transactions

Remy Wang

April 2026

So far we've been treating DBs as inanimate objects: they hold data, we can look at the data, and that's it. But SQL databases are so popular because they deal with *changes* very well. Let's see some examples.

I'm in a good mood today and decided to give everyone free points! I vibed a quick python script to do this:

UID	score
1	39
2	24
3	40
4	35
...	...

I'm in a good mood today and decided to give everyone free points! I vibed a quick python script to do this:

UID	score
1	39
2	24
3	40
4	35
...	...

Uh oh! My computer crashed halfway through, and I don't know who still haven't got the free points!

You don't like that, because some of you got the points and some didn't. You would be fine if either none of you get the points or all of you did.

Let's try another experiment. This time, your fate is in your own hands: steal points from your enemies! Subtract some points from someone and add it to your own. Here's the Google sheet, go!

Well, that was a mess. The total score no longer add up, meaning our DB is not *consistent*!

The problem is that you were *interfering* with each other!

Well, that was a mess. The total score no longer add up, meaning our DB is not *consistent*!

The problem is that you were *interfering* with each other!

One solution is to go *one at a time*, so that every change is made in *isolation*. In other words, we *serialize* the changes.

Another solution is we first claim a *lock* on a piece of data before changing it. This time, wait a few seconds before editing a cell, and only edit it if no one else is trying to edit.

Formally, a group of actions on a DB is called a *transaction*. DBs like SQLite guarantee transactions (tx) are:

- **Atomic**: either all or none of the actions are applied (all or nothing)

Formally, a group of actions on a DB is called a *transaction*. DBs like SQLite guarantee transactions (tx) are:

- **Atomic**: either all or none of the actions are applied (all or nothing)
- **Consistent**: after a tx finishes, the DB is in a consistent state

Formally, a group of actions on a DB is called a *transaction*. DBs like SQLite guarantee transactions (tx) are:

- **Atomic**: either all or none of the actions are applied (all or nothing)
- **Consistent**: after a tx finishes, the DB is in a consistent state
- **Isolated**: txs do not interfere with each other

Formally, a group of actions on a DB is called a *transaction*. DBs like SQLite guarantee transactions (tx) are:

- **Atomic**: either all or none of the actions are applied (all or nothing)
- **Consistent**: after a tx finishes, the DB is in a consistent state
- **Isolated**: txs do not interfere with each other
- **Durable**: once a tx finishes, its effect is kept forever

Formally, a group of actions on a DB is called a *transaction*. DBs like SQLite guarantee transactions (tx) are:

- **Atomic**: either all or none of the actions are applied (all or nothing)
- **Consistent**: after a tx finishes, the DB is in a consistent state
- **Isolated**: txs do not interfere with each other
- **Durable**: once a tx finishes, its effect is kept forever

Formally, a group of actions on a DB is called a *transaction*. DBs like SQLite guarantee transactions (tx) are:

- **Atomic**: either all or none of the actions are applied (all or nothing)
- **Consistent**: after a tx finishes, the DB is in a consistent state
- **Isolated**: txs do not interfere with each other
- **Durable**: once a tx finishes, its effect is kept forever

This is known as *ACID* on the street.

Note that *consistent* is a high-level property, usually achieved with the other 3.

- When only some of you got the extra credit, our tx was not *atomic*
- When you were stealing points and had conflicts, you were not *isolated*, leaving the DB *inconsistent*
- We haven't see *durable* yet, but if the computer dies before writing data to disk, the data would be lost

- When only some of you got the extra credit, our tx was not *atomic*
- When you were stealing points and had conflicts, you were not *isolated*, leaving the DB *inconsistent*
- We haven't see *durable* yet, but if the computer dies before writing data to disk, the data would be lost

Luckily, the DB handles all these for us!

But it's still important to understand what's going on, in case anything goes sideways.

There are several different ways to ensure *atomicity*. The main idea is to a tx to be explicitly `COMMIT`ted:

```
BEGIN TRANSACTION;

SELECT ...;
INSERT ...;

COMMIT;
```

You can think of each tx as making a local copy of the DB, and any operation in that tx only acts upon the local copy, until the tx is committed which writes the DB to the global one.

Try running the following concurrently in 2 SQLite sessions[1]:

```
--      SESSION 1                       SESSION 2

create table r (x int);
insert into r values (1);
                                        select * from r;
begin transaction;
insert into r values (2);
                                        select * from r;
commit;
                                        select * from r;
```

You'll see the change made within the tx does not take effect until after `COMMIT`.

[1] Run `sqlite3 test.db` in 2 different terminal tabs.

The easiest way to guarantee *isolation* is to *serialize* the transactions – running them one at a time. This is pretty much what SQLite does![2] So why don't we just leave it at that?

[2] SQLite serializes writes, but allow concurrent reads.

The easiest way to guarantee *isolation* is to *serialize* the transactions – running them one at a time. This is pretty much what SQLite does![2] So why don't we just leave it at that?

Because performance. When we serialized our score updates, we all had to wait in line, even though we could have used some parallelism.

[2] SQLite serializes writes, but allow concurrent reads.

So on one hand, serialization guarantees consistency but is slow

On the other hand, concurrency is fast but is inconsistent

The key idea of TX processing is to preserve consistency while improving concurrency, which is usually done by giving the *illusion* of serial execution while running concurrently. Magic!

First off, if 2 TX touch different data, they can of course run concurrently:

T_1	T_2
R(A)	R(B)
W(A)	W(B)

Here R(A) means reading a DB item A, and W(A) writes to A. You can think of a DB item as a row, but it depends on the context.

The table here shows the *schedule*[3] of the txs which is a record of actions taken over time.

[3] Formally, a schedule is a *partial ordering* of the actions.

We say a schedule is *serial* if all actions are strictly ordered:

T_1	T_2
R(A)	
W(A)	
	R(B)
	W(B)

In our example, the concurrent schedule happens to be *equivalent* to the serial one, because the TX involve independent items:

T_1	T_2
R(A)	R(B)
W(A)	W(B)

T_1	T_2
R(A)	
W(A)	
	R(B)
	W(B)

Formally, two TX are *equivalent* if they produce the same result no matter what is read or written. . . . And we say the concurrent schedule is *serializable*, i.e., it is equivalent to some serial schedule.

The main idea of DBs is that we want to allow **serializable**, yet concurrent **transaction schedules** like the above.[4]

[4] Make sure you know what the bold words mean by now!

Note that two serial schedules are not necessarily equivalent:

T_1	T_2
A = 2*A	
	A = 2+A

T_1	T_2
	A = 2+A
A = 2*A	

If we start with A = 1, the first schedule would result in A = 4, while the second in A = 6.

Note that two serial schedules are not necessarily equivalent:

T_1	T_2
A = 2*A	
	A = 2+A

T_1	T_2
	A = 2+A
A = 2*A	

If we start with A = 1, the first schedule would result in A = 4, while the second in A = 6.

For a schedule to be *serializable*, it only needs to be equivalent to *some* serial one, but we don't know which one.

There's also the notion of *strict serializability*, which basically says "first come, first serve":

- A schedule is strict-serializable if it's equivalent to a serial schedule that runs the TXs in the same order they arrive.

This can be important for e.g. ticket sales.

To check if a schedule is serializable, we first need to know how to check if 2 schedules are equivalent.

To do that, we check if we can *reorder* one into another, while avoiding *conflicts*:

T_1	T_2
R(A)	R(B)
W(A)	W(B)

T_1	T_2
R(A)	
	R(B)
W(A)	
	W(B)

T_1	T_2
R(A)	
W(A)	
	R(B)
	W(B)

1. Because reading data never causes conflict, we are free to reorder R(A) and R(B) any way we like.

To check if a schedule is serializable, we first need to know how to check if 2 schedules are equivalent.

To do that, we check if we can *reorder* one into another, while avoiding *conflicts*:

T_1	T_2
R(A)	R(B)
W(A)	W(B)

T_1	T_2
R(A)	
	R(B)
W(A)	
	W(B)

T_1	T_2
R(A)	
W(A)	
	R(B)
	W(B)

1. Because reading data never causes conflict, we are free to reorder R(A) and R(B) any way we like.
2. Since writing to different items do not conflict, we can also reorder W(A) and W(B).

To check if a schedule is serializable, we first need to know how to check if 2 schedules are equivalent.

To do that, we check if we can *reorder* one into another, while avoiding *conflicts*:

T_1	T_2
R(A)	R(B)
W(A)	W(B)

T_1	T_2
R(A)	
	R(B)
W(A)	
	W(B)

T_1	T_2
R(A)	
W(A)	
	R(B)
	W(B)

1. Because reading data never causes conflict, we are free to reorder R(A) and R(B) any way we like.
2. Since writing to different items do not conflict, we can also reorder W(A) and W(B).
3. There's also no conflict between W(A) and R(B), so we swap one more time and the schedule is serial.

In general, two actions are in conflict if they:

- operate on the same DB item
- one of them is a write

For example, the following schedules are not equivalent, because they cannot be reordered into each other:

T_1	T_2
R(A)	
	R(A)
W(A)	
	W(A)

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)

Indeed, the first schedule is not *serializable*, and the two TXs are attempting to change the same item at the same time.

For a concrete example:

T_1	T_2
x=R(A)	
y=2x	u=R(A)
W(A)=y	v=2u
	W(A)=v

If the TXs run in sequence, we would multiply A by 4, but the schedule above would only multiply by 2.

In other words, *no serial schedule* would have produced the same result.

This gives us an algorithm to check for serializability:

- First compute all possible serial schedules involving the TXs
- For each serial schedule, check if it can be reordered into the given one

But this is hopelessly slow: there are exponentially many serial schedules, and each can have exponentially many reorderings.

We can check serializability with a powerful tool called the *precedence graph*. Consider the following schedule:

T_1	T_2	T_3
R(B)	R(A)	
	W(A)	
W(B)		R(A)
	R(B)	
	W(B)	

The graph has a node per transaction, and an edge $T_i \rightarrow T_j$ if there's a pair of actions $a_i \in T_i$ and $a_j \in T_j$ s.t.:

- a_i occurs no later than a_j
- they conflict with each other

$$T_1 \rightarrow T_2 \rightarrow T_3$$

Let's also check the schedule that we know is not serializable:

T_1	T_2
x=R(A)	
y=2x	u=R(A)
W(A)=y	v=2u
	W(A)=v

Let's also check the schedule that we know is not serializable:

T_1	T_2
x=R(A)	
y=2x	u=R(A)
W(A)=y	v=2u
	W(A)=v

Its precedence graph has a cycle:

$$T_1 \leftrightarrows T_2$$

A schedule is (conflict-)serializable iff its precedence graph has no cycle

A schedule is (conflict-)serializable iff its precedence graph has no cycle

Intuitively, an edge $T_i \to T_j$ means there's a pair actions a_i, a_j that cannot be reordered due to a conflict, so a_i must occur first in any reordering, meaning some part of T_i must happen before T_j.

But if there's another edge $T_j \to T_i$, some part of T_j would also happen before T_i, which is impossible in a serial schedule.

That was a lot of new concepts to learn! Let's review:

- We want to guarantee ACID when the DB is modified

That was a lot of new concepts to learn! Let's review:

- We want to guarantee ACID when the DB is modified
- So we group actions into TXs as units

That was a lot of new concepts to learn! Let's review:

- We want to guarantee ACID when the DB is modified
- So we group actions into TXs as units
- Serial execution would lead to ACID, but slow

That was a lot of new concepts to learn! Let's review:

- We want to guarantee ACID when the DB is modified
- So we group actions into TXs as units
- Serial execution would lead to ACID, but slow
- We want serializable concurrent schedules

That was a lot of new concepts to learn! Let's review:

- We want to guarantee ACID when the DB is modified
- So we group actions into TXs as units
- Serial execution would lead to ACID, but slow
- We want serializable concurrent schedules
- Which are schedules equivalent to some serial ones

That was a lot of new concepts to learn! Let's review:

- We want to guarantee ACID when the DB is modified
- So we group actions into TXs as units
- Serial execution would lead to ACID, but slow
- We want serializable concurrent schedules
- Which are schedules equivalent to some serial ones
- Which we can check with the precedence graph