The SCADS Toolkit

Nick Lanham, Jesse Trutna

(it’s catching…)
SCADS Overview

- SCADS is a scalable, non-relational datastore for highly concurrent, interactive workloads.

- Scale Independence - as new users join
  - No changes to application
  - Cost per user doesn’t increase
  - Request latency doesn’t change

- Key Innovations
  1. Performance safe query language
  2. Declarative performance/consistency tradeoffs
  3. Automatic scale up and down using machine learning
Investigated other open-source distributed key-value stores
  - Cassandra, Hypertable, CouchDB
  - Monolithic, opaque point solutions
  - Make many decisions about how to architect the system a-prori

Want set of components to rapidly explore the space of systems’ design
  - Extensible components communicate over established APIs
  - Understand the implications and performance bottlenecks of different designs
SCADS Components

- Application
- Machine Learning
- Client Library
- Data Placement
- Cluster Manager
- Storage Layer
Component Responsibilities

- **Storage Layer**
  - Persist and serve data for a specified key responsibility
  - Copy and sync data between storage nodes

- **Data Placement Layer**
  - Assign node key responsibilities
  - Manage replication and partition factors
  - Provide clients with key to node mapping
  - Provides mechanism for machine learning policies

- **Client Library**
  - Hides client interaction with distributed storage system
  - Provides higher-level constructs like indexes and query language
Storage Layer

.Key-value store that supports range queries built on BDB

-API

Record get(NameSpace ns, RecordKey key)
list<Record> get_set(NameSpace ns, RecordSet rs)
bool put(NameSpace ns, Record rec)
i32 count_set(NameSpace ns, RecordSet rs)
bool set_responsibility_policy(NameSpace ns, RecordSet policy)
RecordSet get_responsibility_policy(NameSpace ns)
bool sync_set(NameSpace ns, RecordSet rs, Host h, ConflictPolicy policy)
bool copy_set(NameSpace ns, RecordSet rs, Host h)
bool remove_set(NameSpace ns, RecordSet rs)
Storage Layer

Clients / Data Placement Components

Storage Node

Storage Node
Storage Layer

Copy Times
100 Megabytes

- Internal Copy Port
- C Thrift Client, one force to disk
- C Thrift Client, force each record
- Ruby Thrift Client, force each record

Seconds

- 9.5 s
- 105.7 s
- 995.3 s
- 2025.6 s
Replicas may diverge during network partitions (in order to preserve availability).

Need way to resolve divergence when connectivity is restored.

But, nodes may store arbitrarily large quantities of data

So...

Need efficient way to determine set difference between nodes (key-value pairs with differing values or the presence of new pairs)

Sounds like a job for: Merkle Trees!
Merkle Tree

- Merkle Tree (a.k.a Hash Tree)
  - Tree that computes a signature for a file by recursively hashing the nodes of the tree.
  - Can quickly determine which portions of a file are different

- Quick How-to:
  - Take a file of length $n$
Merkle Tree: Construction

Hash of each node is hash of children

File (n pages)

Merkle Tree
Merkle Tree: Inserts

File (n pages)

Merkle Tree
Merkle Tree: Sync
Alas, Merkle Tree relies on known quantity of data. :(

We have a key-value store, may have inserts or deletions on one side and not the other... Need a dynamic data structure.

Furthermore, we can't use a regular B-Tree.

- Insertions may occur in different orders
- Re-balancing the root would result in entirely different hash for the tree.

We need a tree that has a deterministic structure, given a set of key-value pairs

- Trie!
Trie (a.k.a. Prefix Tree)

- Edges labeled with characters
- Key is path to leaf
- Compute hashes up this tree
Optimization:
- Collapse any node that has only one child
Sync: performance

**Sync Time**

(100 MB)

- Time (sec)
- Divergence (% of total dataset)

- Merkle
- Scan

- Graph showing time vs. divergence for Merkle and Scan.
Merkle Trees are often called Tiger Trees.

We are using the Tiger Hash Algorithm

Thus, we are using a “Patricia Merkle Tiger Trie”

Awesome.
Data Placement & Client Library

Data Placement Layer
- Maintains global view of data placement in cluster via node to key range mappings
- Orchestrates transfer of data and changes in node responsibility
- Polices without interruption in data availability

Client Library
- Receives requests from client applications
- Caches key to node mappings received from DP layer
- Current implementation: ROWA
- Coordinates get_set() requests to nodes to satisfy client
Mechanics of Data Movement

- Machine learning: “move data from node A to node B”
- Copy data from A to B
- Map data assignments to A and B
- Assign B’s responsibility policy
- Update A’s responsibility policy
- Sync A and B
- Remove old data from A
Goal

- Gain experience with how application developers use SCADS
- See what performance problems arise

Twitter clone written in RoR by undergraduates
- Use SCADS instead of ActiveRecord

DEMO!
- [http://scadr.radlab.net](http://scadr.radlab.net)
  - Use it!
Performance Tests: GREP

Load times

Task times
Performance Tests: Storage Layer

- Num clients 1
- Num clients 2
Performance Tests: Storage Layer

Num clients 1 2
Performance Tests: Data Placement
Performance Tests: More Nodes

Performance of Partitioned Cluster

Number of Servers

Total Requests/Sec

0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
160

0 5 10 15 20 25 30 35
Future Work

- Predicting system performance
  - X-Trace track requests through system components
  - Built into Thrift protocol layer