CS 4440 A

Emerging Database Technologies

Lecture 5

01/28/26

Overview of this section

- **Transactions** are a programming abstraction that enables the DBMS to handle *recovery* and *concurrency* for users.

- **Application:** Transactions are critical for users
  - Even casual users of data processing systems!

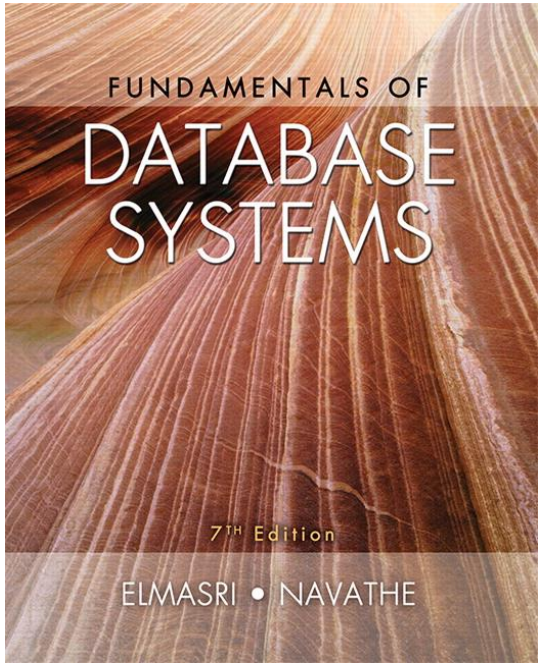
- **Fundamentals:** The basics of **how** TXNs work
  - Transaction processing is part of the debate around new data processing systems

  - Give you enough information to understand how TXNs work, and the main concerns with using them

Reading Materials

Fundamental of Database Systems (7th Edition)

- Chapter 20 - Introduction to Transaction Processing Concepts and Theory



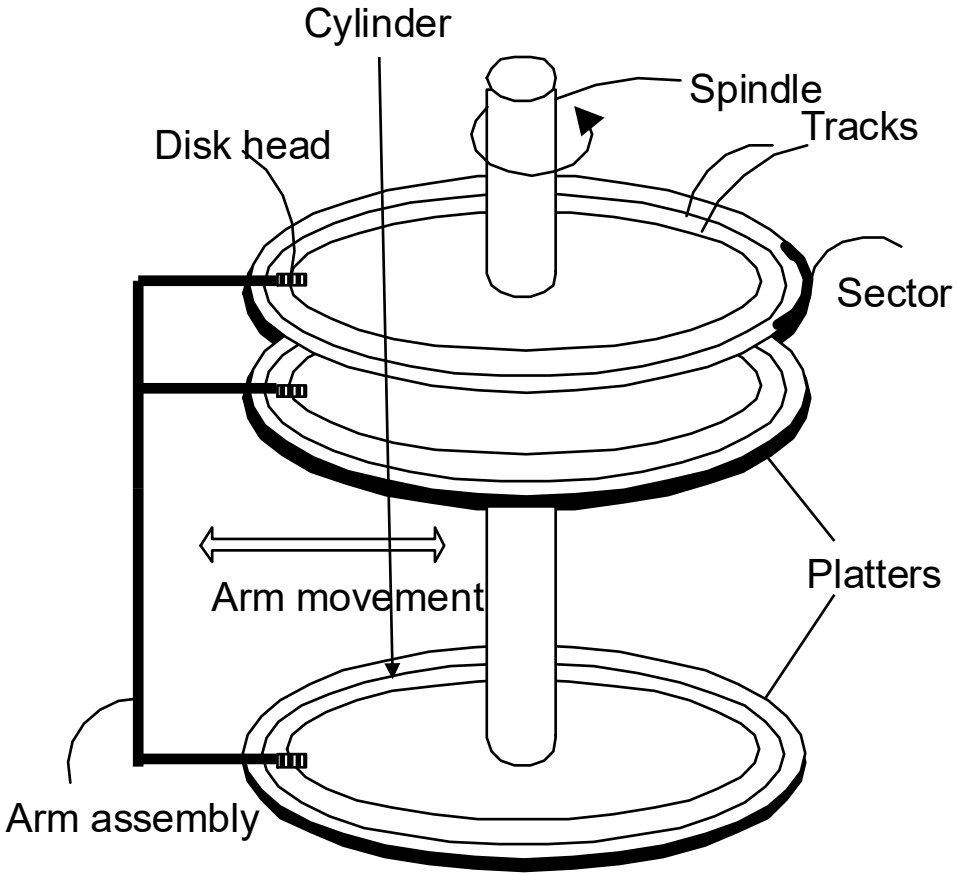
Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

Agenda

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

High-level: Disk vs. Main Memory

- **Disk**:
  - *Slow*
    - Sequential access
      - (although fast sequential reads)
  - *Durable*
    - We will assume that once on disk, data is safe!
  - Cheap



High-level: Disk vs. Main Memory

- Random Access Memory (RAM) or **Main Memory**:
  - *Fast*
    - Random access, byte addressable
      - ~10x faster for sequential access
      - ~100,000x faster for random access!
  - *Volatile*
    - Data can be lost if e.g. crash occurs, power goes out, etc!
  - Expensive
    - For $100, get 16GB of RAM vs. 2TB of disk!

High-level: Disk vs. Main Memory

- Keep in mind the tradeoffs here as motivation for the mechanisms we introduce
  - Main memory: fast but limited capacity, volatile
  - Vs. Disk: slow but large capacity, durable

How do we effectively utilize ***both*** ensuring certain critical guarantees?

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
- Purchase a group of products
- Register for a class (either waitlist or allocated)

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

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.

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: Consistency

The tables must always satisfy user-specified ***integrity constraints***

- *Examples:*
  - Account number is unique
  - Stock amount can't be negative
  - Sum of *debits* and of *credits* is 0

How consistency is achieved:

- Programmer makes sure a txn takes a consistent state to a consistent state
- *System* makes sure that the txn is **atomic**

ACID: Isolation

A transaction executes concurrently with other transactions

Isolation: the effect is as if each transaction executes in *isolation* of the others.

- A user should be able to observe changes from other transactions during the run

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: 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: 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'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)

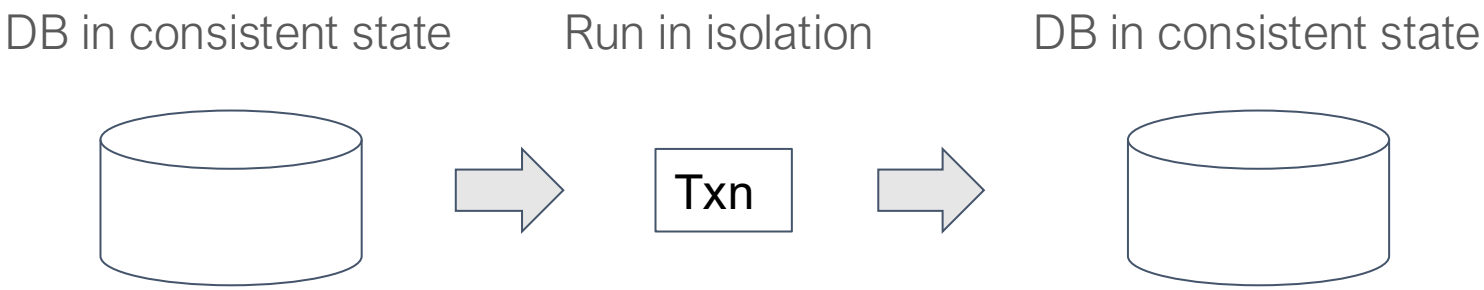
However, the DBMS may be in inconsistent state "during a transaction" between actions

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

The Correctness Principle

A fundamental assumption about transaction is:

If a transaction executes in the absence of any other transactions or system errors, and it starts with the database in a consistent state, then the database is also in a consistent state when the transactions ends.



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

A Note: ACID is contentious!

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

Many "NoSQL" DBMSs relax ACID

In turn, now "NewSQL" reintroduces ACID compliance to NoSQL-style DBMSs…

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: the interleaved execution of transactions will adhere to our notion of serializability.
- However, if any transaction executes at a lower level, then serializability may be violated.

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;
```

Dirty reads

Reading data written by a transaction that has not yet committed

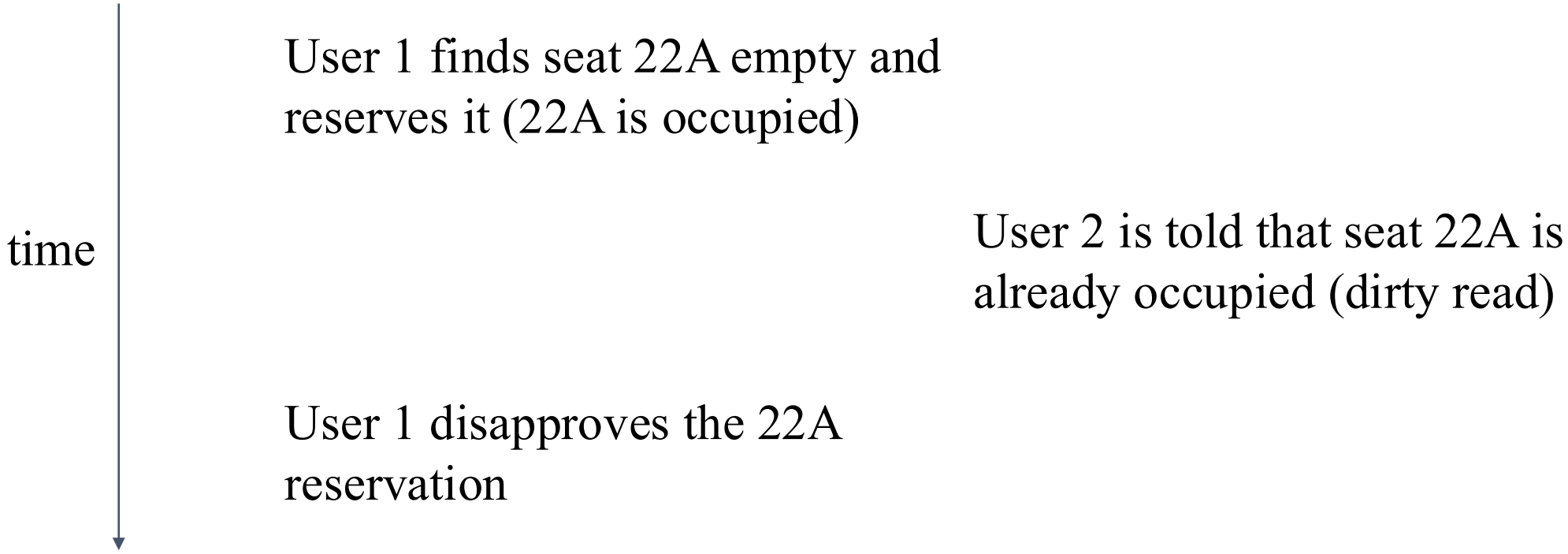
Consider this seat selection example:

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

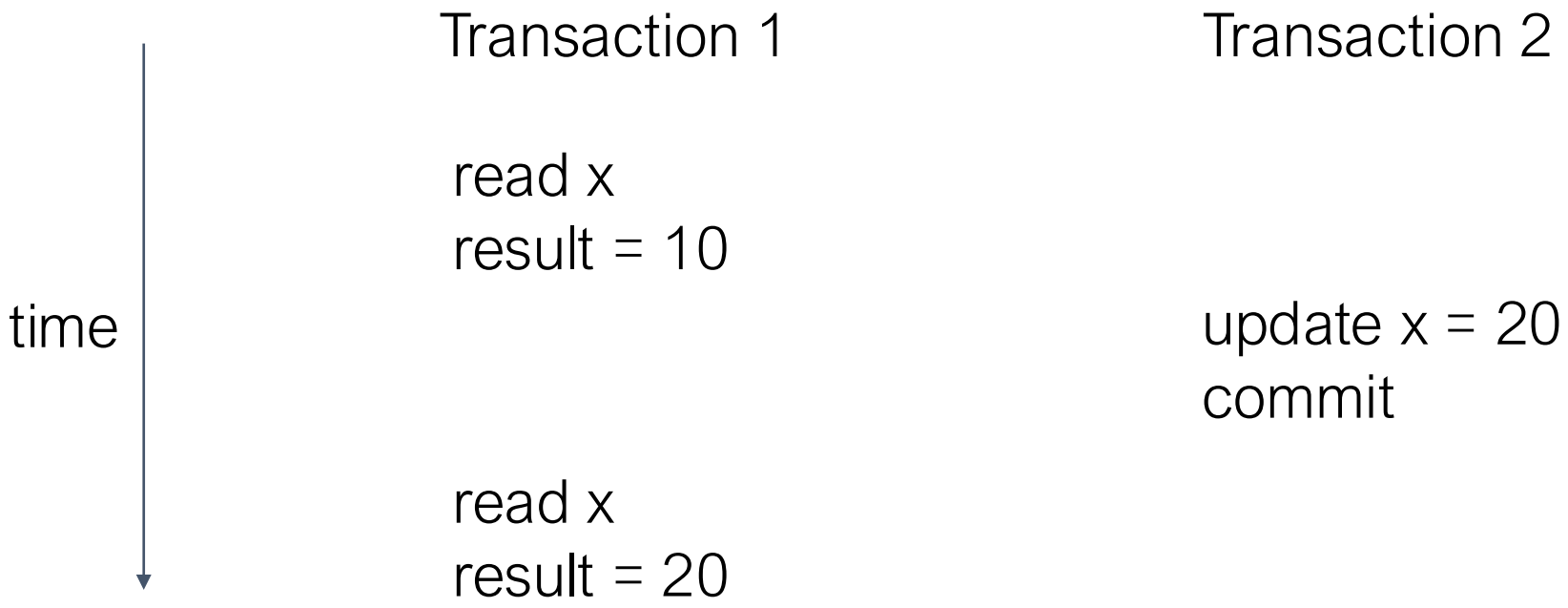
Tell DBMS before running transaction:

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

Read committed

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

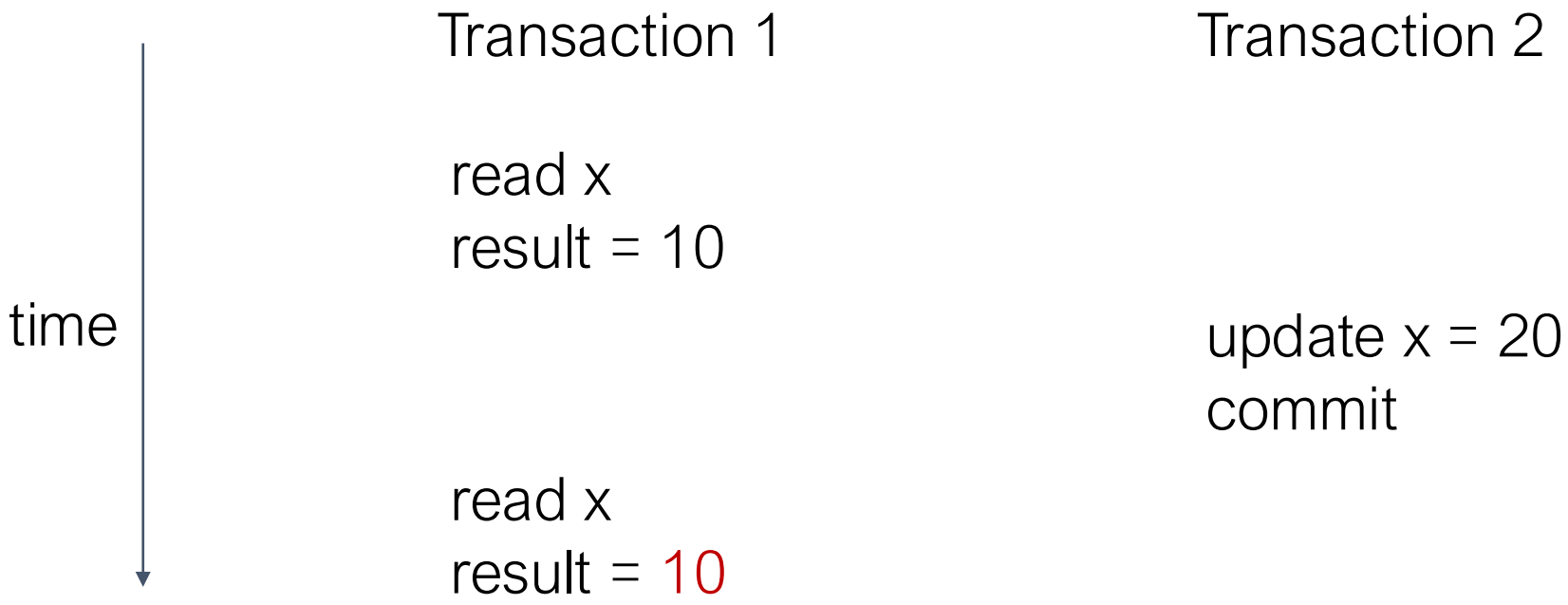
```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```



Repeatable read

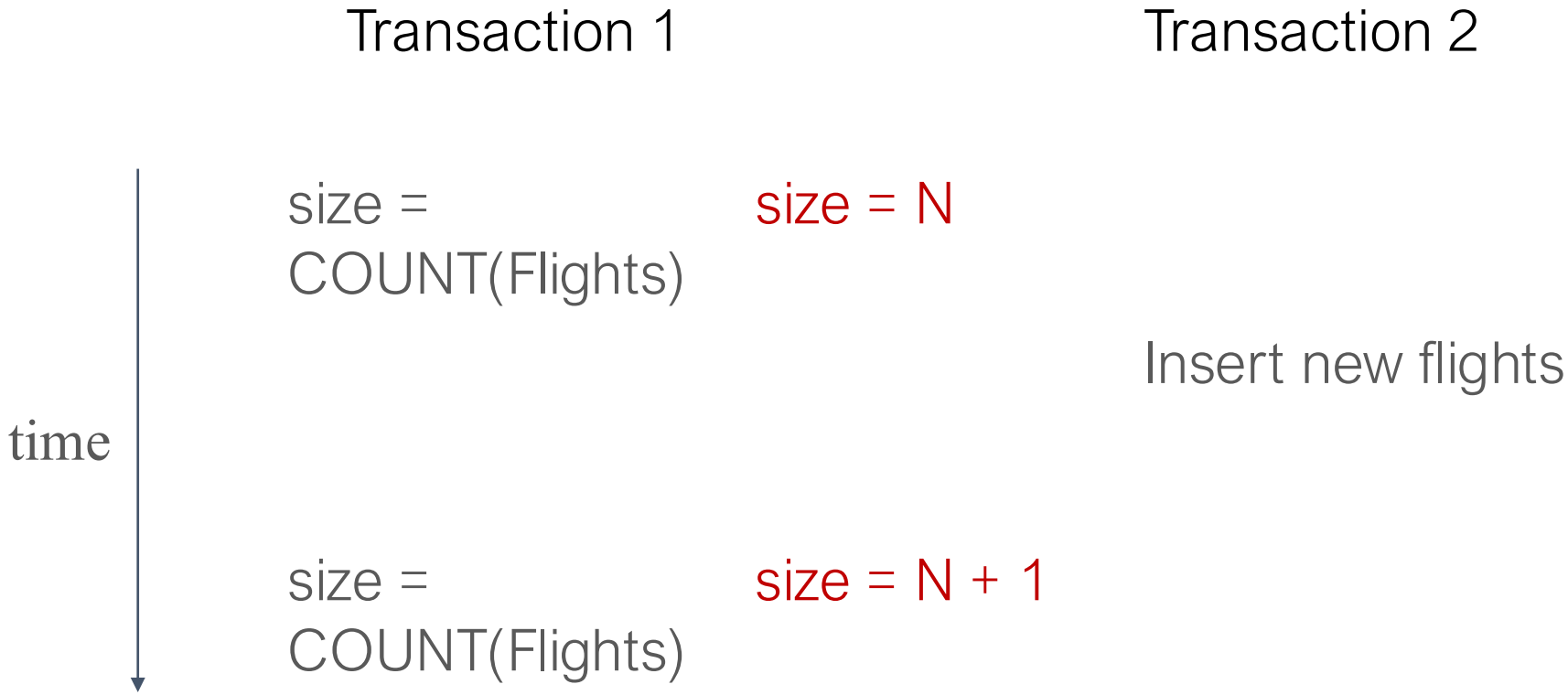
Any tuple that was retrieved will be retrieved again if the same query is repeated, even though other transactions may modify the individual rows that were read.

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



Repeatable read

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



Repeatable Read

Guarantee: rows read by a transaction will not change if read again in that transaction.

- Doesn’t guarantee anything about rows that weren't originally read.

Why Phantom Reads Can Occur

- Locking: Repeatable read typically locks the rows it reads, but not the gaps between rows.
- New Inserts: Without gap locking, new rows could be inserted that match your WHERE clause.

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
- In fact, PostgreSQL doesn't support this isolation level
- No lock on data

Comparison of SQL isolation levels

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

- Fast and simple to use; adequate for many applications
- Shared lock (read lock) on rows when they are read, exclusive lock (write lock) on rows when they are being modified

Comparison of SQL isolation levels

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

- Good for reporting, data warehousing types of workload
- Shared locks on all rows read by a transaction

Comparison of SQL isolation levels

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

- Recommended only when updating transactions contain logic sufficiently complex that they might give wrong answers in READ COMMITED mode
- Locking the entire range of rows that could potentially be accessed by a transaction's queries