Announcements

• Term Paper:
  – Final submission is due on April 18, 2018 @ 11:59 pm.

• MongoDB:
  – For quick testing:
    • https://mws.mongodb.com/
  – Aggregation:
    • https://docs.mongodb.com/manual/aggregation/
    • https://docs.mongodb.com/manual/tutorial/aggregation-zip-code-data-set/
    – $exp, $where, $regex,

• Quiz on this Wednesday:
  – Last problem set
  – Chapter 18 and 19
  – NO question on MongoDB
• Why did we spend so long talking about SQL and so less on MongoDB?

  – Because, you know all these concepts from SQL
  – MongoDB tutorial is really excellent
    • [https://docs.mongodb.com/manual/crud/](https://docs.mongodb.com/manual/crud/)
  – You can work in pair!
Today’s Lecture

1. Transactions

2. Properties of Transactions: ACID

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

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

In the real world, a TXN either happened completely or not at all.
A **transaction ("TXN")** is a sequence of one or more *operations* (reads or writes) which reflects a **single real-world transition**.

**Examples:**

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

In the real world, a TXN either happened completely or not at all.
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 - 100
  WHERE name = 'Bob'
  UPDATE Bank SET amount = amount + 100
  WHERE name = 'Joe'
COMMIT
```
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
```

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

COMMIT
```

Now we’d be fine! We’ll see how / why this lecture

Use ROLLBACK instead of COMMIT if you don’t want to continue
2. **Concurrent** execution of user programs is essential for good DBMS performance.

- 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 humming...
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 next lecture...
2. PROPERTIES OF TRANSACTIONS
What you will learn about in this section

1. Atomicity
2. Consistency
3. Isolation
4. Durability
Transaction Properties: ACID

• **Atomic**
  – State shows either all the effects of txn, or none of them

• **Consistent**
  – Txn moves from a state where integrity holds, to another where integrity holds

• **Isolated**
  – Effect of txns is the same as txns running one after another (ie looks like batch mode)

• **Durable**
  – Once a txn has committed, its effects remain in the database
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 (rollback)*: no changes are made
• 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 (zero)

• 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
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 (we’ll see this next lecture!)

And all this with... Performance!!
Ensuring Atomicity & Durability

• **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)
• Is a list of modifications

• Log is *duplexed* and *archived* on stable storage.

• Can **force write** entries to disk
  – A page goes to disk.

• All log activities are **handled transparently** by the DBMS.
Basic Idea: (Physical) Logging

• Record UNDO information for every update!
  – Sequential writes to log
  – Minimal info (diff) written to log

• The **log** consists of **an ordered list of actions**
  – Log record contains:
    <XID, location, old data, new data>

This is sufficient to UNDO any transaction!
Why do we need logging for atomicity?

• Couldn’t we just write TXN to disk **only** once whole TXN complete?
  – Then, if abort / crash and TXN not complete, it has no effect - atomicity!
  – *With unlimited memory and time, this could work...*

• However, we **need to log partial results of TXNs** because of:
  – Memory constraints (not enough space for full TXN??)
  – Time constraints (what if one TXN takes very long?)

We need to write partial results to disk!
...And so we need a log to be able to **undo** these partial results!
3. ATOMICITY & DURABILITY VIA LOGGING
What you will learn about in this section

1. Logging: An animation of commit protocols
A Picture of Logging
A picture of logging

T: R(A), W(A)

T

Main Memory

A=0
B=5

Log

Data on Disk

A=0

Log on Disk
A picture of logging

T: R(A), W(A)

A: 0 → 1

Main Memory

A=1

B=5

Data on Disk

Log on Disk

Log

CSC 261, Spring 2018, UR
A picture of logging

T: R(A), W(A)

A: 0→1

Main Memory

A=1
B=5

Log

Data on Disk

Log on Disk

Operation recorded in log in main memory!

CSC 261, Spring 2018, UR
What is the correct way to write this all to disk?

• We’ll look at the *Write-Ahead Logging (WAL)* protocol

• We’ll see why it works by looking at other protocols which are incorrect!

Remember: Key idea is to ensure durability *while* maintaining our ability to “undo”!
Write-Ahead Logging (WAL)
TXN Commit Protocol
Transaction Commit Process

1. FORCE Write commit record to log

2. All log records up to last update from this TX are FORCED

3. Commit() returns

Transaction is committed once commit log record is on stable storage
Incorrect Commit Protocol #1

T: R(A), W(A)

A: 0\rightarrow 1

Main Memory

A=1

B=5

Log

Data on Disk

Log on Disk

Let’s try committing before we’ve written either data or log to disk...

OK, Commit!

If we crash now, is T durable?

Lost T’s update!

CSC 261, Spring 2018, UR
Incorrect Commit Protocol #2

T: R(A), W(A)

A: 0 → 1

Let's try committing after we've written data but before we've written log to disk...

OK, Commit!

If we crash now, is T durable? Yes! Except...

How do we know whether T was committed??
Improved Commit Protocol (WAL)
This time, let’s try committing after we’ve written log to disk but before we’ve written data to disk... this is WAL!

If we crash now, is T durable?
Write-ahead Logging (WAL) Commit Protocol

This time, let’s try committing *after we’ve written log to disk but before we’ve written data to disk*... this is WAL!

If we crash now, is T durable?

**OK, Commit!**

*USE THE LOG!*

T: R(A), W(A)

T → Main Memory

A: 0 → 1

A=1

Data on Disk

Log on Disk

CSC 261, Spring 2018, UR
• DB uses **Write-Ahead Logging (WAL) Protocol:**

1. Must *force log record* for an update *before* the corresponding data page goes to storage

2. Must *write all log records* for a TX *before commit*

Each update is logged! Why not reads?

→ **Atomicity**

→ **Durability**
If DB says TX commits, TX effect remains after database crash.

DB can undo actions and help us with atomicity.

This is only half the story...
Acknowledgement

• Some of the slides in this presentation are taken from the slides provided by the authors.
• Many of these slides are taken from cs145 course offered by Stanford University.