

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 → 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 → 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” → 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

6 Lecture 8—9/23/2014

- Pre-relational data models:
 - Network: objects + pointers
 - Hierarchical: nested sets
- Then, *the Relational Revolution*
- But, persistent questioning:
 - In the 80's, nested relational
 - Object-Oriented Database → 80's/90's
 - And then... XML
- Why flatten into relations:
 - Space efficient
 - Update/delete/inserts require work/care
 - Simple model and language
 - Data independence: physical and logical independence
- When is the relational model a pain?
 - Joins are expensive
 - Must know schema ahead of time
 - Read-only workloads are expensive
 - Programming language state
- Engineering versus “Found Structure” → *bricolage*
 - Engineering → collaboration and communication
 - Found structure → exploratory data analytics
- XML database history:
 - WWW + search ate DB lunch
 - Let's query the internets!
 - $\text{DB} \times \text{WWW} \times \text{markup language people} = \text{pandemonium}$
 - 4 data models, 3 query languages, mostly overlap...
- DB vendors kept up with the pace of research
- How to encode XML:
 - Native
 - Shred to relationable tables
 - Relational encoding of trees
 - Path/value encoding
 - Hybrids

7 Lecture 9—9/25/2014

- Few ways to provide/describe isolation; e.g., for a KV store:
 - 2PL → mechanism
 - Avoid anomalies (no lost update, no dirty/fuzzy read) → anomaly prevention
 - Draw conflicts into graph (graph should have no cycle) → graph formalism
 - Every history is view equivalent to a serial history → equivalence formalism
 - If every program preserves an invariant, then every execution will preserve the invariant → integrity
- Full isolation is expensive; let's go for weaker models
- These descriptions don't imply equivalence:
 - Can provide SG without cycle with OCC, $OCC \neq 2PL$
 - Snapshot isolation → no anomalies, but does not preserve invariants
- Strong vs. weak isolation: weaker implies more anomalies allowed, or more executions allowed
- Conditions of *reality*:
 - Phantom: occurs when your program has a complex predicate, and when table modifications are allowed (modifications can change predicate selections)
 - * Can solve with predicate locks (but no one does that)
 - * Can solve with locks on indices, next key locks, etc...
 - *Repeatable read*: in SQL, an isolation level where you can have phantoms, but everything else is OK (IBM defines as full isolation, hence confusion...)
 - *Fuzzy read*: short read locks, I read a single item multiple times and can see different values
 - Statement-level atomicity: hold short read locks for the whole time I'm evaluating a statement
 - `select ... for update`: hint that I'll grab locks later
- Isolation levels:
 0. Short duration write locks
 1. Commit duration write locks
 2. Commit duration write locks, short read locks
 3. Full serializability
- Snapshot isolation: a good demonstration that you can beat anomalies (no lost update, no dirty read, no fuzzy read, no phantoms), but not provide serializability

8 Lecture 10—9/30/2014

- Implementation:
 - Concurrency control: 1 transaction at a time
 - All transactions are stored procedures; not interactive
 - Partition your workload (?)
 - Hot replicas: in relational world, replication is a backup strategy, not a runtime strategy
 - Weak consistency “*may*” be interesting

8.1 Hekaton

- Built from ground-up to *fit* into SQL Server:
 - Limits design space
 - Must coexist with old systems (e.g., here, must have commit time in old system)
- Much else is in other papers

9 Lecture 11—10/2/2014

- This guy fellow thinks that people aren't thinking enough about parametrized queries
 - OLTP is pre-canned; OLAP isn't exactly
 - E.g., web forms
- Query optimization, a primer:
 - So, we want to select some keys from a table
 - There are many ways to execute most queries
 - * Joins are associative and commutative, so can be reordered
 - * Many different algorithms for joins
 - If we have value distribution statistics, we can estimate the cost of a query plan
 - Problem is NP-hard, exponential in the number of tables
 - Getting a plan wrong can cause orders of magnitude performance differences
 - * But, there are often many OK plans

9.1 Adaptive Query Processing

- Eddies:
 - NOW-sort
 - Led to Inktomi
 - Led to River: adaptive parallel streaming/shuffle
 - Control: approximate query processing
 - Can structure work in many ways:
 - * Want to optimize for the expected amount of work

9.2 Robust Query Processing

- How can we make an optimizer more robust?
 - Add a feedback loop and adapt
 - Make query plan “pretty good” even if your estimates are bad—you don't want the *best* plan, you want a good plan that has a very low chance of morphing into an *bad* plan
- Why?
 - Bound your worst-case performance (don't break)
 - If you are robust to a metric, then you can use a cheap approach for collecting that metric
 - Predictable across versions
- Approaches:

- L.C.: You materialized a full intermediate result
- L.C. Eager M.: You add a materialization and look at the full intermediate approach
- E.C.W.C.: Watch a pipe and remember the record IDs, do an anti-join
- E.C.W.B.: Keep a buffer, so not as big as a materialization
- Need dynamic programming algo to search the space

10 Lecture 12—10/7/2014

- Query optimization:
 - We want to find the plan space
 - Estimate the cost!
 - And search (prune) that space
- Should separate a plan space into a logical and physical space
 - Logical space: logical operations
 - Physical space: access operations and disk layout (implementation)
- If you put a rule, you can make things blow up badly
- What happens if an optimizer doesn't exist? You need to take the optimizer that existed before, and modify it for your new optimized implementation.
- Cost estimation:
 - Want to put together summary statistics
 - Specifically, we probably want to generate selectivity estimation for joins
 - Cardinality estimation for group-by's
 - Summary statistics are computed from the data, selectivity/cardinality estimation from summary statistics
 - Cost estimation = statistics
- Logical plan space: SQL, XQuery, Pig, Datalog
- Parallelism:
 - How do I do a parallel database? How do I do joins/etc?
 - How do I allocate resources?
- Physical strategies need to have a logical mapping
- Can define rules mapping logical to logical, physical to logical
- Search strategies:
 - Bottom-up: data oriented, forward chaining
 - Top-down: goal oriented, backward chaining

11 Lecture 13—10/14/2014

- “This is not computer science, it is *sociology*!”
- Ecosystem is important → how does industry impact research?
- **TL; DR:**
 - MapReduce: Introduced mid-query fault tolerance
 - DryadLINQ: “We used cool language shit that we had at Microsoft.”
- Early 2000’s: What happened?
 - “*The Cloud*” → massive user bases
 - Scale-out data:
 1. Data as a source of value → early offers for FB were based on value of network/data
 2. Data has enormous scale and “schema-on-use” → disk is cheap, don’t toss data away!
 - Boom (late ’90’s) → bust (2001) → boom (2004) → bust (GFC) → now
 - Revenue model changed: money is not made on software sales, rather, it is made on advertising
 - Also, RE: revenue: switched from *sales* to *subscription* → shift from CapEx to OpEx
 - Aside: internet company developers (not enterprise developers) are driving design
 - Custom SW is not sufficiently valuable relative to other things, so give it away:
 1. Network is value, not SW
 2. Ops side arises: even if you *had* FB’s software, could you run Facebook?
 3. Open source software attracts top talent
 4. $E[\text{engineer}] = 2$ years: if everyone is going to leave in two years, don’t have proprietary software
 5. “OOO and software reuse is dead”
 - As a result, *buy vs. build* decision is completely different
- Huge implications on the DB community:
 - *Oracle as a punching bag* → “All the bad things about Oracle must have arrived out of bad DB research.”
 - * MySQL was also a dichotomous punching bag
 - * Where was Postgres? Cool software, but performance is worse than MySQL.
 - We’ve changed the optimization function: programmer productivity is #1 priority
 - * Initially, we must harness parallelism for developers!
 - First, Google MapReduce
 - And, Hadoop @ Yahoo
 - Also, Scope at MSFT
 - Since 2004:
 1. Parallel DBs were overpriced and slow
 2. A lot of the MR workloads were SQL-ish:
 - * Hive
 - * Pig
 3. Clear OSS opportunity:
 - * Cloudera
 - * Hortonworks
 - * MapR
 - * ...
 4. Google sponsors *significant* education push around MapReduce
 5. Hybrid workflows rise up

11.1 Questions

1. Who should use the Hadoop ecosystem?

- HDFS: Is a pretty useful storage system.
- Everything else:
 - If you want to program MapReduce (e.g., custom UDFs), MR is a better platform...
 - What do you do once you get a result? With DBs, you can materialize it. With MR, you have many ways to go from there. E.g., *opportunistic vs. agile* pipelines. Assumes sophisticated developers are using the system.
 - How much data should you have? If you've got a program that'll scale across multiple machines, that's a good fit for Hadoop.
 - If you're using Hadoop, your jobs should probably be long running, and not terribly time critical.
 - Machine learning at scale?
 - How does Spark come in? Logical vs. physical fault tolerance is a nice thing to fall back on.

2. What should they pay?

11.2 DryadLINQ and .NET

- LINQ is cool.
- Unfortunately, it is in .NET...
- What *is* LINQ? → a functional DSL for collections
- What *in* .NET are they interested in? → Probably libraries for math/collections.
- Debugging: provide a single node implementation that can be run locally.
- Also have PLINQ → parallel LINQ on a single system; so node + processor parallelism