Distributed Data-Intensive Systems

CSCE 438
Texas A&M University
Department of Computer Science and Engineering
Instructor: Prof. James Caverlee

Cloud Computing
April 22, 2014

Slides remixed from Widom, Ives, Haeberlen, Leskovec, Lin
NoSQL

• “SQL” = Traditional relational DBMS

• Recognition over past decade or so:
  • Not every data management/analysis problem is best solved using a traditional relational DBMS
  • “NoSQL” = “No SQL” = Not using traditional relational DBMS = Not Only SQL

• “No SQL” ≠ Don’t use SQL language
Traditional DBMS

• Database Management System (DBMS)

• Typically provides:
  • “efficient, reliable, convenient, and safe multi-user storage of and access to massive amounts of persistent data.”
  • Often encoded as **ACID properties**
    • Atomicity, Consistency, Isolation, Durability
  • Think: Banking transactions, NORAD defense

• In Web + Big Data + Cloud era:
  • Not every data management/analysis problem is best solved using a traditional DBMS
NoSQL Systems

• Alternative to traditional relational DBMS
  • Flexible schema
  • Quicker/cheaper to set up
  • Massive scalability
  • Relaxed consistency $\rightarrow$ higher performance & availability

• No declarative query language $\rightarrow$ more programming

• Relaxed consistency $\rightarrow$ fewer guarantees

• BASE properties = Basic availability, Soft-state, Eventual Consistency

• Think: Facebook friend feed, Netflix recommenders, …
NoSQL Systems

- Designed to achieve high scalability and high availability for essential functionalities, trading off for some features of traditional DBMS
  - No relational data model, no joins, limited or no transaction
- May offer weaker data consistency such as Eventual Consistency
  - This design choice is often explained by the CAP theorem: Pick partition-tolerance and availability
Eventual Consistency

• A model originally proposed for disconnected operation (e.g., mobile computing)

• Different nodes keep replicas and each update is “eventually” propagated to each replica
  • And eventually, there is agreement on which update is the latest
  • DNS is the most well-known system implementing eventual consistency

• Usual definition is counterfactual: “once updating ceases, and the system stabilizes, then after a long enough period, all replicas will have the same value”
NoSQL Examples:

- Could think of MapReduce as a NoSQL framework
  - No data model, data stored in files
  - User provides functions; System provides fault tolerance, scalability
  - SQL-like queries provided by Hive, Pig (built on top of MapReduce)

- Key-value stores
  - Google BigTable, Amazon Dynamo, Redis, MemcacheDB, HBase, …

- Document stores
  - MongoDB, CouchDB, SimpleDB, …
Key-value stores

• The key-value store (KVS) is a simple abstraction for managing persistent state
  • Data is organized as (key, value) pairs

• Only three basic operations:
  • PUT(key, value)
  • GET(key) → value
  • DELETE(key)
Key-value stores in the cloud

- Many situations need hosting of large data sets
  - Examples: Amazon catalog, eBay listings, Facebook pages, …

- Ideal: Abstraction of a 'big disk in the clouds', which would have:
  - Perfect durability – nothing would ever disappear in a crash
  - 100% availability – we could always get to the service
  - Zero latency from anywhere on earth – no delays!
  - Minimal bandwidth utilization – we only send across the network what we absolutely need
  - Isolation under concurrent updates – make sure data stays consistent
However ...

• Why isn't this feasible?

• The “cloud” exists over a physical network
  • Communication takes time, esp. across the globe
  • Bandwidth is limited, both on the backbone and endpoint

• The “cloud” has imperfect hardware
  • Hard disks crash
  • Servers crash
  • Software has bugs
Finding the right trade-off

• In practice, we can't have everything
  • ... but most applications don't really need 'everything'!

• Some observations:
  • Read-only (or read-mostly) data is easiest to support
    • Replicate it everywhere! No concurrency issues!
    • But only some kinds of data fit this pattern – examples?
  • Granularity matters: “Few large-object” tasks generally tolerate longer latencies than “many small-object” tasks
    • Fewer requests, often more processing at the client
    • But it’s much more expensive to replicate or to update!
  • Maybe it makes sense to develop separate solutions for large read-mostly objects vs. small read-write objects!
    • Different requirements = different technical solutions
Specialized key-value stores

• Cloud KVS are often specialized for a particular tradeoff or usage scenario

• Example: Amazon’s solutions
  • Simple Storage Service (S3):
    • large objects – files, virtual machines, etc.
    • assumes objects change infrequently
    • objects are opaque to the storage system
  • SimpleDB:
    • small objects – Java objects, records, etc.
    • generally updated more frequently; greater need for consistency
    • generally multiple attributes or properties, which are exposed to the storage system
Recap

• Ideally, we would simply like the abstraction of a 'big disk in the cloud'
  • Perfect durability, availability, consistency, throughput, ...

• Practical constraints require compromises
  • Propagation delay, unreliable hardware/software, ...

• Hence, we need to make the right tradeoff
  • For example, specialize KVS for particular workloads
  • No one-size-fits-all solution; different solutions are useful in different situations
Big Objects: Amazon S3

- S3 = Simple Storage System
  - Think roughly of an Internet file system
- Stores large objects (=values) that may have access permissions
  - Used in “cloud backup” services like Jungle Disk
  - Used to distribute software packages
  - Used internally by Amazon to store virtual machines
- “Up to 99.99999999% durability, 99.99% availability” (“ten nines” and “four nines”)
S3: Key concepts

• S3 consists of:
  • objects – named items stored in S3
  • buckets of objects – think of these as volumes in a filesystem
  • the console includes a notion of folders, but these are not intrinsic to S3

• Names within a bucket must uniquely identify a single object
  • i.e., keys must be unique
S3: Keys and objects

• What can we use as keys?
  • Keys can be any string

• What can we use as objects?
  • Objects can be from 1 byte to 5 TB, any format
  • Number of objects is 'unlimited'

• Where can objects be stored?
  • Can be assigned to specific geographic regions (Washington, Virginia, California, Ireland, Singapore, Tokyo, ...)
  • Why is this important?
S3: Different ways to access objects

- Objects in S3 can be accessed
  - ... via REST or SOAP
  - ... via BitTorrent
  - ... over the web: http://s3.amazonaws.com/bucket/key
  - Web Services use HTTP (the Web browser protocol over sockets) and XML to send requests and data
  - AWS Console also enables configuration
S3: Access permissions

- Permissions are assigned through Access Control Lists (ACLs)
  - Essentially, a list of users/groups permissions
  - Bucket permissions are inherited by objects unless overridden at the object level

- What can you control?
  - Can be at the level of buckets or individual objects
  - Available rights: Read, write, read ACL, write ACL
  - Possible grantees: Everyone, authenticated users, specific users (by AWS account email address)
Summary: Cloud key-value stores

• Attempt to provide very high durability, availability in a persistent, geographically distributed storage system

• Need to choose compromises due to limitations of communications, hardware, software
  • Large, seldom-changing objects – eventual consistency and versioned model in S3
  • Small, more frequently changing objects – lower-latency response, conditional updates in SimpleDB

• Both are useful in different situations