Introduction to NoSQL Databases

KDD 2013 Big Data Camp

Dean Wampler, Ph.D.
dean@concurrentthought.com
@deanwampler
Concurrent Thought

Contents

Introduction to NoSQL Databases

- Recap of the NoSQL World.
- Growth of the “Interwebs” – Forces that lead to the creation of NoSQL Databases.
- Kinds of NoSQL Databases.
- Conclusions – Is SQL Dead??

A Brief History of Data

For decades, Relational Databases have been the dominant tool for data persistence. There are other models that are also decades old… and predate it.

Hierarchical and Network Model

- Hierarchical:
  - Tree-based, parent-child relationships.
  - 1:N (one parent, N children).
  - Popular in the late 1960’s through the 1970’s.
- Network:
  - M:N (many to many).
  - Formally defined in 1971 at the Conference on Data Systems Languages (CODASYL).
  - Based on set theory.
  - Can have cycles in the network.

Relational Model

- RDBMS – Relational Database Management System.
- *Relational Model* devised by Edgar F. Codd at IBM in 1970.
- A *relation*: a set of tuples (i.e., rows a table) that have the same “attributes” (i.e., columns).
- Rows are *unique* (not true in actual SQL databases…)
- Columns have types and unique names.
- Each row has the same set of columns.
- The sequence of rows and columns are not significant.
- 1 or more columns may be indexed to speed up queries.

Relational Model

- Actual Relational Database Management Systems and SQL depart from Codd’s model in several ways.

Object-Oriented Model

- Emerged in the 90’s with adoption of OO Programming.
- Attempts to unify the programming and persistence models for languages like C++, Java, Smalltalk, etc.
- Supports inheritance.
- Eliminates the *impedence mismatch* required to persist objects to relational tables (inheritance is particularly tricky).

Object-Oriented Model

- Never really displaced RDBMS, even though most IT development is done using OO languages!
  - Performance was a serious problem in early OODBMSs.
  - Database architects and developers tend to be conservative.
  - Using RDBMS continued to be "good enough".

Object/Relational Model

- Adds to the relational model the ability to define new types and specify the unique operations on those types.
- Examples include geospatial data, time-series data (e.g., stock prices), and binary media like images, audio, and video.
- Doesn't support inheritance in the usual OOP sense.
- Most of the big RDBMS vendors offer some O/R capabilities.
- Compromise solution that also emerged in the 90's.

Take Aways

- (Good, appropriate) conservatism means that extraordinary forces are required to change your data management model.
- SQL is an extremely concise, powerful tool for queries!

Requirement(?) for Data Persistence

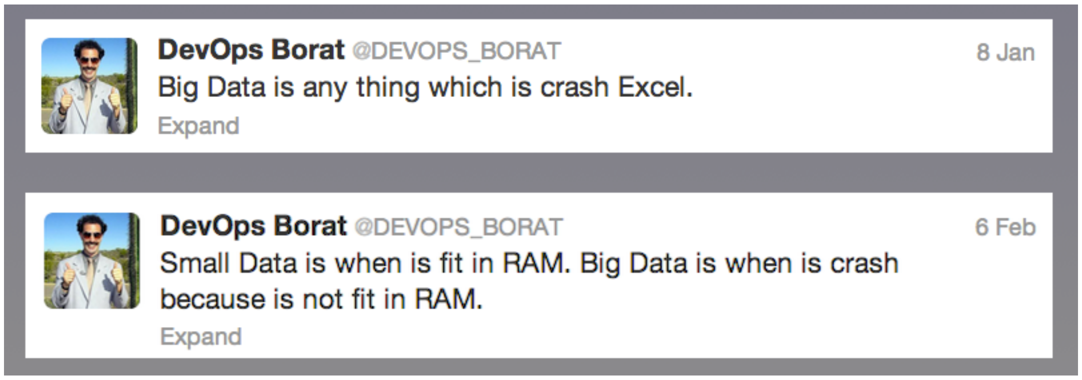
ACID

- *A* – Atomicity
- *C* – Consistency
- *I* – Isolation
- *D* – Durability

Circa 1995 – The "Interwebs"

What changed? What if you're Google?

What Is Big Data?



DevOps Borat @DEVOPS_BORAT 8 Jan
Big Data is any thing which is crash Excel.
Expand

DevOps Borat @DEVOPS_BORAT 6 Feb
Small Data is when is fit in RAM. Big Data is when is crash because is not fit in RAM.
Expand

"Web Scale" Data Persistence

- *High Availability:* If there's one server, we can't tolerate it crashing.
- *Horizontal Scaling:* No one server can "implement" Google.
- *Flexibility, Extensibility:* So we can grow our business with minimal cost.

What If You're Google?

How would you scale your Oracle instance (for example) to be "Google scale"?

Replication:

Duplicate copies of the *same* data.

- Replication Enables Failover
- If you have duplicate data and duplicate software, then if one system goes down, the other(s) can take over the load.

Sharding:

Splitting a large data set across multiple servers.

- Customer orders on the East coast are stored in New Jersey.
- Customer orders on the West coast are stored in Oregon.

Or, split *functionality* across servers:

- Customer orders on one server.
- Inventory on another server.
- …

Distributed Transactions?

- *Distributed Transactions:* Join transactions on individual systems into a larger transaction that spans the systems.
- *Two-Phase Commit:* A *consensus* protocol for whether to complete or abort the distributed transaction…

What Happens at Enormous Scales?

Imagine you're Amazon. You have millions of customers shopping at your site every day. You are saving information about searches, products viewed, products put in wish lists, orders placed, credit cards charged, etc.

Massive sharding and replication don't eliminate new phenomena that happen at large scales…

CAP

Eric Brewer (Professor at the University of California, Berkeley and the cofounder and chief scientist at Inktomi) conjectured in 2000 that you can’t satisfy all three of the following properties at once in World-scale systems.

- *C – Consistency*: To a client, an operation occurs all at once (Note: this is really *Atomicity* in ACID).
- *A – Availability*: Every operation completes to an expected result.
- *P – Partition Tolerance*: Operations will complete, even if some system components are unavailable (“partitioned” from the others).

CAP (cont.)

In The CAP Theorem, Seth Gilbert and Nancy Lynch elaborate on CAP, providing a proof of it, and offering real-world solutions.

CAP (cont.)

Gilbert and Lynch point out:

- *Consistency*: Distributed operations in a transaction appear as if they are on one node.
- *Availability*: A weak form of availability is implied; there is no explicit limit on how long a transaction will take to complete and return a result!
- *Partition Tolerance*: No set of failures less than total network failure is allowed to cause the system to respond incorrectly.

CAP: Pick Two

Brewer observed that you can pick two of the three: CA, AP, or CP.

(Really, it’s about picking A or C when Partitions happen.)

Consistency (Atomicity) and Availability

… but not Partition Tolerance.

E.g., distributed systems in a “reliable” LAN.

- Atomicity is provided by distributed transactions and 2-phase commit.
- Availability is ensured by system reliability.
- But if a partition forms (e.g., a node goes down), atomicity and availability are not guaranteed.
    - I.e., these systems are usually designed with the assumption of no partitions and guaranteed atomicity, so they will appear unavailable when a partition forms.

Availability and Partition Tolerance

… but not Consistency (Atomicity)

E.g., Google, Amazon, …

- Systems remain available, even when someone cuts a trans-ocean data cable…
- However, the data seen on either side of the cut might be different.
- Data caches can exhibit this property. You’re local cache of search results might be slightly stale, but returning a result quickly is more important than taking longer to return a more accurate result.

Consistency and Partition Tolerance

… but not Availability. Useful when it's better to never allow stale data, even if it means that no service is available at all.

E.g., a distributed DBMS in a LAN, using distributed locking.

- There are rarely partitions, so it's usually "tolerant".
- Atomicity is provided by the DBMS and distributed transaction framework.
- But if a partition occurs (e.g., node crash), it might become unavailability, but it never returns inconsistent data.

Distributed Data and Latency

Users expect fast responses (low latency), but the speed of light can't be increased. (It takes about 19ms for light to travel from New York to London) Also, to be globally available requires distribution of services for fault tolerance.

Distributed Data and Latency (cont.)

To meet these goals, your architecture has to tolerate high latency (Dan Pritchett), a.k.a The Tail at Scale. A couple of techniques help.

- Moving data and computing resources close to customers.
  - Lowers the round-trip time
- Doing as much calculation asynchronously as possible.
  - E.g., Google indexes the web asynchronously, so searches use precomputed data and are "instantaneous".

Distributed Data and Latency (cont.)

Suppose you scale by partitioning your data. You would like to maintain ACID compliant transactions across the data partitions. The traditional approach is to rely upon distributed transactions and two-phase commit.

The Problem with Distributed Transactions

But distributed transactions create *synchronous* couplings across the databases, which can increase latency substantially

When partitions are present, the transaction may never complete successfully!

The alternative to *ACID* is *BASE*:

- **B**asically **A**vailable
- **S**oft state
- **E**ventually consistent

When availability is the top priority, then try to always provide some level of service, don't insist on absolutely consistent state (i.e., tolerate some stale data), and expect the system to become consistent, *eventually*.

There's a spectrum between ACID and BASE for *each part* of the system.

ACID vs BASE

To be clear, BASE is a *compromise*.

Everyone would prefer to always have ACID compliance, but they are willing to accept BASE when ACID can't be met.

Back to Databases

The issues of scaling, CAP and BASE vs. ACID apply to all systems, including data systems.

Do We Need an RDBMS?

- Amazon, Google, eBay, Yahoo!, and other large-scale Internet companies have found that the cost of scaling traditional RDBMS is high and not always "worth it".
- Social networking sites like Twitter and Facebook have enormous *graphs* of social relationships to manage.
- If data is easily sharded, because the shards have no references spanning them, then is the relational model essential?
- If ACID isn't essential for all operations, are there lighter-weight persistence alternatives?

"NoSQL"

Not SQL or *Not Only SQL*

- If you have a graph of data, use a *graph* database.
- If you need great flexibility in the data format, use an "informal", easily-changed schema, use a *document-oriented* or *column oriented* database.
- If you need to store opaque *values* retrievable with *keys*, use a *key-value* store.
- If *availability* and *scalability* is more important than *transactional consistency*, use an *eventually-consistent* data store (including many of the above).

Categories of "NoSQL" Databases

- *Column-Oriented* Stores (vs. the *Row* Orientation of traditional RDBMS)
- *Document-Oriented* Stores
- *Key-Value/Tuple* Stores
- *Eventually-Consistent*, Key-Value Stores
- *Graph* Databases

(Compare with NoSQL-Database.org)

Categories of "NoSQL" Databases

NOTE: The actual databases often overlap several categories.

Column-Oriented Stores

E.g., *Cassandra, HBase, Big Table*

Column-oriented storage is often better for OLAP (OnLine Analytical Processing), e.g., Data analytics using Data Warehouses.

Row-oriented storage if often better for OLTP (OnLine Transaction Processing), e.g., typical “live” transactions.

Column-Oriented Stores

Column-oriented storage is often better for…

- optimizing space through compression:
    - A column of similar data is easier to compress.
- optimizing disk access for small subsets of all columns.
    - Versus scanning rows of many columns, most of which are discarded.
- Adding and removing columns frequently.

Many of the largest NoSQL database instances are column databases.

Document-Oriented Stores

E.g., *MongoDB, CouchBase, MarkLogic*

Stores *documents*, usually in the form of JSON (JavaScript Object Notation), YAML (Yet Another Markup Language), or XML (eXtensible Markup Language).

- *Schemaless:* No fixed schema, *semistructured* data.
    - Contrast with RDBMS.
    - But the document must be “well formed” (valid JSON, YAML, or XML).
- Often lightweight, so performance is often very good.

Document-Oriented Stores

- Often support a query capability or language (even though they’re *schemaless*).
- May support a *MapReduce* capability.
- Excellent for medium-sized datasets with rapidly-evolving “schema”.

Key-Value Stores

E.g., *Amazon SimpleDB, Riak, Redis, MemcacheDB, MNesia, …*

Think of hash maps on *steroids*.

- May be purely in-memory (with optional flushing to disk) and resident on one machine.
- Excellent for “persisting” semi-structured data with shorter lifespans (e.g., web sessions).
- Used when durability and consistency are lower priorities compared to speed.

Eventually Consistent Key-Value Stores

E.g., *Amazon DynamoDB, Voldemort*

More emphasis on longer-lived objects that need to be persisted, but for which rapid retrieval is still important.

- Transparent replication of data.
- Transparent clustering of nodes.
- Great for fault tolerance, especially for “active sessions”.

Graph Databases

E.g., *Neo4J, Titan*

Built in semantics for representing graphs, including cycles.

- First-class support for nodes, edges, and properties to associate information with nodes and edges.
- Good when lots of expensive joins would be required in a RDBMS.
- Handle evolving schema easily.

NoSQL vs. Hadoop

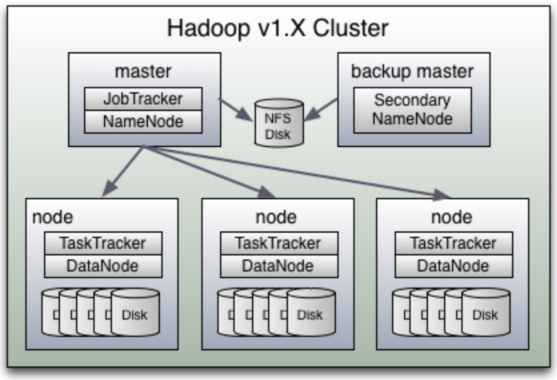
NoSQL stores, like all datastorage technologies, give you choices for:

- A query “API”.
- Speed of reads and writes.
- Integrity controls like transactions (or not…).
- A data model.
- Durability guarantees.

… but *computation* is not the emphasis.

Hadoop

General-purpose, distributed computation.

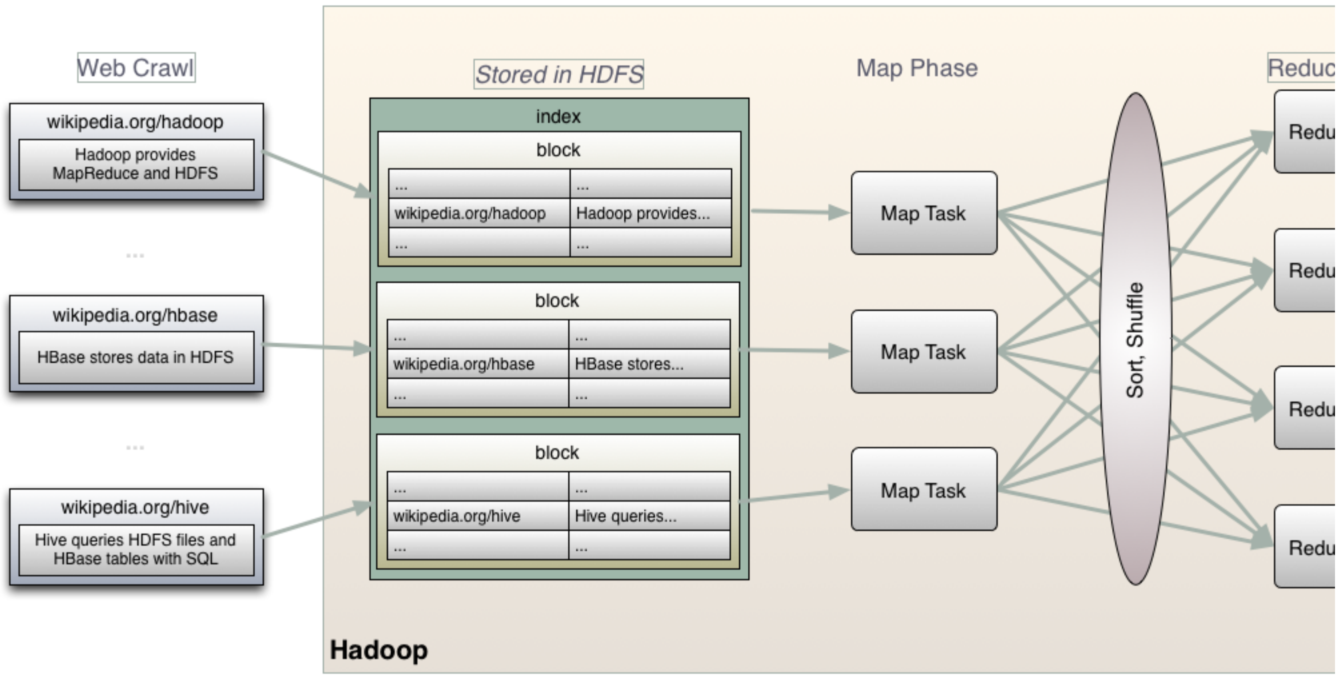


Example: Inverted Index

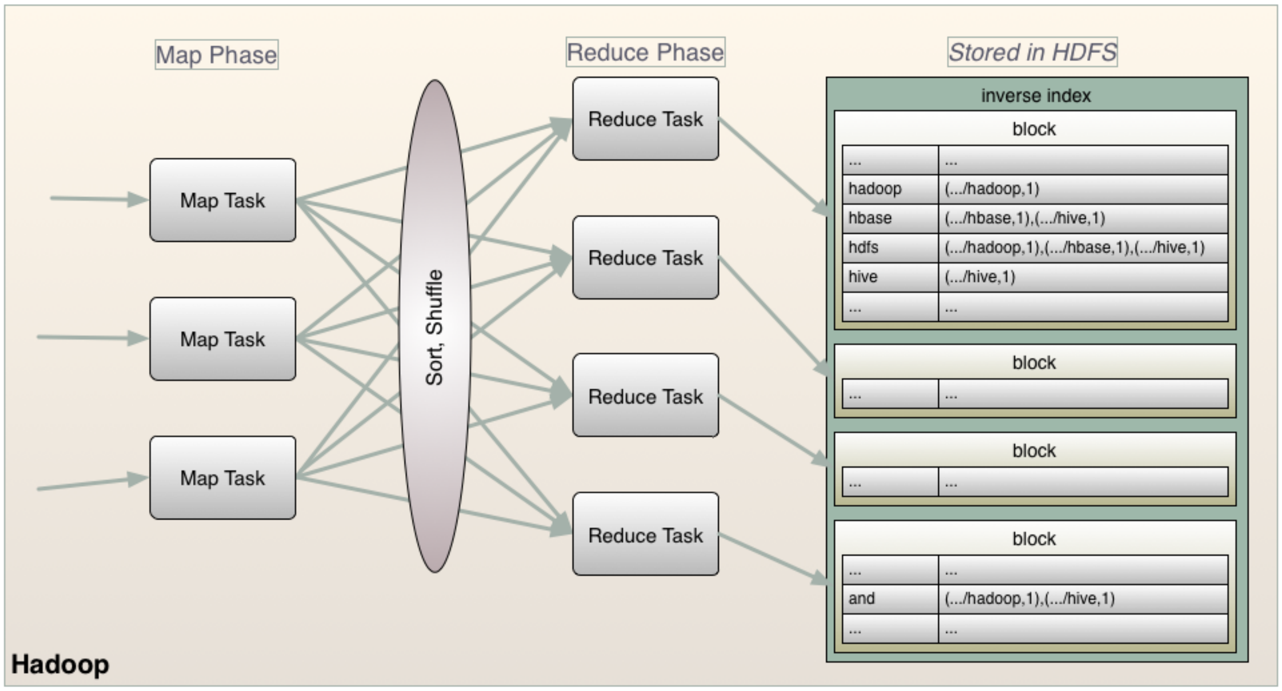
Create an index of all documents containing particular words and phrases.

A very simple example of how the data for a web search engine is computed.

Example: Inverted Index (cont.)



Example: Inverted Index (cont.)



Diving Deeper…

Let’s explore these categories with specific examples.

Column-Oriented Storage – Deeper Dive

Records have many, many columns (sometimes different from row to row), where typical queries concern a small subset.

Examples: *Cassandra, HBase, Big Table*

Cassandra: A Column-Oriented Database

Design goals:

- Reliability through distributed processing, no single point of failure.
- High write throughput.
- Trade RDBMS features for a simple data model with easy, dynamic control over data layout and format.

Cassandra: Developed at Facebook

Originally designed for Facebook's Inbox Search feature.

- Very high write throughput required.
- Replication to geographically-distributed data centers.
    - Minimizes search latencies.

Cassandra: Inspirations

- Inspired by Amazon's DynamoDB and Google's BigTable.
    - But provides better write performance.
- Open-source Apache Project.
    - But Facebook has maintained its own fork.

Cassandra: Columns

Column Oriented means that columns are easy to query for, as well as to add and remove in the "schema". This supports evolution of features.

However, operations are keyed by rows and each row operation is atomic, independent of the number of columns involved.

Cassandra: Columns

- *Column Families:* a group of columns.
    - *Super Column Family*: a column family within a column family.

API

Just 3 methods:

- insert(table, key, rowMutation)
- get(table, key, columnName)
- delete(table, key, columnName)

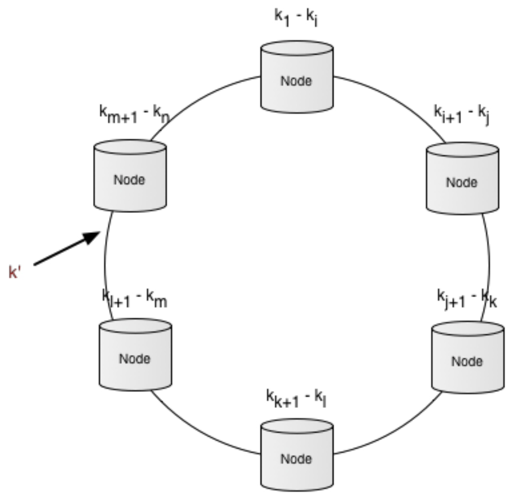
columnName can refer to a single column, or a column family.

Distributed System Techniques

Cassandra uses many distributed system techniques for scaling and availability:

- Partitioning
- Replication
- Membership
- Failure Handling
- Scaling

Partitioning



Partitioning

Dynamically partition the data over the cluster.

- Too hard to do this manually.
- Keys are hashed, using consistent hashing.
- The range of possible hash values is treated as a “ring” that wraps.
- Each node is assigned a random position on the ring.

When a row’s key is hashed, the node with the closest key *greater than* the row’s key is assigned the responsibility for coordinating reads and writes.

Reads and Writes

The responsible node decides what replicas should also handle the request.

- Reads can be configured to use the value of the nearest replica or a *quorum* of replica responses.
- Writes are routed to all the designated replicas and the write is only considered complete when a *quorum* of replicas respond that the write was successful.

Failures and Balancing

If a node fails or a new one is added, only its immediate neighbor nodes in the ring are affected.

The initial distribution of nodes doesn't account for the actual distribution of keys in the data nor the performance of particular nodes. Hence, it can be imbalanced. Cassandra will move nodes on the ring to adjust performance.

Replication

Each Cassandra "instance" is configured with a *replication factor* of N, the number of nodes to which each data item is replicated.

When a node is responsible for a row, it stores the row locally and replicates it to N-1 other nodes.

Cassandra can be configured to ensure that each row is replicated across multiple data centers, so every row remains available even in the event of a total failure of a data center.

Membership

Membership and health of the nodes is managed through a *Gossip* protocol called *Scuttlebutt*. The *Phi Accrual Failure Detector* module is used by each node to publish its "suspicions" about the health of every other node, rather than publishing an "up" or "down" report that may not be accurate. The suspicion level rises, as concerns about a node rise. At the same time, the probability of error decreases.

This turns out to be a much faster way to detect node failure than other detection schemes.

Bootstrapping

When a node starts for the first time, it chooses a random position on the ring, then "Gossips" its position to the cluster. This is how all the other nodes know the members of the cluster, for replication, etc.

Scaling

A new node can be assigned a key that allows it to offload an overloaded node. The overloaded node then uses kernel-to-kernel copying techniques to move the data it is no longer responsible for to the new node.

Local Persistence: Writes

For local storage, Cassandra relies on the local file system. (By contrast, Google's BigTable is designed for GFS, a distributed file system.)

Writes are appended to a commit log. Only *after* this succeeds is the same change written to an in-memory data structure.

Local Persistence: Writes (cont.)

Typically, a dedicated disk is used on each system for the commit log, to minimize seeks and thereby optimize writes.

When the in-memory storage reaches a certain size threshold, it is written to another disk and the rows are indexed for fast retrieval.

Local Persistence: Writes (cont.)

Over time, many such files written from the in-memory data structure will exist. A background process merges them into one file. Also, old commit logs are eventually deleted, once the in-memory structures they represent are persisted.

Hence, durability is ensured through a combination of the commit log and the row files, replicated across the cluster.

Since *no files are ever modified for updates*, no *locks are required*. Cassandra also groups writes to be synchronous, maximizing disk write speeds.

Local Persistence: Reads

First, the in-memory data structure is searched for the row. If not found, the *newest* to *oldest* files are searched, in order.

To avoid searching a file that is unlikely to contain the row, a *bloom filter* that summarizes the keys in the file is also stored in the file and and memory. It is examined to determine if the file has the row.

Local Persistence: Reads (cont.)

Column locations are also indexed, so getting to the correct disk block is optimized.

Column-Oriented Databases: Conclusions

- Great for schema that need to evolve by adding columns.
- Great for fast queries by column(s), without reading entire rows.
- Optimized for write performance, while maintaining fast reads.
- Trade ACID and SQL for flexibile availability, consistency, and performance.

Column-Oriented Databases: Conclusions

Note that some relational databases now support column-oriented storage.

Document Databases – Deeper Dive

Treat *documents* (XML, JSON, YML, etc.) as “records”.

Examples: *MongoDB*, *CouchBase*, *CouchDB*, *MarkLogic*

MongoDB, CouchBase, and CouchDB

- MongoDB: mongodb.org
- CouchDB: couchdb.apache.org
- CouchBase: couchbase.com

All three are *JSON* document oriented, are *schemaless*, and support JavaScript for programming many database operations. But they are very different in other ways, a reflection of their different goals.

MarkLogic

- MarkLogic: marklogic.com

An XML document database.

Differences Between MongoDB and CouchDB

- Persistence Strategy
- Durability, Replication, and Consistency
- Queries
- MapReduce
- Transactions
- “Driver” APIs
- Ideal Applications

Persistence Strategy

CouchDB uses *MVCC* (multiversion concurrency control), while *MongoDB* uses *update in place*.

In MVCC a new version of the document is created with the desired changes. The previous versions are still available, until they are specifically deleted (or *compacted* into fewer versions).

Update in place overwrites previous values… and requires transactions for data integrity, etc.

Advantages of MVCC

- Better for *durability*. Append-only log files and BTrees on disk make restarting after crashes “mostly” straightforward.
- If conflicting updates happen, *both* of them are saved.

Advantages of MVCC (cont.)

- MVCC is very good for syncing master-master database configurations. For example, you can create a copy of the database, take it offline and work on it, then sync it to the other copies when back online again. CouchDB has bi-directional replication.

Note: distributed version control systems like *git* work the same way.

Disadvantages of MVCC

- Making copies of documents is slower, consumes more space over time. Compaction is required occasionally.
- Update conflicts must be resolved manually (although now update is “lost” and CouchDB deterministically picks a default “winner”).

Advantages of Update in Place

- Faster write performance, especially for updates, since the whole document isn’t copied^.
- Compaction isn’t needed to remove unneeded copies of documents, since multiple versions aren’t created.

^ However, disk seek time to find records can be significant. An append-only file on a disk with no other read/write operations will have minimal seek time.

Disadvantages of Update in Place

- If conflicting updates occur, only the last one survives. Transactions are more important to avoid collisions without the recovery option that MVCC provides (e.g., picking a previous version or merging results some how).

Disadvantages of Update in Place (cont.)

- Crashes more easily leave the database in an inconsistent state, requiring database repair on reboot.
- Even with replicated database instances, service may not be interrupted, but restoring replica consistency is difficult.

Durability, Replication, and Consistency

When replicating CouchDB instances, you get *eventual consistency*. Note that if one replica is offline, consistency can’t happen until it comes online again.

Queries

MongoDB provides a full-featured, *ad hoc* query capability using a JavaScript-based query language. You can do most kinds of queries that you can do with SQL (*except joins*) in a SQL database. Instead of joins, *denormalize* the data!

CouchDB does not provide a query mechanism. Instead, *views*, indexing, and *MapReduce* operations are specified to support desired read, aggregation, and updating patterns.

MapReduce

Both systems support *MapReduce*. CouchDB requires writing MapReduce functions to construct the views. MongoDB provides a built-in MapReduce and integration with Hadoop MapReduce, but MongoDB doesn't require the use of MapReduce, like CouchDB does.

Transactions

Neither database supports the full transaction capabilities typically found in RDBMSs. They do this to keep the systems lighter and faster.

Instead, MongoDB offers several atomic operations for *single* documents (i.e., rows, in RDBMS terminology).

Similarly, CouchDB also supports document-level transactions through MVCC, providing optimistic updates, but records when a collision occurs, recording *both* updates, and using a deterministic algorithm to pick the "winner". Normally, though, the application should decide how to resolve the collision.

Ideal Applications

So, while superficially similar, they are very different in important ways, which affects the kinds of applications for which each is ideal.

Ideal Applications: CouchDB

CouchDB is ideal for:

- Productivity applications with built in databases, where offline work is common, with subsequent synching to the "master". Think email, calendaring, and similar apps.
- Persistence with easy replication is more important that update throughput.
- Easy integration with web front ends, through REST.
- Many apps that Ruby on Rails is currently used to implement.

Ideal Applications: MongoDB

MongoDB is ideal for:

- Applications with high-volume writes/updates, especially if some write losses are tolerable (e.g., sensor networks).
- System "logs" (especially with capped collections).
- Applications requiring ad hoc query support.
- Applications requiring fast language-specific drivers.

Document Databases: Conclusions

In general, *Document Databases* are philosophically more like traditional RDBMS than other NoSQL stores.

Key-Value Stores – Deeper Dive

E.g., *Amazon SimpleDB, Riak, Redis, BerkeleyDB, Mnesia*.

Key-Value Stores

- Think of hash maps on *steroids*.
- Used when durability and consistency are lower priorities compared to speed.
- Excellent for "persisting" semi-structured data with shorter lifespans (e.g., web sessions).

Spectrum of Key-Value Stores

- *Cache*-like – Purely in-memory with optional flushing to disk. May reside only on one machine.
- *Distributed Hash Map* – More durability to disk emphasized, high-capacity, distributed, no single points of failure…

Key-Value Store: General Considerations

- How often is data synced to disk?
- How is data replicated to other nodes (for distributed stores)?
- How is fault tolerance supported?

Eventually Consistent Key-Value Stores

E.g., *Amazon DynamoDB, Voldemort*

More emphasis on longer-lived objects that need to be persisted, but for which rapid retrieval is still important.

- Transparent replication of data.
- Transparent clustering of nodes.
- Great for fault tolerance, especially for "active sessions".

Are Key-Value Stores Really Databases?

Memcached has been around a while. It's a popular in-memory, distributed cache. It's great for caching database queries in memory, for faster repeat retrieval, storing user sessions between page views in a web application, etc.

The definition of "database" has expanded in recent years, so now we think of a key-value store, like memcached, as a datastore, *if* it is backed by some sort of durability.

MemcacheDB is built on top of Memcached to provide this durability. It uses another, fairly old, open-source NoSQL store, BerkeleyDB to provide persistence.

MemcacheDB

It's API is compliant with the Memcache protocol:

- get(also multiple get)
- set, add, replace
- append/prepend
- incr, decr
- delete
- stats

Very simple! If all you need is key-based storage and retrieval of "blobs", then a key-value store like this can be ideal.

Redis

Redis provides data-structure aware storage. Not just key-values (e.g., a map or hash), where values are "blobs", but also API calls specifically for working with strings, lists, sets, and sorted sets.

Redis

Other features:

- Many language bindings.
- Very high speed.
- Limited support for fault tolerance and clustering.

So, useful for the same sorts of tasks MemcacheDB would be used for, plus cases where awareness of the additional data structures is useful.

Scalability and Performance

- The extremely simple data model means these systems tend to…
  - … have extraordinary read and write performance for their "weight".
  - … support clustering, replication, failover, etc. easily.

Key-Value Store: Conclusions

The largest category, in terms of available implementations, they are best when the reliability and scalability that comes from simplicity is best. Missing features like rich queries have to be implemented in software.

Graph Databases – Deeper Dive

Model *relationships* as first-class concepts using *edges* between *nodes* in *graphs*.

Graph Databases

Contrast with *relational databases*, where *relationships* are managed through *joins*.

- For M:N relationships, you have to manually create a separate *join table*.
- For data best modeled as a *network* of *nodes*, this isn't very convenient, nor does it perform well, in most cases.

Graph Examples

- The Internet!
- Social networks.
- Physical maps: streets, public transit, geography, etc.
- Complex business rules.
- Parts lists and their relationships (for assembly lines and finished products, e.g., airplanes!).

Graph Databases and OO Middleware

An *object* is actually a graph of smaller objects:

- Smaller graphs (collections).
- "Leaf" nodes, like integers, floats, etc.

(It's sometimes useful to treat other small objects, like strings as leaf nodes, too.)

Hence, for applications with complex object graphs, a graph database might be a better fit than a relational database.

Elements of a Graph Database

At its core a graph database supports the following concepts from standard graph theory.

- Nodes
- Edges (or Arcs)
- Properties of nodes, edges, or both.

Nodes

A *node* is analogous to a row in a relational database. It's the "entity" whose relationships to other entities are of primary interest.

Examples:

- Users of Twitter, Facebook, LinkedIn, etc.
- Airports served by an airlines.
- Addresses to which mail and packages are delivered.

Edges

The connections (relationships) between nodes. Which of these are always bidirectional? Sometimes?

Examples:

- Your followers and who you follow on Twitter.
- Your friends and their friends, etc., on Facebook and LinkedIn.
- Flight "legs" between airports.
- Delivery route between two addresses for mail and package deliveries.

Edges (cont.)

Traversals (paths) are formed from multiple, contiguous edges.

- They may be circular.
  - The may go from one node to another and back again.
- There will be dead ends.
- There will be clumps with relatively few edges between them.

For a real-world example, see this video of a LinkedIn employee's own social graph.

Properties

Key-value pairs associated with nodes and/or edges. Much of the interesting data can be in these properties!

Example Properties for Nodes:

- Where you live, your profession, interests, age, marital status, …
- Airport capacity, number of gates, hours of operation, …
- Business or personal address? Apartment building or house? Hostile dogs?

Properties (cont.)

Example Properties for Edges:

- Is this a professional or personal relationship.
- How often is this relationship used?
  - Twitter direct messages.
  - Emails sent.
- Daily flight schedule (including number of flights, passengers carried per flight, etc.).
- Distance between two addresses.
- Hazards or barriers between two addresses.

Graph Databases vs. Other Kinds

- Most of the database *kinds* we've discussed can scale through *sharding*. That works best if there are *relatively few relationships between the shards*.
- Graph databases *emphasize* connections, so sharding requires careful understanding of the actual clumps in the data.

Graph Databases vs. Other Kinds

Where *joins* can be expensive in an RDBMS, graphs optimize traversals.

Neo4J

Neo4J.

Neo4j is an embedded, disk-based, fully-transactional Java persistence engine that stores data structured in graphs and supports graph traversal algorithms.

Neo4J: Features

- Object-oriented Java API.
- Scalability:
  - Handles graphs of several billion nodes/relationships/properties on a single machine.
  - Can be sharded across multiple machines.
- Support for many RDBMS database features:
  - ACID transactions.
  - Durable persistence.
  - Concurrency control.

Titan

Titan

- Bleeding-edge distributed scalability across clusters.
  - HBase or Cassandra used for storage.
- Sophisticated graph algorithms.

Giraph and Hama

- Apache Giraph
- Apache Hama

Both:

- Run on top of Hadoop.
- Use the *Bulk Synchronous Parallel* graph traversal model.

Graph Databases: Conclusion

Scaling distributed graphs is still a research project, but there have been some impressive results. See for example, how Facebook uses Giraph.

Conclusions – No More SQL??

So, is SQL dead?

More Data Options

- SQL isn't going away because ACID transactions are still the best model for many, many problems.
- Many data problems aren't *Big Data* problems.

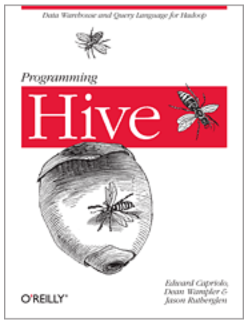
NoSQL Solves…

… problems that became painful during the rise of the Internet.

SQL Is Actually Roaring Back!

The world *loves* SQL, so…

- NewSQL databases combine ACID transactions and the relational model with NoSQL scaling techniques.
- Hive pioneered SQL queries on Hadoop.


Data Warehouse and Query Language for Hadoop
Programming
Hive
O'REILLY
Edward Capriolo, Dean Wampler & Jason Rutherglen

New Approaches – Datomic

Datomic

- Whole database is a *value*.
- No "destructive" updates.
    - All previous *history* retained and usable.
- Invented by *Rich Hickey*, the creator of *Clojure*.

Questions?

- This talk: bit.ly/DW_KDD2013
- Dean Wampler, Ph.D.
- dean@concurrentthought.com
- @deanwampler
- concurrentthought.com
- polyglotprogramming.com/talks

Thank You!