Distributed Databases

Database Systems

UNIVERSITY OF VIRGINIA
DATA SCIENCE
INSTITUTE

A.C.I.D. Review

A.C.I.D. Review

- **ACID** is a set of properties that guarantee that database transactions are processed *reliably*
- A **transaction** represents a single logical operation on the data, which usually consists of multiple queries
- **Transaction safe databases** – guaranteed that the principles of atomicity, consistency, isolation, and durability are going to hold

A.C.I.D. Review

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

How would you describe each of these in as few words as possible?

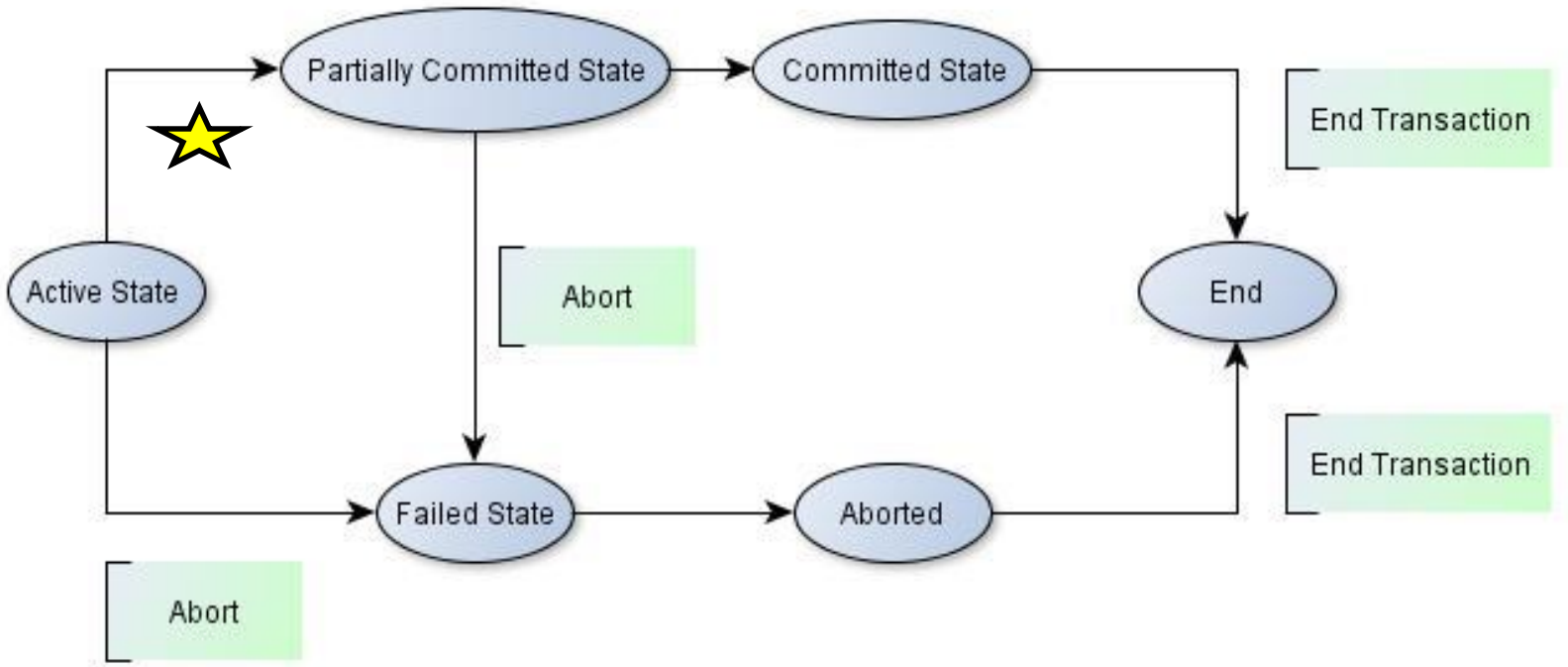
A.C.I.D. Review

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

Atomicity

- *“All Or Nothing”*
- Either all of the queries in a transaction are executed or none of them are
- Each transaction is considered to be atomic
  - If one part of the transaction fails, the entire transaction fails
- Contrast:
  - “auto-commit” (e.g. MyISAM)
  - “begin atomic”, “end atomic”, “commit” (e.g. InnoDB)

Transaction States



Consistency

- *“Models The Real World”*
- The database should always be an accurate representation of the world, without it the DB is useless
- The database must remain in a consistent state before and after transactions are run (*successful or not*)
- Can be enforced via
  - Foreign keys, Primary keys, Unique, Constraints, Triggers, Assertions, … even data types
- Mainly enforced by the application
- Determined by the client – “requirements” of a DB

Consistency

- Databases that we've talked about have been examples of centralized systems
  - Strongly consistent
- Distributed database systems (physically separate from each other) – *don't follow the **ACID** properties*
  - either strongly consistent or has some form of weak consistency
  - At the other end of the spectrum: ***BASE*** properties ("eventual consistency")
  - *This is where our discussions will take us by the end of the lecture*

Isolation

- *"As If You're The Only One"*
- Queries run as if they are the only query running on the database at that time
- No queries will interfere with another query
- Maintain the *performance* (average response time to be low) as well as the *consistency* between transactions
- To ensure isolation, the transaction schedule has to be conflict serializable
  - Ability to interleave Read/Write commands

Durability

- *“Recoverability”*
- If something goes wrong, you have the ability to recover
- System is tolerant to failure
- Implementation of durability via
  - RAID
  - Shadow Copy
  - Logging
- Deadlocks, Deadlock prevention, and Deadlock recovery (Rollbacks)

DB Architecture and Distributed Systems

Times Are Changing…

- For many years (70s→early 2000s) if you wanted a reliable, working database, you made sure your database was ACID compliant
- Things have changed since then
- There are applications that have scaled rapidly
  - Twitter, FB, … & anything cloud related
- Multicore computers have been introduced
- How to leverage these changes to make our systems work better?

Architecture Styles

- Centralized Systems
- Parallel Systems
- Distributed Systems

Centralized Systems

- Running on a single computer
- Have no concurrency (some DB systems are not multi-user)
- Databases written for centralized systems do not split the parts of a query among the processors, but they will run a different query on each processor
- Higher throughput than single processor, but individual transactions don't run any faster (since there still is a precedence in order of queries)

Centralized Systems

- *How to scale up Centralized Systems?*
- SSD instead of HD
- Increase RAM
- System bus – “motherboard super-highway” (bottleneck)
- One core vs. multi-core

Centralized Systems

- How can one improve the performance of a database using multiple cores?
- In a DBMS there is not just one process
  - Server process – listening to the network
  - Query processor process (DB writer)
  - Lock manager process – in charge of what tables are locked
  - Log writer process – in charge of writing to log files, incase there is a need to do cascading rollback
  - Process monitor process – if something dies it handles it
- So can *split up the duties* among the cores

Centralized Systems

- *Split up the tasks* – run different transactions on different cores
- *Splitting a query* – e.g. a query with 4 nested select statements – but this is harder to do
- What's the next best thing you can do to improve upon a single-processor system?
- Instead of one CPU with multiple cores… have multiple CPUs

Parallel Systems

- Parallel Architecture – when there are multiple CPUs
- *Fewer processors*, but each more powerful & more expensive
  - Can execute a query faster
- *Many processors*, but each less powerful & cheaper
  - Can execute more queries at the same time
- Increase speed-up - can execute *more* queries
- Increase scale-up - can execute more *complex* queries
- All a matter of scale
  - it's usually easier/cheaper to add more parallelism than to replace a centralized system with something "more"

Parallel Systems

- A parallel system will be on a single bus, with multiple CPUs (that have multiple cores) They all share RAM and HD
- Shared-memory architecture

Architectures involving multiple CPUs

- **Shared-memory architecture**
  - all processors and disks have access to common memory on a bus - *bottleneck*: bus speed
  - If all the tables are in RAM, then each of these CPUs can all be working together on the database

Architectures involving multiple CPUs

- **Shared-disk architecture**
  - all processors access all disks via a network, but have private memory (RAM) - *bottleneck*: connection
  - Advantage: if splitting transactions on different cores, or working on different applications on each core – works well
  - Disadvantage: if working on the same application, or on the same tables – you have consistency issues
    - The version of the DB in RAM in one location may not be the same in another location
    - At some point they have to agree before data is written back to the HD
    - Introduces overhead

Architectures involving multiple CPUs

- The problem/challenge regarding the hardware of the previous two architecture systems is that they require very customized hardware – you cannot normally upgrade a motherboard by inserting extra CPUs
- Kept evolving…

Architectures involving multiple CPUs

- **Shared-nothing architecture**
  - Use a network to communicate - *bottleneck*: non-local disk access
  - The fundamental components of the computer are not shared
  - Separated physically – individual machines are connected via Ethernet

Distributed Systems

- **Distributed System** – a *shared-nothing architecture* in which the machines are in different physical locations
- Some advantages
  - Sharing data
  - Autonomy
  - Scalability
  - Availability through replication
    - *Less danger of single-point failure*

Evolution to Distributed Systems

- *Centralized systems* with single core CPUs→
- Multiple cores →
- Multiple CPUs →
- *Parallel systems* each with own RAM →
- Connected a number of computers together in one location →
- Put computers in physically different locations →
- *Distributed systems*

Distributed Database Systems

- Depending on RAID system chosen – failure tolerance could be better, but…
- **Challenges**
  - Security – using the Internet
  - Consistency issues
  - Increased storage requirements

  - Transactions may access data at one or more sites

Distributed Database Systems

- **Distributed Strategies**
  - Homogeneous
  - Heterogeneous
- *How is the data stored?*
  - Replication
  - Fragmentation
    - Horizontal
    - Vertical

Distributed Strategies

Homogeneous

- All sites have identical software
- All sites have similar schema

Sites are aware of each other and agree to cooperate in processing user requests

Each site surrenders part of its autonomy in terms of right to change schemas or software

Heterogeneous

Sites may have different software

Sites may use different schemas

- Sites may not be aware of each other and may provide only limited facilities for cooperation in transaction processing

Difference in schema is a major problem for query processing

Difference in software is a major problem for transaction processing

Distributed Data Storage

- Assume relational data model
- **Replication**
  - System maintains multiple copies of data, stored in different sites, for faster retrieval and fault tolerance
- **Fragmentation**
  - Relation is partitioned into several fragments stored in distinct sites [Horizontal & Vertical]
- Replication and fragmentation can be combined
  - Relation is partitioned into several fragments: system maintains several identical replicas of each such fragment

Distributed Data Storage

- Division of relation r into fragments r_1, r_2, …, r_n which contain sufficient information to *reconstruct* relation r
- **Horizontal fragmentation**: each tuple of *r* is assigned to one or more "closest" sites
  - Tuples split by **region**
- **Vertical fragmentation**: the schema for relation *r* is split into several smaller schemas, or certain tables are stored in certain sites
  - Relations split by **purpose**

Distributed Data Storage

- Advantages of Fragmentation
- Horizontal:
  - allows parallel processing on fragments of a relation
  - allows a relation to be split so that tuples are located where they are most frequently accessed
- Vertical:
  - allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed
  - tuple-id attribute allows efficient joining of vertical fragments
  - allows parallel processing on a relation

Trading Consistency for Availability

- It is challenging to have a good distributed database system
  - Threatens the **ACID properties** we spoke of earlier
  - Atomicity – can send a transaction to one CPU
  - Isolation – difficult! How to do that with multiple CPUs?
  - Durability – perhaps each can have own form of durability?
  - Consistency - …?!

Trading Consistency for Availability

- Consistency in Databases (ACID):
  - Database has a set of integrity constraints
  - A consistent database state is one where all integrity constraints are satisfied
  - Each transaction run individually on a consistent database state must leave the database in a consistent state
- Consistency in distributed systems with replication
  - Strong consistency: a schedule with read and write operations on a *replicated* object should give results and final state equivalent to some schedule on a *single copy* of the object, with order of operations from a single site preserved
  - Weak consistency (several forms)

CAP Theorem

- Developed about 12 years ago to describe the change in moving from ACID and *single centralized systems* to *distributed systems*

- **C**onsistency (all copies have same value)
- **A**vailability (system can run even if parts have failed)
  - Via replication
- **P**artitions (network can break into two or more parts, each with active systems that can't talk to other parts)
  - If network experiences problems it can handle it

CAP Theorem

- Brewer's CAP "Theorem": You can have at most two of these three properties for any system
- Very large systems will partition at some point

→Choose one of consistency or availability

  - Traditional database choose *consistency*
  - Most Web applications choose *availability*
    - Except for specific (important) parts such as order processing

CAP Theorem

- C/P
  - Query executed on one site (for insert/update/delete)
  - Query is passed to all other sites so they all execute that query
- A/P
  - Each system acting independently – if network goes down or if one system dies… don't care. Just executes queries that comes its way → inconsistent DBs! How to reconcile?
- C/A
  - Each site treating itself as its own system

ACID to BASE

- Nowadays you **scale** C/A/P – can't have all 3 perfectly but don't have to completely give up any one of them
- Instead of ACID properties we now have **BASE** properties
- "**B**asically **A**vailable, **S**oft state, **E**ventually consistent"
- Soft state – DBs may not all be in the same state at the same time – there is fluctuation
- Much more relaxed properties

Facebook Example

- Consider the Facebook problem
  - **Availability** is the number one concern
  - Partitions will occur
  - Thus consistency might be sacrificed sometimes
    - E.g. for chat
  - Later versions of data overwrite earlier versions when systems re-sync
  - Give up availability sometimes
    - E.g. for photos
    - Based on how often you've looked at photos, older photos stored in slow disks with low power – slow access but that's OK

Relational…?

- *Not everything has to be relational!*
- One of the things that has changed, particularly with distributed databases (because it is easier to break the data up) is the idea of a ***NoSQL database***
- Effectively the XML data store which we talked about
  - Idea of key-value store DB
  - Not everything needs relational system, all that is needed is a key and the values that go with that key
- Works well in distributed systems because only keeping up with single value pairs – they scale well

Example of Distributed DB System

- **Amazon DynamoDB**
- [Video] ~ 2 minutes

- Video mentions a lot of topics that we've talked about in class