CMSC724: Parallel/Distributed Databases[1]; MapReduce

Amol Deshpande

University of Maryland, College Park

March 15, 2011

[1] Based on notes from Joe Hellerstein

Outline

1. Parallel Databases
2. Map Reduce
3. Friends or Foes?
4. Pig Latin
5. Distributed Data Stores/Key-Value Stores

Database Machines

- Proposals in late 70's, early 80's for specialized hardware
  - Database Machines: An idea whose time has passed ?
    - Boral, DeWitt, 1983
  - Processor-per-track:
    - Database specific storage
    - Evaluate selections directly on the CPs etc...
  - Processor-per-head
    - Need parallel readout
    - Combined with indexes, gives good performance
  - Off-the-track
    - Something like a shared-memory machine
    - Used special DB-specific processors

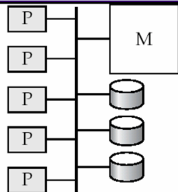
Database Machines

- Didn't work
  - Processor-per-track:
    - Not cost-effective
    - Based on fixed-head disks, or solid-state storage
  - Processor-per-head
    - Parallel readouts are hard to do ??
  - Off-the-track
    - Disk bandwidth is actually decreasing ??? (maybe true then)

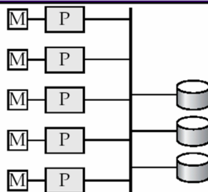
Database Machines

- Didn't work
  - Processor-per-track:
    - Not cost-effective
    - Based on fixed-head disks, or solid-state storage
  - Processor-per-head
    - Parallel readouts are hard to do ??
  - Off-the-track
    - Disk bandwidth is actually decreasing ??? (maybe true then)
  - Generally, specialized hardware is hard to make work
    - Too expensive, slow-to-evolve, requires a tool set
    - Doesn't help too much with sorts/joins anyway
    - General-purpose hardware improves faster
    - Soon catches up
- IDISKs ? (late 90's)

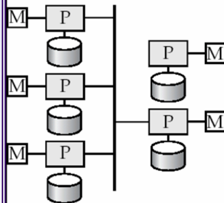
Types of Parallelism



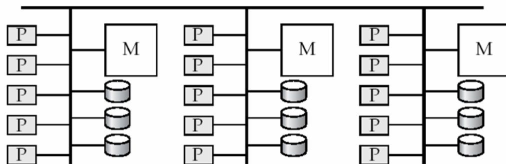
(a) shared memory



(b) shared disk



(c) shared nothing



(d) hierarchical

Types of Parallelism

- Shared memory
  - One of the last remaining “cash cows”
  - Direct mapping from uni-processor
  - Data structures shared between processors
    - Process models extend naturally: processes or threads assigned to different processes
    - Cache-coherency can be issues: typically left to the hardware
  - Resurgence as multi-core
    - Separate caches, but usually not large enough to call shared-nothing

Types of Parallelism

- Shared memory
  - One of the last remaining “cash cows”
  - Direct mapping from uni-processor
  - Data structures shared between processors
    - Process models extend naturally: processes or threads assigned to different processes
    - Cache-coherency can be issues: typically left to the hardware
  - Resurgence as multi-core
    - Separate caches, but usually not large enough to call shared-nothing
- Shared disk
  - Increasingly common because of SAN (storage area network)
  - Better failure behavior (since data still available)
  - Distributed lock managers, cache-coherency etc...

Types of Parallelism

- Shared nothing
  - Perhaps most common and most scalable
  - Horizontal data partitioning
    - Good partitioning schemes essential
    - More burden on the DBA
  - Query processing and optimization challenging
  - Partial failures
    - Option 1: Can skip the data on the machine that failed
    - Option 2: Bring down the whole system (“Data skip”)
    - Option 3: Redundancy (usually “hot standby”)

Types of Parallelism

- Shared nothing
  - Perhaps most common and most scalable
  - Horizontal data partitioning
    - Good partitioning schemes essential
    - More burden on the DBA
  - Query processing and optimization challenging
  - Partial failures
    - Option 1: Can skip the data on the machine that failed
    - Option 2: Bring down the whole system ("Data skip")
    - Option 3: Redundancy (usually "hot standby")
- NUMA (Non-uniform Memory Architecture)
  - Processors have different access costs for different parts of memory
  - Option 1: ignore non-uniformity (treat as shared-memory)
  - Option 2: minimize cross-processor access to memory (treat as shared-nothing/shared-disk)

Concepts...

- Database operations “embarrassingly parallel”
- Speedup vs Scaleup
  - Speedup: old time/new time
  - Scaleup: how many more queries/how much larger query can you solve

Concepts...

- Database operations “embarrassingly parallel”
- Speedup vs Scaleup
  - Speedup: old time/new time
  - Scaleup: how many more queries/how much larger query can you solve
- Types of parallelism:
  - Pipelined
    - Each “operator” on a different processor
    - Easier to setup, but low parallelism
  - Partitioned
    - Split relations horizontally, replicate the operators
    - Exploits all processors, but much harder to setup
    - Optimization messy: Need to make decisions about how to split etc..

Concepts...

- Storage: Round-robin vs Hash-based vs Range-based
  - Bubba used “heat” to partition
- Barriers to linearity
  - Startup overheads (remember Amdahl’s law)
  - Interference
    - Communication overhead, waiting on queues etc..
    - If the interference just 1%, the maximum speedup < 37
  - Skew
    - Partitioning may turn out to be non-uniform (common cause: duplicates)
    - Solution 1: Carefully design hash functions
    - Solution 2: Use a very fine-grained partitioning function, and adjust the assignment of partitions to processors

Concepts...

- Autonomy ?
  - Not autonomous, centralized decision-making
- Concurrency/locking ?
  - Two-phase locking (2PL)
  - Probably two-phase commit (2PC)
  - Centralized deadlock detection

Concepts...

- Autonomy ?
  - Not autonomous, centralized decision-making
- Concurrency/locking ?
  - Two-phase locking (2PL)
  - Probably two-phase commit (2PC)
  - Centralized deadlock detection
- Recovery
  - ARIES-based (similar to centralized)
- Failures ?
  - Chained declustering
    - Each relation partition replicated on one other site
    - Many similarities to RAID

Query execution

- New operators: Hash joins, replicate-all strategy etc...
- Left-deep trees good for pipelining
  - Beware: Some people (eg. DeWitt) calls these “right-deep”
- Selections: Indexes etc (individual at each site)
- Joins: Hash joins, replicate-all
  - Symmetric hash join operator

Query execution

- New operators: Hash joins, replicate-all strategy etc...
- Left-deep trees good for pipelining
  - Beware: Some people (eg. DeWitt) calls these “right-deep”
- Selections: Indexes etc (individual at each site)
- Joins: Hash joins, replicate-all
  - Symmetric hash join operator
- Sorting
  - Sort partitions in parallel, merge is computationally trivial
- Aggregation: Do separately, and combine
  - Can all aggregates be done like this ?

Engineering issues

- Re-use existing code
- Gamma: Split/merge operators
- Volcano:
  - Exchange operator
  - Allows arbitrary interleavings
  - An operator can directly call another operator (within the process), across processes or across network

Engineering issues

- Re-use existing code
- Gamma: Split/merge operators
- Volcano:
  - Exchange operator
  - Allows arbitrary interleavings
  - An operator can directly call another operator (within the process), across processes or across network
  - Data-driven vs Demand-driven dataflows (pull vs push)
  - Semaphores used for controlling producer vs consumer rates (flow control)
  - Also, rule-based extensible optimizer (became the basis for MSSQL Server Optimizer framework)
  - Later work by Mehul Shah on extending Exchange

Query optimization

- Much larger plan space
  - Need to worry about partitioning, different indexes at different sites..
- Cost metric:
  - Communication cost ?
  - Response time is not a nice metric
    - Conflicts with traditional total work metric
    - May prefer to optimize for total work, and handle more queries instead

Query optimization

- Much larger plan space
  - Need to worry about partitioning, different indexes at different sites..
- Cost metric:
  - Communication cost ?
  - Response time is not a nice metric
    - Conflicts with traditional total work metric
    - May prefer to optimize for total work, and handle more queries instead
- 2-phase optimization (XPRS)
  - Phase 1: Optimize for total work
  - Phase 2: Parallelize the plan
- Load balancing/skew: Recursive partitioning for hash joins

Distributed Databases

- R* etc...
- Communication costs much higher
  - Use semi-joins/bloom filters etc
- More autonomy per machine
  - Typically different administrative domains
  - Different schemas, even different machines
- Federated ?
  - Mariposa – Used the economic paradigm
  - For query processing, replication etc.

Outline

1 Parallel Databases

2 Map Reduce

3 Friends or Foes?

4 Pig Latin

5 Distributed Data Stores/Key-Value Stores

Discussion/Thoughts

- From Curt Monash's Blog, especially Mapreduce part
- New key industry players in large-scale data analysis/data warehousing
  - Netezza, Aster, Greenplum, Vertica (Stonebraker) etc...
  - Along with Oracle (Exadata), DB2, Teradata, many more
  - Many have a few customers each

Discussion/Thoughts

- From Curt Monash's Blog, especially Mapreduce part
- New key industry players in large-scale data analysis/data warehousing
  - Netezza, Aster, Greenplum, Vertica (Stonebraker) etc...
  - Along with Oracle (Exadata), DB2, Teradata, many more
  - Many have a few customers each
- Netezza, Aster, Greenplum offer Mapreduce functionality by now
  - Aster: Highly parallel data warehousing solution – very nice whitepaper on Mapreduce
  - We will see some syntax later
- SIGMOD 2009 paper: A Comparison of Large-scale Data Analysis

MapReduce

- Goal: efficient parallelization of various tasks across 1000's of machines without the user having to worry about the details such as:
  - How to parallelize
  - How to distribute the data
  - How to handle failures

MapReduce

- Goal: efficient parallelization of various tasks across 1000's of machines without the user having to worry about the details such as:
  - How to parallelize
  - How to distribute the data
  - How to handle failures
- Basic Idea:
  - If you force programs to be written using two primitives (*map* and *reduce*), parallelism can be gotten for free
    - Replace: map-reduce with SQL, parallelism with speed/ease-of-use
  - More programs than you might think can be written this way

MapReduce: Applications

- From Nice Overview by Curt Monash
- Three major classes:
  - Text tokenization, indexing, and search
  - Creation of other kinds of data structures (e.g., graphs)
  - Data mining and machine learning
- See this blog post for a long list of applications
- Or See Hadoop List
- For Machine Learning algorithms, see MAHOUT

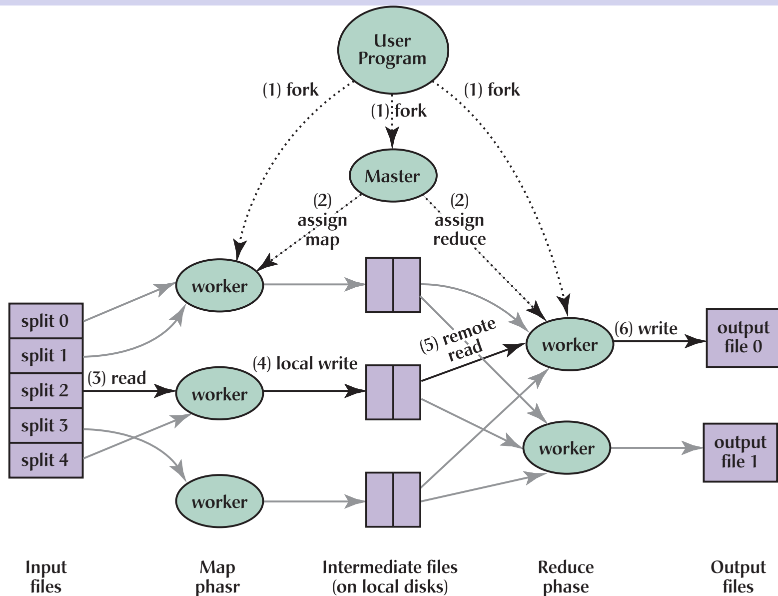
Mapreduce

- Users needs to write two key functions:
  - Map: generate a set of (key, value) pairs
  - Reduce: group the pairs by *key's* and combine them (GROUP BY)
- Borrowed from Lisp

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Mapreduce: Execution Overview



Mapreduce: Implementation

- A master for each tasks, assigns tasks to workers
- Data transfers using the file system (by passing file-names)
- Master pings the workers to make sure they are alive
  - If not, reassign the task to some other worker
- Work is divided into a large number of small chunks
  - Similar ideas used in parallel database for handling data skew
- Atomic commits using the file system

Mapreduce: Implementation

- Google File System
  - A distributed, fault-tolerant file system
  - Data divided into blocks of 64MB
  - Each block stored on several machines (typically 3)
- Mapreduce uses the location information to assign work

Mapreduce: Implementation

- Google File System
  - A distributed, fault-tolerant file system
  - Data divided into blocks of 64MB
  - Each block stored on several machines (typically 3)
- Mapreduce uses the location information to assign work
- Many other optimizations
  - Backup tasks to handle “straggler”
  - Control over partitioning functions
  - Ability to skip “bad” records

Mapreduce

- Has been used within Google for:
  - Large-scale machine learning problems
  - Clustering problems for Google News etc..
  - Generating summary reports
  - Large-scale graph computations
- Also replaced the original tools for large-scale indexing
  - ie., generating the inverted indexes etc.
  - runs as a sequence of 5 to 10 Mapreduce operations

Mapreduce: Thoughts

- Hadoop
  - Open-source implementation of Mapreduce
    - Has support for both the distributed file system and Mapreduce
  - University of Maryland is a major player in this
    - Jimmy Lin is running several projects related to NLP
    - If you want to play with this, let me know
  - IBM, Yahoo, other major players interested in this

Mapreduce: Thoughts

- Hadoop
  - Open-source implementation of Mapreduce
    - Has support for both the distributed file system and Mapreduce
  - University of Maryland is a major player in this
    - Jimmy Lin is running several projects related to NLP
    - If you want to play with this, let me know
  - IBM, Yahoo, other major players interested in this
- Cloud Computing
  - Somewhat vague term, but quite related

Pig Project @ Yahoo

- Generalization of Mapreduce
- An IDE for developing large-scale data analysis tasks
  - Appears a simplification of SQL (at this point at least)

Operators

LOAD | GROUP | COGROUP | FILTER | FOREACH | ORDER

= LOAD USING Default AS ( )

Generate Query

Program		Output
visits = LOAD 'visits.txt' AS (user, url, time);	visits:	(Amy, cnn.com, 8am) (Amy, frogs.com, 9am) (Fred, snails.com, 11am)
pages = LOAD 'pages.txt' AS (url, pagerank);	pages:	(cnn.com, 0.8) (frogs.com, 0.8) (snails.com, 0.3)
v_p = JOIN visits BY url, pages BY url;	v_p:	(Amy, cnn.com, 8am, cnn.com, 0.8) (Amy, frogs.com, 9am, frogs.com, 0.8) (Fred, snails.com, 11am, snails.com, 0.3)
users = GROUP v_p BY user;	users:	(Amy, { (Amy, cnn.com, 8am, cnn.com, 0.8), (Amy, frogs.com, 9am, frogs.com, 0.8) }) (Fred, { (Fred, snails.com, 11am, snails.com, 0.3) })
useravg = FOREACH users GENERATE group, AVG(v_p.pagerank) AS avgpr;	useravg:	(Amy, 0.8) (Fred, 0.3)
answer = FILTER useravg BY avgpr > '0.5';	answer:	(Amy, 0.8)

Figure 4: Pig Pen screenshot; displayed program finds users who tend to visit high-pagerank pages.

Mapreduce + Databases: Thoughts

- Abstract ideas have been known before
  - See Mapreduce: A Major Step Backwards; DeWitt and Stonebraker
  - Can be implemented using user-defined aggregates in PostgreSQL quite easily
  - Top-down, declarative design
    - The user specifies what is to be done, not how many machines to use etc...

Mapreduce + Databases: Thoughts

- Abstract ideas have been known before
  - See Mapreduce: A Major Step Backwards; DeWitt and Stonebraker
  - Can be implemented using user-defined aggregates in PostgreSQL quite easily
  - Top-down, declarative design
    - The user specifies what is to be done, not how many machines to use etc...
- The strength comes from simplicity and ease of use
  - No database system can come close to the performance of Mapreduce infrastructure
  - RDBMSs can't scale to that degree, are not as fault-tolerant etc...
    - Again: this is mainly because of ACID
    - Databases were designed to support it
    - Most of the Google tasks don't worry about that

Mapreduce + Databases: Thoughts

- Mapreduce is very good at what it was designed for
  - But may not be ideal for more complex tasks
    - E.g. no notion of “Query Optimization” (in particular, operator order optimization)
    - The sequence of Mapreduce tasks makes it procedural within a single machine
  - Joins are tricky to do
    - Mapreduce assumes a single input

Mapreduce + Databases: Thoughts

- Mapreduce is very good at what it was designed for
  - But may not be ideal for more complex tasks
    - E.g. no notion of “Query Optimization” (in particular, operator order optimization)
    - The sequence of Mapreduce tasks makes it procedural within a single machine
  - Joins are tricky to do
    - Mapreduce assumes a single input
- Trying to force use of Mapreduce may not be the best option
- However, much work in recent years on extending the functionality
  - See Pig project at Yahoo, Map-reduce-merge etc..

Mapreduce + Databases: Aster

- From the Aster White Paper
- Write two functions using your favorite language
  - *Map* and *Reduce*
- Use them directly in SQL
- Aster will take care of pipelining, parallel execution etc..

```
select token, sum(occurrences) as globalOccurrence
from map ( ON
  select word, count(*) as occurrences
  from WordOccurrences
  group by word )
group by token;
```

Outline

1. Parallel Databases
2. Map Reduce
3. Friends or Foes?
4. Pig Latin
5. Distributed Data Stores/Key-Value Stores

MR: A Major Step Backwards?

- An (in)famous blog post by DeWitt and Stonebraker
  - Discussed why MR wasn't a new idea, and how most of the concepts were developed in parallel databases a long time ago
    - Still an interesting read
- Later changed their position quite a bit
  - Result: this paper

Key points

- MR very good at extract-transform-load tasks
  - Experiments indicate loading data is much slower in databases
- But not good at tasks that are best suited for DBMSes
- UDF functionality in databases can cover many of other intended MR uses

Possible applications of MR

- (According to the authors)
- ETL and "read once" data sets
    - ETL has typically been distinct from databases
- Complex analytics
- Semi-structured data
- Quick and dirty analyses
    - MR has much shorter latency with such tasks

Architecture differences

- In the representative implementations
- Repetitive record parsing
  - Databases convert data into an internal format
- Compression
- Pipelining vs Materialization
  - Addressed by "Mapreduce Online" line of work
- Another important issue
  - Parallel Databases are very very expensive

Outline

1. Parallel Databases
2. Map Reduce
3. Friends or Foes?
4. Pig Latin
5. Distributed Data Stores/Key-Value Stores

Overview

- Something that fits in between SQL and MapReduce
  - To make it easy for programmers to write procedural, non-SQL code
- Open source, on top of Hadoop
- No transactions – read-only analysis queries
- Supports nested data model (i.e., not in 1NF)
  - Allows sets/maps as fields
  - Interestingly: need this for GROUP operator
- UDFs written in Java

Example

Example 1. *Suppose we have a table* `urls: (url, category, pagerank)`. *The following is a simple SQL query that finds, for each sufficiently large category, the average pagerank of high-pagerank urls in that category.*

```
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 10^6
```

An equivalent Pig Latin program is the following. (Pig Latin is described in detail in Section 3; a detailed understanding of the language is not required to follow this example.)

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls)>10^6;
output = FOREACH big_groups GENERATE
            category, AVG(good_urls.pagerank);
```

Debugging

- Basic Idea: Show the results of the operations on a small sample of the data
  - Technical challenges: how to make sure that these are actually useful?
  - e.g., Joins: if you take random samples of the relations, the result may contain nothing
    - Need to take biased samples
  - Pig Pen: a visual debugging environment

Pig Pen

Operators

LOAD | GROUP | COGROUP | FILTER | FOREACH | ORDER

= LOAD USING Default AS ( )

Generate Query

visits = LOAD 'visits.txt' AS (user, url, time);	visits: (Amy, cnn.com, 8am) (Amy, frogs.com, 9am) (Fred, snails.com, 11am)
pages = LOAD 'pages.txt' AS (url, pagerank);	pages: (cnn.com, 0.8) (frogs.com, 0.8) (snails.com, 0.3)
v_p = JOIN visits BY url, pages BY url;	v_p: (Amy, cnn.com, 8am, cnn.com, 0.8) (Amy, frogs.com, 9am, frogs.com, 0.8) (Fred, snails.com, 11am, snails.com, 0.3)
users = GROUP v_p BY user;	users: (Amy, { (Amy, cnn.com, 8am, cnn.com, 0.8), (Amy, frogs.com, 9am, frogs.com, 0.8) }) (Fred, { (Fred, snails.com, 11am, snails.com, 0.3) })
useravg = FOREACH users GENERATE group, AVG(v_p.pagerank) AS avgpr;	useravg: (Amy, 0.8) (Fred, 0.3)
answer = FILTER useravg BY avgpr > '0.5';	answer: (Amy, 0.8)

Figure 4: Pig Pen screenshot; displayed program finds users who tend to visit high-pagerank pages.

Outline

1 Parallel Databases

2 Map Reduce

3 Friends or Foes?

4 Pig Latin

5 Distributed Data Stores/Key-Value Stores

Motivation/Issues

- Geared toward the second use case
  - Large-scale web application that need real-time access
    - With a few ms latencies (e.g., Facebook == 4ms for reads)
- Problems with using databases
  - Too slow
  - Don't need ACID properties (??)
  - Too expensive

Systems

- Interesting numbers (http://highscalability.com)
  - Twitter: 177M tweets sent on 3/1/2011 (nothing special about the date), 572,000 accounts added on 3/12/2011
  - Dropbox: 1M files saved every 15 mins
  - Stackoverflow: 3M page views a day (Redis for caching)
  - Wordnik: 10 million API Requests a Day on MongoDB and Scala
  - Mollom: Killing Over 373 Million Spams at 100 Requests Per Second (Cassandra)
  - Facebook's New Real-time Messaging System: HBase to Store 135+ Billion Messages a Month
    - Interestingly: decided to move away from Cassandra because not happy with the eventual consistency model
  - Reddit: 270 Million Page Views a Month in May 2010 (Memcache)

Motivation/Issues: CAP

- Distributed systems
- CAP theorem: can have two of: consistency, availability, and tolerance to network partitions
    - Originally a conjecture (Eric Brewer), but made formal later (Gilbert, Lynch, 2002)

Motivation/Issues: CAP

- Distributed systems
- CAP theorem: can have two of: consistency, availability, and tolerance to network partitions
  - Originally a conjecture (Eric Brewer), but made formal later (Gilbert, Lynch, 2002)
- Theorem: *It is impossible in the asynchronous network model to implement a read/write data object that guarantees the following properties:*
  - *Availability*
  - *Atomic consistency in all fair executions (including those in which messages are lost).*
  - In other words, if there is a network partition, we can:
    - Go down (sacrifice availability), or
    - Allow inconsistency

Motivation/Issues: Distributed transactions

- Distributed transactions: If a transaction spans multiple machines, how to do it?
- Correct solution: Two-phase Commit
  - Follow a protocol exchanging multiple rounds of messages
  - Problems: high latency, no tolerance to partitions
  - Three-phase commit solves the latter problem with more rounds of messages
- A simpler solution
  - Send the update to a bunch of geographically distributed machines – pray no meteoroid strikes!
- Generally very hard to provide ACID properties with low latencies
  - Especially for transactions spanning multiple sites

Motivation/Issues: Dealing with replication

- Replicas are needed
- How to keep them updated?
  - Eager/synchronous replication: use two-phase commit across the replicas
    - surely you are joking..
    - Sometimes called *active-active*
  - Lazy/asynchronous replication (*master-slave*)
    - Choose one replica as the *master*
    - Propagate the updates to the *slave* replicas in background
    - Usually done using *log shipping*
    - Results in possibility of stale reads

Eventual Consistency

- Amazon formalized as:
  - *the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value*
  - May lead to out-of-order reads
  - DNS is an example
  - Many key-value stores (including the most popular ones like MongoDB)

Eventual Consistency

- Amazon formalized as:
  - *the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value*
  - May lead to out-of-order reads
  - DNS is an example
  - Many key-value stores (including the most popular ones like MongoDB)
- But many points in between full consistency and eventual consistency with out-of-order reads
  - e.g., *timeline consistency* that PNUTS provides (i.e., no out-of-order reads)
- Same system may support multiple options, under multiple configurations
  - E.g., MongoDB, Cassandra (at least now), PNUTS etc...

Systems

- Numerous systems designed in last 10 years that look very similar
  - Differences often subtle, and if not hard to pin down, hard to understand
  - Often the differences are about the implementations
- Often called key-value stores
  - The main provided functionality is that of a hashtable
- Some earlier solutions
  - Still very popular
  - Memcached + MySQL + Sharding
    - Sharding == partitioning
    - Store data in MySQL – use Memcached to cache the data
    - Memcached not really a database, just a cache
    - All kinds of consistency issues
    - But... very very fast

Systems

- Tokyo, Redis
  - Very efficient key value stores
- BigTable (Google), HBase (Apache open source), Cassandra (original Facebook, open sourced), Voldemort (originally LinkedIn)...
  - At least in original iterations, focused on performance
  - Cassandra later developed more sophisticated *tunable* consistency (maybe others too)
- PNUTS (Yahoo!)
  - Focus on geographically distributed stuff
    - Easier to deal with some issues if we assume everything is a single data center
  - Support tunable consistency for reads: *read-any*, *read-latest* etc..
  - Form of master-slave replication
  - No real support for multi-record transactions

Systems

- MySQL + Memcached: End of an era?
  - *If you look at the early days of this blog, when web scalability was still in its heady bloom of youth, many of the articles had to do with leveraging MySQL and memcached. Exciting times. Shard MySQL to handle high write loads, cache objects in memcached to handle high read loads, and then write a lot of glue code to make it all work together. That was state of the art, that was how it was done. The architecture of many major sites still follow this pattern today, largely because with enough elbow grease, it works.*
  - Digg moved to Cassandra in 2009; LinkedIn to Voldemort
  - Twitter moved to Cassandra recently
    - *.. the rate of growth is accelerating.. a system in place based on shared mysql + memcache .. quickly becoming prohibitively costly (in terms of manpower) to operate.*

Systems

- Megastore (Google)
  - Recent CIDR paper
  - Built on top of BigTable – powers Google App Engine
  - Full ACID using Paxos
    - Using active-active replication/two-phase commit
    - However many details to make this fast
  - Supports notion of *entity groups*
    - e.g., all emails of a user is a single entity group
  - Transactions that span a single entity group are generally fine
  - Transactions that span multiple entity groups would use two-phase commit – not preferred

Systems

- VoltDB (Stonebraker startup)
  - Rethinking OLTP databases altogether
  - Provides full ACID with eager replication
    - Just like Megastore, better if your transaction does not span multiple cores
  - Remove all obstacles to efficiency
    - No locking, no latching, no buffer management, no recovery
  - Give up on "tolerance to network partitions"
    - How often they happen anyway?
  - Very nice talk by Stonebraker
    - Six Urban Myths about SQL
    - Very nice analysis with some counter-points

Systems

- Mongodb
  - Perhaps the poster child of key-value NoSQL stores
  - Very scalable
  - Document-oriented storage
    - JSON-style documents
    - JSON may be becoming more popular than XML as the data interchange format
  - Some form of eventual consistency

Two interesting points

- Non-determinism
  - Why allow any kind of non-determinism in the system?
    - What if we enforce that a transaction that enters the system first should commit first?
    - Would simply certain things
  - Problem: Locking
    - If a transaction is delayed or blocked, other transactions (after it) should be allowed to continue
  - But what if all transactions are extremely short? (See Abadi's VLDB 2010 paper)
- Commutativity
  - Consider: Transaction 1 increases A by 100, Transaction B increases it by 1%
  - They are not commutative
  - This causes problems in distributed settings
  - Disallowing it may simplify things