Scaling Data and Services

Jeff Chase
Duke University
Challenge: data management

- Data volumes are growing enormously.
- Mega-services are “grounded” in data.
- **How to scale the data tier?**
  - Scaling requires *dynamic placement* of data items across data servers, so we can grow the number of servers.
  - **Sharding** divides data across multiple servers or storage units.
  - **Caching** helps to reduce load on the data tier.
  - **Replication** helps to survive failures and balance read/write load.
  - Caching and replication require *careful update protocols* to ensure that servers see a consistent view of the data.
Concept: load spreading

- Spread (“deal”) the data across a set of storage units.
  - Make it “look like one big unit”, e.g., “one big disk”.
  - Redirect requests for a data item to the right unit.

- The concept appears in many different settings/contexts.
  - We can spread load across many servers too, to make a server cluster look like “one big server”.
  - We can spread out different data items: objects, records, blocks, chunks, tables, buckets, keys…
  - Keep track using maps or a deterministic function (e.g., a hash).

- Also called sharding, declustering, striping, “bricks”.
“Sharding”

https://code.msdn.microsoft.com/windowsazure/sharding-in-azure-using-0171324f
Key-value stores

- Many mega-services are built on **key-value stores**.
  - Store variable-length content objects: think “tiny files” (**value**)
  - Each object is named by a **key**, usually fixed-size.
  - Key is also called a **token**: not to be confused with a crypto key! Although it may be a content hash (SHAx or MD5).
  - Simple **put/get** interface with no offsets or transactions (yet).
  - Goes back to literature on Distributed Data Structures [Gribble 2000] and **Distributed Hash Tables (DHTs)**.

[Image from Sean Rhea, opendht.org, 2004]
Scalable key-value stores

- Can we build massively scalable key/value stores?
  - Balance the load: distribute the keys across the nodes.
  - Find the “right” server(s) for a given key.
  - Adapt to change (growth and “churn”) efficiently and reliably.
  - Bound the “spread” of each object (to reduce cost).

- Warning: it’s a consensus problem!

- What is the consistency model for massive stores?
  - Can we relax consistency for better scaling? Do we have to?
Key-value stores

- Data objects named in a “flat” key space (e.g., “serial numbers”)
- K-V is a simple and clean abstraction that admits a scalable, reliable implementation: a major focus of R&D.
- Is put/get sufficient to implement non-trivial apps?
Service-oriented architecture of Amazon’s platform

Dynamo is a scalable, replicated key-value store.
Memcached is a scalable in-memory key-value cache.
Storage services: 31 flavors

- Can we build rich-functioned services on a scalable data tier that is “less” than an ACID database or even a consistent file system?

People talk about the “NoSQL Movement” to scale the data tier beyond classic databases. There’s a long history.

Today most of the active development in scalable storage is in key-value stores.
Load spreading and performance

- What effect does load spreading across N units have on performance, relative to 1 unit?
- What effect does it have on throughput?
- What effect does it have on response time?
- How does the workload affect the answers?
- What if the accesses follow a skewed distribution, so some items are more “popular” than others?
“Hot spot” bottlenecks

What happens if the workload references items according to a skewed popularity distribution?

• Some items are “hot” (popular) and some are “cold” (rarely used).
• A read or write of a stored item must execute where the item resides.
• The servers/disks/units that store the “hot” items get more requests, resulting in an unbalanced load: they become “hot” units.
• The “hot” units saturate before the others (bottleneck or hot spot).
• Requests for items on “hot” units have longer response times. (Why?)
What about failures?

- **Systems fail.** Here’s a reasonable set of assumptions about failure properties for servers/bricks (or disks)
  - **Fail-stop** or **fail-fast** fault model
  - Nodes either function correctly or remain silent
  - A failed node may restart, or not
  - A restarted node loses its memory state, and recovers its secondary (disk) state

- If failures are random/independent, the probability of **some** failure is linear with the number of units.
  - Higher scale $\rightarrow$ less reliable!
“Declustering” data

Bricks

Clients

[drawing adapted from Barbara Liskov]
“Declustering” data

Bricks

Clients

[drawing adapted from Barbara Liskov]
Replicating data

Bricks

Coordinators

[drawing adapted from Barbara Liskov]
Replicating data

Bricks

Coordinators

[drawing adapted from Barbara Liskov]
Replicating data

[Drawing adapted from Barbara Liskov]
Replicating data

Bricks

Coordinators

[drawing adapted from Barbara Liskov]
Scalable storage: summary of drawings

- The items A, B, C could be blocks, or objects (files), or any other kind of read/write service request.
- The system can write different items to different nodes, to enable reads/writes on those items to proceed in parallel (declustering).
  - How does declustering affect throughput and response time?
- The system can write copies of the same item to multiple nodes (replication), to protect the data against failure of one of the nodes.
  - How does replication affect throughput and response time?
- Replication → multiple reads of the same item may proceed in parallel.
- When a client reads an item, it can only read it from a node that has an up-to-date copy.
- Where to put the data? How to keep track of where it is? How to keep the data up to date? How to adjust to failures (node “churn”)?
Recap: scalable data
An abstract model

This model applies to a service cluster serving clients, or to an I/O system receiving block I/O requests from a host, or both.

Pending requests build up on one or more queues, as modeled by queueing theory (if assumptions of the theory are met).

Throughput (as a function of N) depends in part on the redundancy policy chosen to protect against failures of individual bricks.

Requests (e.g., reads and writes on blocks) arrive.

A dispatcher with a request routing policy draws requests from the queues and dispatches them to an array of N functional units (“bricks”: disks, or servers, or disk servers). Throughput depends on a balanced distribution, ideally with low spread (for locality and cache performance).