CS286: Database Systems

1 Lecture 3—9/4/2014

1.1 R*

- Assumptions:
  - There are administrative causes behind distributed data
  - Network: unreliable transport, in-order, packets are intact
  - Independent node failure
  - Slow-ish network
- Research goals:
  - "Site autonomy": No centralized state or control
    * Data you touch should determine the sites you talk to
    * "Distributed system is a system that fails because a machine you've never heard of fails"
    * Load sharing and decentralization
    * Less communication
    * Harder to coordinate data consistency
    * More network connections beyond hub and spoke
    * Metadata management is harder
  - Location transparency → emulate a centralized DB
  - Don't assume much about the network or OS
- Highlights:
  - Query optimizer cost modeling
  - Data layouts → horizontal partitioning
  - Replication
  - Distribution
  - Query compilation—unclear as to balance between compilation overhead and work saving
  - Spent a lot of time talking about 2PC → presumed commit

1.2 Gamma

- Assumptions:
  - Fast interconnect—hypercube, more network bandwidth than aggregate disk bandwidth
  - Shared nothing—no disk or memory sharing
- Research goals:
  - Scale

- Highlights:
  - Parallel hybrid-hash join
  - Chained declustering
- Assess:
  - Linear speedup + scale-up
  - Superlinear speedup due to minimized seek count at scale

2 Lecture 4—9/9/2014

- ACID
  - Consistency is not what we typically think
  - Distributed systems: data has a consistent value across sites
  - Databases: data meets contract when transaction completes
- Serializability mathematically gives atomicity and isolation
- Logging gives atomicity and durability
- Ordering:
  - Determines outcome (unless operations are not associative and commutative)
  - Some things are commutable/associable
  - Ordering must be equivalent to some serializable order
  - Implicitly, this provides an API—people don't *need* to reason about concurrency
- What is storage?
  - *Spacial-temporal rendezvous makes everything work!!!!*
- Want to avoid/undo conflicts in space and time
  - *Space:* Shared names
  - *Time:* Ordering
- 2PL: Provides a conflict serialized schedule
  - Ordered by race for locks
  - Ordered by the end of the first phase ("lock point")
- Multi-version timestamp ordering
  - Every transaction gets a timestamp—this is the only synchronization point
  - For every object:
    - Writes generate a new version for an object
    - Reads annotate the version for the object

3 Lecture 5—9/11/2014

- Good graphs:
  - Crossover points
  - Non-monotonicity
  - Good breadth of X
  - Smooth $\rightarrow$ variance was accounted for
- Infinite resources:
  - Why run infinite resources? Many people assumed infinite resources in their papers.
  - OCC wins because it allows higher parallelism, at the cost of restarting transactions
  - Blocking (2PL) performs well at start, low at the end. Why?
    * Deadlock starts to cause performance to fail
    * Lock contention starts to cause transactions to get in each other's way
    * Locking is a feedback loop—it lengthens transaction time
- Takeaways:
  - MPL is a control variable—choose your infrastructure for your system
- When do we have "infinite" resources?
  - When we have user interaction (Computer $\gg$ human)
  - Vastly overprovisioned compute
  - Work is not going on inside the serving infrastructure (e.g., work is done by clients)

3.1 What happens when you go distributed?

- Why go distributed?
  - Capacity (storage and throughput)
  - Low latency (tolerance)
  - Fault tolerance (durability vs. availability)
- Techniques
  - Sharding—split dataset across many nodes
  - Replication

4 Lecture 6—9/16/2014

- You need replication $\rightarrow$ resilience to failure
- Tradeoff between replication and performance
- NoSQL:
  - Typically, a key-value store (data/programming model)
  - Typically distributed and sharded/partitioned
  - Usually weaker consistency model
  - No transactions/weak isolation model
  - "Not MySQL" $\rightarrow$ lots of work at AOL/etc. with MySQL on `memcached`

  - Typically OSS, not enterprise
  - "Scalable", especially incremental scale → improves organization/administration/ops
  - "Evaporation" of the DBA

- Motivations:
  - Bayou: I want to operate when disconnected
  - Dynamo: Nodes gonna fail

- CAP theorem: if partitions occur, then we can either have consistency or availability
  - Availability: As long as a client can access a server, I can access data (concurrent operations don't need to communicate)
  - Consistency: "linearizable registers" → if I make a write, you can read my write

5 Lecture 7—9/18/2014

- In traditional database, have disk page with tuples stored at continuous offsets.
  - Pointers ("slots") are at end of page and point back to tuples.
  - Can then compress and compact by looking at slot pointers.
  - Fixed length fields stored in tuples
  - Tuples contain pointers to variable length fields

- What changed between 1980 and 2010?
  - CPUs 10,000× faster
  - Disk BW grew 100×
  - Disk seek time improved 10×

- Specifically, gulf between disk performance and processor performance grew

- Research methodology: if area is fairly static, change parameters and see what you can do

- MonetDB
  - Vector/block processing:

- Traditional iterator processing model:
  - Build a tree of operators that run on top of iterators
  - Algorithms have init method (set up state), get next (give me a tuple), and close operators
  - "Pull" model → data and control flow are coupled

- "Late materialization:" query optimizer should defer reading columns until as late as it can

- "Invisible joins:" joins that batch reordering
  - Semijoin: Filter R for all items that have a match in S

- Database cracking: opportunistically reorder blocks in order to improve performance