Facebook ODS
Real-time monitoring system built on HBase

Vinod Venkataraman (vinodv@fb.com)
Charles Thayer (cgthayer@fb.com)
Liyin Tang (liyin.tang@fb.com)
October 24, 2012
# Agenda

<table>
<thead>
<tr>
<th></th>
<th>Topic</th>
<th>Presenter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction (Vinod)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Architecture (Vinod)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Pitfalls (Charles)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Failover (Charles)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>HBase performance tuning (Liyin)</td>
<td></td>
</tr>
</tbody>
</table>
Introduction
ODS

- Time series data for real-time monitoring and trends
- Collects metrics from each server
- Aggregates in useful ways
- Detects and alerts on anomalies
- 2.5 B data points written per minute, 16 k reads per minute
Motivation

Troubles with the old MySQL-based system

- Unavailability on hardware failures
- MySQL table size limitations
- Sharding scheme created hotspots
- Data growth manageability headaches
Motivation

Why HBase?

- Optimized for append-heavy, light read workloads
- Inherent sharding, connection handling, failure recovery
- Map-Reduce a natural fit for time rollups
- Strong in-house engineering and operational support
- Prototype provided promising results even when unoptimized
Architecture
New Architecture Overview

- HBase Cluster
  - Time Rollup (Map-Reduce)
  - Storage Servers
    - Server
    - Server
    - Server
    - Server

- ODS Router
  - Aggregator

- Clients
  - Search Indexer
  - Detection/Alerting

- Tailer
  - Tailer
  - Tailer
Split cluster experiment

- ODS uses very little disk
- More memory allows for more cached data
- Limited server types
- What about running HBase and HDFS separately?
- In theory, can scale memory or disk separately as needed

👍 Capacity Engineers, Users like this.
Split cluster experiment

So what went wrong?

- Locality, locality, locality!
- Map-Reduce jobs overload HDFS rack switches
- Compactions, cluster maintenance all overload HDFS rack switches
- Operational overhead
- Fell back to co-locating HBase/ HDFS
Pitfalls
Design: Schema & Migration
Migration: Notes

- Bulkload is **fast** and you’ll need it.
  - We migrated about **6** times.
  - No rewriting data, just **moves** (make backups with distcp)
  - Region **splits** must match.
  - Before reviving, run major **compaction**.

- Can **not** **rename** tables.

- Our process: copy SQL tables to HDFS -> MR -> **bulkload**
You’ll want tools:
- To generate a smaller dataset for testing against.
- To produce text output, for checking results
Migration: Hashes

- Found hotspots.
  - Running MR would kill one RS at a time
  - Watch your puts() per RS for balance
  - Cause: bad rowkey hashing
- Java’s default hashCode() is bad for HBase rowkeys.
  - Poor spread
  - Went with murmur: balance of speed and effectiveness.
- Beware: Migrating a bad rowkey is very different from column data
  - It was a lot of work to cut-over live, and to minimize downtime.
Migration: Endnotes

- Numeric rowkey problems:
  - Avoid writing monotonically increasing sequence-ids
  - Beware of other “sequential” data types:
    - IP addrs, dates, hostnames, etc...

- Natural strings
  - Combine data and hash: “apple” is bad but “f5ac8127:apple” isn’t
  - NoSQL: Ok to have redundant info (especially with compression)

- When deletes don’t delete: versions/timestamps
  - When you delete the last cell, and reveal an older version.
Practice: Threads & Memory
Threads & Memory

- You have more threads than you think (we had >28K)
  - 9 tables, 100 RS, 32 htable objects in a pool
- Limit the HTables with HTablePool.
  - We wrote a new pool on top of HTablePool.
  - Went from 28K -> 400 @idle & 1400 (typical)
- Limit stack size (-Xss128k)
- Memory “Leaks”
  - -XX:MaxDirectMemorySize=256m
Ops: Failover
Failover: Why do you need it?

- **Unavoidable** outages
  - When HDFS **file formats change**, you need conversion time.
  - Namenodes die and bringing up the secondary is **manual**.
- **Rollback**: New software pushes sometimes trigger unexpected performance dips.
Failover:

- **Alert Layer**
- **Clients**
- **ODS Router**
- **Aggregator**
- **Primary Storage Server**
  - **HDFS**
  - **HBase**
  - **Shadow Storage Server**
    - **HDFS**
    - **HBase**
- **Secondary Storage Server**
  - **HDFS**
  - **HBase**
  - **Slave (East)**

**Double-writing**: to primary and secondary (Not replication)

- **Master (West)**
- **Slave (East)**
- **Tailor**
- **Sherlock**
- **Detection**
- **Testing**
Failover: The Reality

- **Easy** -> all the time
- Having a Failover cluster has **saved** us many times. E.g.
  - Region servers **bouncing** up and down (still heartbeating)
  - HDFS can have various failures
  - **MR** can bring down the cluster
  - **Rack failures** survive, but **kill** performance
Failover: Benefits

- Upgrades can be “canary” tested on the secondary cluster.
- Upgrade problems are never experienced by users.
- We can failover first, then investigate problems.
- You have a live backup in a Disaster Event.
- Some MR can be run on the secondary cluster.
- Backups can run on the secondary cluster.
- You can compare both sides to help debug many issues.
Lessons for HBase Performance Tuning
Takeaway I: Shard the application traffic as similar as HBase shards the row key space.
Sharding Overview

- **Aggregator**
- **ODS Router**
- **Clients**
  - Sharding here
  - Sharding here again

**Server Instances**
- **Storage Server/HBase Client**
- **Region Server**

**HBase cluster**
- Sharding here again
Sharding Overview

- Clients
- Aggregator
- ODS Router
- Storage Server/ HBase Client
- Region Server
- HBase clusters

Sharding here

Sharding here again
Sharding Overview

Clients

Aggregator

ODS Router

Use the similar sharding here

Server Instances

Storage Server/HBase Client

Storage Server/HBase Client

Storage Server/HBase Client

Storage Server/HBase Client

Region Server

Region Server

Region Server

Region Server

HBase clusters
Takeaway II: Use the async put API provided by HTableMultiplexer to improve the write throughput
Sync MultiPut

Application Thread I
HTable Instance
htable.put(multiPuts)
r1: cf1:c1:v1
r2: cf1:c1:v2
r3: cf1:c2:v3

Sync API Call

Application Thread II
HTable Instance
htable.put(multiPuts)
r3’: cf1:c1:v1
r4: cf1:c1:v2
r5: cf1:c2:v3

Hconnection Manager:
- r1 → RS A
- r2 → RS B
- r3 → RS A

Region Server A
Region Server B
Region Server C
Region Server D
Region Server E

Slow / crashed
**HTableMultiplexer**

- Application Thread I
  - Shared HTableMultiplexer
    - r1: cf1:c1:v1
    - r2: cf1:c1:v2
    - r3: cf1:c2:v3

- Application Thread II
  - Shared HTableMultiplexer
    - r3': cf1:c1:v1
    - r4: cf1:c1:v2
    - r5: cf1:c2:v3

- Async API Call

- Multiplexer
  - Queue/RS
    - RS A -> r1: r1 -> RS A
    - RS B -> r2: r2 -> RS B
    - RS C -> r3, r3': r3, r3' -> RS C
    - RS D -> r4: r4 -> RS A
    - RS E -> r5: r5 -> RS A

- Region Servers:
  - A
  - B
  - C
  - D
  - E

- Slow / Crashed
Takeaway III: Enhance the block placement to retain the data locality during the region movement
Locality drops when region moves

Region Server A

DataNode A

b1 b2 b3 b4

b5 b6 b7 b8

Region a: HFile a1, a2, .. an

Region Server B

DataNode B

b1 b3

b6 b8

Region Server C

DataNode C

b2 b7

100 % locality

50 % locality

20 % locality
Locality-Based Region Placement

- Assignment Plan
  - Region Placement -> Block Placement
  - Region -> Primary RS, Secondary RS, Tertiary RS
  - Region will be failed over among these 3 locations
  - All the blocks belonging to that region allocated on these 3 locations
  - The plan is calculated offline and stored in the META table
  - Easy to refresh or customize the plan
  - Master executes the assignment plan
Goal: retain the locality during the region movement
Takeaway IV: Keep the hot recent data live in the block cache.
Compaction invalidates the block cache
Tier based compaction (HBase-6371)

- Tier 0
  - Sequence ID (Time)
  - Region
    - MemStore
- Tier 1
  - Compaction Tier 1
    - Block Cache
      - Cached KVs for that region
- Tier 2
  - Compaction Tier 2

New Files -> Old Files

TTL

New Files ->

Sequence ID (Time)
## Takeaways

1. Shard the application traffic similar to HBase
2. Use async puts from HTable Multiplexer
3. Place regions based on locality
4. Keep hot, recent data live in the block cache
Acknowledgements

ODS

HBase Engineering

HBase Production Engineering
Q&A

Email
{ vinodv, cgthayer, liyin.tang } @fb.com

Find us on Facebook

Facebook /UsingHBase