Transactions and Concurrency Control

Database Systems

UNIVERSITY OF VIRGINIA
DATA SCIENCE INSTITUTE

A.C.I.D.

- A: Atomicity
- C: Consistency
- I: Isolation
- D: Durability

- **ACID** properties define what makes a good database
- It defines the rules for consistency – what makes a database transactional, so that when a change is made it is actually *reflecting the world around it*

Atomicity [A.C.I.D.]

- When a query runs, either ***all*** of it runs or ***none*** of it runs
- E.g. Can stipulate running a set of queries as a full transaction
  - Take out $50 from checking account
  - Deposit $50 into savings account
  - Want to do all (*both*) or nothing – don’t want to take out $50 but not put it in savings
  - That is bad, and doesn’t reflect the world around you

Consistency [A.C.I.D.]

- The database itself should always be an accurate representation of the world – without it the DB is useless
- It is difficult to do this at the database level. Can do some things to **enforce consistency**:
  - Foreign keys
  - Primary keys
  - Unique
  - Constraints
  - Triggers
  - Assertions
  - …

Consistency

- The database must remain in a consistent state before and after transactions are run (*successful or not*)
- Mainly enforced by the **application** that you are writing
- Consistency is determined by the client (customer driven)
  - The customer stipulates these constraints
  - E.g. A student can only enroll in this class if they exist in another table
  - "*Requirements* modeling" of a database

Isolation [A.C.I.D.]

- Every query runs as if it is the only query running on the database
- No query can at any time interfere with the query execution of another query
- E.g. Bank database – millions of customers, all trying to access ATMs at the same time
  - This becomes a concern
  - How to handles lots of transactions… which execute first? …

Durability [A.C.I.D.]

- If something goes wrong, the database is still preserved – everything needs to be recoverable. This is the concept of durability
- Questions
  - What happens if the power goes out in the middle of a transaction?
  - What happens if an Ethernet cable is damaged/pulled out/… in the middle of a transaction?
  - What happens if some one clicks cancel in the middle of a transaction?
- What happens? Everything has to be recoverable

A.C.I.D. Review

- A: Atomicity - *“All Or Nothing”*
- C: Consistency - *“Models The Real World”*
- I: Isolation - *“As If You’re The Only One”*
- D: Durability - *“Recoverability”*

They are Intertwined

- In a sense, *Atomicity*, *Isolation*, and *Durability* is to enforce <u>*Consistency*</u>
- All three ensure you are representing the world and "not messing it up"
- Therefore it is difficult to come up with an example that enforces one property but *not* another property
- So they are <u>related</u>

Transaction Safe

- ACID relational databases are the only databases considered transaction safe
- **Transaction safe databases** – guaranteed that the principles of *atomicity*, *consistency*, *isolation*, and *durability* are going to hold
- And therefore the queries you run will not mess up the database

ACID Compliance

- Does <u>not</u> follow ACID properties
  - NoSQL databases
  - Distributed databases
  - MyISAM
- Does follow ACID properties
  - InnoDB

ACID Compliance

- Atomicity
- E.g. +$50 and -$50 – one transaction (atomic unit)
- MyISAM uses “auto commit”
  - When you execute a query, decide what happened, write the results to disk. Then automatically commits the results to disk
- InnoDB can turn auto commit off
  - Begin atomic (starts a set of queries)
    - {Transactions} – stipulate which queries go together
  - End atomic
  - Commit – verify all that has occurred is fine *then* write to memory

Transaction “States”

- You can have transactions that can be in various states
- States a transaction can be in:
  - Active state
  - Partially Committed state
  - End
- MyISAM doesn’t have a partially committed state (you run the query, it immediately commits and you’re done)
  - At any time the query can *fail*
  - When that query fails – the query becomes aborted
  - When the query is aborted you have the option to **restart** or **end**

States

- Active State
  - A transaction begins
- Partially Committed State
  - All of the queries have executed
  - The DB has done a preliminary run of everything – verified that everything can execute
  - At this stage have not yet called the Commit command
- End State
  - Commit is called, everything is written to memory, query ends

States

- For Atomicity
  - Means turning on the Partially committed state
  - Transactions can move to this state and will have to wait until you explicitly say "OK now you can Commit" [moving to End state]

States

- For Durability
  - If something fails you can decide what to do when you are in the Partially Committed state
  - You will not go ahead and Commit (write results to disk) if something goes wrong, until all queries in the transaction have been executed successfully

States

- For Durability
  - The RAID level you choose also plays a part with the durability of a database
  - Something you can do without RAID that is similar is to use a "**Shadow Copy**"
  - Another thing you can do for durability is to use "**logging**"

Shadow Copy

- Create a “**Shadow Copy**” of a table
- When you are in a Partially Committed state it makes a copy of that table, runs the queries on it, makes sure everything is alright, then as soon as you’re ready to Commit…
  - Instead of taking all of the executions necessary and execute all again on the original table, it just **changes the pointer to that table**
  - This option takes up extra space and incurs ***overhead***

Logging

- A good database will log all of the commands (queries) that are occurring when going to active and partially committed states
- If it knows what commands have been occurring, if it gets into a wrong state or something happens, it can **rollback** by looking at the log files and executing a corresponding transaction that can undo what has happened
- To rollback a transaction is to **undo** what ever you have done so that the original consistency is preserved.

States

- For Consistency
  - Program driven
  - We talked about other things that enforce consistency on a DB
    - Foreign keys, primary keys, constraints, triggers, data types…
  - Consistent transactions means it must be atomic – otherwise it won't model the world accurately

Isolation

- For Isolation … is where things get *interesting*
- Isolation and concurrency control go hand-in-hand
- We have already said that one of the advantages of databases over a flat-file system is the ability to have **multiple users using the database at the same time** (multiple users getting their own data at a time, but also multiple users could be hitting the same data at the same time)
- Who gets access first?... Why one user over the other? … How do we know two things can “happen at the same time”?

Isolation

- Let's boil down the idea of the interactions between the program and the hard drive as simply being reads (R) and writes (W)
- Notation:
  - R_A = reading a data value A (variable A)
  - W_A = writing a data value A (variable A)
- Might have two transactions coming into the system:
  - **T1**: R_A, A=A+50, W_A
  - **T2**: R_A, A=A-50, W_A
- This is an example of a **SERIAL SCHEDULE**
  - Every transaction *appears to run independently*

Serial Schedule

- Every transaction appears to run independently, that is, appears to have its own access to the DB, looks like nothing is running concurrently
- It appears like:
  - one thing runs to completion, then
  - the next one starts and it runs to completion, then
  - the next one starts…
- **Simplifying to single-thread, single-execution environment**

Serial Schedule

- For those of you who have taken OS, this is <u>not</u> a good way to schedule things to a processor
- Problems? Can run really slow, average response time for users is very high
- If there is a short transaction that is doing R/W to a different location, there should be no harm in…
    - *Pausing* a longer transaction
    - Jumping ahead and executing that shorter transaction
    - Finish executing
    - *Resume* the longer transaction

Pre-emptive scheduling

- A pre-emptive schedule is a common idea in processing
- T1 and T2 read and write the same data so these two transactions shouldn't move between each other – should only go one at a time
- But what if we had…
  - T1: R_A, W_A, R_B, W_B
  - T2: R_C, W_C, R_B, W_B
- Reading C and Writing B are not conflicting operations
- You *can* move things around

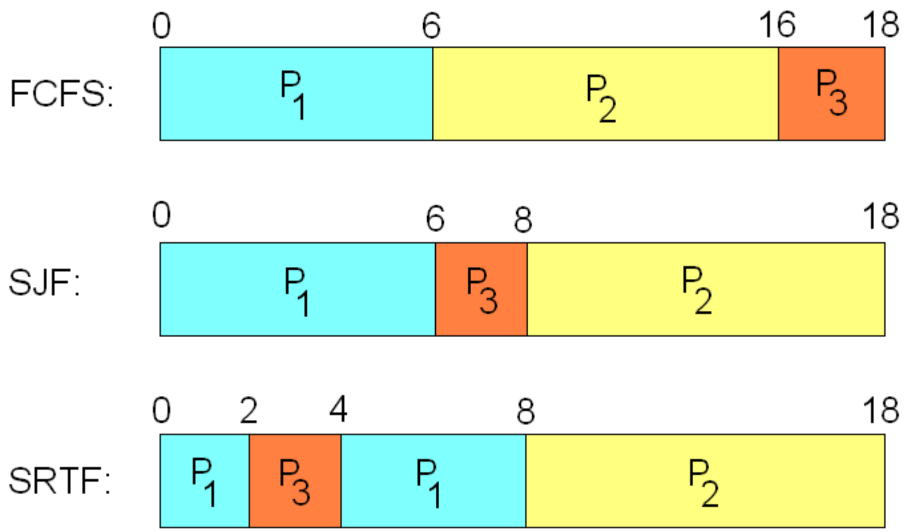
Scheduling

Examples of Scheduling Types

FCFS = First Come First Served
(non pre-emptive)

SJF = Shortest Job First
(non pre-emptive)

SRTF = pre-emptive version of SJF



Suppose that a system has three processes, with the following table of arrival times and burst times:

process number	arrival time	burst time
1	0	6
2	0	10
3	2	2

*[Reminder: **average response time** is improved; however, overall raw time remains the same.]*

Conflict Serializable Schedule

- If you can move things around you'll end up with a schedule called **CONFLICT SERIALIZABLE**
- If you can move non-conflicting operations, you can eventually get to a **serial schedule** (rearranging the order in which the reads/writes are executed)
- For a DB to be considered ACID compliant with isolation, it must be able to generate **conflict serializable schedules** of execution (from the list of transactions coming in) → this is the definition of **isolation**

Isolation

- Isolation means, when a transaction runs it “has the DB to itself”
  - In reality we don’t care if it has the DB to itself – we care about whether it has its data items to itself
- The idea of conflict serialization: Able to do analysis of priority queue of commands/queries about to be executed and if we see the following: T5: [________________________]

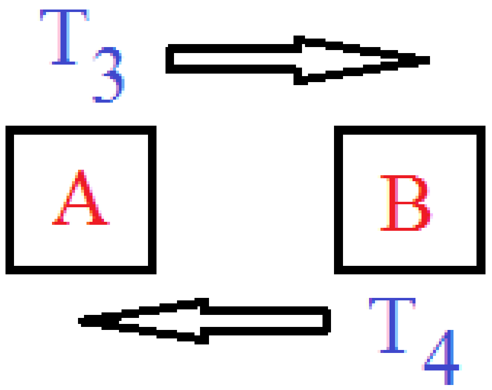
T6: [__]

Allow T6 to jump ahead of T5 without T5 getting messed up

- Average response time goes *down* (faster) – gives illusion that each user has the DB to him/herself
- Example of isolation and concurrency control mixed together
- Not improving raw performance but average response time

Deadlocks

- Deadlocks still do sometimes occur
- “exclusive locks”
- “shared locks”
- *To rollback from a deadlock*
  - “Youngest query dies”
  - Forced to fail state
  - Goes to this state and is automatically restarted
  - Forced to give up its exclusive lock on resources
  - Other transactions finishes by grabbing the lock on the resource



T3: XLockA, R_A, XLockB, R_B
T4: XLockB, R_B, Xlock A, R_A
DEADLOCK!

Cascading Rollback

- How bad can things get?
  - T7: R_A, R_B, W_A ←Assume something happened with this transaction
  - T8: R_A, W_A ← Ideally the read on "**A**" shouldn't be allowed to execute until the write in **T7** was confirmed to be good
  - T9: R_A, …
- Result? **Cascading Rollback**
- If T7 is aborted, T8 has to be undone since it read a value that was written to by T7. Since T9 read a value that was written to by T8 but read and written on a "bad" value (from T7)… etc
- This is not good, could be 100s of people writing to a similar data structure and if one transaction in the middle is screwed up → have to undo everything!
- Avoiding cascading rollback: goal of CONFLICT SERIALIZABILITY