Transactions

Spring 2025

Today…

We dig into two problems that turn out to be related:

- how do we let multiple users interact with a database at the same time?
- how do we recover from a database crash without messing up the data?

Transaction management
Concurrency control
Crash and recovery

Handling multiple users

How do we ensure multiple users can use the database at the same time without getting in each other's way?

- why do we care?
- imagine the database underpins a website
- users of the website are basically users of the database

What is the issues with concurrent users reading and writing to a database?

- think **git** — multiple users writing to a repo, you get merge conflicts
- what's the equivalent of merge conflicts in a database?

Transactions

A **transaction** is a sequence of instructions to read/write from tables in the DB

Transactions bundle together operations that need to be done "at the same time"

Example: transferring money between bank accounts:

- remove from source account
- add to destination account

Users do not submit queries to a database but transactions

- could be one-query transactions

Transactions

Transactions bundle together operations that need to be done "at the same time"

Transactions from concurrent users should not conflict

- otherwise, chaos!

Easy way to prevent this: put transactions in a queue, run them one at a time

- that's super inefficient at scale
- many transactions can run at the same time without conflicting

(What does that even mean? What's a conflict?)

Transactions

Relational database guarantee a set of properties for transactions that have come to almost *define* the relational model:

the **ACID** properties

They ensure that transactions do not conflict, and that the data in the data is persistent and consistent.

Vocabulary:

- a transaction that completes (every action is done) is said to **commit**
- a transaction that does not complete (e.g., due to an error) is said to **abort**

ACID properties

Atomicity
All actions in a transaction happen when the transaction commits,
or none do when the transaction aborts.

Consistency
A transaction that commits leaves the database in a consistent state
(with respect to integrity constraints)

Isolation
Each transaction's execution is isolated from other transactions

Durability
The effects of a transaction persist if and only if it commits

Atomicity

Transactions are **atomic**

- atom = "indecomposable unit"
- you should not be able to observe half a transaction
- either **all** the actions in the transaction are performed
- or **none** of the actions in the transaction are performed

How is that achieved?

- when a transaction starts, it performs its actions
- if the transaction commits, we're done
- if the transaction aborts, it *rolls back* whatever changes it has made

Example

Sequence of reads and writes to tables

READ(A)
WRITE(C)
READ(B)
WRITE(C)
error → *abort*

Need to roll back the changes to C

Aborting a transaction

How does a database abort a transaction?

Maintain a log

- record every write to a table as (old value, new value) before it happens
- record transaction commits and transaction aborts before it happens
- every record is tagged with the transaction ID

To commit a transaction, record the commit in the log

To abort a transaction:

- record the abort in the log
- walk *back* the log for the transaction, undoing every write

Isolation: a study in interleavings

Transaction are **isolated**

- they run as though no other transaction is running at the same time
- to study this, we need to think about interleavings of actions
- the database sequences all actions
- by virtue of concurrency, speed of individual operations, network delays, these actions will occur in some order, in some interleaving (aka, a **schedule**)

Some interleavings are okay, some are not

- the database needs a way to ensure only okay interleavings happen

Example

Here's an example, not from databases, but from programming:

P_1 = BEGIN

```
        A = A + 100
        B = B - 100
    END
```

P_2 = BEGIN

```
        A = 1.5 * A
        B = 1.5 * B
    END
```

Example

Here's an example, not from databases, but from programming

P_1 = BEGIN

x ← READ(A)
WRITE(A) ← x + 100
x ← READ(B)
WRITE(B) ← x - 100

END

P_2 = BEGIN

y ← READ(A)
WRITE(A) ← 1.5 * y
y ← READ(B)
WRITE(B) ← 1.5 * y

END

Notation:

BEGIN … END
define a transaction

x ← READ(A)
read from table A into a local variable

WRITE(A) ← v
write v to table A

No interleaving

Run P_1, wait for it to be done, then run P_2 A = 200 B = 200

x ← READ(A)		
WRITE(A) ← x + 100		A = 300 B = 200
x ← READ(B)		
WRITE(B) ← x - 100		A = 300 B = 100
commit		
	y ← READ(A)	
	WRITE(A) ← 1.5 * y	A = 450 B = 100
	y ← READ(B)	
	WRITE(B) ← 1.5 * y	A = 450 B = 150
	commit	

An okay interleaving

		A = 200 B = 200
x ← READ(A)		
WRITE(A) ← x + 100		A = 300 B = 200
	y ← READ(A)	
	WRITE(A) ← 1.5 * y	A = 450 B = 200
x ← READ(B)		
WRITE(B) ← x - 100		A = 450 B = 100
commit		
	y ← READ(B)	
	WRITE(B) ← 1.5 * y	A = 450 B = 150
	commit	

A not okay interleaving

		A = 200 B = 200
x ← READ(A)		
WRITE(A) ← x + 100		A = 300 B = 200
	y ← READ(A)	
	WRITE(A) ← 1.5 * y	A = 450 B = 200
	y ← READ(B)	
	WRITE(B) ← 1.5 * y	A = 450 B = 300
	commit	
x ← READ(B)		
WRITE(B) ← x - 100		A = 450 **B = 200**
commit		

Serial and serializable schedules

A **serial schedule** is a one that does not interleave actions from two transactions

Two schedules are **equivalent** if both schedules yield the same results/effects in any database state

A schedule is **serializable** if it is equivalent to some serial schedule

Intuition: a serializable schedule has the same results/effects as running transactions one after the other (in some order) even if actions are interleaved

How does a database ensures that it only runs serializable schedules?

Non-serializable schedule: WR conflict

Reading uncommitted data (WR conflict)

	x ← READ(A)
	WRITE(A) ← x + 1
y ← READ(A)	
WRITE(B) ← 2 * y	
commit	
	WRITE(A) ← x
	commit

Effect of left transaction alone:
A = n, B = m ⇒ A = n, B = 2n

Effect of right transaction alone:
A = n, B = m ⇒ A = n, B = m

This schedule takes
A = 0, B = 0
to
A = 0, B = 2

which is not achievable by a serial schedule

Non-serializable schedule: RW conflict

Unrepeatable reads (RW conflicts)

$x \leftarrow$ READ(A)	
	WRITE(A) $\leftarrow$ 3
	commit
$y \leftarrow$ READ(A)	
WRITE(A) $\leftarrow x + y$	
commit	

Effect of left transaction alone:

$A = n \;\Rightarrow\; A = 2n$

Effect of right transaction alone:

$A = n \;\Rightarrow\; A = 3$

This schedule takes

$A = 1$

to

$A = 4$

which is not achievable by a serial schedule

Ensuring serializability

How does a database ensures that it only runs serializable schedules?

- **two-phase locking protocol**

Every table has two associated locks:

- a shared read lock (S)
- an exclusive write lock (X)

A transaction must wait to acquire an S_A lock on table A before reading from it

A transaction must wait to acquire an X_A lock on table A before writing to it

Transactions release their locks when they commit (or abort).

Properties

if a transaction holds an X lock on a table:

- no other transaction can get a lock on the table
- i.e., no other transaction can read or write to it

if a transaction holds an S lock on a table:

- no other transaction can get an X lock on the table
- i.e., no other transaction can write to it
- but another transaction can get an S lock on the table to read from it

Two-phase locking protocol

Two phases:

1. **Grow** phase
   a transaction acquires locks and does not release any lock

2. **Contract** phase
   a transaction releases all locks and does not acquire any

Many variants:

- can acquire all locks needed at once — less concurrency, but no deadlock
- acquire only when needed — more concurrency, but deadlock

Deadlock

Transactions that are blocked indefinitely, each waiting for the other to release a lock they need

P_1: acquire X_A
write(A)
acquire X_B
write(B)

P_2: acquire X_B
write(B)
acquire X_A
write(A)

Possible interleaving:

X_A		P_1 holds X_A
WRITE(A)		
	X_B	P_2 holds X_B
	WRITE(B)	
X_B		P_1 waiting for X_B
	X_A	P_2 waiting for X_A

Deadlock detection

Check for deadlocks at regular intervals

When you detect a deadlock:

- pick a transaction to abort
- iterate until no more deadlock

Deadlock prevention

Instead of detecting deadlocks, prevent them

- use timestamp on transactions to order them
- still leads to aborted transactions

Two approaches — wait-die and wound-wait

Suppose T_2 needs a lock that T_1 holds

wait-die: if $T_2 > T_1$ then T_2 waits for T_1 to release the lock
if $T_2 < T_1$ then T_2 aborts

wound-wait: if $T_2 > T_1$ then T_1 aborts
if $T_2 < T_1$ then T_2 waits for T_1 to release the lock

Durability

Transactions are **durable** — if they commit, they persist (and only then).

What happens if a database crashes before the database commits?

- because *databases do crash*, and it should not be catastrophic

For atomicity, we maintain a log of all transaction actions

- to be able to unde them in case of transaction abort

We can use a **write-ahead log** to also handle recovery after a database crash

- write-ahead log = write to the log *before* making the changes to blocks

Recovery

ARIES algorithm

1. When the database starts, scan the log for the most recent checkpoint
    - checkpoint = when the blocks content = log content, and no ongoing transaction
    - can also clear the log at checkpoint

2. Redo all the writes in the log, in order
    - this restores the state of the database to what it was when it crashed

3. Abort all transactions that are not committed

Challenge: database crashing during recovery!

That's all, folks!