TRANSACTION MANAGEMENT

CS 564- Spring 2018

ACKs: Jeff Naughton, Jignesh Patel, AnHai Doan
WHAT IS THIS LECTURE ABOUT?

• Transaction (TXN) management
• **ACID** properties
  – atomicity
  – consistency
  – isolation
  – durability
• Logging
• Scheduling & locking
TRANSACTIONS
DBMS MEMORY MODEL

Local: each process in a DBMS has its own local memory, where it stores values that only it “sees”

Global: each process can read from / write to shared data in main memory

Disk: global memory can read from / flush to disk

Log: Assume on stable disk storage- spans both main memory and disk
A **transaction** is a collection of **operations** that form a single **atomic** logical unit

```
BEGIN TRANSACTION;
{SQL}
COMMIT;
```

- **Operations:** READ / WRITE
- In the real world, a TXN either happens completely or not at all
TRANSACTION EXAMPLES

• Bank transfer of money between two accounts
• Purchase a group of products online
• Register for a class (either waitlist or allocated)
In SQL, multiple statements can be grouped together as a transaction:

```
BEGIN TRANSACTION ;
  UPDATE account
    SET balance = balance - 1000
    WHERE account_no = 1;
  UPDATE account
    SET balance = balance + 1000
    WHERE account_no = 2;
COMMIT ;
```
WHY TRANSACTIONS?

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

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

**Concurrency:** achieving better performance by parallelizing TXNs *without* inconsistencies.
RECOVERY & DURABILITY

- Data must be durable in the face of:
  - system crashes
  - TXN aborts 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
RECOVERY & DURABILITY: EXAMPLE

What can happen if the system crashes after the first SQL query is executed?

```
UPDATE account
    SET balance = balance - 1000
    WHERE account_no = 1;
UPDATE account
    SET balance = balance + 1000
    WHERE account_no = 2;
```
**CONCURRENCY**

*Concurrent* execution of user programs is essential for good DBMS performance

- better utilization: CPU/IO overlap
- avoids the situation where long running queries starve other queries
- provides the users with an illusion of a single-user system, called *isolation*
- maintains *consistency* during the concurrent execution
CONCURRENCY: EXAMPLE

What can happen if the two SQL queries are executed at the same time?

1: UPDATE account
   SET balance = balance - 1000
   WHERE account_no = 1;

2: UPDATE account
   SET balance = balance * 1.5
   WHERE account_no = 1;
The ACID Properties
ACID PROPERTIES

**Atomicity**: all actions in the TXN happen, or none happen

**Consistency**: a database in a consistent state will remain in a consistent state after the TXN

**Isolation**: the execution of one TXN is isolated from other (possibly interleaved) TXNs

**Durability**: once a TXN commits, its effects must persist
ATOMICITY

Atomicity: All actions in the transaction happen, or none happen

- Two possible outcomes for a TXN
  - **commit**: all the changes are made
  - **abort**: no changes are made
**ACID: CONSISTENCY**

**Consistency**: a database in a consistent state will remain in a consistent state after the transaction

- **Examples:**
  - account number is unique
  - stock amount can’t be negative
- **How consistency is achieved:**
  - the *programmer* makes sure a TXN takes a consistent state to a consistent state
  - the *DBMS* makes sure that the TXN is *atomic*
**ACID: ISOLATION**

**Isolation:** the execution of one transaction is isolated from other (possibly interleaved) transactions

**Example:**
- if T1, T2 are interleaved, the result should be the same as executing first T1 then T2, or first T2 then T1
**ACID: DURABILITY**

**Durability**: if a transaction commits, its effects must persist

- for example, if the system crashes after a commit, the effects must remain
- essentially, this means that we have to write to disk
CHALLENGES FOR ACID

• in spite of failures: power failures, but not media failures
• users may abort the program: need to “rollback the changes”
  – we need to log what happened!
• many users can execute concurrently
  – locking (we’ll see this next lecture!)

all these must be done while keeping performance in mind!
LOGGING
WHY LOGGING?

• Can we just write the modified pages to disk only once whole TXN is complete?
  – if abort/crash and the TXN is not complete, it has no effect: atomicity + durability!

• However, we need to log partial results of TXNs:
  – memory constraints (the buffer pool may want to write pages to disk earlier!)
  – time constraints (what if one TXN takes very long?)
LOGGING

The **log** is a list of modifications

- it records **REDO/UNDO** information for every **update**
  - only minimal info (diff) written to log
- it is **duplexed** and **archived** on stable storage (disk)
- it can **force** pages to disk

- it consists of an **ordered list** of actions of the form
  \[<TXNID, location, old-data, new-data>\]
LOGGING: EXAMPLE

A = 0
B = 2

T: Read(A), Write(A)
LOGGING: EXAMPLE

The log records the operation in the main memory!

T: Read(A), Write(A)
HOW DO WE WRITE THIS TO DISK?

- We will see the Write-Ahead Logging (WAL) protocol
- WAL guarantees atomicity & durability
- We will also see why other ideas don’t work!
WRITE-AHEAD LOGGING

1. we **force** the log record for an update before the corresponding page goes to disk

2. we write all log records for a TXN before commit

**Note:** WAL does not record any reads, only updates!
What happens if we commit the TXN before writing page/log to disk?
- if crash, not durable!

**T: Write(A), Write(C)**
What happens if we write the page to disk before the log writes to disk?

- if crash/abort, not atomicity!

**T: Write(A), Write(C)**
LOGGING: WAL PROTOCOL

$A = 0$
$B = 2$

$C = 4$
$D = 6$

$A = 0$
$B = 2$

$C = 4$
$D = 6$

$T$: Write(A), Write(C)
LOGGING: WAL PROTOCOL

A = 0
B = 2
C = 4
D = 6

So far no writing to disk. If crash/abort now, we are fine!

T: Write(A), Write(C)
If the buffer decides to write to disk, we must write the log before!
- if crash/abort, we can UNDO using log

T: Write(A), Write(C)
LOGGING: WAL PROTOCOL

A = 1  C = 4
B = 2  D = 6

log on disk
A: 0 → 1

main memory
A = 1
B = 2
C = 5
D = 6

log in memory
A: 0 → 1, C: 4 → 5

We don’t write to disk right away

T: Write(A), Write(C)
LOGGING: WAL PROTOCOL

Upon commit of the TXN, we force write the log to disk!
• the page does not need to be written!

T: Write(A), Write(C)
LOGGING: WAL PROTOCOL

Upon crash, we can use the log to reconstruct (REDO) the modified pages.

T: Write(A), Write(C)
ARIES

- The WAL protocol still has to force multiple pages to disk, which can limit performance
- **ARIES** is a (very) complex recovery algorithm that improves performance and has 3 phases:
  - Analysis
  - UNDO (rollback)
  - REDO (replay)

- For more on crashes and recovery, take CS 764!