

Relational Database System Implementation

CS122 – Lecture 14

Winter Term, 2018-2019

Database Limitations

- Can create a pretty sophisticated database by now
 - Can parse, plan, and execute SQL queries
 - Provide faster access-paths to records using indexes
 - Perform query-plan optimization
- Our database still lacks an essential set of capabilities:
- **The database isn't reliable when failures occur!**
 - Logical failures – an operation against the database violates some constraint and cannot be completed
 - System failures – the hardware or OS suffers a failure
- **The database also cannot handle concurrent access**

Transaction Processing

- Databases provide transactions to properly handle these situations
- A *transaction* is a collection of one or more operations that form a single logical unit of work
- Clients must tell DB when a transaction begins or ends
 - Start a transaction:
 - Standard: START TRANSACTION
 - Also: BEGIN [TRANSACTION | WORK]
 - Complete a transaction:
 - Standard: COMMIT [WORK]
 - Also: COMMIT TRANSACTION, END [TRANSACTION | WORK]

Transaction Processing (2)

- A transaction is a single logical unit of work
 - Should be indivisible: either all operations affect the database, or none of them do
- Clients can *abort* an in-progress transaction:
 - Client tells DB to undo all changes made by transaction
 - Also called *rolling back* a transaction
 - Commands:
 - Standard: ROLLBACK [WORK]
 - Also: ROLLBACK TRANSACTION, ABORT [TRANSACTION | WORK]
- The DB itself can also abort transactions, in some cases
 - e.g. if a constraint is violated during a transaction

ACID Properties

- Transaction processing systems should satisfy specific properties, called the *ACID properties*
- ACID properties were originally devised by Jim Gray
 - A critical contribution to databases and transaction processing systems
 - Gray won a Turing award in 1998 for this work
 - “ACID” acronym was later coined by other researchers

ACID Properties (2)

- Atomicity
 - Either all operations in the transaction are reflected in the database, or none of them are
- Consistency
 - Execution of the transaction (in isolation from any other transactions) preserves all database constraints
 - Given: the database starts in a consistent state
 - The completed transaction should also leave the database in a consistent state

ACID Properties (3)

- Isolation
 - When multiple transactions are executed concurrently, they must appear to execute in isolation of each other
 - For every pair of transactions T_i and T_j , it appears that either T_i completes before T_j starts, or that T_j completes before T_i starts
- Durability
 - After a transaction completes successfully (i.e. after it is reported as committed), all changes it made are persistent, even if there are system failures

Example: Account Transfer

- Classic transaction-processing example: transfer money from one bank account to another account
- Database transactions involve complex SQL statements
- Model them as a sequence of read and write operations
- Example: transfer \$50 from account A to account B

```
 $T_i$ : read( $A$ );  
       $A := A - 50$ ;  
      write( $A$ );  
      read( $B$ );  
       $B := B + 50$ ;  
      write( $B$ );
```


Example: Account Transfer (2)

- Consistency:
 - If database was in a consistent state before the transaction, it will still be consistent afterward
- Often involves constraints that aren't specifically modeled in the database
- Example: sum of account balances $A + B$ should be unchanged by this transaction
- May have other constraints as well, such as:
 - All accounts must have a non-NULL balance
 - Account balance is not allowed to be negative
- DB may be able to enforce some of these constraints, but the application is also responsible for ensuring consistency!

```
 $T_i$ : read(A);  
      A := A - 50;  
      write(A);  
      read(B);  
      B := B + 50;  
      write(B);
```

Example: Account Transfer (3)

- Scenario 1: In the process of this transaction, a failure occurs after $\text{write}(A)$, but before $\text{write}(B)$
 - e.g. perhaps account B doesn't exist, or the system crashes, etc.
- Atomicity:
 - Either all of the transaction's operations complete, or none of them do
- In this case, if atomicity is violated:
 - The database loses \$50! Consistency is also violated.
- The database enters into an *inconsistent state*

```
 $T_i$ : read(A);  
      A := A - 50;  
      write(A);  
      read(B);  
      B := B + 50;  
      write(B);
```

Example: Account Transfer (4)

- Scenario 2: This time the transaction completes, but *then* the system crashes
 - e.g. power failure, disk crash, BSOD/kernel-panic, or database software crashes (least likely! 😊)
- Durability:
 - If the transaction is durable then the database will still reflect the changes after recovery has been completed
 - The client doesn't need to repeat the transaction again
- When DB responds that the transaction is committed, this is a guarantee that the changes will persist

```
Ti: read(A);  
      A := A - 50;  
      write(A);  
      read(B);  
      B := B + 50;  
      write(B);
```

Example: Account Transfer (5)

- Most DBs allow concurrent access by multiple clients
 - Multiple transactions can occur at the same time
- Two account transfers occurring at the same time:

T_i :	read(A);	T_j :	read(A);
	$A := A - 50$;		$A := A - 30$;
	write(A);		write(A);
	read(B);		read(C);
	$B := B + 50$;		$C := C + 30$;
	write(B);		write(C);

- Isolation:
 - Txns must appear to execute in isolation of each other
 - Either T_i executes and then T_j executes, or vice versa

Example: Account Transfer (6)

- Database could literally follow this rule:
 - Either T_i executes and then T_j executes, or vice versa
 - Execute only one transaction at a time
- Called a *serial* execution schedule
- Problem: this is really slow

T_i :	read(A); $A := A - 50$; write(A); read(B); $B := B + 50$; write(B);	T_j :	read(A); $A := A - 30$; write(A); read(C); $C := C + 30$; write(C);
---------	--	---------	--

 - Doesn't maximize utilization of database server resources
 - Transaction throughput will be really low
- Most of the time, transactions work with completely different records. *Why slow things down?!*

Example: Account Transfer (7)

- Most databases interleave transaction operations
 - Simply must ensure that the transactions *appear* to execute in isolation of each other

- Example: what if T_i and T_j are executed like this?

T_i :
 read(A);
 A := A - 50;
 write(A);

- This will properly maintain transaction isolation

T_j :
 read(A);
 A := A - 30;
 write(A);

- It is *equivalent* to a serial execution schedule
- It is a *serializable* schedule

read(B);
 B := B + 50;
 write(B);

read(C);
 C := C + 30;
 write(C);

Example: Account Transfer (8)

- What if T_i and T_j were are executed like this instead?
- This execution schedule will produce an inconsistent state!
 - T_i clobbers T_j 's update of A
 - In this case, our database creates \$30 out of thin air...
- Transaction isolation is very important when a database supports concurrent access

T_i : read(A);
A := A - 50;

T_j : read(A);
A := A - 30;
write(A);

write(A);
read(B);

read(C);
C := C + 30;
write(C);

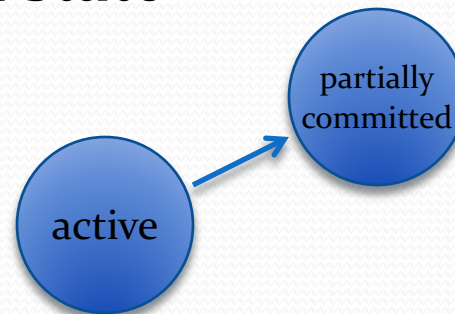
B := B + 50;
write(B);

Implementing ACID Properties

- Atomicity, Consistency and Durability are important whether the database is single-user or multi-user
 - Still need these transaction properties even when database is only used by one client at a time!
- Isolation is only important when a database can be used by multiple clients at the same time
 - (And it's much more complicated...)
- Will discuss Atomicity, Consistency and Durability first
- Talk about Isolation afterward

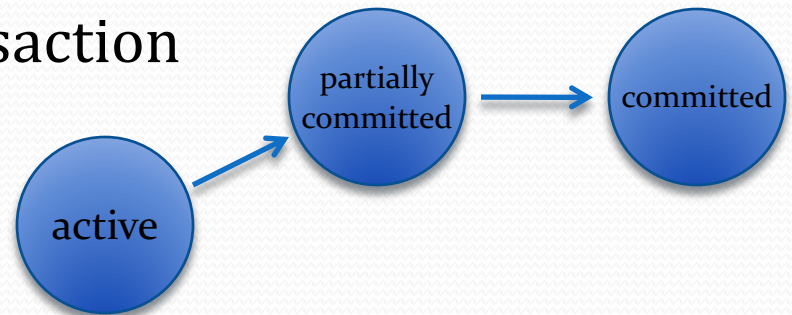
Transaction States

- Each transaction goes through a series of states
- Initially, transactions are in the Active state
 - More operations can be performed in the context of this transaction
- When last operation has been performed, transaction enters the Partially Committed state
 - e.g. client issues COMMIT
 - No more operations can be performed in the transaction
 - Database still has work to do!



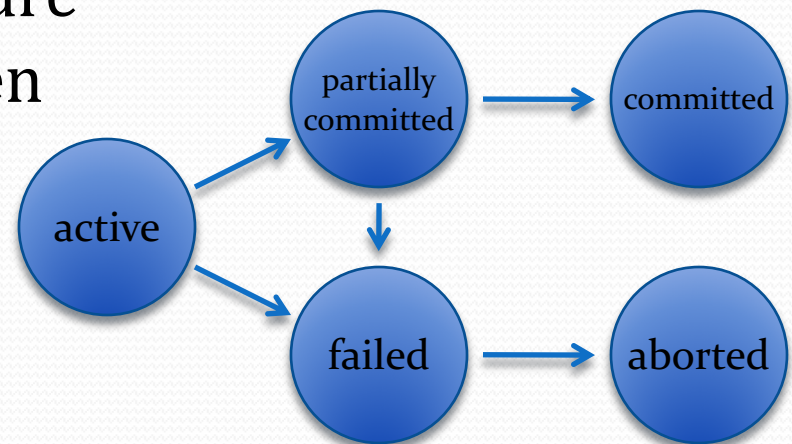
Transaction States (2)

- Partially Committed state:
 - Client can't do anything else, but database must now commit the transaction
- Transaction's state-changes may still reside in memory
 - Database may still need to write data to disk
 - DB may need to verify constraints that have been deferred to the end of the transaction
- If these operations succeed, transaction enters the Committed state
 - DB has recorded that the transaction is committed
 - Txn will be durable and atomic



Transaction States (3)

- If database cannot complete commit-operations, the transaction enters the Failed state
- Transaction can also enter the Failed state while Active
 - Will occur if an operation violates a database constraint
 - Or, client may issue a ROLLBACK command
- At this point, the DB must ensure that all state-changes have been rolled back to previous state
 - Once this is done, the txn enters the Aborted state



Storage Characteristics

- Ability to implement durable and atomic transactions depends on characteristics of storage media
- Previously discussed storage hierarchy
- Primary storage – main memory, caches
 - This storage is *volatile*: data won't survive a power loss
 - Also usually doesn't survive through a system crash
- Secondary/tertiary storage – disks, SSD, tapes, optical
 - This storage is *nonvolatile*: data survives loss of power
 - Can still suffer data corruption or data loss, e.g. if a hard disk crashes, or if the system crashes during a write

Storage Characteristics (2)

- Storage characteristics broken down by reliability:
 - Volatile storage – doesn't survive system failure
 - Nonvolatile storage – survives a system failure, but still susceptible to data loss
- A third category of storage reliability:
 - *Stable* storage – data is never lost or corrupted
- Stable storage is an “ideal” to strive for
 - Requires very careful engineering to achieve (e.g. redundant storage devices, off-site backups, etc.)
 - Most systems don't require that data is *never* lost; just aim to ensure that data loss is extremely unlikely

Storage Characteristics (3)

- Transacted operations are usually performed in volatile memory
 - Supports fast random access, use in computations
- To make a transaction durable:
 - Must ensure that all effects are properly recorded in nonvolatile storage (or stable storage, ideally)
- To make a transaction atomic:
 - Must record transaction's effects to nonvolatile storage in such a way that all effects become "committed" at once

Platform Requirements

- To make a transaction durable:
 - Must ensure that all effects are properly recorded in nonvolatile storage (or stable storage, ideally)
- Most platforms provide caching between memory and disk
 - Dramatically improves performance by avoiding I/O operations that can be completed using data in memory
- Platform/OS must provide a way to force all cached writes to nonvolatile storage
 - When operation completes, platform guarantees that all modified data has been written to nonvolatile storage
 - e.g. UNIX has `fsync()` operation – synchronizes a file to disk
 - If system crashes after `fsync()` completes, data is still there (barring filesystem corruption, of course)

Platform Requirements (2)

- To make a transaction atomic:
 - Must record transaction's effects to nonvolatile storage in such a way that all effects become "committed" at once
- Platform/OS must ensure that certain operations against nonvolatile storage are also atomic
 - The operation either completes successfully, or it doesn't complete at all (no partial failures!)
 - e.g. most UNIX file-IO operations are atomic, such as `write()` (for certain data sizes), `rename()`, `unlink()`, ...
 - Also atomic in the context of concurrent usage

Platform Requirements (3)

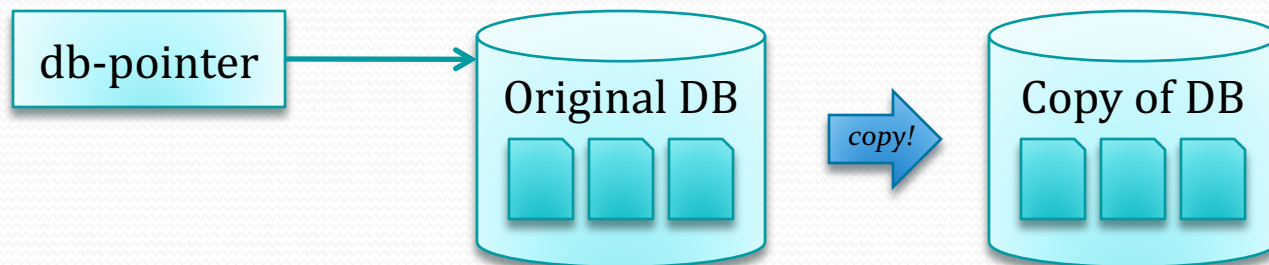
- Platform/OS can't always guarantee that operations against nonvolatile storage will be atomic in context of operating system or hardware failures
 - e.g. during a `fsync()` or `write()` operation, power fails
 - File being written may sustain a limited amount of data-loss or corruption
- Can employ some strategies to mitigate this issue...
 - *(Aim to provide as much durability as possible)*
- Database server is really only as good as the operating system and hardware that it's running on
 - e.g. want journaling filesystem, RAID, reliable power, etc.

Atomic, Durable Transactions

- Tables usually live in different files...
 - Multiple files may be written by a given transaction
 - A transaction may write to multiple parts of a given file
- Really isn't a way to update or modify multiple files in a single atomic operation
- Example commit operation:
 - Database writes each dirty page to disk, then calls `fsync()` on each modified table file in order...
 - ...but if the database or operating system crashes during this process, the transaction will not be durable or atomic! ☹️
- Instead, we must find a way to turn our “commit” operation into a single atomic update against a single file

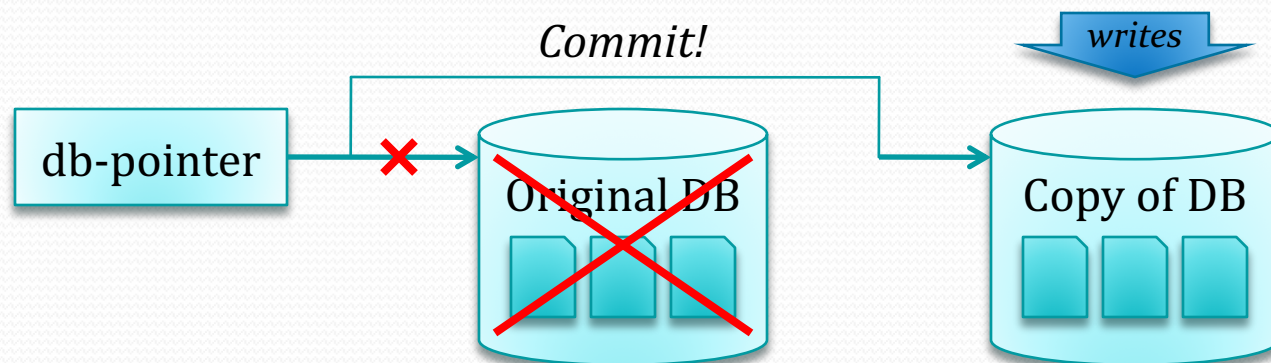
Another Strategy

- For this strategy, require only one transaction at a time
- When a transaction modifies the database, the DB server creates a complete copy of the database
 - All table files, all indexes, etc.
- The DB server keeps track of the “current” database with a single pointer to which copy is current
 - Initially points to the original set of files



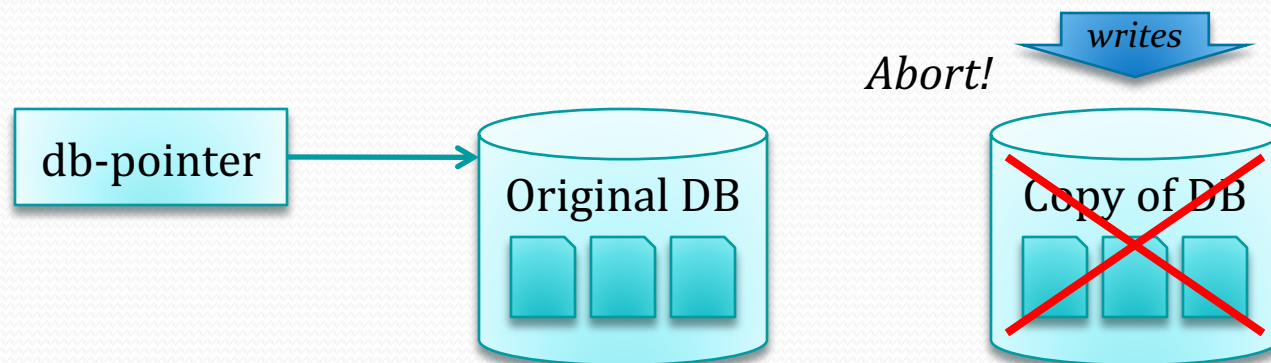
Another Strategy (2)

- All reads and writes are performed against copy of DB
- At commit time, DB server performs this sequence:
 - Write all dirty pages to disk, and fsync() each data file
 - db-pointer is updated to point to new copy
 - db-pointer is updated on disk, and then fsync()ed as well
 - At this point, the transaction is considered “committed”
 - Finally, old copy of DB is deleted



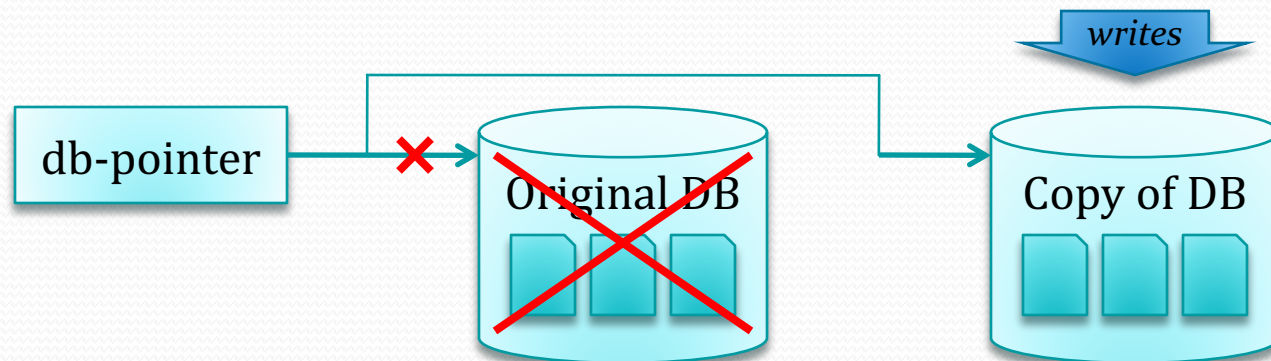
Another Strategy (3)

- If a transaction must be aborted, DB server simply deletes the new copy of the database
 - All changes were made against the copy
 - Original version is still completely unchanged
- Satisfies our requirements for transaction atomicity



Shadow Copies

- This approach is called *shadow-copy*
- Obviously very slow...
 - Can be greatly improved by dividing data into pages, and then employing a copy-on-write strategy with pages
 - Called *shadow-paging*
- Main issue is it only allows one transaction at a time
 - This strategy is rarely employed due to this limitation



Shadow Copies (2)

- Too limited for general use, but still captures the essential requirement:
 - Committing a transaction must involve a single atomic operation against non-volatile storage
 - Made all changes into a copy of the database
 - Final commit operation simply required updating the db-pointer value, then syncing it to disk
- If system crashes before db-pointer is sync'd to disk:
 - At recovery, DB considers the transaction to be aborted
 - (It has to, because there is no other record that the transaction completed successfully.)