M146 Database Systems
Spring 2021

Georgia Koutrika

Overview

- So far, we have focused on query processing
  - In other words, reading and manipulating data
- A database system, however, not only reads, but also stores data
  - At the same time as others are querying it

Challenges with Many Users

Suppose that a bank application serves 1000’s of users or more. What are some challenges?

Security: Different users, different roles

Concurrency: Need to provide performance through concurrent access

Disk/SSD access is slow, DBMS hides the latency by doing more CPU work concurrently

Consistency: Concurrency can lead to update problems

DBMS allows user to write data as if they were the only user

Overview (cont.)

- The basic concept is **transaction processing**
- Every transaction needs to satisfy **four basic properties**
  - Atomicity, consistency, isolation, durability
- To achieve performance
  - Solution: by interleaving transactions (**concurrency control**)
- How does the system **guarantee** these properties?

Overview (cont.)

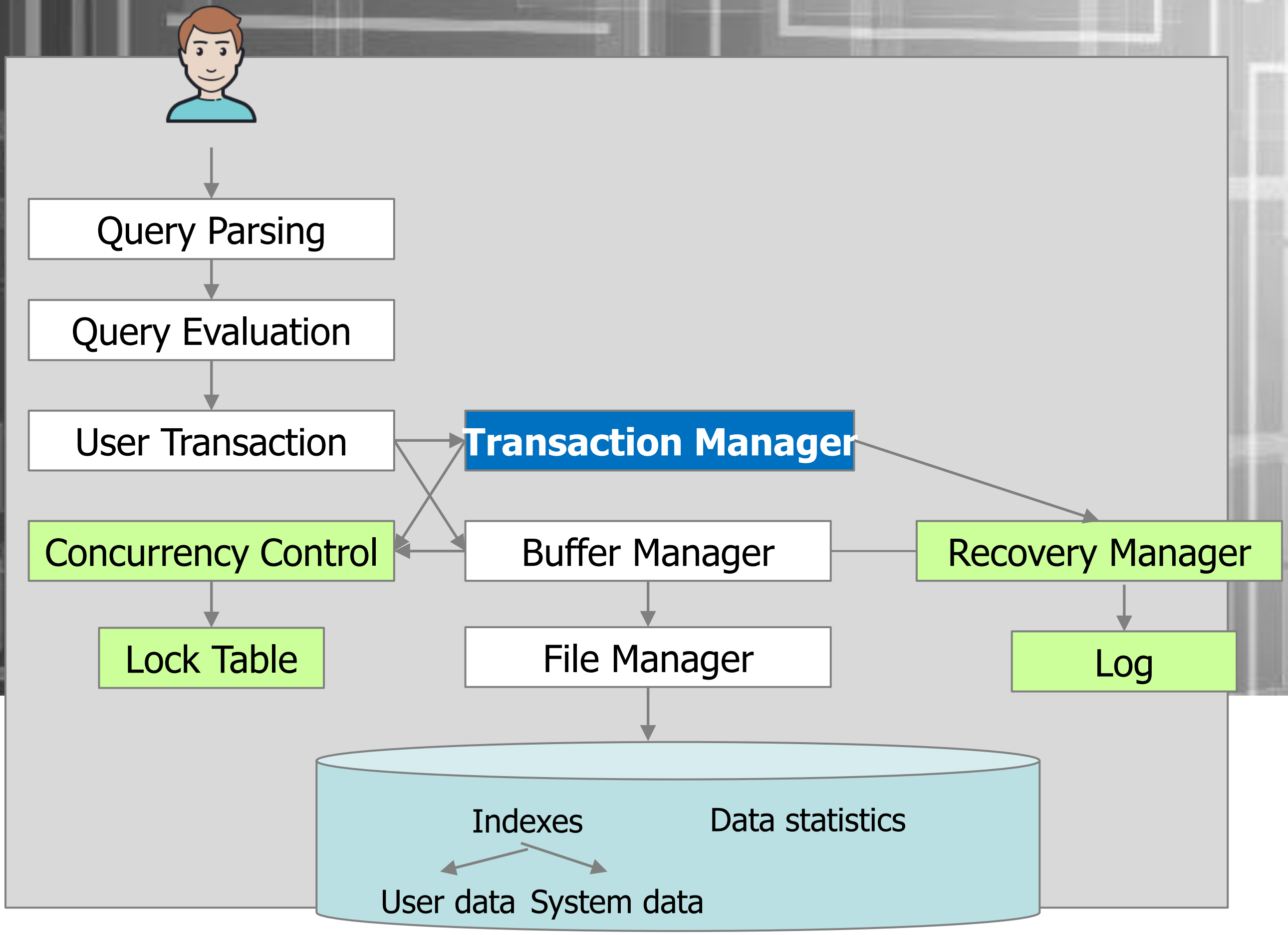
- Interleaving transactions causes certain anomalies
  - How can we decide if, after we have interleaved transactions, the **result is correct?**
  - Solution: the system uses **locks** to ensure correctness
- How are locks used?
  - Lock granularity, degrees of consistency and **two-phase locking**
- What impact do locks have on performance?

Overview (cont.)

- Locking poses significant overhead
  - Luckily, however, this overhead can be tuned by the user

- But what if the worse comes to worst? (e.g., system crashes)
  - **Transactional semantics**
  - **Write-ahead logging**
  - **Recovery**

TRANSACTIONS



Motivation for Transactions

Example:

Acct	Balance
a10	20.000
a20	15.000

Transfer $3k from a10 to a20:
1. Debit $3k from a10
2. Credit $3k to a20

Acct	Balance
a10	17.000
a20	18.000

There are two critical questions:

what happens if:
- Crash before 1
- After 1 but before 2
- After 2.

what happens if:
- We interleave other operations?

Transactions

A **transaction** can be defined as a group of tasks.
A single task is the minimum processing unit which cannot be divided further.

Acct	Balance
a10	20.000
a20	15.000

This is an example transaction

Transfer $3k from a10 to a20:
1. Debit $3k from a10
2. Credit $3k to a20

Acct	Balance
a10	17.000
a20	18.000

Transactions in SQL

- In "ad-hoc" SQL:
  - Default: each statement = one transaction
- In a program, multiple statements can be grouped together as a transaction:

```
START TRANSACTION
        UPDATE Bank SET amount = amount – 3000
        WHERE name = 'Bob'
        UPDATE Bank SET amount = amount + 3000
        WHERE name = 'Joe'
COMMIT
```

Motivation for Transactions

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

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

2. **<u>Concurrency:</u>** Achieving better performance by parallelizing TXNs *without* creating anomalies

Motivation for Transactions

<u>1. Recovery & Durability</u> 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

DELETE Product
        WHERE price <=0.99
```

Crash / abort!

What goes wrong?

Protection against crashes / aborts

If we enclose both operations in one transaction, we will be fine (we will see how).

```
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
```

Motivation for Transactions

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 busy…
> The DBMS uses locks to ensure correctness

Concurrent execution

- When a user submits a transaction it is as if the transaction is executing by itself
    - The DBMS achieves concurrency by interleaving transactions
    - If the transaction begins with the DB in a consistent state, it must leave the DB in a consistent state after it finishes
- The semantics of the transactions are unknown to the system
    - Whether the transaction updates a bank account or it fires a rocket missile, the DBMS will never know!

Multiple operations

```
UPDATE Bank SET amount = amount – 3000
WHERE name = ‘Bob’
```

```
UPDATE Bank SET amount = amount + 3000
WHERE name = ‘Joe’
```

If we execute them separately,
what could go wrong with interleaving?

```
START TRANSACTION
        UPDATE Bank SET amount = amount – 3000
        WHERE name = ‘Bob’
        UPDATE Bank SET amount = amount + 3000
        WHERE name = ‘Joe’
COMMIT
```

If we enclose both operations in one transaction,
we will be fine (we will see how).

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

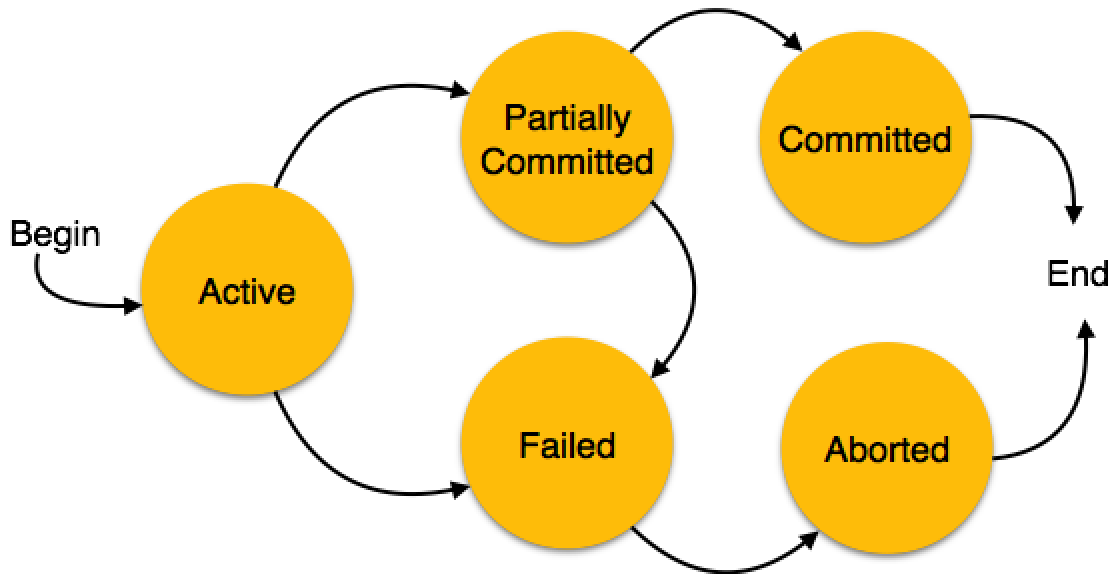
```
Client 1: START TRANSACTION
                    UPDATE Product
                    SET Price = Price – 1.99
                    WHERE pname = ‘Gizmo’
          COMMIT

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 …

States of Transactions

- **Active** – The transaction is being executed. This is the initial state of every transaction.
- **Partially Committed** – When a transaction executes its final operation.
- **Failed** – If any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.
- **Aborted** – If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state prior to the execution of the transaction. The database recovery module can select to–
  - Re-start the transaction
  - Kill the transaction
- **Committed** – If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.



https://www.tutorialspoint.com/dbms/dbms_transaction.htm

What you will learn about in this section

1. **<u>A</u>**tomicity
2. **<u>C</u>**onsistency
3. **<u>I</u>**solation
4. **<u>D</u>**urability

Transaction Properties: ACID

- Atomicity: all the actions in a transaction are executed as a single atomic operation; either they are all carried out or none are
- Consistency: if a transaction begins with the DB in a consistent state, it must finish with the DB in a consistent state
- Isolation: a transaction should execute as if it is the only one executing; it is protected (isolated) from the effects of concurrently running transactions
- Durability: if a transaction has been successfully completed, its effects should be permanent

ACID continues to be a source of great debate!

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.

  - E.g. Should not 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**

Challenges for ACID properties

- In spite of failures: Power failures, but not media failures
- Users may abort the program: need to “rollback the changes”
  - Need to *log* what happened
- Many users executing concurrently
  - Can be solved via *locking*

And all this with… Performance!! Performance!! Performance!!

A Note: ACID is contentious!

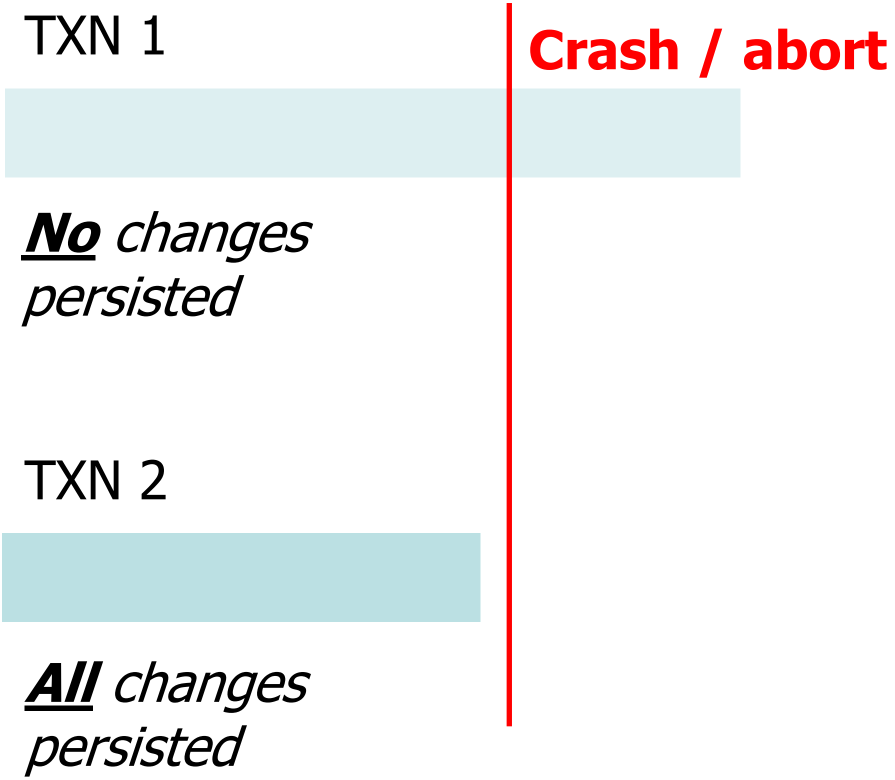
- Many debates over ACID, both **historically** and **currently**
- Many newer “NoSQL” DBMSs relax ACID
- In turn, now “NewSQL” reintroduces ACID compliance to NoSQL-style DBMSs…

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

Ensuring Atomicity & Durability

ACID

- **Atomicity:**
  - TXNs should either happen completely or not at all
  - If abort / crash during TXN, *no* effects should be seen



- **Durability:**
  - If DBMS stops running, changes due to completed TXNs should all persist
  - *Just store on stable disk*

We'll focus on how to accomplish atomicity (via logging)