Project Voldemort

Jay Kreps

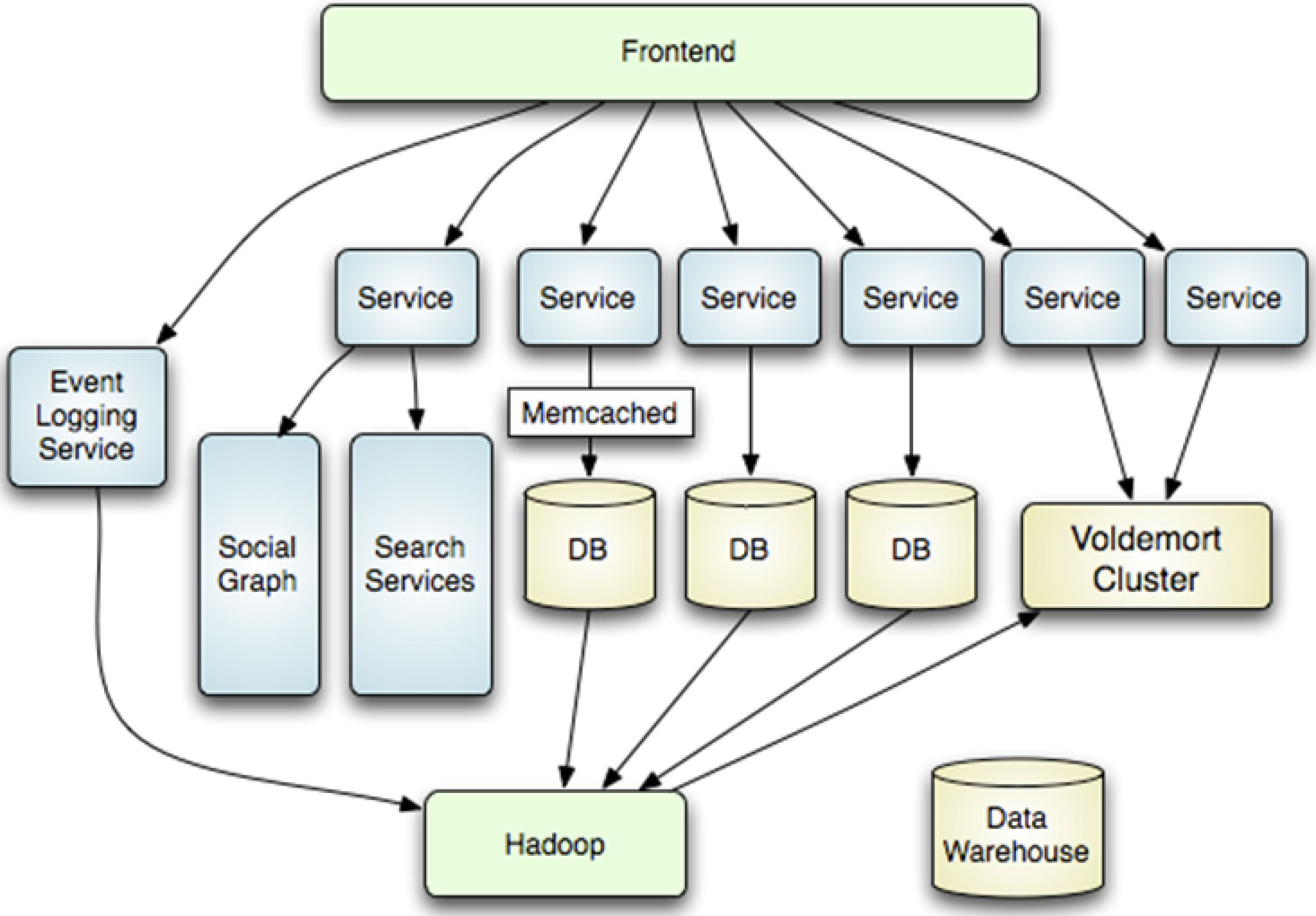
Where was it born?

- LinkedIn’s Data & Analytics Team
  - Analysis & Research
  - Hadoop and data pipeline
  - Search
  - Social Graph
- Caltrain
- Very lenient boss

Two Cheers for the relational data model

- The relational view is a triumph of computer science, but…
- Pasting together strings to get at your data is silly
- Hard to build re-usable data structures
- Don’t hide the memory hierarchy!
  - Good: Filesystem API
  - Bad: SQL, some RPCs

LinkedIn from 20,000 feet



Services Break Relational DBs

- No real joins
- Lots of denormalization
- ORM is pointless
- Most constraints, triggers, etc disappear
- Making data access APIs cachable means lots of simple GETs
- No-downtime releases are painful
- LinkedIn isn’t horizontally partitioned
- Latency is key

Other Considerations

- Who is responsible for performance (engineers? DBA? site operations?)
- Can you do capacity planning?
- Can you simulate the problem early in the design phase?
- How do you do upgrades?
- Can you mock your database?

Some problems we wanted to solve

- Application Examples
  - People You May Know
  - Item-Item Recommendations
  - Type ahead selection
  - Member and Company Derived Data
  - Network statistics
  - Who Viewed My Profile?
  - Relevance data
  - Crawler detection
- Some data is batch computed and served as read only
- Some data is very high write load
- Voldemort is only for real-time problems
- Latency is key

Some constraints

- Data set is large and persistent
  - Cannot be all in memory
  - Must partition data between machines
- 90% of caching tiers are fixing problems that shouldn’t exist
- Need control over system availability and data durability
  - Must replicate data on multiple machines
- Cost of scalability can’t be too high
- Must support diverse usages

Inspired By Amazon Dynamo & Memcached

- Amazon’s Dynamo storage system
  - Works across data centers
  - Eventual consistency
  - Commodity hardware
  - Not too hard to build
- Memcached
  - Actually works
  - Really fast
  - Really simple
- Decisions:
  - Multiple reads/writes
  - Consistent hashing for data distribution
  - Key-Value model
  - Data versioning

Priorities

1. Performance and scalability
2. Actually works
3. Community
4. Data consistency
5. Flexible & Extensible
6. Everything else

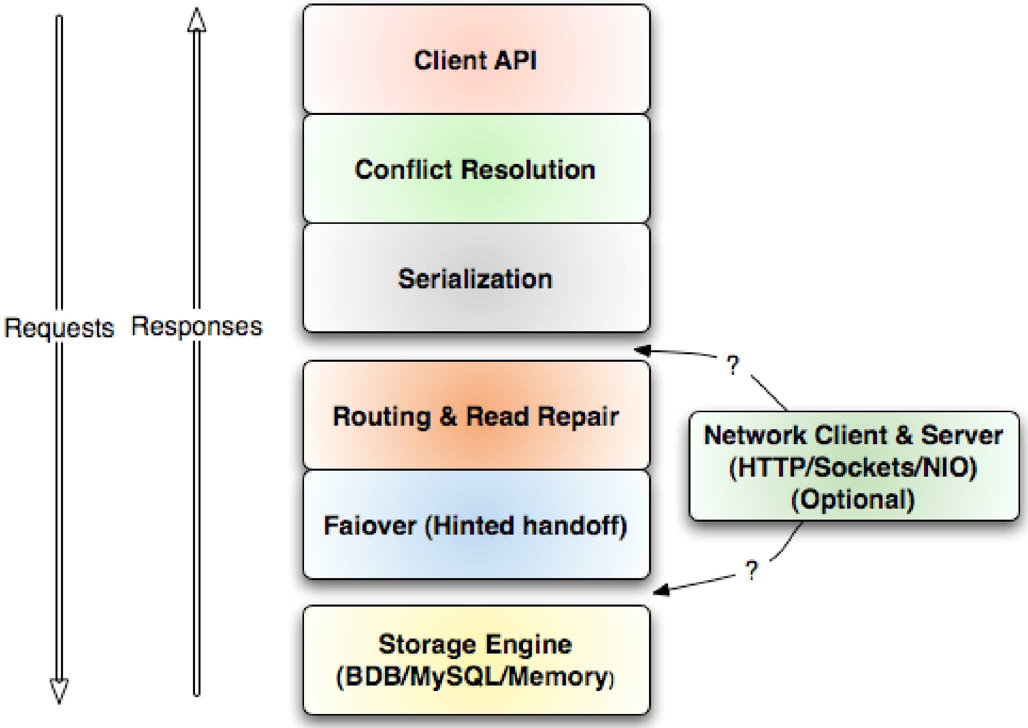
Why Is This Hard?

- Failures in a distributed system are much more complicated
  - A can talk to B does not imply B can talk to A
  - A can talk to B does not imply C can talk to B
- Getting a consistent view of the cluster is as hard as getting a consistent view of the data
- Nodes will fail and come back to life with stale data
- I/O has high request latency variance
- I/O on commodity disks is even worse
- Intermittent failures are common
- User must be isolated from these problems
- There are fundamental trade-offs between availability and consistency

Voldemort Design

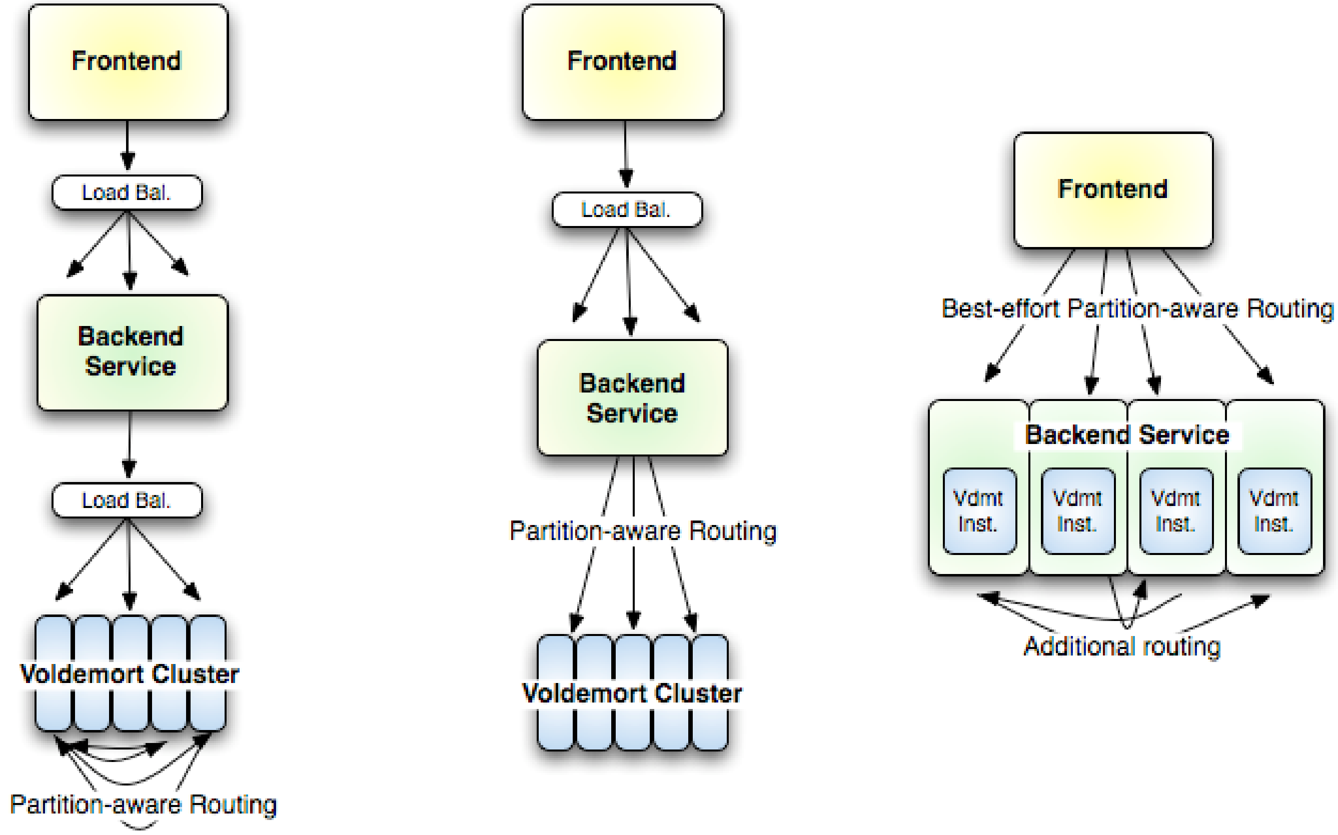
- Layered design
- One interface for all layers:
    - put/get/delete
    - Each layer decorates the next
    - Very flexible
    - Easy to test

Logical Architecture



Voldemort Physical Deployment

Physical Architecture Options



3-Tier, Server-Routed

3-Tier, Client-Routed

2-Tier, Frontend-Routed

Client API

- Key-value only
- Rich values give denormalized one-many relationships
- Four operations: PUT, GET, GET_ALL, DELETE
- Data is organized into “stores”, i.e. tables
- Key is unique to a store
- For PUT and DELETE you can specify the version you are updating
- Simple optimistic locking to support multi-row updates and consistent read-update-delete

Versioning & Conflict Resolution

- Vector clocks for consistency
  - A partial order on values
  - Improved version of optimistic locking
  - Comes from best known distributed system paper "Time, Clocks, and the Ordering of Events in a Distributed System"
- Conflicts resolved at read time and write time
- No locking or blocking necessary
- Vector clocks resolve any non-concurrent writes
- User can supply strategy for handling concurrent writes
- Tradeoffs when compared to Paxos or 2PC

Vector Clock Example

```
# two servers simultaneously fetch a value
[client 1] get(1234) => {"name":"jay", "email":"jkreps@linkedin.com"}
[client 2] get(1234) => {"name":"jay", "email":"jkreps@linkedin.com"}

# client 1 modifies the name and does a put
[client 1] put(1234), {"name":"jay2", "email":"jkreps@linkedin.com"})

# client 2 modifies the email and does a put
[client 2] put(1234, {"name":"jay3", "email":"jay.kreps@gmail.com"})

# We now have the following conflicting versions:
{"name":"jay", "email":"jkreps@linkedin.com"}
{"name":"jay kreps", "email":"jkreps@linkedin.com"}
{"name":"jay", "email":"jay.kreps@gmail.com"}
```

Serialization

- Really important--data is forever
- But really boring!
- Many ways to do it
  - Compressed JSON, Protocol Buffers, Thrift
  - They all suck!
- Bytes <=> objects <=> strings?
- Schema-free?
- Support real data structures

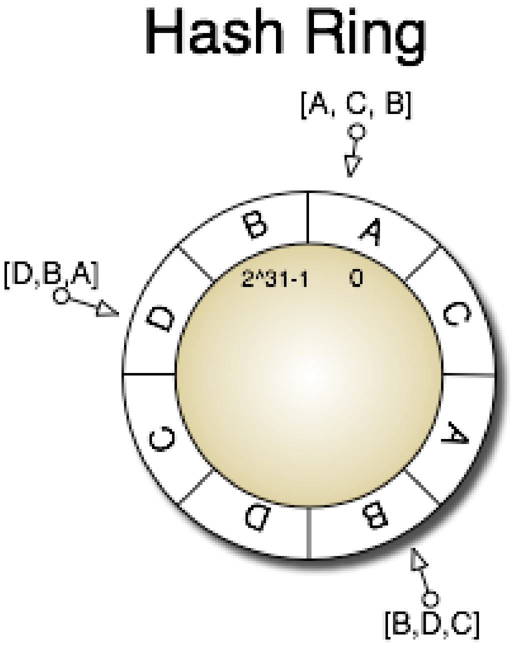
Routing

- Routing layer turns a single GET, PUT, or DELETE into multiple, parallel operations
- Client- or server-side
- Data partitioning uses a consistent hashing variant
  - Allows for incremental expansion
  - Allows for unbalanced nodes (some servers may be better)
- Routing layer handles repair of stale data at read time
- Easy to add domain specific strategies for data placement
  - E.g. only do synchronous operations on nodes in the local data center

Routing Parameters

- N - The replication factor (how many copies of each key-value pair we store)
- R - The number of reads required
- W - The number of writes we block for
- If R+W > N then we have a quorum-like algorithm, and we will read our writes

Routing Algorithm



- To route a GET:
  - Calculate an ordered preference list of N nodes that handle the given key, skipping any known-failed nodes
  - Read from the first R
  - If any reads fail continue down the preference list until R reads have completed
  - Compare all fetched values and repair any nodes that have stale data
- To route a PUT/DELETE:
  - Calculate an ordered preference list of N nodes that handle the given key, skipping any failed nodes
  - Create a latch with W counters
  - Issue the N writes, and decrement the counter when each is complete
  - Block until W successful writes occur

Routing With Failures

- Load balancing is in the software
    - either server or client
- No master
- View of server state may be inconsistent (A may think B is down, C may disagree)
- If a write fails put it somewhere else
- A node that gets one of these failed writes will attempt to deliver it to the failed node periodically until the node returns
- Value may be read-repaired first, but delivering stale data will be detected from the vector clock and ignored
- All requests must have aggressive timeouts

Network Layer

- Network is the major bottleneck in many uses
- Client performance turns out to be harder than server (client must wait!)
- Server is also a Client
- Two implementations
    - HTTP + servlet container
    - Simple socket protocol + custom server
- HTTP server is great, but http client is 5-10X slower
- Socket protocol is what we use in production
- Blocking IO and new non-blocking connectors

Persistence

- Single machine key-value storage is a commodity
- All disk data structures are bad in different ways
- Btrees are still the best all-purpose structure
- Huge variety of needs
- SSDs may completely change this layer
- Plugins are better than tying yourself to a single strategy

Persistence II

- A good Btree takes 2 years to get right, so we just use BDB
- Even so, data corruption really scares me
- BDB, MySQL, and mmap'd file implementations
    - Also 4 others that are more specialized
- In-memory implementation for unit testing (or caching)
- Test suite for conformance to interface contract
- No flush on write is a huge, huge win
- Have a crazy idea you want to try?

State of the Project

- Active mailing list
- 4-5 regular committers outside LinkedIn
- Lots of contributors
- Equal contribution from in and out of LinkedIn
- Project basics
  - IRC
  - Some documentation
  - Lots more to do
- > 300 unit tests that run on every checkin (and pass)
- Pretty clean code
- Moved to GitHub (by popular demand)
- Production usage at a half dozen companies
- Not a LinkedIn project anymore
- But LinkedIn is really committed to it (and we are hiring to work on it)

Glaring Weaknesses

- Not nearly enough documentation
- Need a rigorous performance and multi-machine failure tests running NIGHTLY
- No online cluster expansion (without reduced guarantees)
- Need more clients in other languages (Java and python only, very alpha C++ in development)
- Better tools for cluster-wide control and monitoring

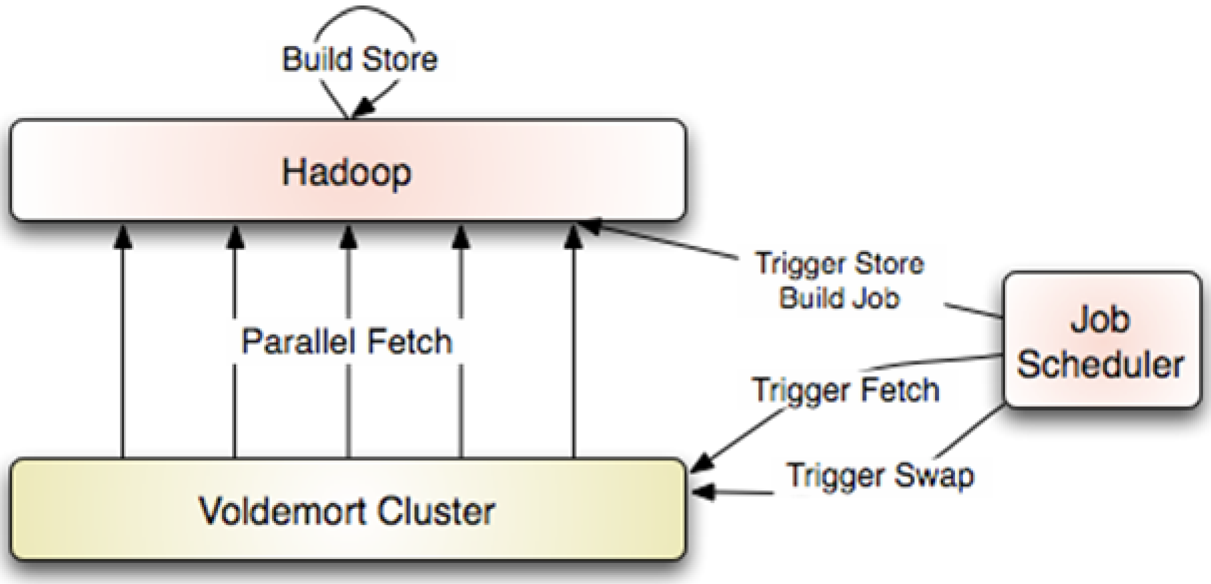
Example of LinkedIn's usage

- 4 Clusters, 4 teams
    - Wide variety of data sizes, clients, needs
- My team:
    - 12 machines
    - Nice servers
    - 300M operations/day
    - ~4 billion events in 10 stores (one per event type)
- Other teams: news article data, email related data, UI settings
- Some really terrifying projects on the horizon

Hadoop and Voldemort sitting in a tree…

- Now a completely different problem: Big batch data processing
- One major focus of our batch jobs is relationships, matching, and relevance
- Many types of matching people, jobs, questions, news articles, etc
- O(N^2) :-(
- End result is hundreds of gigabytes or terrabytes of output
- cycles are threatening to get rapid
- Building an index of this size is a huge operation
- Huge impact on live request latency

Read-Only Store Build and Swap Process



1. Index build runs 100% in Hadoop
2. MapReduce job outputs Voldemort Stores to HDFS
3. Nodes all download their pre-built stores in parallel
4. Atomic swap to make the data live
5. Heavily optimized storage engine for read-only data
6. I/O Throttling on the transfer to protect the live servers

Some performance numbers

- Production stats
  - Median: 0.1 ms
  - 99.9 percentile GET: 3 ms
- Single node max throughput (1 client node, 1 server node):
  - 19,384 reads/sec
  - 16,559 writes/sec
- These numbers are for mostly in-memory problems

Some new & upcoming things

- New
  - Python client
  - Non-blocking socket server
  - Alpha round on online cluster expansion
  - Read-only store and Hadoop integration
  - Improved monitoring stats
- Future
  - Publish/Subscribe model to track changes
  - Great performance and integration tests

Shameless promotion

- Check it out: project-voldemort.com
- We love getting patches.
- We kind of love getting bug reports.
- LinkedIn is hiring, so you can work on this full time.
  - Email me if interested
  - jkreps@linkedin.com

The End