Search Engine Architecture

2. Big Data Storage
Cloud Computing
Computing in the Cloud

- Before clouds...
  - Grids
  - Connection machine
  - Vector supercomputers
  - ...

- Cloud computing means many different things:
  - Big data
  - Rebranding of web 2.0
  - Utility computing
  - Everything as a service

What is Cloud Computing?

• Cloud computing means computing resources available “on demand”
  • Resources can include storage, compute cycles, or software built on top (e.g. database as a service)
  • On demand means fast setup/teardown, pay-as-you-go
• For big data, clouds are attractive for several reasons
  • Access to large infrastructure that is hard to operate
  • Bursty workloads benefiting from pay-as-you-go
  • Recent years have seen major growth of cloud computing in most software domains

Everything as a Service

- Utility computing = Infrastructure as a Service (IaaS)
  - Why buy machines when you can rent cycles?
  - Examples: Amazon’s EC2, Rackspace
- Platform as a Service (PaaS)
  - Give me nice API and take care of the maintenance, upgrades, ...
  - Example: Google App Engine, Amazon RDS
- Software as a Service (SaaS)
  - Just run it for me!
  - Example: Gmail, Salesforce

Benefits for Users

• Fast deployment
  • Cloud services can start in minutes, without long setup
• Outsourced management
  • Provider handles administration, reliability, security
• Lower costs
  • Benefit from economies of scale of provider
  • Only pay for resources while in use
• Elasticity
  • Easy to acquire lots of infrastructure for a short period

Benefits for Providers

- **Economies of scale**
  - Share expertise and resources across many customers
  - Lower costs per user due to scale
- **Fast deployment**
  - Compared to traditional software sales cycles, new features reach users directly
- **Optimization across users**

Utility Computing

• What?
  • Computing resources as a metered service ("pay as you go")
  • Ability to dynamically provision virtual machines

• Why?
  • Cost: capital vs. operating expenses
  • Scalability: "infinite" capacity
  • Elasticity: scale up or down on demand

• Does it make sense?
  • Benefits to cloud users
  • Business case for cloud providers

Virtualization

Traditional Stack

Virtualized Stack

Who cares?

- **Ready-made big data problems**
  - Social media, user-generated content = big data
  - Examples: Facebook friend suggestions, Google ad placement
  - Business intelligence: gather everything in a data warehouse and run analytics to generate insight

- **Utility computing provides:**
  - Ability to provision clusters on-demand in the cloud
  - Lower barrier to entry for tackling big data problems
  - Commoditization and democratization of big data capabilities

Big Data Storage
The Fundamental Problem

• We want to keep track of *mutable* state in a *scalable* manner

• Assumptions:
  • State organized in terms of many “records”
  • State unlikely to fit on single machine, must be distributed

Three Core Ideas

• Partitioning (sharding)
  • For scalability
  • For latency

• Replication
  • For robustness (availability)
  • For throughput

• Caching
  • For latency

Lots of problems...

- How do we keep replicas in sync?
- How do we synchronize transactions across multiple partitions?
- What happens to the cache when the underlying data changes?
What do RDBMSes provide?

- Relational model with schemas
- Powerful, flexible query language
- Transactional semantics: ACID
- Rich ecosystem, lots of tool support

How do RDBMSes do it?

- Transactions on a single machine: (relatively) easy!
- Partition tables to keep transactions on a single machine
  - Example: partition by user
- What about transactions that require multiple machines?
  - Example: transactions involving multiple users

Solution: Two-Phase Commit

Okay everyone, PREPARE!

Good. COMMIT!

DONE!

YES

ACK!

YES

ACK!

YES

ACK!

2PC: Sketch

Okay everyone, PREPARE!

ROLLBACK!

DONE!

NO!

ACK!

YES

ACK!

Subordinates

2PC: Assumptions and Limitations

• Assumptions:
  • Persistent storage and write-ahead log at every node
  • WAL is never permanently lost

• Limitations:
  • It’s blocking and slow
  • What if the coordinator dies?

Solution: Paxos!
(details beyond scope of this course)

RDBMSes: Pain Points

Source: www.flickr.com/photos/spencerdahl/6075142688/ via Big Data Infrastructure 2015
#1: Must design up front, painful to evolve

#2: 2PC is slow!

Source: Lin et al. Big Data Infrastructure. UMD Spring 2015
#3: Cost!
What do RDBMSes provide?

- Relational model with schemas
- Powerful, flexible query language
- Transactional semantics: ACID
- Rich ecosystem, lots of tool support

What if we want *a la carte*?

Source: www.flickr.com/photos/vidiot/18556565/
Features *a la carte*?

- What if I’m willing to give up consistency for scalability?
- What if I’m willing to give up the relational model for something more flexible?
- What if I just want a cheaper solution?

Enter... NoSQL!

NoSQL  (Not only SQL)

- Horizontally scale “simple operations”
- Replicate/distribute data over many servers
- Simple call interface
- Weaker concurrency model than ACID
- Efficient use of distributed indexes and RAM
- Flexible schemas

Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. SIGMOD Record.
(Major) Types of NoSQL databases

- Key-value stores
- Column-oriented databases
- Document stores
- Graph databases

Key-Value Stores

Source: Wikipedia (Keychain)
Key-Value Stores: Data Model

• Stores associations between keys and values
• Keys are usually primitives
  • For example, ints, strings, raw bytes, etc.
• Values can be primitive or complex: usually opaque to store
  • Primitives: ints, strings, etc.
  • Complex: JSON, HTML fragments, etc.

Key-Value Stores: Operations

- **Very simple API:**
  - Get – fetch value associated with key
  - Put – set value associated with key

- **Optional operations:**
  - Multi-get
  - Multi-put
  - Range queries

- **Consistency model:**
  - Atomic puts (usually)
  - Cross-key operations: who knows?

Key-Value Stores: Implementation

• Non-persistent:
  • Just a big in-memory hash table
• Persistent
  • Wrapper around a traditional RDBMS

What if data doesn’t fit on a single machine?

Simple Solution: Partition!

• Partition the key space across multiple machines
  • Let’s say, hash partitioning
  • For $n$ machines, store key $k$ at machine $h(k) \mod n$

• Okay... But:
  1. How do we know which physical machine to contact?
  2. How do we add a new machine to the cluster?
  3. What happens if a machine fails?

See the problems here?

Clever Solution

- Hash the keys
- Hash the machines also!

Distributed hash tables!

Routing: Which machine holds the key?

Each machine holds pointers to predecessor and successor

Send request to any node, gets routed to correct one in $O(n)$ hops

Can we do better?

Routing: Which machine holds the key?

Each machine holds pointers to predecessor and successor
+ “finger table” (+2, +4, +8, ...)

Send request to any node, gets routed to correct one in $O(\log n)$ hops

Simpler Solution

Routing: Which machine holds the key?

New machine joins: What happens?

How do we rebuild the predecessor, successor, finger tables?


Stoica et al. (2001). Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. SIGCOMM.

Cf. Gossip Protocols
Another Refinement: Virtual Nodes

• Don’t directly hash servers
• Create a large number of virtual nodes, map to physical servers
  • Better load redistribution in event of machine failure
  • When new server joins, evenly shed load from other servers

**Data Model**

- A table in Bigtable is a sparse, distributed, persistent multidimensional sorted map
- Map indexed by a row key, column key, and a timestamp
  - \((\text{row:string}, \text{column:string}, \text{time:int64}) \rightarrow \text{uninterpreted byte array})\)
- Supports lookups, inserts, deletes
  - Single row transactions only

Rows and Columns

• Rows maintained in sorted lexicographic order
  • Applications can exploit this property for efficient row scans
  • Row ranges dynamically partitioned into tablets
• Columns grouped into column families
  • Column key = \textit{family}:\textit{qualifier}
  • Column families provide locality hints
  • Unbounded number of columns
  • Column entries are sparse for each row

At the end of the day, it’s all key-value pairs!

Bigtable Building Blocks

- GFS
- Chubby
- SSTable

SSTable

- “Sorted String Table” – a variant of log-structured merge trees
- Persistent, ordered immutable map from keys to values
  - Stored in GFS
- Sequence of blocks on disk plus an index for block lookup
  - Can be completely mapped into memory
- Supported operations:
  - Look up value associated with key
  - Iterate key/value pairs within a key range

Source: Graphic from slides by Erik Paulson via Big Data Infrastructure 2015
**Tablet**

- Dynamically partitioned range of rows
- Built from multiple SSTables

Table

- Multiple tablets make up the table
- SSTables can be shared

Architecture

- Client library
- Single master server
- Tablet servers

Bigtable Master

- Assigns tablets to tablet servers
- Detects addition and expiration of tablet servers
- Balances tablet server load
- Handles garbage collection
- Handles schema changes

Bigtable Tablet Servers

- Each tablet server manages a set of tablets
  - Typically between ten to a thousand tablets
  - Each 100-200 MB by default
- Handles read and write requests to the tablets
- Splits tablets that have grown too large

Tablet Serving

Compactions

• Minor compaction
  • Converts the memtable into an SSTable
  • Reduces memory usage and log traffic on restart

• Merging compaction
  • Reads the contents of a few SSTables and the memtable, and writes out a new SSTable
  • Reduces number of SSTables

• Major compaction
  • Merging compaction that results in only one SSTable
  • No deletion records, only live data

• Key point: sequential writes only
  • What about SSDs? Is this point obsolete?

Bigtable Applications

- Data source and data sink for MapReduce
- Google’s web crawl
- Google Earth
- Google Analytics

Challenges

• Keep it simple!
• Keep it flexible!
• Push functionality onto application

... and it’s still hard to get right!
Example: splitting and compaction storms

Back to consistency...
Consistency Scenarios

- People you don’t want seeing your pictures:
  - Alice removes Mom from list of people who can view photos
  - Alice posts embarrassing pictures from Spring Break
  - Can Mom see Alice’s photo?
- Why am I still getting messages?
  - Bob unsubscribes from mailing list
  - Message sent to mailing list right after
  - Does Bob receive the message?

Three Core Ideas

• Partitioning (sharding)
  • For scalability
  • For latency

• Replication
  • For robustness (availability)
  • For throughput

• Caching
  • For latency

CAP “Theorem” (Brewer, 2000)

Consistency
Availability
Partition tolerance

... pick two

CAP Tradeoffs

- CA = consistency + availability
  - E.g., parallel databases that use 2PC
- AP = availability + tolerance to partitions
  - E.g., DNS, web caching

Is this helpful?

- CAP not really even a “theorem” because vague definitions
- More precise formulation came a few years later

Wait a sec, that doesn’t sound right!

Abadi Says...

• CP makes no sense!
• CAP says, in the presence of P, choose A or C
  • But you’d want to make this tradeoff even when there is no P
• Fundamental tradeoff is between consistency and latency
  • Not available = (very) long latency

Move over, CAP

- PACELC ("pass-elk")
- PAC
  - If there’s a partition, do we choose A or C?
- ELC
  - Else, do we choose latency or consistency?

Replication possibilities

• Update sent to all replicas at the same time
  • To guarantee consistency you need something like Paxos
• Update sent to a master
  • Replication is synchronous
  • Replication is asynchronous
  • Combination of both
• Update sent to an arbitrary replica

All these possibilities involve tradeoffs! “eventual consistency”

Three Core Ideas

• Partitioning (sharding)
  • For scalability
  • For latency

• Replication
  • For robustness (availability)
  • For throughput

• Caching
  • For latency

“Units of Consistency”

- Single record:
  - Relatively straightforward
  - Complex application logic to handle multi-record transactions
- Arbitrary transactions:
  - Requires 2PC/Paxos
- Middle ground: entity groups
  - Groups of entities that share affinity
  - Co-locate entity groups
  - Provide transaction support within entity groups
  - Example: user + user’s photos + user’s posts etc.

More “Units of Consistency”

- CRDTs – “Conflict-free Replicated Data Type”
  - E.g. sets, counters
- Vector clocks
  - First read old record and vector clock value
  - Then send this vector clock with new, updated value
  - Push conflict-resolution back to the application
- Tunable availability vs. consistency
  - All, One, Quorum
  - \( R + W \geq N \)

Three Core Ideas

- Partitioning (sharding)
  - For scalability
  - For latency
- Replication
  - For robustness (availability)
  - For throughput
- Caching
  - For latency

This is really hard!

Now imagine multiple datacenters... What’s different?

Three Core Ideas

• Partitioning (sharding)
  • For scalability
  • For latency
• Replication
  • For robustness (availability)
  • For throughput
• Caching
  • For latency

Facebook Architecture

Read path:
Look in memcached
Look in MySQL
Populate in memcached

Write path:
Write in MySQL
Remove in memcached

Subsequent read:
Look in MySQL
Populate in memcached

1. User updates first name from “Jason” to “Monkey”.
2. Write “Monkey” in master DB in CA, delete memcached entry in CA and VA.
3. Someone goes to profile in Virginia, read VA slave DB, get “Jason”.
4. Update VA memcache with first name as “Jason”.
5. Replication catches up. “Jason” stuck in memcached until another write!

Facebook Architecture

Solution: Piggyback on replication stream, tweak SQL

```sql
REPLACE INTO profile (`first_name`) VALUES ('Monkey')
WHERE `user_id`='jsobel' MEMCACHE_DIRTY 'jsobel:first_name'
```
Three Core Ideas

- Partitioning (sharding)
  - For scalability
  - For latency
- Replication
  - For robustness (availability)
  - For throughput
- Caching
  - For latency

Google’s Spanner

• Features:
  • Full ACID transactions across multiple datacenters, across continents!
  • External consistency: wrt globally-consistent timestamps!

• How?
  • TrueTime: globally synchronized API using GPSes and atomic clocks exposes uncertainty
  • Use 2PC but use Paxos to replicate state

• Tradeoffs?

Applications

- Cassandra
  - DHT, column families, LSM trees
- Riak
  - DHT, key-value, LSM trees
- LevelDB
  - Embeddable, key-value, LSM trees
- RocksDB
  - Embeddable, key-value, LSM trees
- Elasticsearch
  - Consistent hashing, inverted index, LSM trees
- Redis
  - Consistent hashing, key-value, AOF + snapshots