SnappyData:
Apache Spark Meets
Embedded In-Memory Database

Masaki Yamakawa
UL Systems, Inc.
About me

Masaki Yamakawa
UL Systems, Inc.
Managing Consultant

{ "Sector": "Financial",
  "Skills": ["Distributed system",
             "In-memory computing"],
  "Hobbies": "Marathon running"
}
Agenda

1. Current Issues of Real-Time Analytics Solutions
2. SnappyData Features
3. Our SnappyData Case Study
Current Issues of Real-Time Analytics Solutions

PART 1
Are you satisfied with real-time analytics solutions?

- Complex
- Slow
- Bad performance
- Loading data to memory required
- Difficulty with updates
What are common demands for data processing platforms?

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Analytics</th>
<th>Streaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional data processing</td>
<td>RDBMS</td>
<td>DWH</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="RDBMS Icon" /></td>
<td><img src="image" alt="DWH Icon" /></td>
</tr>
<tr>
<td>Bigdata processing</td>
<td>NoSQL</td>
<td>SQL on Hadoop</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="NoSQL Icons" /></td>
<td><img src="image" alt="SQL on Hadoop Icons" /></td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="HBase" /></td>
<td><img src="image" alt="Hive" /></td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="MongoDB" /></td>
<td><img src="image" alt="Drill" /></td>
</tr>
</tbody>
</table>
Tends to become complex system when integrates with multiple products

- **Enterprise systems**
  - Store enterprise data
  - Store and process big data

- **Web/B2C services etc.**
  - Store and process big data

- **IoT / sensor / real-time data etc.**
  - Process real-time data

- **RDBMS**
  - ETL processing

- **DWH**
  - ETL processing
  - Data visualization and analysis

- **BI/Analytic systems AP**
  - Data visualization and analysis

- **Stream data processing**
  - Notification, Alert

- **Real-time AP**
Tends to become complex system when integrates with multiple products

Increased TCO

It takes time to analyze

Inefficiency

Difficult to maintain data consistency

Increased TCO

It takes time to analyze

Inefficiency

Difficult to maintain data consistency
Although it became quite simple after Spark released…?
SnappyData can build simpler real-time analytics solutions!

- **Enterprise systems**
  - Store enterprise data

- **Web/B2C services etc.**
  - Store and process big data

- **IoT / sensor / real-time data etc.**
  - Process real-time data

---

**SnappyData**

- **Data visualization and analysis**
  - BI/Analytic
  - AP

- **Notification, Alert**
  - Real-time
  - AP
SnappyData, the Spark Database.

SnappyData is a high performance in-memory data platform for mixed workload applications. Built on Apache Spark, SnappyData provides a unified programming model for streaming, transactions, machine learning and SQL Analytics in a single cluster. SnappyData unifies real time data sources with external data sources that have a Spark connector. SnappyData is 100% compatible with Apache Spark and is Open Sourced with an Apache V2 license.
Apache Spark + Distributed In-memory DB + Own features

SNAPPYDATA

Batch processing
Analytics
Stream processing

Distributed computing framework
Spark

Row database
Transaction

Distributed In-memory database
GemFire XD

Columnar database
Synapsis Data Engine

SnappyData's own features
What is SnappyData's core component?

- Seamless integration of Spark and in-memory database components

![Diagram showing SnappyData components]

- Spark
- GemFire XD
- Micro-batch Streaming
- Spark Core
- Spark SQL Catalyst
- Continuous Query
- Transaction
- OLTP Query
- OLAP Query
- Synopsis Data Engine
- In-Memory Database
- Stream Table
- Row Table
- Index
- Column Table
- Sample/TopK Table
- P2P Cluster Management
- Replication/Partition
- HDFS
- Distributed file system
- SnappyData's additional features
Key to Spark program’s accelerations

1. In-memory database
2. In-memory data format
3. Unified cluster
4. Optimized Spark SQL
Key#1: Data exists in in-memory database

In case of Spark:

- Spark program
- Spark
- HDFS

In case of SnappyData:

- Spark program
- Spark
- Distributed in-memory database

In-memory and on-disk storage options are illustrated.
Key#1: Data access code example

**In case of Spark**

```scala
// load data from HDFS
val df = spark.sqlContext.read.format("com.databricks.spark.csv").
  option("header", "true").load("hdfs://...")
df.createOrReplaceTempView("SparkTable")

// create new DataFrame using SparkSQL
val filteredDf = spark.sql("SELECT * FROM SparkTable WHERE ...")
val newDf = filteredDf.

// save processing results
newDf.write.format("com.databricks.spark.csv").
  option("header", "false").save("hdfs://...")
```

**In case of SnappyData**

```scala
// create SnappySession from SparkContext
val snappy = new org.apache.spark.sql.
  SnappySession(spark.sparkContext)

// create new DataFrame using SparkSQL
val filteredDf = snappy.sql("SELECT * FROM SnappyTable WHERE ...")
val newDf = filteredDf.

// save processing results
newDf.write.format("com.databricks.spark.csv").
  option("header", "false").save("hdfs://...")
```

No need to load data
Key#2: SnappyData same data format as Spark’s

In case of Spark
- Spark
- DataFrame
- O/R mapping
- HDFS/data storage
- CSV file
- reading/writing data
- serialization/deserialization

In case of SnappyData
- Spark
- DataFrame
- No serialization/deserialization, No O/R mapping
- GemFire XD: In-memory database
- reading/writing data
Key#3: Spark and GemFire XD cluster can be integrated

Unified cluster mode

Spark with GemFire XD cluster

Spark Context

SnappyData DataServer

Spark Executor

DataFrame

DataFrame

database

JVM

SnappyData DataServer

Spark Executor

DataFrame

DataFrame

database

JVM

SnappyData DataServer

Spark Executor

DataFrame

DataFrame

database

JVM

SnappyData DataServer

Spark Executor

DataFrame

DataFrame

database

JVM

SnappyData Locator
Key#3: Another cluster mode (for your reference)

Split cluster mode

Spark cluster
- Spark Executor
  - DataFrame
  - DataFrame
  - JVM
- GemFire XD cluster
  - SnappyData DataServer
  - JVM
  - Database

Spark Context
- SnappyData Leader
- Spark Driver

SnappyData Locator
- JVM
Key#4: SparkSQL Acceleration

In case of Spark

```
SELECT
    A.CardNumber,
    SUM(A.TxAmount)
FROM
    CreditCardTx1 A,
    CreditCardComm B
WHERE
    A.CardNumber = B.CardNumber AND
    A.TxAmount + B.Comm < 1000
GROUP BY
    A.CardNumber
ORDER BY
    A.CardNumber
```

In case of SnappyData

- Unique DAG is generated, less shuffle and faster
- SnappyHashJoin
- SnappyHashAggregate

Accelerate the processing by modifying some workload of SparkSQL.
Our SnappyData Case Study: How to use SnappyData

PART 3
Example of use: Production plan simulation system

Production results stream

Machine sensor stream

BOM stream

Production results table

Machine sensor table

BOM table

Simulation parameters table

In-memory database

Real-time notification

APP BI Tool
Architecture with SnappyData

- Use SnappyData to realize all data processings such as stream processings, transactions, analytics
- The key is that it includes in-memory database and can be processed by SQL
A) Stream Data Processing

- The stream data is inserted into the table
- Stream data processing can be executed by SQL

Difference from plain Spark

- Messaging Middleware
- SQL

In-memory database

APP
SnappyData implements stream data processing using SQL.

### Stream table

<table>
<thead>
<tr>
<th>SensorId</th>
<th>VIN</th>
<th>MachineNo</th>
<th>Point</th>
<th>Value</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11AA</td>
<td>111</td>
<td>1</td>
<td>28.076</td>
<td>2017/11/05 10:10:01</td>
</tr>
<tr>
<td>2</td>
<td>22BB</td>
<td>222</td>
<td>37</td>
<td>60.069</td>
<td>2017/11/05 10:10:20</td>
</tr>
<tr>
<td>3</td>
<td>11AA</td>
<td>111</td>
<td>2</td>
<td>37.528</td>
<td>2017/11/05 10:10:21</td>
</tr>
<tr>
<td>4</td>
<td>33CC</td>
<td>333</td>
<td>25</td>
<td>1.740</td>
<td>2017/11/05 10:11:05</td>
</tr>
<tr>
<td>5</td>
<td>11AA</td>
<td>111</td>
<td>3</td>
<td>88.654</td>
<td>2017/11/05 10:11:15</td>
</tr>
<tr>
<td>6</td>
<td>11AA</td>
<td>111</td>
<td>4</td>
<td>394.39</td>
<td>2017/11/05 10:11:16</td>
</tr>
</tbody>
</table>

### Process (Continuous Query)

```sql
SELECT * FROM MachineSensorStream
WINDOW (DURATION 10 SECONDS, SLIDE 2 SECONDS)
WHERE Point=1;
```
CREATE STREAM TABLE MachineSensorStream
(SensorId long, VIN string, MachineNo int, Point long, Value double, Timestamp timestamp)
USING KAFKA_STREAM
OPTIONS
(storagelevel 'MEMORY_AND_DISK_SER_2', rowConverter 'uls.snappy.KafkaToRowsConverter',
kafkaParams 'zookeeper.connect->localhost:2181,xx', topics 'MachineSensorStream');
Implements StreamToRowsConverter and converts to table format

class KafkaToRowsConverter extends StreamToRowsConverter with Serializable {

  override def toRows(message: Any): Seq[Row] = {

    Seq(Row.fromSeq(Seq(
      sensor.getSensorId,
      sensor.getVin,
      sensor.getMachineNo,
      sensor.getPoint,
      sensor.getValue,
      sensor.getTimestamp)))
  }
}

Data for one row of stream table
Stream data processing using SQL

SELECT *
FROM MachineSensorStream
WINDOW (DURATION 10 SECONDS, SLIDE 2 SECONDS)
WHERE Point=1;

Point acquires "1" data in 2 secs sliding window
In Continuous Query, only data included in WINDOW is acquired.

<table>
<thead>
<tr>
<th>SensorId</th>
<th>VIN</th>
<th>Machine No</th>
<th>Point</th>
<th>Value</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11AA</td>
<td>111</td>
<td>1</td>
<td>28.0760</td>
<td>2017/11/05 10:10:01</td>
</tr>
<tr>
<td>2</td>
<td>22BB</td>
<td>222</td>
<td>37</td>
<td>60.069</td>
<td>2017/11/05 10:10:20</td>
</tr>
<tr>
<td>3</td>
<td>11AA</td>
<td>111</td>
<td>2</td>
<td>37.528</td>
<td>2017/11/05 10:10:21</td>
</tr>
<tr>
<td>4</td>
<td>33CC</td>
<td>333</td>
<td>25</td>
<td>1.740</td>
<td>2017/11/05 10:11:05</td>
</tr>
<tr>
<td>5</td>
<td>11AA</td>
<td>111</td>
<td>3</td>
<td>88.654</td>
<td>2017/11/05 10:11:15</td>
</tr>
<tr>
<td>6</td>
<td>11AA</td>
<td>111</td>
<td>4</td>
<td>394.390</td>
<td>2017/11/05 10:11:16</td>
</tr>
</tbody>
</table>

After 2 seconds, the data updates again.
Stream data processing code example

// create SnappyStreamingContext from SparkStreamingContext
val snappy = new SnappyStreamingContext(sc, 10)

// register Continuous Query
val machineSensorStream : SchemaDStream = snappy.registerCQ(s""
SELECT
  SensorId, VIN, MachineNo, Point, Value, Timestamp
FROM MachineSensorStream
  WINDOW
  (DURATION 10 SECONDS, SLIDE 2 SECONDS)
WHERE
  Point=1
"")

// process stream data
machineSensorStream.foreachDataFrame(df => {
  ...
  df.write.insertInto("MachineSensorHistory")
  ...
})
B) Transaction

- Insert, update, and delete to DataFrame are reflected in In-memory database
- SnappyData has the same function as RDBMS
Data insert, update, delete code example

```scala
bomStream.foreachRDD(rdd => {
  val streamDS = rdd.toDS()

  // Delete from BOM table
  streamDS.where("ACTION = 'DELETE'").write.deleteFrom("BOM")
  // Insert/Update to BOM table
  streamDS.where("ACTION = 'INSERT'").write.putInto("BOM")
})

machineSensorStream.foreachRDD(rdd => {
  val streamDS = rdd.toDS()

  // Create BOM table DataFrame
  val bom = snappy.table("BOM")

  // Register join result in faulty parts table
  val faultyParts = streamDS.join(bom, "$PartsNo" === "$PartsNo", "leftsemi")
  val faulty = faultyParts.select("SensorId", "VIN", "MachineNo", "PartsNo", "Timestamp")
  faulty.write.insertInto("FaultyParts")
})
```

Possible to insert, update, and delete in standard SQL using SnappySession
Possible to use different table formats depending on data characteristics

**Row table**  
(For master data / transaction data)

<table>
<thead>
<tr>
<th>Parts No</th>
<th>Parts Type</th>
<th>Effective Date</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>9999999999</td>
<td>1</td>
<td>2020/12/01</td>
<td></td>
</tr>
<tr>
<td>876543210</td>
<td>1</td>
<td>2019/04/02</td>
<td></td>
</tr>
<tr>
<td>213757211</td>
<td>2</td>
<td>2020/02/02</td>
<td></td>
</tr>
<tr>
<td>555444777</td>
<td>1</td>
<td>2018/08/13</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>987654321</td>
<td>2</td>
<td>2022/09/30</td>
<td></td>
</tr>
</tbody>
</table>

- frequently inserted or updated
- lookup by key

**Column table**  
(For aggregate / analysis data)

<table>
<thead>
<tr>
<th>Sensor Id</th>
<th>VIN</th>
<th>Machin eNo</th>
<th>Poin t</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11AA</td>
<td>111</td>
<td>1</td>
<td>28.0760</td>
</tr>
<tr>
<td>2</td>
<td>22BB</td>
<td>222</td>
<td>37</td>
<td>60.069</td>
</tr>
<tr>
<td>3</td>
<td>11AA</td>
<td>111</td>
<td>2</td>
<td>37.528</td>
</tr>
<tr>
<td>4</td>
<td>33CC</td>
<td>333</td>
<td>25</td>
<td>1.740</td>
</tr>
<tr>
<td>6</td>
<td>11AA</td>
<td>111</td>
<td>4</td>
<td>394.390</td>
</tr>
</tbody>
</table>

- aggregate or group by specified columns
You can create tables with DDL like RDB

- Additional settings for data distribution and persistence are required

```sql
CREATE TABLE BOM
(PartsNo CHAR(16) NOT NULL PRIMARY KEY,
PartsType CHAR(1) NOT NULL,
EffectiveDate DATE NOT NULL,
... CHAR(3) ,
... CHAR(1) ,
... DATE ,
... DECIMAL(9, 2))
USING ROW OPTIONS

PARTITION_BY 'PartsNo',
COLOCATE_WITH 'PartsType',
REDDUNDANCY '1',
EVICTON_BY 'LRMEMSIZE 10240',
OVERFLOW 'true',
DISKSTORE 'LOCALSTORE',
PERSISTENCE 'ASYNC',
EXPIRE '86400');
```

**Row table**

**Column table**

```sql
CREATE TABLE MachineSensor
(SensorId BIGINT ,
VIN CHAR(20) ,
MachineNo CHAR(16) ,
Value DECIMAL(15, 2) ,
Point CHAR(2) ,
... DATE ,
... DECIMAL(9, 2))
USING COLUMN OPTIONS

PARTITION_BY 'SensorID',
COLOCATE_WITH 'PartsType',
REDDUNDANCY '1',
EVICTON_BY 'LRMEMSIZE 10240',
OVERFLOW 'true',
DISKSTORE 'LOCALSTORE',
PERSISTENCE 'ASYNC';
```

**Database engine**

**Data distribution setting**

**Persistence setting**

**Expire option**
Needs to use replication and partition properly

- Replication can be used instead of broadcast variable
- Partition can be used instead of RDD

**Replication (For master data)**

- Node A: Data A, Data B, Data C, Data D
- Node B: Data A, Data B, Data C, Data D
- Node C: Data A, Data B, Data C, Data D
- Node D: Data A, Data B, Data C, Data D

**Partition (For transaction data)**

- Node A: Data A, Data B
- Node B: Data A, Data B
- Node C: Data A, Data B
- Node D: Data C, Data D

- Node A with a Prim symbol
- Node B with a Prim symbol
- Node C with a Prim symbol
- Node D with a Prim symbol

**Distribute data by multiple nodes**

- Keep same data on all nodes
Transactions can be used like RDBMS

<table>
<thead>
<tr>
<th>Supported transaction isolation level</th>
<th>READ UNCOMMITTED</th>
<th>READ COMMITTED</th>
<th>REPEATABLE READ</th>
<th>SERIALIZABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT FOR UPDATE</td>
<td>Conflict exception occurs at COMMIT when data is updated during transaction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT/ROLLBACK</td>
<td>If the cluster member goes down during a transaction, an exception is raised that COMMIT failed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCK TABLE</td>
<td>Not supported</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CASCADE DELETE</td>
<td>Not supported</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
P2P Architecture

- SnappyData is masterless
- Possible to scale out not only the reading processes but also the writing processes
C) Analytics

- SnappyData has changed SparkSQL to become faster
- Approximate query, TOP K table can be used
SnappyData is 10-20X faster than SparkSQL

TPC-H: 10X-20X faster than Spark 2.0
Implement unique features that enable real-time analysis:

- Real-time analytics can't wait more than 10 seconds for one query.

**Synopsis Data Engine**

<table>
<thead>
<tr>
<th>Approximate query</th>
<th>Stratified Sampling</th>
<th>Query against sampled data</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOP K table</td>
<td>Random Sampling</td>
<td>Use Min / Max to detect outliers</td>
</tr>
</tbody>
</table>
Possible to sample based on cardinality of specific column

- Specify information for sampling in standard DDL

CREATE SAMPLE TABLE MachineSensorSample ON MachineSensor OPTIONS (qcs '"PartsNo, Year, Month"', fraction '0.03') AS (SELECT * FROM MachineSensor)

Sample table

| Sr Id | VIN | Year | Month | Value | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>111</td>
<td>2017</td>
<td>11</td>
<td>10,000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>222</td>
<td>2017</td>
<td>11</td>
<td>3,980</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>111</td>
<td>2017</td>
<td>11</td>
<td>5,130</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>222</td>
<td>2017</td>
<td>11</td>
<td>323,456</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>111</td>
<td>2017</td>
<td>11</td>
<td>1,980</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>111</td>
<td>2017</td>
<td>11</td>
<td>23,456</td>
<td></td>
</tr>
</tbody>
</table>

Base table

MachineSensor

Create Sample table

Sample table

| Sr Id | VIN | Year | Month | Value | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>111</td>
<td>2017</td>
<td>11</td>
<td>10,000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>222</td>
<td>2017</td>
<td>11</td>
<td>3,980</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>111</td>
<td>2017</td>
<td>11</td>
<td>5,130</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>222</td>
<td>2017</td>
<td>11</td>
<td>323,456</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>111</td>
<td>2017</td>
<td>11</td>
<td>1,980</td>
<td></td>
</tr>
</tbody>
</table>

It is sampled based on cardinality of the specified column
For random sampling, no need to create sample tables

• Specify sampling information in SQL

```
SELECT VIN, AVG(Value)
FROM MachineSensor
GROUP BY VIN
ORDER BY VIN
WITH ERROR 0.10
CONFIDENCE 0.95
```

**Approximate Query**

<table>
<thead>
<tr>
<th>Sr Id</th>
<th>VIN</th>
<th>Year</th>
<th>Month</th>
<th>Tx Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>111</td>
<td>2017</td>
<td>11</td>
<td>￥10,000</td>
</tr>
<tr>
<td>2</td>
<td>222</td>
<td>2017</td>
<td>11</td>
<td>￥3,980</td>
</tr>
<tr>
<td>3</td>
<td>111</td>
<td>2017</td>
<td>11</td>
<td>￥5,130</td>
</tr>
<tr>
<td>4</td>
<td>222</td>
<td>2017</td>
<td>11</td>
<td>￥323,45</td>
</tr>
<tr>
<td>5</td>
<td>111</td>
<td>2017</td>
<td>11</td>
<td>￥1,980</td>
</tr>
<tr>
<td>6</td>
<td>111</td>
<td>2017</td>
<td>11</td>
<td>￥23,456</td>
</tr>
</tbody>
</table>

Base table
MachineSensor

Randomly sampled

SQL

Query results
Faster query by utilizing sampling technique

Average usage values

<table>
<thead>
<tr>
<th>VIN</th>
<th>111</th>
<th>222</th>
<th>333</th>
<th>444</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>182.531</td>
<td>132.712</td>
<td>79.521</td>
<td>12.903</td>
</tr>
</tbody>
</table>

Processing time 10 sec

Sampling

Average usage values

<table>
<thead>
<tr>
<th>VIN</th>
<th>111</th>
<th>222</th>
<th>333</th>
<th>444</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>182.294</td>
<td>132.801</td>
<td>79.582</td>
<td>12.912</td>
</tr>
</tbody>
</table>

Processing time 1.2 sec
Comparing confidence of query results and query performance...

<table>
<thead>
<tr>
<th></th>
<th>Std SQL</th>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
<th>Case 4</th>
<th>Case 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>-</td>
<td>0.20</td>
<td>0.05</td>
<td>0.20</td>
<td>0.30</td>
<td>0.20</td>
</tr>
<tr>
<td>Confidence</td>
<td>-</td>
<td>0.70</td>
<td>0.95</td>
<td>0.80</td>
<td>0.80</td>
<td>0.90</td>
</tr>
<tr>
<td>Query time</td>
<td>10.0 sec</td>
<td>1.2 sec</td>
<td>1.4 sec</td>
<td>1.3 sec</td>
<td>1.2 sec</td>
<td>1.1 sec</td>
</tr>
</tbody>
</table>
TOP K table

Collecting the top 50 cases with higher value at 1 minute intervals

```
CREATE TOPK TABLE HighestSensorValue
    on MachineSensor
OPTIONS (
    key              'MachineNo',
    timeInterval     '60000ms',
    size             '50',
    frequencyCol     'Value',
    timeSeriesColumn 'Timestamp'
)
```

<table>
<thead>
<tr>
<th>MachineNo</th>
<th>Value</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6666666666</td>
<td>1,2345,678</td>
<td></td>
</tr>
<tr>
<td>197654532</td>
<td>10,000,000</td>
<td></td>
</tr>
<tr>
<td>197654532</td>
<td>5,048,600</td>
<td></td>
</tr>
<tr>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
</tr>
</tbody>
</table>
Connect to SnappyData from BI tool
Code example: Connect to SnappyData from application

```scala
// connect to SnappyData
val conn = DriverManager.getConnection("jdbc:snappydata://localhost:1527/APP")

// insert data
val psInsert = conn.prepareStatement("INSERT INTO MachineSensor VALUES(?, ?, ?, ?, ...)")
psInsert.setString(1, "1000200030004000")
psInsert.setBigDecimal(2, java.math.BigDecimal.valueOf(100.2))
...
psInsert.executeUpdate()

// select data
val psSelect = conn.prepareStatement("SELECT * FROM MachineSensor WHERE PartsNo=?")
psSelect.setString(1, "1000200030004000")
val rs = psSelect.executeQuery()

// disconnect from SnappyData
conn.close()
```
SnappyData Summary

PART 4
SnappyData Summary

100% compatible with Spark

Spark and In-memory database integrated and simple

Unified with table and SQL

Well designed for faster performance
Thanks!

Contact Information
mailto: info@ulsystems.co.jp
http://www.ulsystems.co.jp
twitter: @MasakiYamakawa