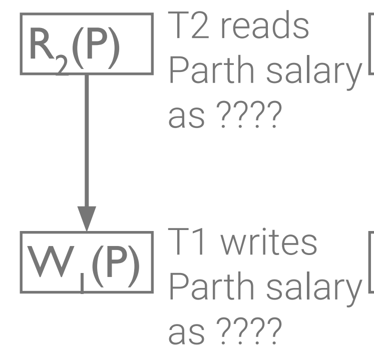
hypothetically

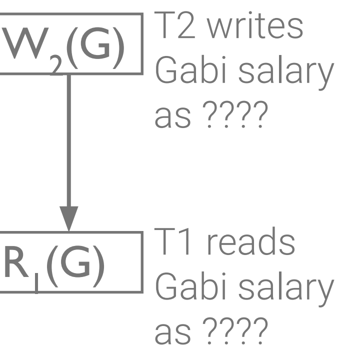
hypothetically

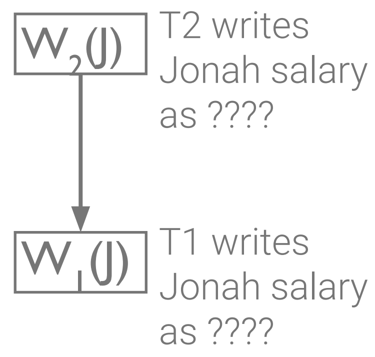
hypothetically

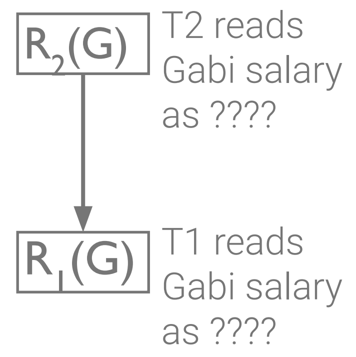
hypothetically

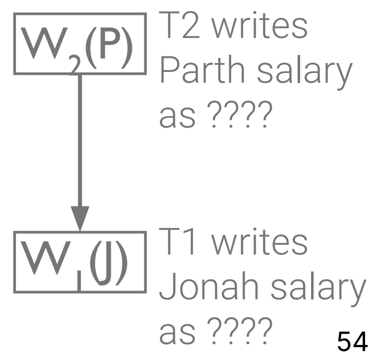
hypothetically











Which of the following are conflicting actions?

#transactions

Alt Def: If T1 and T2 have **conflicting actions**, then every **equivalent serial schedule** (i.e., with the same database outcome) must have T1 and T2 in some **specific order**.

Suppose T1 → T2 in a schedule, i.e., T1 comes before T2.

> For which of the following would the resulting flip of actions mean that this transaction order would change, i.e., that now T2 → T1? Select all.

A. $W_1(P)$ → $R_2(P)$
T1 writes Parth salary as 110000
T2 reads Parth salary as 110000

B. $R_1(G)$ → $W_2(G)$
T1 reads Gabi salary as 100000
T2 writes Gabi salary as 121000

C. $W_1(J)$ → $W_2(J)$
T1 writes Jonah salary as 300000
T2 writes Jonah salary as 0

D. $R_1(G)$ → $R_2(G)$
T1 reads Gabi salary as 121000
T2 reads Gabi salary as 121000

E. $W_1(J)$ → $W_2(P)$
T1 writes Jonah salary as 0
T2 writes Parth salary as 200000

review next time

Which of the following are conflicting actions? review next time

#transactions

Alt Def: If T1 and T2 have **conflicting actions**, then every **equivalent serial schedule** (i.e., with the same database outcome) must have T1 and T2 in some **specific order**.

Suppose T1 → T2 in a schedule, i.e., T1 comes before T2.

For which of the following would the resulting flip of actions mean that this transaction order would change, i.e., that now T2 → T1? Select all.

- $W_1(P)$ T1 writes Parth salary as 110000 → $R_2(P)$ T2 reads Parth salary as 110000
- $R_1(G)$ T1 reads Gabi salary as 100000 → $W_2(G)$ T2 writes Gabi salary as 121000
- $W_1(J)$ T1 writes Jonah salary as 300000 → $W_2(J)$ T2 writes Jonah salary as 0

cannot be flipped! **conflicting actions**!

- $R_1(G)$ T1 reads Gabi salary as 121000 → $R_2(G)$ T2 reads Gabi salary as 121000
- $W_1(J)$ T1 writes Jonah salary as 0 → $W_2(P)$ T2 writes Parth salary as 200000

can be flipped! **no conflicts**!

CC BY NC SA

How do we know if a schedule is serializable?

#transactions

We like **serializable schedules**.

- **Isolation**, again: After the dust settles, transactions appear to have happened in some order (which may seem “arbitrary”). However, the order means that:
  - the txns appear to have followed a serial schedule.
  - that txns can be “rolled back” one-by-one.

A schedule is serializable if **all conflicting actions dictate a specific ordering of the transactions** (with no cycles)

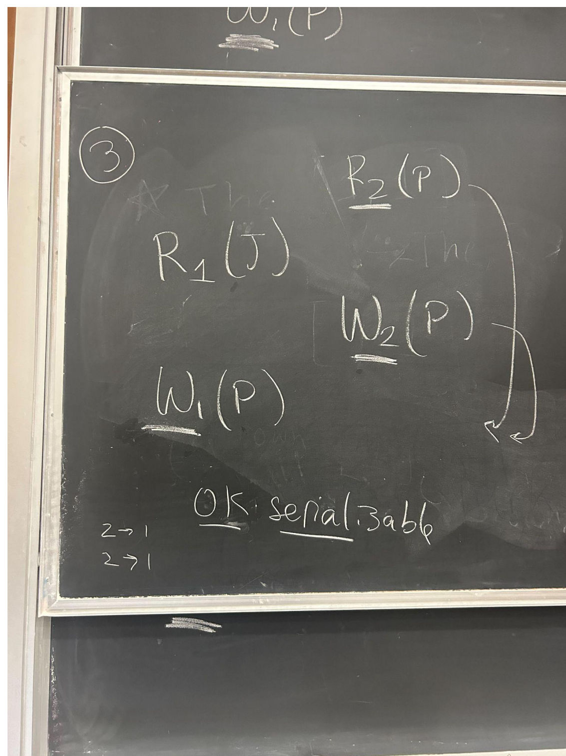
- A topological sort on the graph of conflicts between transactions.

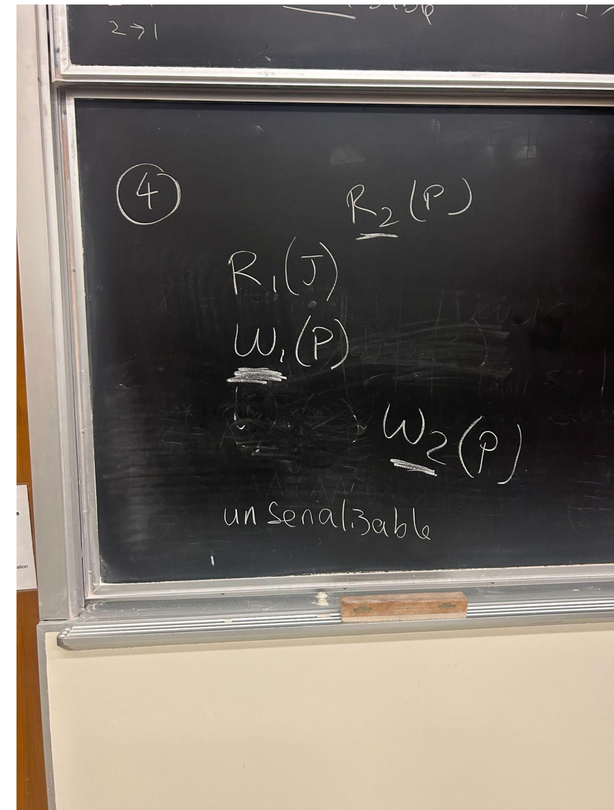
Conflicting actions between transactions determine if a schedule is **serializable**.

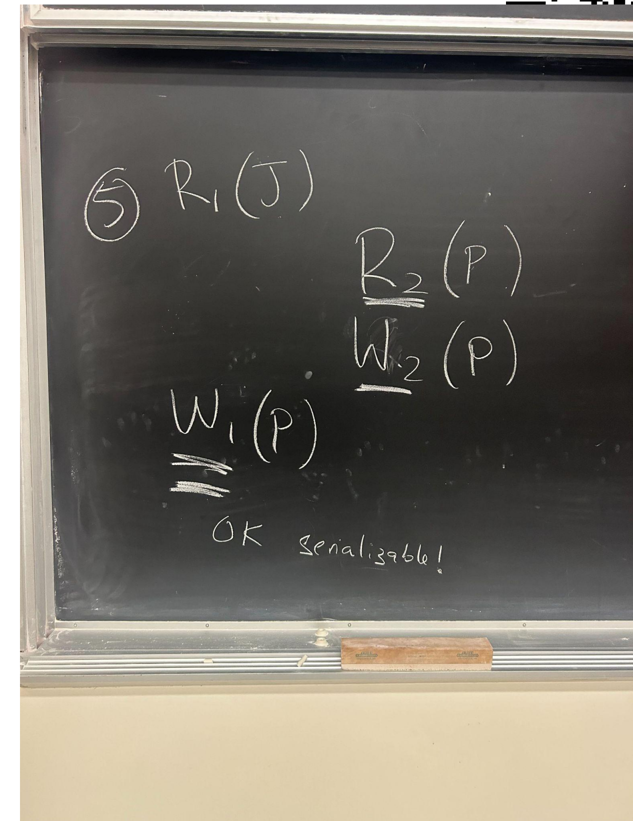
review next time

CC BY NC SA

From the board







review next time

CC BY NC SA

The previous slide, in table form

serializable
3.

T_1	T_2
	$R_2(P)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	

unserializable
4.

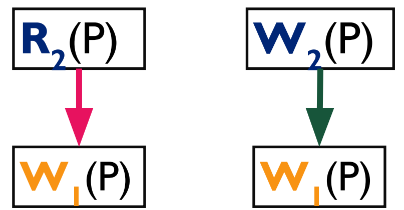
T_1	T_2
	$R_2(P)$
$R_1(J)$	
$W_1(P)$	
⚠️	$W_2(P)$

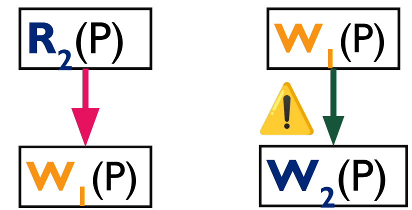
serializable
5.

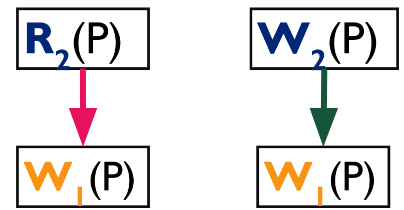
T_1	T_2
$R_1(J)$	
	$R_2(P)$
	$W_2(P)$
$W_1(P)$	

T_1: $R_1(J)$, $W_1(P)$

T_2: $R_2(P)$, $W_2(P)$







Schedule 4: The two pairs of conflicting actions imply two different orders to T1 and T2.

for next time

Performance Tradeoffs: Snapshot Isolation

Lecture 21, Data 101, Fall 2024

#transactions

Transactions
The ACID Principle
Isolation and Serializability
Strict 2-Phase Locking
Conflicting Actions
[Extra] Conflict Graphs
[Extra] Conflict Serializable
Weak Isolation
[Extra] Additional slides

CC BY NC SA

Final thoughts: Why Should Data Engineers Know Transactions?

#transactions

You may think that transactions (and serializability) are very much in the weeds of DBMS design, which we don't particularly implement in this course. However...

Inevitably you will update a database and manage data from transactional databases!

- This means you should have a sense of its characteristics.

If your DB is slow for transactional reasons:

- You should understand why
- And how you can trade-off speed and "correctness," i.e., redefine your transactions.

Finally, transaction concepts are also quite useful outside of databases.

- Examples: Queueing systems, e.g., RabbitMQ or Kafka.

CC BY NC SA

#transactions

BY NC SA

Serialized Transactions: A summary

#transactions

Serialized transactions ensure ACID properties of shared access, particularly Isolation.

- **Strict 2PL** is a common implementation of serialization, though it is not the only one.

Life is good?...Except...

- `SELECT avg(gpa)  FROM students;`
  - Locks all students!
  - But we likely don’t need this to be 100% correct!
- Sometimes we prefer to trade **correctness** for a little more **performance**.

CC BY NC SA

Approximating Serialized Transactions with Weak Isolation

#transactions

Serialized transactions ensure ACID properties of shared access, particularly Isolation.

- **Strict 2PL** is a common implementation of serialization, though it is not the only one.

Life is good? Except…

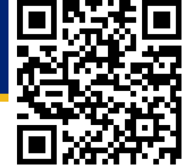
- `SELECT avg(gpa)  FROM students;`
  - Locks all students!
  - But we likely don't need this to be 100% correct!
- Sometimes we prefer to trade **correctness** for a little more **performance**.

Enter: **Weak Isolation**.

- Each isolation can choose to be a "bit sloppy"…
- …as long as it doesn't mess up other transaction's choices to do so.
- The most common weak isolation implementation is **snapshot isolation**.
- This is a much weaker property of isolation than serialized transactions, but it's good enough when we prefer more concurrency/higher performance.

cc BY NC SA

Snapshot Isolation



#transactions

Snapshot isolation is a weaker form of isolation than serialization, but it's good enough when we prefer more concurrency/higher performance.

- Database system requirements: Keep multiple versions of tuples.

At transaction start: Take a "snapshot" of the database, off which to do reads/writes.

- **snapshot reads**: All reads of this transaction are from this snapshot.
- **write validation**: This transaction can commit if none of its writes conflict with other transactions since the snapshot was taken.
  - If write-write conflicts, then abort this transaction.

cc BY NC SA

Snapshot Isolation is Actually Popular

#transactions

Isolation levels (both default and maximum) vary in support across different database engines.

Marketing also varies!

- When Oracle says "Serializable," they actually are giving you Snapshot Isolation!!

The maximum levels of many cloud DBMSs is not always the theoretical maximum, which is "serializable" transactions.

- **Serializable**: Google Cloud Spanner, CockroachDB, Azure SQL Server
- Read Commit: Snowflake, AWS Aurora
  - For more about Read Commit and others, check out the bonus slides.

Database	Default	Maximum
Actian Ingres 10.0/10S	S	S
Aerospike	RC	RC
Akiban Persistit	SI	SI
Clustrix CLX 4100	RR	RR
Greenplum 4.1	RC	S
IBM DB2 10 for z/OS	CS	S
IBM Informix 11.50	Depends	S
MySQL 5.6	RR	S
MemSQL 1b	RC	RC
MS SQL Server 2012	RC	S
NuoDB	CR	CR
Oracle 11g	RC	SI
Oracle Berkeley DB	S	S
Oracle Berkeley DB JE	RR	S
Postgres 9.2.2	RC	S
SAP HANA	RC	SI
ScaleDB 1.02	RC	RC
VoltDB	S	S

RC: read committed, RR: repeatable read, SI: snapshot isolation, S: serializability, CS: cursor stability, CR: consistent read

Table 2: Default and maximum isolation levels for ACID and NewSQL databases as of January 2013 (from [9]).

cc BY NC SA

[Bonus] Strict 2-Phase Locking: Details

Lecture 21, Data 101, Fall 2024

#transactions

Transactions

The ACID Principle

Isolation and Serializability

Strict 2-Phase Locking

Conflicting Actions

[Extra] Conflict Graphs

[Extra] Conflict Serializable

Weak Isolation

[Extra] Additional slides

CC BY NC SA

Database locking

#transactions

How do databases ensure **serializability**?

One of the most straightforward implementations is called **Strict Two-Phase Locking** (**Strict 2PL**).

- This is a **conservative** method to guarantee serializability.
- It prevents certain serializable schedules and therefore may suffer some performance hits, but overall there is no harm done because **it is always correct / satisfies ACID principle**.
- What theoretical guarantees? See **conflict serializability**

Locking is the process of ensuring that 2 conflicting actions happen in order.

- The **first action** that arrives should "**lock**" the shared object.
- The **second action** that arrives needs to **wait** until the first action's **transaction completes**.
- (we'll define **conflicting action** more precisely later)

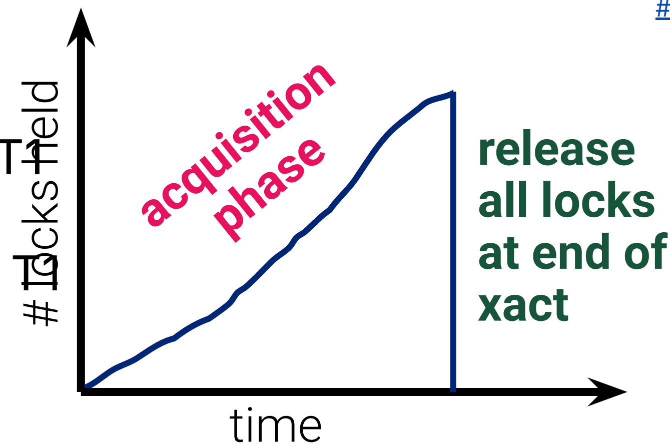
CC BY NC SA

Strict Two-Phase Locking (Strict 2PL)

Phase 1: During the transaction, **lock objects before use**.

Two types of locks:

- **S lock**: Before executing R1(O), transaction T1 must acquire a **shared lock** on O.
- **X lock**: Before executing W1(O), transaction T1 must acquire an **exclusive lock** on O.



Phase 2: At the end of the transaction (i.e., COMMIT or ROLLBACK), **release all locks at once**.

The Strict 2PL algorithm allows only serializable schedules!

Note that schedules can result in **deadlock**. See Discussion for more info/practice!

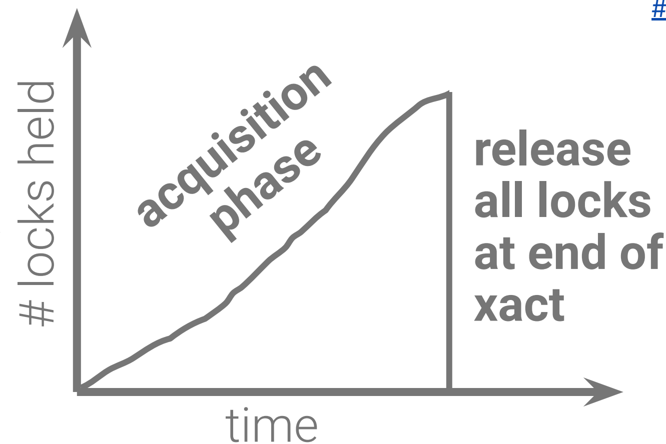
CC BY NC SA

Strict Two-Phase Locking (Strict 2PL), Practically

#transactions

What objects are we locking?

- For most purposes, assume the DBMS is locking **individual records**.
- It is sometimes useful to lock entire tables at once (e.g., to change a schema/a default attribute), but we won't go into detail.



What does it mean to "**acquire**" or "**release**" a lock?

- Under the hood: DBMS maintains some of "lock table" according to an **internal protocol**.
- The system ensures that all transactions follow the internal protocol's locking rules.
  - Analogy: red lights at intersections. You trust the protocol.

CC BY NC SA

[Extra] Determining Serializability: Conflict Graphs

Lecture 21, Data 101, Fall 2024

#transactions

Transactions
The ACID Principle
Isolation and Serializability
Strict 2-Phase Locking
Conflicting Actions
[Extra] Conflict Graphs
[Extra] Conflict Serializable
Weak Isolation
[Extra] Additional slides

CC BY NC SA

[Exercise] Determining Conflicting Actions

What are the **conflicting actions** in each of the schedules?

time ↓

1.

T_1	T_2
	$R_2(J)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	

2.

T_1	T_2
$R_1(G)$	
	$R_2(P)$
$W_1(P)$	
	$W_2(G)$

CC BY NC SA

[Exercise] Determining Conflicting Actions

#transactions

What are the **conflicting actions** in each of the schedules?

1. W1(P) and W2(P)
// write/write to same object

2. R1(G) and W2(G); // read/write same obj
W1(P) and R2(P) // read/write same obj

time ↓

1.

T_1	T_2
	$R_2(J)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	

2.

T_1	T_2
$R_1(G)$	
	$R_2(P)$
$W_1(P)$	
	$W_2(G)$

CC BY NC SA

[Exercise] Determining Serializability

#transactions

Suppose we have the following conflicting actions:

1. W1(P) and W2(P)
2. R1(G) and

Which of the following schedules are **serializable**?

Which of the following schedules are **serializable**?

time ↓

1.

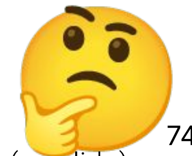
T_1	T_2
	$R_2(J)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	

2.

T_1	T_2
$R_1(G)$	
	$R_2(P)$
$W_1(P)$	
	$W_2(G)$

A. Serializable schedule, i.e., equivalent to some serial schedule of T1 and T2

B. Unserializable schedule, i.e., no equivalent serial schedule exists



(no slido)

[Exercise] Determining Serializability

#transactions

Suppose we have the following conflicting actions:

1. W1(P) and W2(P)
2. R1(G) and W2(G); W1(P) and R2(P)

Which of the following schedules are **serializable**?

time →

1.

T_1	T_2
	$R_2(J)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	

2.

T_1	T_2
$R_1(G)$	
	$R_2(P)$
$W_1(P)$	
	$W_2(G)$

A. **Serializable!** Equivalent to T2 happening before T1.

B. **Unserializable!** Conflicting actions can't be "flipped."

CC BY NC SA

Serializability of a schedule can be determined by drawing its **conflict graph**

#transactions

- One node per transaction Ti.
- Edge from Ti to Tj if:
  - Action a in Ti **conflicts** with Action b in Tj, AND
  - Action a happens **before** Action b in the schedule.

CC BY NC SA

#transactions

Serializability of a schedule can be determined by drawing its **conflict graph.**

- One node per transaction Ti.
- Edge from Ti to Tj if:
  - Action a in Ti **conflicts** with Action b in Tj, AND
  - Action a happens **before** Action b in the schedule.

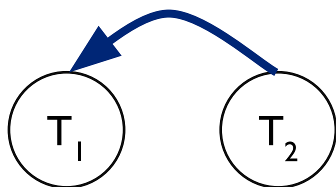
Given: Conflicting actions

1. W1(P) and W2(P)
2. R1(G) and W2(G);
   W1(P) and R2(P)

time ↓

1.

T_1	T_2
	$R_2(J)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	



Serializable!

2.

T_1	T_2
$R_1(G)$	
	$R_2(P)$
$W_1(P)$	
	$W_2(G)$

Unserializable!

CC BY NC SA

Serializability of a schedule can be determined by drawing its **conflict graph**.

#transactions

- One node per transaction Ti.
- Edge from Ti to Tj if:
  - Action a in Ti **conflicts** with Action b in Tj, AND
  - Action a happens **before** Action b in the schedule.

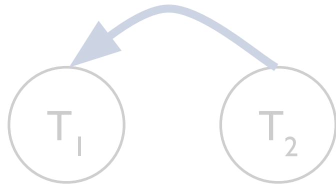
Given: Conflicting actions

1. W1(P) and W2(P)
2. R1(G) and W2(G); W1(P) and R2(P)

time

1.

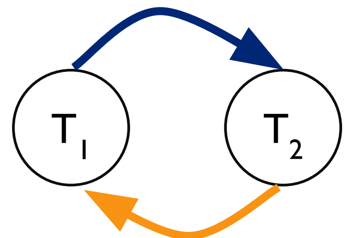
T_1	T_2
	$R_2(J)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	



Serializable!

2.

T_1	T_2
$R_1(G)$	
	$R_2(P)$
$W_1(P)$	
	$W_2(G)$



Unserializable!

#transactions

If the conflict graph has no cycles (**acyclic**), then the schedule is **serializable**. Otherwise, it has cycles and it is unserializable.

(proof later)

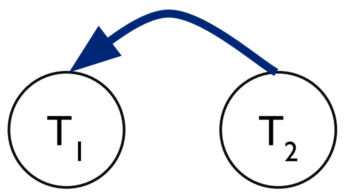
Given: Conflicting actions

1. W1(P) and W2(P)
2. R1(G) and W2(G); W1(P) and R2(P)

time

1.

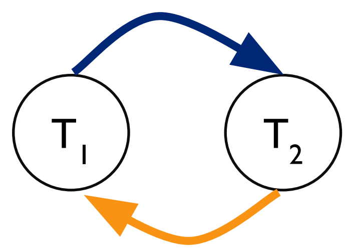
T_1	T_2
	$R_2(J)$
$R_1(J)$	
	$W_2(P)$
$W_1(P)$	



Serializable! Equivalent to T2 happening before T1.

2.

T_1	T_2
$R_1(G)$	
	$R_2(P)$
$W_1(P)$	
	$W_2(G)$



Unserializable! Conflicting actions can't be "flipped."

How do we know if a schedule is serializable?

We like **serializable schedules**.

- **Isolation**, again: For multiple concurrent transactions, after the dust settles, transactions appear to have happened in some order (which may seem "arbitrary"). However:
  - The order means that the transactions appear to have followed a serial schedule.
  - The order means that transactions can be "rolled back" one-by-one.

We did it!

The strategy for a determining serializability of a given schedule of interleaved transactions:

1. Identify the **conflicting actions**.
2. Draw the **conflict graph**.
3. If the conflict graph is **acyclic**, then the schedule is **serializable**. Else, it is **unserializable**.

Conflicting actions between transactions will determine if a schedule is **serializable**.

CC BY NC SA

[Extra] Formal Terminology: Conflict Serializable

Lecture 21, Data 101, Fall 2024

#transactions

Transactions

The ACID Principle

Isolation and Serializability

Strict 2-Phase Locking

Conflicting Actions

[Extra] Conflict Graphs

[Extra] Conflict Serializable

Weak Isolation

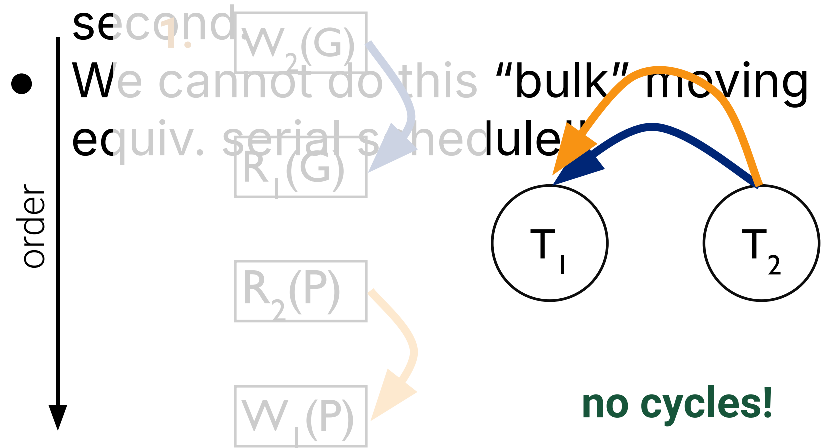
[Extra] Additional slides

BY NC SA

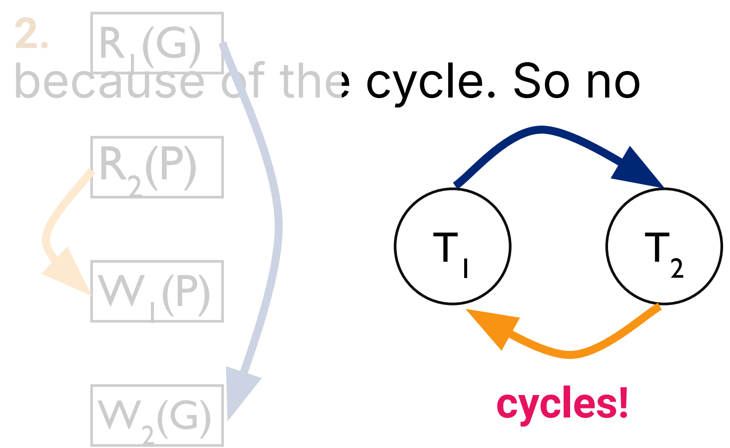
#transactions

Observations:

- The acyclic graph has a natural traversal order.
- An **order of conflicting actions** will guide us to an equivalent serial schedule.
- To make #1's serial schedule: Move all T2 actions first, move all T1 actions second.
- We cannot do this "bulk" moving for #2, because of the cycle. So no equiv. serial schedule!



Serializable! Equivalent to T2 happening before T1.



Unserializable! Conflicting actions can't be "flipped."

CC BY NC SA

Formal Terminology: Conflict Serializable Schedule

#transactions

Observations:

- The acyclic graph has a natural traversal order.
- An **order of conflicting actions** will guide us to an equivalent serial schedule.
- To make #1's serial schedule: Move all T2 actions first, move all T1 actions second.
- We cannot do this "bulk" moving for #2, because of the cycle. So no equiv. serial schedule!!

Def A schedule is **conflict serializable** if and only if the conflict graph is **acyclic** (has no cycles).

CC BY NC SA

Formal Terminology: Conflict Serializable Schedule

#transactions

Observations:

- The acyclic graph has a natural traversal order.
- An **order of conflicting actions** will guide us to an equivalent serial schedule.
- To make #1's serial schedule: Move all T2 actions first, move all T1 actions second.
- We cannot do this "bulk" moving for #2 because of the cycle. So no

Def   A schedule is **conflict serializable** if and only if the conflict graph is **ac**

Lemma

If a schedule is conflict serializable, then it is serializable.

Proof:

- By definition of **conflict serializable**, the given schedule has an acyclic conflict graph.
- Any serial schedule that follows the edges of the given conflict graph has the **same ordering** of conflicting actions and is therefore **equivalent** to the given schedule.
- By definition of serializable, the given schedule is therefore **serializable**.

CC BY NC SA

[Extra] The converse is not true

#transactions

Note that some serializable schedules are not necessarily conflict serializable!

From R&G *Database Ma* ition, Section 17.1, Figure 17.1 (p.550-1):

T1	T2	T3
R(A)		
	W(A)	
	Commit	
W(A)		
Commit		
		W(A)
		Commit

This schedule is equivalent to executing the transactions serially in the order T1, T2, T3, but it is not conflict equivalent to this serial schedule because the writes of T1 and T2 are ordered differently.

CC BY NC SA

[Extra] Weak Isolation: Read Commit

Lecture 21, Data 101, Fall 2024

#transactions

Transactions
The ACID Principle
Isolation and Serializability
Strict 2-Phase Locking
Conflicting Actions
[Extra] Conflict Graphs
[Extra] Conflict Serializable
Weak Isolation
[Extra] Additional slides

CC BY NC SA

“Read Committed” Isolation level

#transactions

- What if we dropped each Shared lock right after reading
  - But kept our eXclusive locks until COMMIT/ROLLBACK?
- Prevents “dirty” (uncommitted) reads from other transactions
  - Each read is of an unlocked/committed item!
- Doesn’t promise much more!

- This isolation level is called Read Committed
  - Note: respects the locks of other, Strict 2PL transactions

CC BY NC SA

Does it help us?

#transactions

Locks 1 student but must be serializable!

```sql
BEGIN
 ISOLATION LEVEL serializable;
UPDATE students
   SET gpa = 4.0
 WHERE sid = 1234;
END;
```

Locks *every* student but doesn't need to be 100% correct!

```sql
BEGIN
 ISOLATION LEVEL read committed;
SELECT avg(gpa)
  FROM students;
END;
```


CC BY NC SA

What could go wrong in Read Committed?

#transactions

- *Non-repeatable reads*
  - Suppose you read a tuple twice in your transaction
  - Another transaction could run between the two reads and update it!

- *Phantoms*
  - Suppose you run a query with a non-key WHERE clause
    - E.g. “find all students with an A grade”
  - If you run it again, some brand new tuples (phantoms) could appear!

- *Staleness:* Technically you could read a very old (but committed) “version”
  - Still satisfies the definition!

CC BY NC SA

Repeatable Read Isolation

#transactions

- Prevents dirty reads *and* non-repeatable reads

- A locking-based way to think about it:
  - All locks are held until COMMIT/ROLLBACK
  - But could be only tuple-level locks

- So phantoms are still possible!

BY NC SA

Snapshot Isolation is not (quite) Serializable

#transactions

Pat (T_1)

Set my salary to be 10% bigger than Gabi's

Set my salary to be 10% bigger than Pat's

Gabi (T_2)

P_0: 200000

Time

G_0: 100000

Pat (T_1)	Gabi (T_2)
$x = R_1(G_0)$	
	$y = R_2(P_0)$
$W_2(P_1 = 110000)$	
	$W_2(G_1 = 220000)$

NOT equivalent to either order (not serializable!)

Write skew anomaly: concurrent reads, and writes reflect the fact that they didn't read each other's writes!

CC BY NC SA