10 Million Smart Meter Data with Apache HBase

5/31/2017
OSS Solution Center
Hitachi, Ltd.
Masahiro Ito

Open Source Summit Japan 2017
Who am I?

• Masahiro Ito
  ➢ Software Engineer at Hitachi, Ltd.
  ➢ Focus on development of Big Data Solution with Apache Hadoop and its related OSS.
  ➢ Mail: masahiro.ito.ph@hitachi.com

➢ Book and Web-articles (in Japanese)
  • Apache Spark ビッグデータ性能検証 (Think IT Books)
  • ユースケースで徹底検証！HBaseでIoT時代のビッグデータ管理機能を試す
    – https://thinkit.co.jp/series/6465
1. Motivation
2. What is NoSQL?
3. Overview of HBase architecture
4. Performance evaluation with 10 million smart meter data
5. Summary
1. Motivation
Motivation

- The internet of things (IoT) and NoSQL
  - Various sensor devices generate large amounts of data.
  - NoSQL has higher performance and scalability than RDB.
  - HBase is one of NoSQL.

- Is HBase suitable for sensor data management?
  - HBase seems to be suitable for managing time series data such as sensor data.
  - I will introduce the result of performance evaluation of HBase with 10 million smart meter data.
2. What is NoSQL?
NoSQL (Not only SQL)

- NoSQL refers to databases other than RDB (Relational DataBase).

- Motivations of NoSQL include:
  - More flexible data model (not tabular relations).
  - High performance and large disk capacity.
    - With simpler "horizontal" scaling to clusters of machines.
  - etc.

- NoSQL databases are increasingly used in big data and real-time web applications.
Features of RDB

Relational model

- Table format (tabular relations)
- SQL interface
  - Supports complex queries

ACID Transaction

- Atomicity
- Consistency
- Isolation
- Durability
3 Vs of Big Data: Challenges of RDB for big data

**Volume**: Need to manage large amount of distributed data.

- Transaction control over distributed data is difficult.

**Velocity**: Need to process large number of requests in real time.

- Exclusive control of transaction is overhead.

**Variety**: Need to manage data of various structures.

- It is incompatible with the predefined table.

- Log
- Pictures
- SNS
- Sensor data
3 Vs of Big Data: Challenges of RDB for big data

**Volume**
Need to manage large amount of distributed data.

**Velocity**
Need to process large number of requests in real time.

**Variety**
Need to manage data of various structures.

**RDB**
Transaction control over distributed data is difficult.

**Exclusive control of transaction is overhead.**

**It is incompatible with the predefined table.**

**NoSQL**
Limiting the scope of transaction control makes it possible to improve performance and disk capacity with scale out.
3 Vs of Big Data: Challenges of RDB for big data

**Volume**
Need to manage large amount of distributed data.

**Velocity**
Need to process large number of requests in real time.

**Variety**
Need to manage data of various structures.

**RDB**
- Transaction control over distributed data is difficult.
- Exclusive control of transaction is overhead.
- It is incompatible with the predefined table.

**NoSQL**
- Limiting the scope of transaction control makes it possible to improve performance and disk capacity with scale out.
- Adopted flexible data structure other than table.

- GB
- PB
- Log
- Pictures
- SNS
- Sensor data
There are lots of NoSQL in the world (many others)

- Riak
- Redis
- MongoDB
- Couchbase
- Cassandra
- Neo4j
- TITAN
- HBase
NoSQL is generally classified by data model

Key value store
- Riak
- Redis

Wide column store
- Cassandra
- HBase

Document store
- MongoDB
- Couchbase

Graph database
- Neo4j
- TITAN
NoSQL is generally classified by data model

**Key value store**

Low latency access with simple data structure.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Wide column store**

Each row has different number of columns.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Document store**

Store structure data such as JSON.

<table>
<thead>
<tr>
<th>Key</th>
<th>Document</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>ID: 001</td>
</tr>
<tr>
<td></td>
<td>User: {</td>
</tr>
</tbody>
</table>
|     |     Name: "Engineer"
|     | }       |

**Graph database**

Represent relationship between data as graph structure.

Node —> Node —> Node —> Node
3. Overview of HBase architecture
HBase overview

• HBase is distributed, scalable, versioned, and non-relational (wide column type) big data store.

• A Google Bigtable clone.
  ➢ Implemented in Java based on the paper of Bigtable.

• One of the OSS in Apache Hadoop eco-system.
Relationship between HBase and Hadoop (HDFS)

- HBase build on HDFS (Hadoop Distributed File System).
Relationship between HBase and Hadoop (HDFS)

- HBase build on HDFS (Hadoop Distributed File System).

HADOOP

- MapReduce
  - [Parallel processing framework]

- YARN (Yet Another Resource Negotiator)
  - [Cluster resource management framework]

HBASE

- Distributed database

HDFS

- Hadoop Distributed File System
  - [Distributed File System]

Commodity servers

- HDFS can read/write large files with high throughput.
- However, it is not suitable for read/write small data.
Relationship between HBase and Hadoop (HDFS)

- HBase build on HDFS (Hadoop Distributed File System).

Hadoop
- MapReduce [Parallel processing framework]
- YARN (Yet Another Resource Negotiator) [Cluster resource management framework]

HBase
[ Distributed database ]

HDFS (Hadoop Distributed File System)
[ Distributed File System ]

Commodity servers

- HDFS can read/write large files with high throughput.
- However, it is not suitable for read/write small data.

HBase can read/write many small data with low latency.
⇒ HBase is a complement to HDFS.
HBase architecture: Master/Slave model

- HBase processes the request and HDFS saves the data.
Data model: Conceptual view

This table looks like a RDB’s table.

Rows in a table are sorted by RowKey.

Each row can have a different number of columns.

Value is stored in Cell. The past values are stored together with Timestamp.

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>20170310</td>
<td>CCC</td>
</tr>
<tr>
<td>20170124</td>
<td>BBB</td>
</tr>
<tr>
<td>20160930</td>
<td>AAA</td>
</tr>
</tbody>
</table>
Data model: Physical view

- Data is stored as key value.
  - The keys are sorted in the order of **RowKey, Column** (ColumnFamily:qualifier), **Timestamp**.
  - It is a “multi-dimensional sorted map”.
    - SortedMap<**RowKey**, SortedMap<**Column**, SortedMap<**Timestamp**, **Value**>>>
Operations and functions

- **Operations**
  - Put, Get, Scan, Delete, etc.

- **Functions**
  - **Index**
    - Only be set to RowKey and Column.
  - **Transaction**
    - Only within one Row.

<table>
<thead>
<tr>
<th>RowKey</th>
<th>Column</th>
<th>Timestamp</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 1</td>
<td>fam1:Col1</td>
<td>20170310</td>
<td>Delete</td>
<td>-</td>
</tr>
<tr>
<td>Row 1</td>
<td>fam1:Col1</td>
<td>20170310</td>
<td>Put</td>
<td>Val_01</td>
</tr>
<tr>
<td>Row 2</td>
<td>fam2:Col3</td>
<td>20170215</td>
<td>Put</td>
<td>Val_03</td>
</tr>
<tr>
<td>Row 2</td>
<td>fam2:Col4</td>
<td>20170309</td>
<td>Put</td>
<td>Val_04</td>
</tr>
<tr>
<td>Row 3</td>
<td>fam1:Col1</td>
<td>20170310</td>
<td>Put</td>
<td>Val_05</td>
</tr>
<tr>
<td>Row 3</td>
<td>fam1:Col2</td>
<td>20160104</td>
<td>Put</td>
<td>Val_06</td>
</tr>
<tr>
<td>Row 4</td>
<td>fam2:Col3</td>
<td>20170221</td>
<td>Delete</td>
<td>-</td>
</tr>
<tr>
<td>Row 4</td>
<td>fam2:Col3</td>
<td>20170204</td>
<td>Put</td>
<td>Val_07</td>
</tr>
</tbody>
</table>
Distributed data management

- How is a table physically divided?

<table>
<thead>
<tr>
<th>RowKey</th>
<th>Column</th>
<th>⋮</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 1</td>
<td>fam1:Col1</td>
<td>⋮</td>
<td>Val_01</td>
</tr>
<tr>
<td>Row 1</td>
<td>fam1:Col2</td>
<td>⋮</td>
<td>Val_02</td>
</tr>
<tr>
<td>Row 1</td>
<td>fam1:Col3</td>
<td>⋮</td>
<td>Val_03</td>
</tr>
<tr>
<td>Row 1</td>
<td>fam2:Col1</td>
<td>⋮</td>
<td>Val_04</td>
</tr>
<tr>
<td>Row 2</td>
<td>fam1:Col1</td>
<td>⋮</td>
<td>Val_05</td>
</tr>
<tr>
<td>Row 2</td>
<td>fam2:Col2</td>
<td>⋮</td>
<td>Val_06</td>
</tr>
<tr>
<td>Row 2</td>
<td>fam2:Col3</td>
<td>⋮</td>
<td>Val_07</td>
</tr>
<tr>
<td>Row 3</td>
<td>fam1:Col1</td>
<td>⋮</td>
<td>Val_08</td>
</tr>
<tr>
<td>Row 3</td>
<td>fam2:Col1</td>
<td>⋮</td>
<td>Val_09</td>
</tr>
<tr>
<td>Row 4</td>
<td>fam1:Col2</td>
<td>⋮</td>
<td>Val_10</td>
</tr>
<tr>
<td>Row 4</td>
<td>fam1:Col4</td>
<td>⋮</td>
<td>Val_11</td>
</tr>
<tr>
<td>Row 4</td>
<td>fam2:Col3</td>
<td>⋮</td>
<td>Val_12</td>
</tr>
<tr>
<td>Row 4</td>
<td>fam2:Col5</td>
<td>⋮</td>
<td>Val_13</td>
</tr>
</tbody>
</table>
Table is divided into **Region** with the range of RowKey

<table>
<thead>
<tr>
<th>Region (Row1-2)</th>
<th>RowKey</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Row 1</td>
<td>fam1:Col1</td>
<td>Val_01</td>
</tr>
<tr>
<td></td>
<td>Row 1</td>
<td>fam1:Col2</td>
<td>Val_02</td>
</tr>
<tr>
<td></td>
<td>Row 1</td>
<td>fam1:Col3</td>
<td>Val_03</td>
</tr>
<tr>
<td></td>
<td>Row 1</td>
<td>fam2:Col1</td>
<td>Val_04</td>
</tr>
<tr>
<td></td>
<td>Row 2</td>
<td>fam1:Col1</td>
<td>Val_05</td>
</tr>
<tr>
<td></td>
<td>Row 2</td>
<td>fam2:Col2</td>
<td>Val_06</td>
</tr>
<tr>
<td></td>
<td>Row 2</td>
<td>fam2:Col3</td>
<td>Val_07</td>
</tr>
<tr>
<td></td>
<td>Row 1</td>
<td>fam2:Col5</td>
<td>Val_13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Region (Row3-4)</th>
<th>RowKey</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Row 3</td>
<td>fam1:Col1</td>
<td>Val_08</td>
</tr>
<tr>
<td></td>
<td>Row 3</td>
<td>fam2:Col1</td>
<td>Val_09</td>
</tr>
<tr>
<td></td>
<td>Row 4</td>
<td>fam1:Col2</td>
<td>Val_10</td>
</tr>
<tr>
<td></td>
<td>Row 4</td>
<td>fam1:Col4</td>
<td>Val_11</td>
</tr>
<tr>
<td></td>
<td>Row 4</td>
<td>fam2:Col3</td>
<td>Val_12</td>
</tr>
<tr>
<td></td>
<td>Row 4</td>
<td>fam2:Col5</td>
<td>Val_13</td>
</tr>
</tbody>
</table>
Data is distributed on the cluster via Regions

- **Automatic sharding**
  - Regions are automatically split and re-distributed as data grows.

- **Simple horizontal scaling**
  - Adding slave nodes improves performance and expands disk capacity.

Region holds data across HBase (as cache in memory) and HDFS (as file in disk).
Summary of HBase architecture

• Simple horizontal scaling
  - Adding slave nodes improves performance and expands disk capacity

• Data is stored as sorted key value
  - Like multi-dimensional sorted map.
  - By designing RowKey carefully, data that are accessed together are physically co-located.

• Limited the index and transaction
  - Index: Only be set to RowKey and Column.
  - Transaction: Only within one Row.
4. Performance evaluation with 10 million smart meter data
i. Evaluation scenario
Smart meter data management

• We assumed the Meter Data Management System for 10 million smart meters.
  ➢ Smart meters collect consumption of electric energy from customers.
    • Send the collected data to the Meter Data Management System every 30 minutes.
  ➢ The collected data is used for power charge calculation and demand forecast analysis, etc.
System overview

- Write 10 million records every 30 minutes in HBase.
- Read to analyze records stored in HBase.
Contents of performance evaluation

1. **Write performance**
   - Measure write time and throughput of 10 million records.

2. **Data compression performance**
   - Measure data compression ratio and compression / decompression time.

3. **Read performance**
   - Measure read time and throughput in two kinds of analysis use cases.
Evaluation environment

**Software version**
CDH5.9 (HBase1.2.0 + Hadoop2.6.0)

<table>
<thead>
<tr>
<th></th>
<th>Client Node</th>
<th>Master Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Core</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>Memory</td>
<td>12 GB</td>
<td>16 GB</td>
</tr>
<tr>
<td># of disk</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Capacity of disk</td>
<td>80 GB</td>
<td>160 GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Per slave node</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Core</td>
<td>32</td>
<td>128</td>
</tr>
<tr>
<td>Memory</td>
<td>128 GB</td>
<td>512 GB</td>
</tr>
<tr>
<td># of disk</td>
<td>6</td>
<td>24</td>
</tr>
<tr>
<td>Capacity of disk</td>
<td>900 GB</td>
<td>-</td>
</tr>
<tr>
<td>Total capacity of disks</td>
<td>5.4 TB (5,400 GB)</td>
<td>21.6 TB (21,600 GB)</td>
</tr>
</tbody>
</table>
Table design

- Divided the table into 400 Regions in advance.
  - 100 Regions per RegionServer
  - Region split key: 0001, 0002, …, 0399

To distribute data among Regions, add 0000 to 0399 (meter ID modulo 400) to the head of RowKey. This technique is called “Salt”.

<table>
<thead>
<tr>
<th>RowKey (Salt&lt;Meter ID&gt;Date-Time)</th>
<th>Column (ColumnFamily:qualifier)</th>
<th>Timestamp</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000-00000000001-20170310-1100</td>
<td>CF:</td>
<td></td>
<td>Put</td>
<td>3.241</td>
</tr>
<tr>
<td>0000-00000000001-20170310-1030</td>
<td>CF:</td>
<td></td>
<td>Put</td>
<td>0.863</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td></td>
<td>Put</td>
<td>…</td>
</tr>
<tr>
<td>0000-00000000001-20160910-1100</td>
<td>CF:</td>
<td></td>
<td>Put</td>
<td>0.044</td>
</tr>
<tr>
<td>0001-00000000002-20170310-1100</td>
<td>CF:</td>
<td></td>
<td>Put</td>
<td>2.390</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td></td>
<td>Put</td>
<td>1.432</td>
</tr>
</tbody>
</table>
ii. Evaluation of write performance
Evaluation of write performance

- Generate 10 million records with HBase clients.
- Send put request using multi clients.
- Measured the write time and throughput of 10 million records.

Tuning parameters
① # of clients
② # of send records per request
③ # of Regions
Write performance

- Write time and throughput of 10 million records.

- Stored multiple records by one request:
  - Records per request: 1 to 10,000  \(\Rightarrow\)  Throughput: 526 to 46,729 records/sec (89x)

- Increased the number of clients:
  - # of Clients: 1 to 64  \(\Rightarrow\)  Throughput: 46,729 to 327,869 records/sec (7x)
iii. Evaluation of Compression performance
Compressor and data block encoding

• HBase tends to increase data size for the following reasons.
  ➢ The number of records increases because data is stored in key value format.
  ➢ Each record length is long because a key is composed of many fields.

• Compress data with a combination of compressor and data block encoding.

  - Measured the file size, write time, and read time of 10 million records.
Data block encoding performance with 10 million records

### HFile size

<table>
<thead>
<tr>
<th>Encoding</th>
<th>HFile size</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>586 MB</td>
</tr>
<tr>
<td>PREFIX</td>
<td>425 MB</td>
</tr>
<tr>
<td>PREFIX_TREE</td>
<td>404 MB</td>
</tr>
<tr>
<td>FAST_DIFF</td>
<td>311 MB</td>
</tr>
<tr>
<td>DIFF</td>
<td>311 MB</td>
</tr>
</tbody>
</table>

- **Reduced to 53% by DIFF encoding**

### Write time

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Write time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>31 sec</td>
</tr>
<tr>
<td>PREFIX</td>
<td>47 sec</td>
</tr>
<tr>
<td>PREFIX_TREE</td>
<td>50 sec</td>
</tr>
<tr>
<td>FAST_DIFF</td>
<td>46 sec</td>
</tr>
<tr>
<td>DIFF</td>
<td>43 sec</td>
</tr>
</tbody>
</table>

- **Increased 48% by DIFF encoding**

### Read time

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Read time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>45 sec</td>
</tr>
<tr>
<td>PREFIX</td>
<td>50 sec</td>
</tr>
<tr>
<td>PREFIX_TREE</td>
<td>46 sec</td>
</tr>
<tr>
<td>FAST_DIFF</td>
<td>43 sec</td>
</tr>
<tr>
<td>DIFF</td>
<td>43 sec</td>
</tr>
</tbody>
</table>

- **Reduced 4% by DIFF encoding**
Compressor performance with 10 million records

**HFile size**
- NONE: 586 MB
- LZ4: 175 MB
- SNAPPY: 162 MB
- GZ: 126 MB

**Write time**
- NONE: 51 sec
- LZ4: 63 sec
- SNAPPY: 51 sec
- GZ: 45 sec

**Read time**
- NONE: 45 sec
- LZ4: 51 sec
- SNAPPY: 46 sec
- GZ: 52 sec

*Reduced to 22% by GZip algorithm*

*Increased 68% by GZip algorithm*

*Increased 15% by GZip algorithm*
Compressor and data block encoding performance with 10 million records

<table>
<thead>
<tr>
<th>Compressor + Encoding</th>
<th>HFile size</th>
<th>Write time</th>
<th>Read time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE + NONE</td>
<td>586 MB</td>
<td>31 sec</td>
<td>44 sec</td>
</tr>
<tr>
<td>NONE + PREFIX</td>
<td>425 MB</td>
<td>41 sec</td>
<td>45 sec</td>
</tr>
<tr>
<td>NONE + PREFIX_TREE</td>
<td>404 MB</td>
<td>46 sec</td>
<td>46 sec</td>
</tr>
<tr>
<td>NONE + FAST_DIFF</td>
<td>311 MB</td>
<td>45 sec</td>
<td>47 sec</td>
</tr>
<tr>
<td>NONE + DIFF</td>
<td>311 MB</td>
<td>52 sec</td>
<td>49 sec</td>
</tr>
<tr>
<td>LZ4 + PREFIX_TREE</td>
<td>189 MB</td>
<td>41 sec</td>
<td>50 sec</td>
</tr>
<tr>
<td>SNAPPY + PREFIX_TREE</td>
<td>188 MB</td>
<td>63 sec</td>
<td>51 sec</td>
</tr>
<tr>
<td>LZ4 + NONE</td>
<td>175 MB</td>
<td>41 sec</td>
<td>51 sec</td>
</tr>
<tr>
<td>LZ4 + PREFIX</td>
<td>163 MB</td>
<td>49 sec</td>
<td>47 sec</td>
</tr>
<tr>
<td>SNAPPY + NONE</td>
<td>162 MB</td>
<td>42 sec</td>
<td>47 sec</td>
</tr>
<tr>
<td>LZ4 + FAST_DIFF</td>
<td>154 MB</td>
<td>41 sec</td>
<td>46 sec</td>
</tr>
<tr>
<td>SNAPPY + PREFIX</td>
<td>151 MB</td>
<td>49 sec</td>
<td>52 sec</td>
</tr>
<tr>
<td>SNAPPY + FAST_DIFF</td>
<td>149 MB</td>
<td>41 sec</td>
<td>46 sec</td>
</tr>
<tr>
<td>GZ + PREFIX_TREE</td>
<td>146 MB</td>
<td>41 sec</td>
<td>47 sec</td>
</tr>
<tr>
<td>LZ4 + DIFF</td>
<td>145 MB</td>
<td>42 sec</td>
<td>52 sec</td>
</tr>
<tr>
<td>SNAPPY + DIFF</td>
<td>138 MB</td>
<td>46 sec</td>
<td>53 sec</td>
</tr>
<tr>
<td>GZ + NONE</td>
<td>126 MB</td>
<td>46 sec</td>
<td>51 sec</td>
</tr>
<tr>
<td>GZ + PREFIX</td>
<td>120 MB</td>
<td>45 sec</td>
<td>50 sec</td>
</tr>
<tr>
<td>GZ + FAST_DIFF</td>
<td>118 MB</td>
<td>41 sec</td>
<td>50 sec</td>
</tr>
<tr>
<td>GZ + DIFF</td>
<td>110 MB</td>
<td>51 sec</td>
<td>52 sec</td>
</tr>
</tbody>
</table>

- **HFile size**: Reduced to 19% by GZip + FAST_DIFF
- **Write time**: Increased 33% by GZip + FAST_DIFF
- **Read time**: Increased 14% by GZip + FAST_DIFF
iv. Evaluation of read performance
Evaluation of read performance

• Measure the read time and throughput in two kinds of analysis use cases.
  - **Use case A**: Scan time series data of a few meters.
    • To display the transition of power consumption per meter in the line chart.
  - **Use case B**: Get the latest data of many meters.
    • To calculate the average and total value of the latest power consumption.

• Evaluation settings
  • Dataset: 10 million meter * 180 days records (Compressed by FAST_DIFF + GZ)
  • Disabled caches and make sure to read data from disk.
Use case A: Scan time series data of a few meters

- Scan meter data for **1-180 days** of **1-100 meters**.
  - Scan time series data of one meter by one scan.

Since read multiple data with one Scan, the throughput improves as the term was longer.
  - Term: 1 to 180 days  ⇒  Throughput: 247 to 51,128 records/sec (207x)
Use case A: Scan time series data of a few meters (with multi thread)

- Scan meter data for **180 days of 1-100 meters**.
  - Scan request was executed in multi thread. (Maximum 1 Scan 1 thread)

Throughput was improved by running Scan requests in parallel.

- # of threads: 1 to 100 ⇒ Throughput: 51,128 to 356,387 records/sec (7x)
Use case B: Get the latest data of many meters (with multi thread)

- Get the latest time (30 minutes) data of **10,000 to 10 million meters**.
  - Scan request can not be applied to these data.
  - Requests are executed in multi thread.
  - Batch execution of multiple “Get” request by one “batch” request.

Throughput was improved by running Get requests in parallel.

- # of threads: 1 to 100 ⇒ Throughput: 1,002 to 7,574 records/sec (7.5x)
Comparison of Scan request with Get request

Use case A:
Scan 180 days time series data of 100 meters with 100 thread.
= Throughput 356,387 records/second

Use case B:
Get the latest 30 min. data of 10,000,000 meters with 100 thread.
= Throughput 7,574 records/second

<table>
<thead>
<tr>
<th>RowKey</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Salt&gt;-&lt;Meter ID&gt;-&lt;Date&gt;-&lt;Time&gt;</td>
<td></td>
</tr>
<tr>
<td>0000-0000000001-20170310-1100</td>
<td>3.241</td>
</tr>
<tr>
<td>0000-0000000001-20170310-1030</td>
<td>0.863</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0000-0000000001-20160910-1100</td>
<td>0.044</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0200-0000000201-20170310-1100</td>
<td>10.390</td>
</tr>
<tr>
<td>0200-0000000201-20170310-1030</td>
<td>14.325</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0200-0000000201-20160910-1100</td>
<td>9.32</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

• Scan request’s throughput was about 47x higher than the Get request.
• Careful RowKey design is important.
  ➢ Place the data that are accessed together physically co-located.
5. Summary
Summary

• HBase is suitable for storing time series data generated by sensor devices.

• Lessons from performance evaluation:
  ➢ Careful RowKey design to be able to scan data is important.
    • Scan request’s throughput was more than 47x that of Get request.
  
  ➢ HBase has high multi-client / multi-thread concurrency.
    • Throughput of the Put / Scan / Get request with multi-client / multi-thread is 7x faster than single-client / single-thread.

  ➢ Choosing the appropriate compression setting.
    • The storage size of time series data could be reduced to 19%.
Trademarks

- Apache HBase and Apache Hadoop are either a registered trademark or a trademark of Apache Software Foundation in the United States and/or other countries.
- Other company and product names mentioned in this document may be the trademarks of their respective owners.