Scaling an index to the exabytes

Andrew Baptist
Cleversafe
Problem statement

- Problem: Create a system to index trillions of objects (data and metadata) with high reliability and availability.
  - Examined existing SQL and NoSQL options
- No existing solutions met our needs
  - Due to the constraints of the CAP theorem.
  - We ended up building a new scalable index.
What is the CAP theorem

- You can't have your cake and eat it too
  - Consistency
    - All clients have the same view of the data
  - Availability
    - All clients can read or write data
  - Partition Tolerance
    - The system runs across multiple nodes

Traditional Databases (MySQL)

Highly available storage (CouchDB, Amazon S3)

Distributed Databases (MongoDB, Zookeeper)
CAP theorem demonstrated

Dealing with a network partition

Option 1 – Block the withdrawal
Not Available

Option 2 – Allow both withdrawals
Not Consistent

Option 3 – Don’t use multiple databases, (and don’t allow remote ATM’s)
Not partition tolerant
Relational databases

- Choose Consistency and Availability
  - Are Partition Intolerant
- Examples are Oracle, MySQL
- Distributed versions choose consistency over availability
  - Relational databases can not deal with inconsistency / eventual consistency
“NoSQL” databases

- Relax the constraints of Relational databases to allow a scale-out architecture
  - Run over a network, so need to choose between availability and consistency
- Consistency option
  - MongoDB – single master
- Availability option
  - CouchDB – allows writes and “syncs up” later
- Let the user decide
  - Amazon DynamoDB – user specifies “consistency” or “availability” for each read and write
Quorum agreement (Zookeeper)

- Any 4 nodes are a quorum (can read/write)
- Each node stores a copy of all data
- Can tolerate up to 3 failures anywhere in the system (networks, nodes, ...)

Withdraw $100 from account

Account balance $150

Withdraw $100 from account
From quorum to erasure coding

- Quorum methods can provide high reliability, but also high data overhead cost (7 copies in this example).
- Erasure coding can provide the same guarantees as quorum (for reliability and availability) but at much lower cost.
- Strictly speaking, they choose consistency over availability.
  - Availability can be made arbitrarily high by increasing the number of nodes.
- Example 24 servers at 8 sites with a quorum threshold of 14.
  - System will be available 99.999999% of the time.
  - To achieve this without erasure coding requires 24 copies.
  - With erasure coding overhead is only 24/14 ≈ 1.71.
An index performs the following 4 operations

- **Insert** \((k, v)\) - add \((k, v)\) to index
- **Delete** \((k)\) - deletes \((k, v)\) from index
- **Read** \((k)\) - returns \((v)\) s.t. \((k, v)\) exists in index
- **GetNext** \((k)\) -> returns smallest \(k' > k\) in index

Depending on index flavor, Read and GetNext may be combined/modified.
On-Disk vs In-Memory index

- In-Memory, a structure like a Binary Tree implements these operations efficiently Log(n) for all operations

- On-disk, typically a B-Tree is used
  Still Log(n), but lower constant factor
B-Tree, B⁺-Tree properties

- A B⁺-Tree is like a B-Tree, with data at the leaves
  - Smaller internal nodes are more easily cached
- B-Trees (and B⁺ trees) minimizes disk seeks
  - 1 billion entries can be indexed with 4 levels
- Latches are required for index page access and updates
  - Since a write or delete to a page may split/join that page/path, the entire path needs to be locked
  - B-Tree latches are different than database locks which have a higher level semantic meaning
Optimistic concurrency storage interface

Read \((k)\) – returns revision \((r)\) and data \((v)\) for key \((k)\)

Write\((k, v, r_{\text{old}}, r_{\text{new}})\) – try to overwrite \(r_{\text{old}}\) with \(r_{\text{new}}\)

- Read operations always read highest revision
- Atomic “Read modify write” for updates
  - Write succeeds if \(r_{\text{old}}\) matches the existing revision
- Special write cases
  - A “new write” is a write with \(r_{\text{old}} = \text{null}\)
  - A “delete” is a write with \(v = \text{null}\)
Performing a consistent update

- Optimistic because we expect it will usually succeed
- Implementing “read modify write” updates

```
do
  Read(k) -> (v_{old}, r_{old})
  modify(v_{old}) -> v_{new}
  Write(k, v_{new}, r_{old}, r_{new}) -> result
while (result != conflict)
```

- Note: this is only on a single key/object at a time
- This model works best when infrequent conflicts, but will succeed even with conflicts (after a number of retries)
A Blink-Tree is just like a B⁺-Tree but does not require latches for reads.
- Not traditionally used since a latch is normal “cheap”
- Adds some complexity to algorithm, but same runtime for ops

General idea – links between neighbor nodes

Tree traversal is top-down, left-to-right
- Insertion is bottom up

Splits and joins can be performed concurrently with reads
- Joins were not addressed in original paper, but added later
Adding a new entry – with split

Add the new entry “10”

Create new “unlinked” node

Update the “left” node

Update the “parent” node
Joining nodes, adding height

- Joining nodes is the inverse process of splitting nodes
  - 1) Update parent, 2) update left sibling, 3) delete node
  - Set thresholds for joining/splitting to avoid thrashing
    - Split size – 2000
    - Join size – 500

- Adding height – when the root node gets too full
  - Add a new root node (with one child), then split old root node

- If any of the split, join or add height operations fail (due to client failures, timeouts, ...), they can be retried by the next “write” operation on the affected nodes
Optimizations for concurrency

- Tuning the desired node size
  - Larger size means less levels (less round trips)
  - Smaller size means less conflicts on updates
  - Empirical testing has resulted in an optimal size of about 2000 entries
- Multiple concurrent updates to the same node cause repeated conflicts
  - For a single client, batch them together
  - For different clients, use exponential backoff
- Cache high level index nodes on clients
  - Writes are always consistent
  - Reads return stale data, but can bypass the index if necessary
  - Periodically recheck cached nodes for updates
- To avoid write conflicts on splits, the “full/empty” level where joins/splits happens is probabilistic rather than a hard limit
  - At 1900 nodes small chance of split, at 2000 high chance
  - Reduces the case everyone trying to split at the same time
Performance results
Cleversafe has built a highly reliable, highly available index to scale to exabytes of data and metadata.

By using quorum agreement on updates with erasure coding for storage, it is possible to get consistency and arbitrarily high availability in a networked system.

Performance increases as the index gets larger.
Questions?