Searching and Analyzing Information Inside Hadoop Platform

Abinasha Karana
25th Feb, 2013
To Search a large dataset

Text Search, Range Search
Faceting, Sorting, Aggregating

1000 columns, multi page document in Billions
What Didn’t Work for US

Map-Reduce
Result not in a mouse click
Lucene is a Java based search engine. To handle large amount of records, the index is partitioned on a dimension and distributed to multiple machines.

What Didn’t Work for US

Lucene Search Engine

- Builds an index and answers queries using the index.
- Read optimized using inverted index.
- Non-Transactional

Lucene Database

- Builds an index and answers queries using the index.
- Write optimized
- Transactional

Didn’t work Because...
Unequal shards leading to hot spotting as well as replication challenges.
What we did

We built a new search/analytics engine on HBase Platform

Leveraging HBase’s auto-sharding and auto-replication
HBase

Hadoop Family Open Source columnar database modeled after Google Big Table.

<table>
<thead>
<tr>
<th>Cell1(int)</th>
<th>Cell2(Float)</th>
<th>Cell3(Float)</th>
<th>Cell4(Float)</th>
<th>Total Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>4.3</td>
<td>87.34</td>
<td>23.11</td>
<td>16 Bytes</td>
</tr>
<tr>
<td>12</td>
<td>8.9</td>
<td>91.12</td>
<td>19.00</td>
<td>16 Bytes</td>
</tr>
<tr>
<td>13</td>
<td>9.1</td>
<td>101.00</td>
<td>27.17</td>
<td>16 Bytes</td>
</tr>
<tr>
<td>12 Bytes</td>
<td>12 Bytes</td>
<td>12 Bytes</td>
<td>12 Bytes</td>
<td>48 Bytes</td>
</tr>
</tbody>
</table>

It is a distributed multi dimensional sorted map with each row having key value maps. Underlying Hadoop-HDFS data storage provides auto replication and auto sharding.
HBase shards the data automatically...

But HBase is designed for write heavy load...

Sharding is just spreading and not replication/clustering

Distributed File System
Hadoop HDFS

But HBase is designed for write heavy load...
In the next few slides you will hear about

My learning from designing, developing and benchmarking HSearch - a real-time search engine whose index is stored and served from HBase

https://github.com/bizosys/
**HSearch Benchmarks**

**Version 2**

**Wikipedia Pages**
- 100 Million Wikipedia pages of total 270GB and no stopwords.
- Data generated by repeating 10 Million Pages 10 Times.
- Search Query Response (Id + Teaser)
  1. Regular word 1.5 Sec.
  2. Common word such as hill found 1.6 million matches and sorted in 7 secs.

Amazon Large instance 7.5GB memory * 11 machines with a single 7.5 K SATA drive

**Version 3**

@ Leading Pharma Research

1. Table Size : 1.2 Billion rows * 800 columns + 1.2 Billion Observation data.
2. A complex query returned 1.4 Million matched rows in 600ms
3. Indexing time 8 Hours.

Amazon Large instance 7.5GB memory * 4 machines with a single 7.5 K SATA drive
HSearch architecture

A Search Query

Female | test day 1 | human | penicillin | blood count | [19-20] AND Rash in face (From Unstructured Descriptions)

Java App Server

Devices

Internet Browser

Hbase + HSearch

HDFS
Where we slowed Down

1. Time spent on reading from disk

2. De-Serialization

3. Network Time
Time spent on reading from disks...

Strategy Applied: Club records to save metadata overhead

1. Stored large cells by merging multiple cols/rows
2. Used a single character as family name
3. Reduced the qualifier name to 1 character.
SSD improved HSearch response time by 66% over SATA.

However, SSD is costlier

We used SSDs for Index only.
### Serialization – De-Serialization …

Strategy Applied: De-Serialized needed segments

<table>
<thead>
<tr>
<th>Student</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>91</td>
</tr>
<tr>
<td>008</td>
<td>92</td>
</tr>
<tr>
<td>002</td>
<td>93</td>
</tr>
<tr>
<td>007</td>
<td>98</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>91</th>
<th>92</th>
<th>93</th>
<th>98</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>008</td>
<td>002</td>
<td>007</td>
</tr>
</tbody>
</table>

**Match on Location Index**

De-Serialize 16+4 Bytes to find the student index(2) scored 93 marks.

Further optimize using binary search on Byte Arrays
From multiple tabular records - To sorted tree structure

<table>
<thead>
<tr>
<th>Tree Id</th>
<th>Cell2</th>
<th>Cell3</th>
<th>Cell4</th>
<th>Cell5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>-8</td>
<td>1001</td>
<td>24.01</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>-8</td>
<td>1003</td>
<td>26.44</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>-8</td>
<td>1002</td>
<td>29.30</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>-7</td>
<td>1001</td>
<td>20.81</td>
</tr>
</tbody>
</table>
where each root level node is serialized to form a HBase Cell.
Inverted Index – Enter By Value and Not Key.

<table>
<thead>
<tr>
<th>Value Range 1</th>
<th>Value Range 2</th>
<th>Value Range 3</th>
<th>Value Range 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
... with parallel processing of Bytes Blocks for each region servers.

Find Cell 5 = 47.10

Partition/Range 1,2,3

Thread 1
Region 1
Machine 1

Each Partition
In a Thread

Thread 2
Region 2
Machine 1

Each Partition
In a Thread

Thread 3
Region 3
Machine 2

Each Partition
In a Thread
Processing moved near to DATA: Filter and Coprocessors

Client Code

Co-Processor
Region Server
Hbase Filter
Table Read

Co-Processor
Region Server
Hbase Filter
Table Read

Like Database Stored Procedure
Useful for SUM, AVG, MIN/MAX

Like Database Stored Procedure
Useful for Filter/Modify a cell
Network Time

Strategy Applied: Bytes Block Caching

Object Cache = 7 bytes + 56 bytes (pointer)

Bytes Cache = 7 bytes + 8 bytes (pointer)
To process Big Data in small time, it is needed to balance Network vs CPU vs I/O vs Memory while leveraging multiple machines.
Disk I/O

Memory

CPU

Network

Block Caching
Keeping program log in memory
And flush on Exception/read finish

Concurrent GC
Object Reuse

IPC Caching
Sending on Chunks

Compression
Data Partitioning

Snappy/ LZO
Compressed Data
And It’s Configuration...

• Network
  • Increased IPC Cache Limit (hbase.client.scanner.caching)

• CPU
  • JVM agressive heap
    (“-server -XX:+UseParallelGC -XX:ParallelGCThreads=4
     XX:+AggressiveHeap ”)

• I/O
  • LZO index compression (“Inbuilt oberhumer LZO” or “Intel IPP
    native LZO”)

• Memory
  • HBase block caching (hfile.block.cache.size) and overall memory
    allocation for data-node and region-server.
and parallelized to multiple machines...

- Htable.batch (Sending/Receiving data from Region Servers in chunk)
- Parallel Htable (Multi threaded Scans)
- Co-Processors, Filters

Allocating appropriate resources
dfs.datanode.max.xcievers,
hbase.regionserver.handler.count and
dfs.datanode.handler.count
THANK YOU