CS 6400 A

Database Systems Concepts and Design

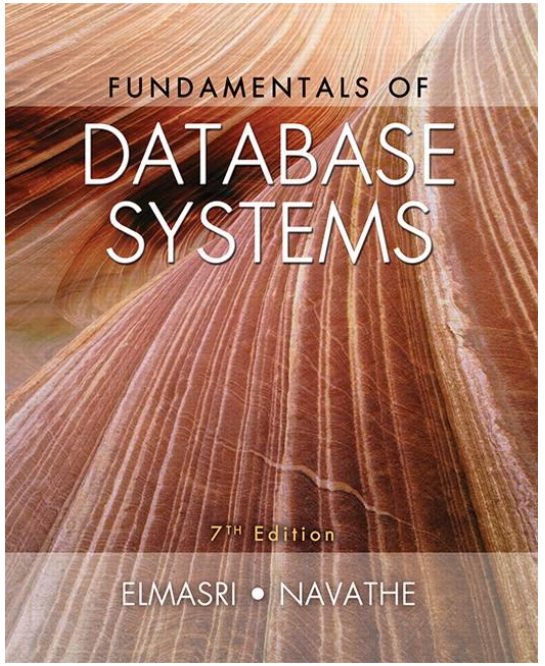
Lecture 17

10/28/26

Reading Materials

Fundamental of Database Systems (7th Edition)

- Chapter 20 - Introduction to Transaction Processing Concepts and Theory



Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

Agenda

1. Transaction Basics
2. ACID properties
3. Using transactions in SQL

1. Transaction Basics

Transactions: Basic Definition

A transaction ("TXN") is a sequence of one or more operations (reads or writes) which reflects a **single real-world transition**.

In the real world, a TXN either happened completely or not at all

```
START TRANSACTION
        UPDATE Product
        SET Price = Price – 1.99
        WHERE pname = 'Gizmo'
COMMIT
```

Transactions: Basic Definition

A transaction ("TXN") is a sequence of one or more operations (reads or writes) which reflects a **single real-world transition**.

In the real world, a TXN either happened completely or not at all

Examples:

- Transfer money between accounts
- Register for a class (either waitlist or allocated)
- Purchase a group of products

Transactions in SQL

In “ad-hoc” SQL:

- Default: each statement = one transaction
- No need to explicitly start or end a transaction.

In a program, multiple statements can be grouped together as a transaction:

```
START TRANSACTION
        UPDATE Bank SET amount = amount – 100
        WHERE name = ‘Bob’
        UPDATE Bank SET amount = amount + 100
        WHERE name = ‘Joe’
COMMIT
```

Model of Transaction in this class

We assume that the DBMS is only concerned about reads and writes to data

- It doesn't care about what the user's program does with the data **outside the database**.

A transaction is the DBMS's abstract view of a user program

- The same program executed multiple times would be considered as different transactions
- The DBMS does not really understand the "semantics" of the data, it only cares about read and write sequences

Motivation for Transactions

Grouping user actions (reads & writes) into *transactions* helps with two goals:

1. Recovery & Durability: Keeping the DBMS data consistent and durable in the face of crashes, aborts, system shutdowns, etc.

2. Concurrency: Achieving better performance by parallelizing TXNs *without* creating anomalies

Motivation

1. Recovery & Durability of user data is essential for reliable DBMS usage

- The DBMS may experience crashes (e.g. power outages, etc.)
- Individual TXNs may be aborted (e.g. by the user)

Idea: Make sure that TXNs are either **durably stored in full**, **or not at all**; keep log to be able to "roll-back" TXNs

Protection against crashes / aborts

Client 1:

```
INSERT INTO SmallProduct(name, price)
        SELECT pname, price
        FROM Product
        WHERE price <= 0.99
```

Crash / abort!

```
DELETE Product
        WHERE price <=0.99
```

What goes wrong?

Protection against crashes / aborts

```
Client 1:
        START TRANSACTION
                INSERT INTO SmallProduct(name, price)
                        SELECT pname, price
                        FROM Product
                        WHERE price <= 0.99

                DELETE Product
                        WHERE price <=0.99
        COMMIT OR ROLLBACK
```

Now we'd be fine!

Motivation

2. Concurrent execution of user programs is essential for good DBMS performance.

- Disk accesses may be frequent and **slow**- optimize for throughput (# of TXNs), trade for latency (time for any one TXN)

- Users should still be able to execute TXNs as if in **isolation** and such that **consistency** is maintained

Idea: Have the DBMS handle running several user TXNs concurrently, in order to keep CPUs buzy…

Multiple users: single statements

```
Client 1: UPDATE Product
                SET Price = Price – 1.99
                WHERE pname = ‘Gizmo’

Client 2:       UPDATE Product
                SET Price = Price*0.5
                WHERE pname=‘Gizmo’
```

Two managers attempt to discount products *concurrently*-
What could go wrong?

Multiple users: single statements

```
Client 1: START TRANSACTION
                    UPDATE Product
                    SET Price = Price – 1.99
                    WHERE pname = ‘Gizmo’
          COMMIT

Client 2: START TRANSACTION
                    UPDATE Product
                    SET Price = Price*0.5
                    WHERE pname=‘Gizmo’
          COMMIT
```

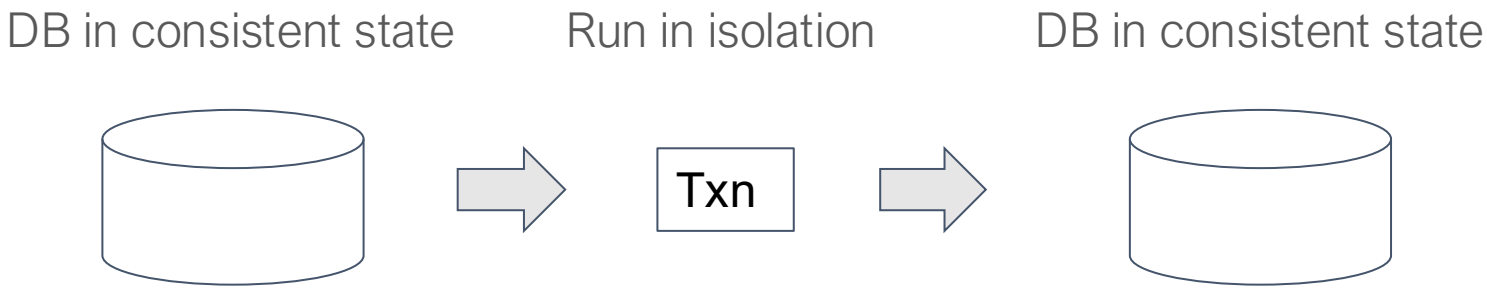
Now works like a charm - we’ll see how / why in the following lectures…

2. ACID Properties

The Correctness Principle

What does it mean for a transaction to be correct?

A correct transaction takes the database from one valid state to another, assuming it executes alone and without failures.



In practice, transactions may fail, execute concurrently, or encounter system crashes.

How can the DBMS preserve correctness under these conditions?

ACID Properties

Atomicity: What if a transaction fails halfway through?

Consistency: Does the transaction preserve the database's rules?

Isolation: What if multiple transactions run concurrently?

Durability: What if the system crashes after commit?

ACID: Atomicity

TXN's activities are atomic: **all or nothing**

- Intuitively: in the real world, a transaction is something that would either occur *completely* or *not at all*

Two possible outcomes for a TXN

- It *commits*: all the changes are made
- It *aborts*: no changes are made

ACID: Isolation

A transaction executes concurrently with other transactions

Isolation: Concurrent transactions should behave as if they executed one at a time, in some order.

- Transactions should not interfere with each other in ways that produce incorrect results.
- The DBMS ensures the outcome is equivalent to a valid serial execution order.

ACID: Consistency

Consistency: A transaction takes the database from one valid state to another valid state.

A valid state satisfies all defined integrity constraints. Examples:

- Account numbers must be unique.
- Stock quantities cannot be negative.
- Total debits must equal total credits.

How consistency is achieved:

- Programmer: Writes transactions that preserve application rules.
- DBMS: Enforces declared constraints; atomicity and isolation help preserve consistency.

ACID: Durability

The effect of a TXN must continue to exist (***"persist"***) after the TXN

- And after the whole program has terminated
- And even if there are power failures, crashes, etc.
- And etc…

- Means: Write data to **disk**

Change on the horizon?
Non-Volatile Ram (NVRam).
Byte addressable.

In-class Exercise

- Scenario: Two customers simultaneously try to book the last seat on a flight. Both see 'seat available', but only one successfully completes the booking. The other gets an error message saying the seat is no longer available.

- Q: Which ACID property prevented double-booking?

In-class Exercise

- Scenario: You place an order and receive confirmation. Five minutes later, the data center experiences a power outage. When the system comes back online, your order is still in the system.

- Q: Which ACID property ensures your order wasn't lost?

In-class Exercise

- Scenario: You're transferring $500 from your checking account to savings. The system deducts $500 from checking, but crashes before adding it to savings. When the system restarts, the $500 has been restored to your checking account.

- Q: Which ACID property prevented you from losing $500?

In-class Exercise

- Scenario: An e-commerce system sells electronics. After processing an order, the inventory count is updated from 10 to 9 items. The database also maintains a rule that inventory can never be negative, and total_items_sold must equal (starting_inventory - current_inventory). The transaction successfully commits.

- Q: Which ACID property ensures all these business rules remain satisfied?

Ensuring Consistency

User's responsibility to maintain the integrity constraints, as the DBMS may not be able to catch such errors in user program's logic

```
START TRANSACTION
UPDATE accounts
SET balance = balance - 100 WHERE id =
1;
COMMIT;
```

Database ends up inconsistent (money disappeared)

However, the DBMS may be in inconsistent state "during a transaction" between actions

- which is ok, but it should leave the database at a consistent state when it commits or aborts

Ensuring Atomicity

A transaction interrupted in the middle can leave the database in an inconsistent state

- DBMS has to remove the effects of partial transactions from the database

DBMS ensures atomicity by “undoing” the actions of incomplete transactions

DBMS maintains a “log” of all changes to do so

Ensuring Durability

The log also ensures durability

If the system crashes before the changes made by a completed transactions are written to the disk, the log is used to remember and restore these changes when the system restarts

"recovery manager"

- takes care of atomicity and durability

Ensuring Isolation

DBMS guarantees isolation

- If T1 and T2 are executed **concurrently**, either the effect would be T1->T2 or T2->T1 (as if they ran **serially**)

DBMS provides no guarantee on which of these order is chosen, just that the result is equivalent to *some* serial order.

Often ensured by "locks" but there are other methods too

Desirable Properties of Transactions: ACID

Atomicity: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.

Consistency: A correct execution of the transaction must take the database from one consistent state to another.

Isolation: A transaction should not make its updates visible to other transactions until it is committed.

Durability: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

A Note: ACID is contentious!

Many debates over ACID, both **historically** and **currently**

- **Traditional relational DBMSs**: Strong transactional guarantees are a central design goal.
- **NoSQL (2000s)**: Some systems relaxed transaction guarantees to prioritize scalability and availability.
- **NewSQL (2010s onward)**: Systems aim to combine distributed scalability with ACID transactions.





ACID is an extremely important & successful paradigm, but still debated!

3. Using Transactions in SQL

Using Transactions in SQL

- SQL allows the programmer to group several statements in a single *transaction*
- Either all operations are performed or none are
- A single SQL statement is always considered to be atomic.

Marks beginning of transaction

```
START TRANSACTION

UPDATE Accounts
SET balance = balance + 100
WHERE acctNo = 456;

UPDATE Accounts
SET balance = balance - 100
WHERE acctNo = 123;

COMMIT;
```

Causes transaction to end successfully

Using Transactions in SQL

- ROLLBACK causes the transaction to abort and undo any changes

We find that there are insufficient funds to make transfer

```
START TRANSACTION

UPDATE Accounts
SET balance = balance + 100
WHERE acctNo = 456;




ROLLBACK;
```

Using Transactions in SQL

SET TRANSACTION ***transaction_mode*** [, ...]

where ***transaction_mode*** is one of:

- ISOLATION LEVEL {
  SERIALIZABLE
  | REPEATABLE READ
  | READ COMMITTED
  | READ UNCOMMITTED }
- READ WRITE | READ ONLY

Isolation Levels

- With SERIALIZABLE: concurrent transactions behave as if they executed in some serial order..
- Weaker isolation levels: Allow more concurrency but may allow executions that are not serializable.

Access Mode

- The default is READ WRITE unless the isolation level of READ UNCOMITTED is specified, in which case READ ONLY is assumed.

Source: https://www.postgresql.org/docs/current/sql-set-transaction.html

Read-only transactions

Transactions that only read data and do not write can be executed in parallel

Tell DBMS before running transaction:

```
SET TRANSACTION READ ONLY;
```

Why Do We Need Different Isolation Levels?

Concurrent transactions can interfere with each other, leading to unexpected results called concurrency anomalies.

Examples:

- Dirty read
- Nonrepeatable read
- Phantom read

These are three classic anomalies used to distinguish SQL isolation levels.

Different isolation levels prevent different anomalies.

Dirty reads

Reading data written by a transaction that has not yet committed

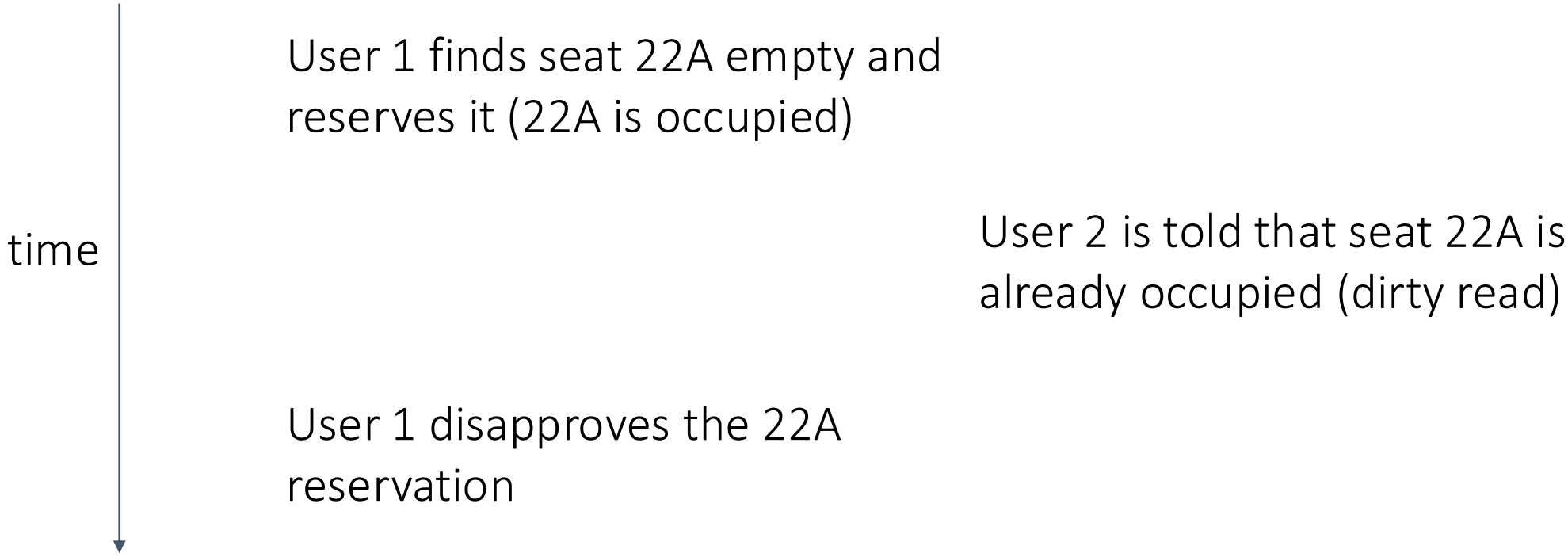
Consider this seat selection example:

1. Find available seat and reserve by setting *seatStatus* to ‘occupied’

2. Ask customer for approval of seat
   a. If so, commit
   b. If not, release seat by setting seatStatus to ‘available’ and repeat Step (1)

Dirty read

If we allow dirty reads, this can happen



Dirty reads

If this result is acceptable, the transaction processing can be done faster

- DBMS does not have to prevent dirty reads
- Allows more parallelism

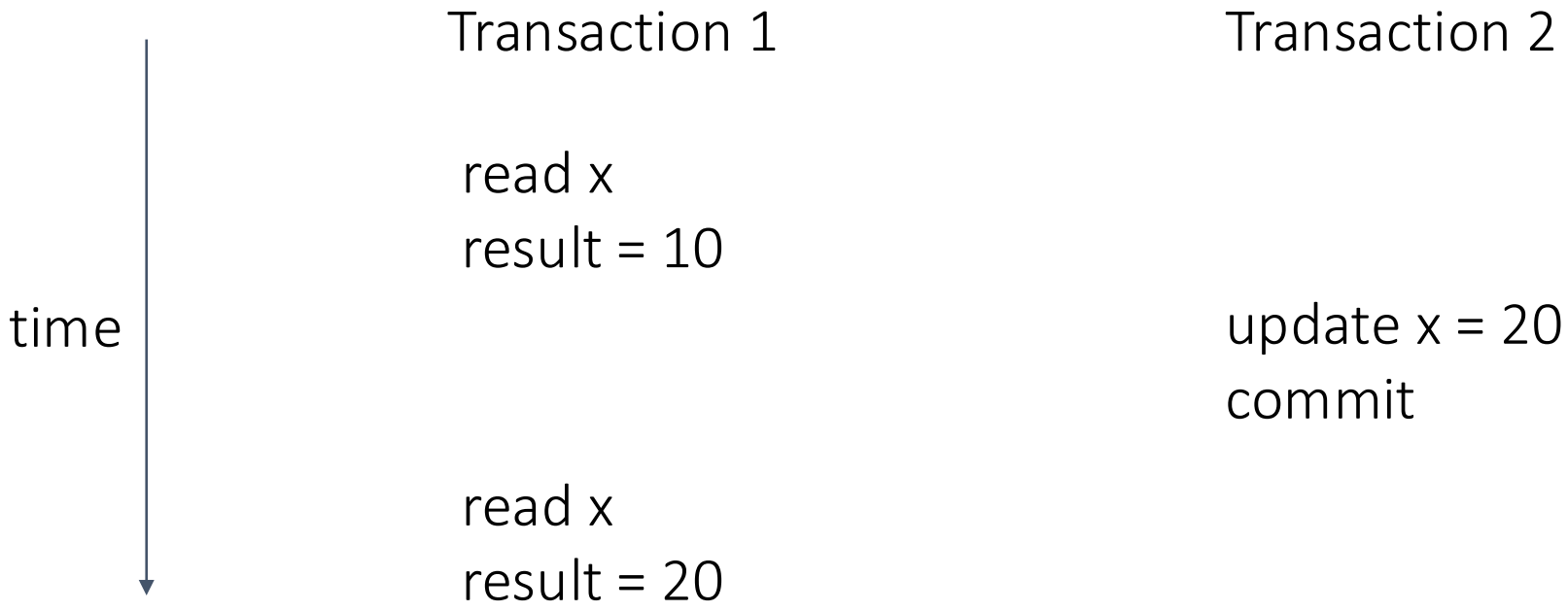
Tell DBMS before running transaction:

```
SET TRANSACTION READ WRITE
   ISOLATION LEVEL READ UNCOMMITTED;
```

Read committed

Only allow reads from committed data, but same query may get different answers

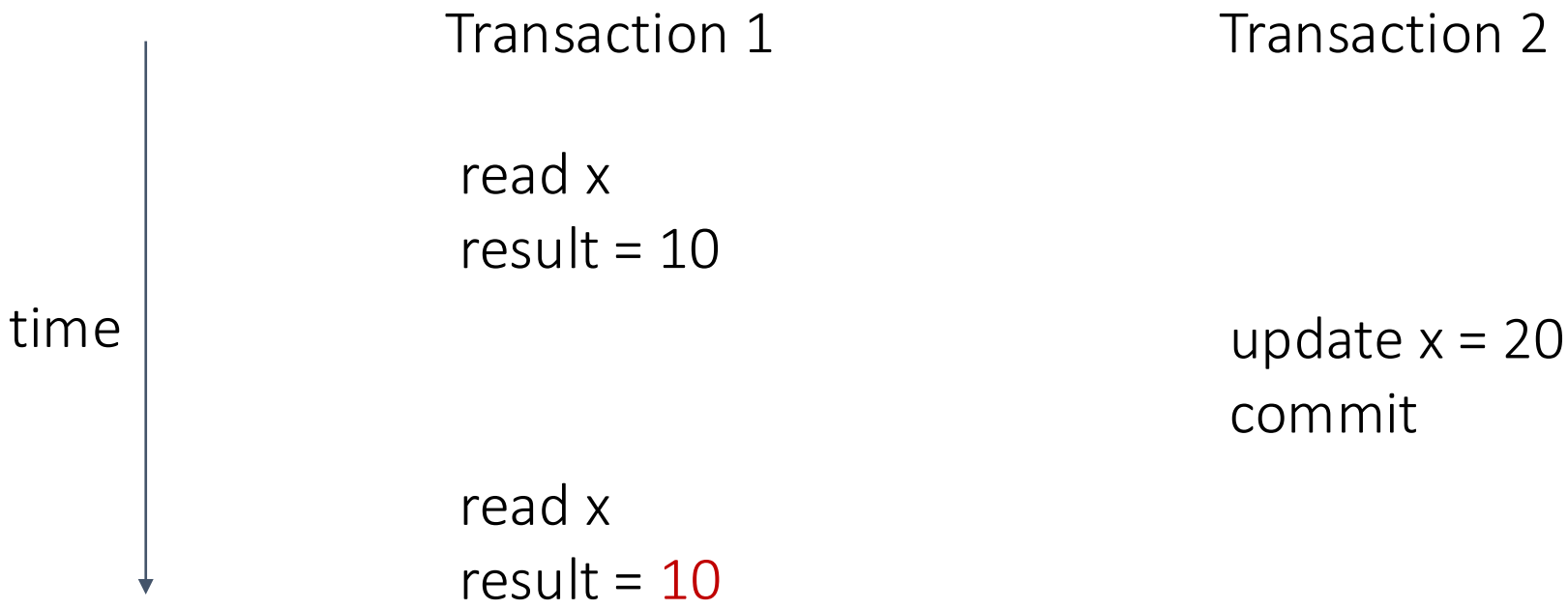
```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```



Repeatable read

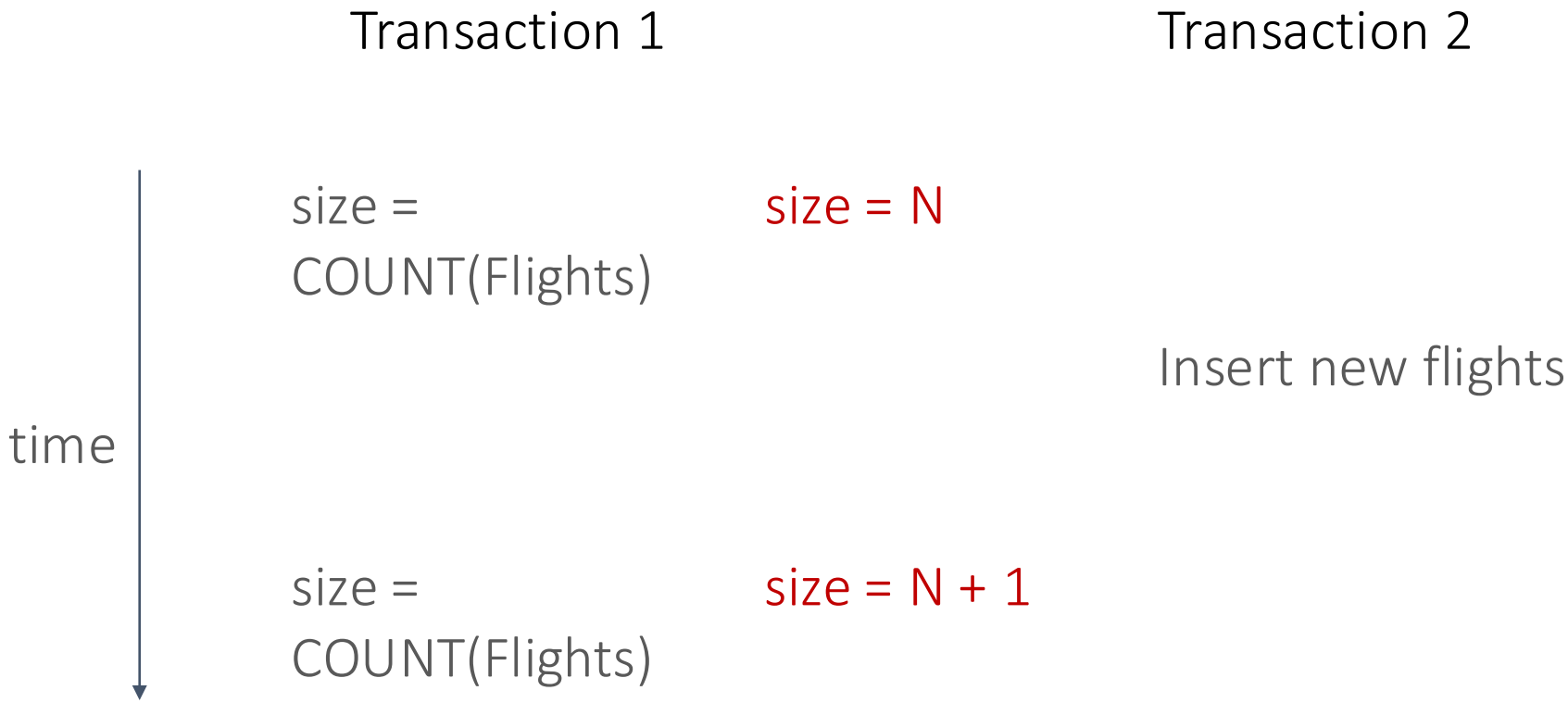
If a transaction reads a row, reading it again will return the same values, even if other transactions update that row.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```



Repeatable read

May allow “phantom” tuples, which are new tuples inserted between queries



Repeatable Read

Guarantee:

- Rows already read by a transaction will have the same values if read again.
- However, repeating a query may return new matching rows (phantom reads).

Why can phantom reads occur? (Lock-based implementation)

- Row locks: Prevent other transactions from modifying rows already read.
- No gap/range locks: Other transactions can insert new rows that satisfy the query's WHERE condition.

Comparison of SQL isolation levels

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
READ UNCOMMITTED	✓	✓	✓
READ COMMITTED	⊘	✓	✓
REPEATABLE READ	⊘	⊘	✓
SERIALIZABLE	⊘	⊘	⊘

Comparison of SQL isolation levels

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
READ UNCOMMITTED	✓	✓	✓
READ COMMITTED	⊘	✓	✓
REPEATABLE READ	⊘	⊘	✓
SERIALIZABLE	⊘	⊘	⊘

- Rarely used in practice, as the performance is not much better than other levels and data read may later be rolled back.
- In fact, PostgreSQL doesn't support this isolation level

Comparison of SQL isolation levels

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
READ UNCOMMITTED	✓	✓	✓
READ COMMITTED	⊘	✓	✓
REPEATABLE READ	⊘	⊘	✓
SERIALIZABLE	⊘	⊘	⊘

- Prevents dirty reads: transactions see only committed data.
- Common default isolation level.

Comparison of SQL isolation levels

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
READ UNCOMMITTED	✓	✓	✓
READ COMMITTED	⊘	✓	✓
REPEATABLE READ	⊘	⊘	✓
SERIALIZABLE	⊘	⊘	⊘

- Useful when a transaction needs a stable view of previously read data (e.g., reporting, data warehousing types of workload)

Comparison of SQL isolation levels

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
READ UNCOMMITTED	✓	✓	✓
READ COMMITTED	⊘	✓	✓
REPEATABLE READ	⊘	⊘	✓
SERIALIZABLE	⊘	⊘	⊘

- **Strongest standard isolation level**: Concurrent transactions behave as if executed one at a time, in some order.
- Some transactions, especially those involving complex logic or multiple reads and writes, require serializable isolation to ensure correctness
- Additional coordination may reduce concurrency and cause transactions to wait or retry.