Scaling HBase for Big Data

Salesforce

Ranjeeth Kathiresan
Senior Software Engineer
rkathiresan@salesforce.com

Gurpreet Multani
Principal Software Engineer
gmultani@salesforce.com
Introduction

Ranjeeth Kathiresan is a Senior Software Engineer at Salesforce, where he focuses primarily on improving the performance, scalability, and availability of applications by assessing and tuning the server-side components in terms of code, design, configuration, and so on, particularly with Apache HBase. Ranjeeth is an admirer of performance engineering and is especially fond of tuning an application to perform better.

Gurpreet Multani is a Principal Software Engineer at Salesforce. At Salesforce, Gurpreet has lead initiatives to scale various Big Data technologies such as Apache HBase, Apache Solr, Apache Kafka. He is particularly interested in finding ways to optimize code to reduce bottlenecks, consume lesser resources and achieve more out of available capacity in the process.
Agenda

- HBase @ Salesforce
- CAP Theorem
- HBase Refresher
- HBase Internals
- Data Loading Use Case
- Write Bottlenecks
- Tuning Writes
- Best Practices
- Q&A
HBase @ Salesforce

2200+ Nodes Across All Clusters

120 TB Typical Cluster Data Volume

100+ HBase Clusters

Variety

Simple Row Store
Denormalization
Messaging
Event Log
Analytics
Metrics
Graphs
Cache
CAP Theorem

It is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees\(^1\):

- **Availability**
  Every request receives a (non-error) response

- **Partition tolerance**
  The system works well despite messages being dropped/delayed by network

- **Consistency**
  Every read receives the most recent write or an error

\(^1\) goo.gl/etoDTE
You cannot *always* choose both consistency and availability in a distributed system
High Availability → Replication

Consistency / Latency Tradeoff
if (partitioning) {
    AC_TRADEOFF = true;
} else {
    LC_TRADEOFF = true;
}

aka PACE/LC theorem¹
HBase is PC/EC

It will always strive to achieve Consistency, and pay the Availability and Latency cost to do so.
HBase Refresher

“A sparse, distributed, persistent, multidimensional, sorted map”*

- **Map**: Key-value store
- **Sparse**
- **Distributed**: Runs on top of HDFS
- **Persistent**
- **Multidimensional**: Column Families
- **Sorted** by the keys

* goo.gl/7mVbaC
HBase Internals

Warm up

- **Zookeeper** is used for co-ordination service
- **META Region** stores the data about other HBase tables
- HBase Data is stored as **HFiles** in HDFS
- HBase uses two level **Log-structured merge-tree** algorithm
HBase Internals
Write Operation
HBase Internals

Compaction

Main purpose of compaction is to optimize read performance by reducing the number of disk seeks

Minor Compaction

**Trigger**: Automatic based on configurations

**Mechanism**
- Reads a configurable number of smaller HFiles and writes into a single large HFile

Major Compaction

**Trigger**: Scheduled or Manual

**Mechanism**
- Reads all HFiles of a region and writes to a single large HFile
- Physical deletion of records
- Tries to achieve high data locality
**HBase Internals**

**Read Operation**

1. Get .META. location
2. Get Region location
3. Get
4. Read
5. Read
6. Read

**Diagram:***

- **Zookeeper**
  - 1. Get .META. location

- **Client**
  - 2. Get Region location
  - 3. Get

- **Region Server**
  - .META. Region
  - 4. Read
  - 5. Read
  - 6. Read

- **Region**
  - Memstore
  - Memstore
  - …

- **Block Cache**

- **HDFS**
  - WAL
  - HFile
  - HFile
  - HFile

- **HBase Internals**

---

*Read Operation Diagram*
One of the use cases is to store and process data in text format.

Lookups from HBase using row key is more efficient.

A subset of data is stored in Solr for effective lookups from HBase.
Data Insights

Volume
- Data Size/Cycle: 200GB
- HBase Data Size/Cycle: 3300GB

Velocity
- Data Influx/Min: 500MB
- Records/Min: 600K

Variety
- Data Format: Text
- Data Format: CSV, JSON

Throughput SLA
- Records/Min: 175K
- Records/Day: 250MM
Write Operation Bottlenecks

**Influx Rate:**
600K Records/Min

**Write Rate:**
60K Records/Min

**Write Operation in progress for >3 days**

**Write Rate dropped to <5K Records/Min after few hours**
Write Operation Tunings

Initial Throughput: 60K Records/Min

Achieved Throughput: 480K Records/Min

Improved throughput by ~8 times & achieved ~3 times more than expected throughput
Region Hot Spotting

Outline: Region Hot Spotting refers to over utilizing a single region server, despite of having multiple nodes in the cluster, during write operation because of using sequential rowkeys.

Scenario

Hey Buddy! I'm overloaded
Not our turn, Yet!!
Not our turn, Yet!!

Impact

Node1
Node2
Node3

Utilization

Time
Write Operation Tunings

Initial Throughput: 60K Records/Min

Achieved Throughput: 480K Records/Min

Salting

Pre-Splitting

Optimal Configuration

Row Size Optimization

Compression

Optimal Read vs. Write Consistency Check
**Salting**

**Outline:** Salting helps to distribute writes over multiple regions by using random row keys

**How do I implement Salting?**

Salting is implemented by defining the rowkeys wisely by adding a salt prefix (random character) to the original key

**Two Common ways of salting**
- Adding a random number as prefix based on modulo
- Hashing the rowkey
Salting

Does it help?

Salting does not resolve Region Hot spotting for the entire write cycle.

**Reason:** HBase creates only one region by default and uses default auto split policy to create more regions.
Write Operation Tunings

Initial Throughput: 60K Records/Min

Achieved Throughput: 480K Records/Min

Salting

Pre-Splitting

Optimal Configuration

Row Size Optimization

Compression

Optimal Read vs. Write Consistency Check
Pre-Splitting

Outline: Pre-Splitting helps to create multiple regions during table creation which will help to reap the benefits of salting

How do I pre-split a HBase table?

Pre-splitting can be done by providing split points during table creation

*Example:* `create 'table_name', 'cf_name', SPLITS => ['a', 'm']`

```
1000
1001
1002
1003
1004
1005
```

Bucket 1[" -> ‘a’]

Bucket 2[‘a’ -> ‘m’]

Bucket 3[‘m’ -> ‘’]
Pre-Splitting

Scenario

GO Regions!!!

Improvement

Utilization

Time

Node1  Node2  Node3
Optimization Benefit

Salting

Pre-Splitting

Current Throughput: 60K Records/Min

Improved Throughput: 150K Records/Min

Throughput Improvement
Write Operation Tunings

Initial Throughput: 60K Records/Min

Achieved Throughput: 480K Records/Min

Salting

Pre-Splitting

Optimal Configuration

Row Size Optimization

Compression

Optimal Read vs. Write Consistency Check
Configuration Tuning

Outline: Default configurations may not work for all use cases. We need to tune configurations based on our use case.

It is 9!!

No. It is 6!!
Configuration Tuning
Region Server Handler Count

- Region Server Handlers (Default Count=10)

**Tuning Benefit**
- Increasing it could help in improving throughput by increasing concurrency
- Thumb Rule -> Low for high payload and high for low payload

**Caution**
- Can increase heap utilization eventually leading to OOM
- High GC pauses impacting the throughput
Configuration Tuning
Region Memstore Size

- Thread which checks Memstore size (Default – 128 MB)

**Tuning Benefit**
- Increasing Memstore size will generate larger HFiles which will minimize compaction impact and improves throughput

**Caution**
- Can increase heap utilization eventually leading to OOM
- High GC pauses impacting the throughput
Configuration Tuning
HStore Blocking Store Files

- Increasing blocking store files improves throughput by allowing client to write more with less pauses
- Can increase heap utilization eventually leading to OOM
- Leading to more HFiles impacting read performance

Default Blocking Store Files - 10
Configuration Tuning
HStore Blocking Wait Time

- **Tuning Benefit**: Decreasing blocking wait time will allow client to write more with less pauses and improves throughput.
- **Caution**: Compaction could take more time as more files could be written without blocking client.

- Time for which writes on Region is blocked (Default – 90 Secs)

---

Client -> Memstore -> Region Store -> Region Server

Region Server

HDFS

- HFile
- HFile
- HFile
- HFile
- HFile

---

- Client
- Memstore
- Region Store
Optimization Benefit

Optimal Configuration

Current Throughput: 150K Records/Min

Improved Throughput: 260K Records/Min

Throughput Improvement

Reduced Resource Utilization
Write Operation Tunings

Initial Throughput: 60K Records/Min

Achieved Throughput: 480K Records/Min

Salting

Pre-Splitting

Optimal Configuration

Row Size Optimization

Compression

Optimal Read vs. Write Consistency Check
Optimal Read vs. Write Consistency Check

Multi Version Concurrency Control (MVCC) is used to achieve Consistency. Read point has to catch up to write point to avoid high delay between read and write versions.

**Issue:** MVCC stuck after few hours of write operation impacting the write throughput drastically as there are 140+ columns per row.

![Scenario Diagram](image)

![Impact Diagram](image)
Optimal Read vs. Write Consistency Check

Solution: Reduce the pressure on MVCC by storing all the 140+ columns in a single cell

Scenario | Throughput | Time | Improvement
---|---|---|---
abc | def | ghi |

Column Representation

```
{  
  "col1":"abc",  
  "col2":"def",  
  "col3":"ghi"  
}
```
Optimization Benefit

- Optimal Read vs. Write Consistency Check
- Stability Improvement
- Steady Resource Utilization
Write Operation Tunings

Initial Throughput: 60K Records/Min

Achieved Throughput: 480K Records/Min

Salting

Pre-Splitting

Optimal Configuration

Row Size Optimization

Compression

Optimal Read vs. Write Consistency Check
Storage Optimization

**Compression:** GZip is better in compression ratio. Snappy is better in resource consumption.

**Row Size Optimization:** Avoid storing empty values when using JSON.

<table>
<thead>
<tr>
<th></th>
<th>Before Optimization</th>
<th>After Optimization</th>
<th>% Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Storage</strong></td>
<td>3300 GB</td>
<td>784 GB</td>
<td>~76%</td>
</tr>
<tr>
<td><strong>Throughput</strong></td>
<td>260K Records/Min</td>
<td>480K Records/Min</td>
<td>~92%</td>
</tr>
</tbody>
</table>

Improved Throughput

Reduced Storage Costs

Productivity Improvement
RECAP
Recap

Data Format → Text

Influx Rate → 500 MB/Min

SLA → 175K Records/Min

Write Throughput

Initial: 60K Records/Min
Achieved: 480K Records/Min

Optimization

- Salting
- Pre-Splitting
- Optimal Row Size
- Compress-ion
- Optimal MVCC
- Config-Tuning

Reduced Storage

Improved Stability

Reduced Resource Utilization
Best Practices

Row key design
• Know your data better before pre-splitting
• Shorter row key but long enough for data access

Minimize Disk I/O
• Less number of Column Families
• Shorter Column Family and Qualifier name

Locality
• Review the locality of regions periodically
• Co-locate Region server and Data node

Maximize Throughput
• Minimize major compactions
• Use high throughput disk
When HBase?

HBase is for you

- Random read/write access to high volumes of data in real time
- No dependency on RDBMS features
- Variable schema with flexibility to add columns
- Single/Range of key based lookups for de-normalized data
- Multiple versions of Big Data

HBase is NOT for you

- Replacement for RDBMS
- Low data volume
- Scanning and aggregation on large volumes of data
- Replacement for batch processing engines like MapReduce/Spark
Q & A