Time Series Storage with Apache Kudu (incubating)

Dan Burkert (Committer)
dan@cloudera.com
@danburkert

Tweet about this talk: @getkudu or #kudu
Time Series

• machine metrics
• event logs
• sensor telemetry
Time Series

Series → Time → Value
## Time Series

<table>
<thead>
<tr>
<th>Series</th>
<th>Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>us-east.appserver01.loadavg.1min</td>
<td>2016-05-09T15:14:30Z</td>
<td>0.44</td>
</tr>
<tr>
<td>us-east.appserver01.loadavg.1min</td>
<td>2016-05-09T15:14:40Z</td>
<td>0.53</td>
</tr>
<tr>
<td>us-west.dbserver03.rss</td>
<td>2016-05-09T15:14:30Z</td>
<td>1572864</td>
</tr>
<tr>
<td>us-west.dbserver03.rss</td>
<td>2016-05-09T15:15:00Z</td>
<td>2097152</td>
</tr>
</tbody>
</table>
Time Series — Design Criteria

- Insert Performance (throughput & latency)
- Read Performance (throughput & latency)
- Storage Overhead (compression ratio)
Time Series — Common Patterns

• Datapoints are inserted in time order across all series

• Reads specify a series and a time range, containing hundreds to many thousands of datapoints

```
SELECT time, value FROM timeseries
WHERE series = "us-west.dbserver03.rss"
AND time >= 2016-05-08T00:00:00;
```
Storage Landscape in Hadoop Ecosystem

**HDFS (GFS) excels at:**
- Batch ingest only (e.g., hourly)
- Efficiently scanning large amounts of data (analytics)

**HBase (BigTable) excels at:**
- Efficiently finding and writing individual rows
- Making data mutable

Gaps exist when these properties are needed *simultaneously*
**Kudu Design Goals**

- **High throughput** for big scans  
  *Goal*: Within 2x of Parquet

- **Low-latency** for short accesses  
  *Goal*: 1ms read/write on SSD

- **Database-like** semantics (initially single-row ACID)

- **Relational data model**  
  - SQL queries are easy  
  - “NoSQL” style scan/insert/update (Java, C++, Python clients)
Novel Kudu Features

• Columnar Storage

• Flexible Data Partitioning

• Next generation consistency & replication
Columnar Storage
Row Storage

(us-east.appserver01.loadavg.1min, 2016-05-09T15:14:30Z, 0.56)
(us-east.appserver01.loadavg.1min, 2016-05-09T15:14:40Z, 0.70)
(us-east.appserver01.loadavg.1min, 2016-05-09T15:14:50Z, 0.72)
(us-east.appserver01.loadavg.1min, 2016-05-09T15:15:00Z, 0.55)
Columnar Storage

us-east.appserver01.loadavg.1min
us-east.appserver01.loadavg.1min
us-east.appserver01.loadavg.1min
us-east.appserver01.loadavg.1min

2016-05-09T15:14:30Z
2016-05-09T15:14:40Z
2016-05-09T15:15:00Z
2016-05-09T15:15:10Z

0.56
0.70
0.72
0.55
Columnar Storage

- Improved Scan performance
  - Predicates (e.g. `WHERE time >= 2016-05-08T00:00:00`) can be evaluated without reading unnecessary data from other columns
  - Efficient encodings can dramatically improve compression ratios, which reduces effective IO load
  - Typed, homogenous data plays well to modern processor strengths (vectorization, pipelining)

- At the cost of random access performance
  - Single row access requires a number of seeks proportional to the number of columns
  - BUT, random access is becoming cheaper (Cheap RAM, SSDs, NVRAM)
Columnar Storage - Dictionary Encoding

- Transform blocks of low cardinality values (e.g., series) into an offset array and a dictionary lookup table
Columnar Storage - Dictionary Encoding

- Transform blocks of low cardinality values (e.g. series) into an offset array and a dictionary lookup table

| us-east.appserver01.loadavg.1min | 0 |
| us-west.dbserver03.rss           | 1 |
| us-east.appserver01.loadavg.1min | 0 |
| asia.appserver02.rss             | 2 |
| us-west.dbserver03.rss           | 1 |
Columnar Storage - Bitshuffle Encoding

- Transform blocks of similar data by transposing values at the bit level
- Dramatically improves the effectiveness of high throughput general purpose compression algorithms (LZ4, snappy)
Columnar Storage - Bitshuffle Encoding

- Transform blocks of similar data by transposing values at the bit level
- Dramatically improves the effectiveness of high throughput general purpose compression algorithms (LZ4, snappy)
Columnar Storage - Bitshuffle Encoding

- Transform blocks of similar data by transposing values at the bit level
- Dramatically improves the effectiveness of high throughput general purpose compression algorithms (LZ4, snappy)

```
00000000 0000101 00110010 01101010 01000011 11011100 10000001 10000000
00000000 0000101 00110010 01101010 01000100 01110101 00011000 00000000
00000000 0000101 00110010 01101010 01000101 00001101 10101110 10000000
00000000 0000101 00110010 01101010 01000101 10100110 01000101 00000000
```

time bits (µs since unix epoch)
Columnar Storage - Bitshuffle Encoding

- Transform blocks of similar data by transposing values at the bit level
- Dramatically improves the effectiveness of high throughput general purpose compression algorithms (LZ4, snappy)

```
00000000 00000101 00110010 01101010 01000011 11011100 10000001 10000000
00000000 00000101 00110010 01101010 01000100 01110101 00011000 00000000
00000000 00000101 00110010 01101010 01000101 00001101 10101110 10000000
00000000 00000101 00110010 01101010 01000101 10100110 01000101 00000000
```

time bits (µs since unix epoch)
Columnar Storage - Bitshuffle Encoding

- Transform blocks of similar data by transposing values at the bit level
- Dramatically improves the effectiveness of high throughput general purpose compression algorithms (LZ4, snappy)

```
00000000 00000000 00000000 00000000 00000000 00000000 00011111 00011111
00000000 11111111 00000000 11110000 00011111 11110000 11110000 11110000
00001111 00000000 00001111 10001011 10011100 01011100 10101111 00010110
10100001 00100100 01100011 00101001 10100000 00000000 00000000 00000000
```

transposed time bits (µs since unix epoch)

individual runs of N bits are highly correlated
(int this case N = 4 datapoints)
Flexible Data Partitioning
Partitioning vs Indexing

- **Partitioning** how datapoints are distributed among partitions
  - Kudu: tablet
  - HBase: region
  - Cassandra: VNode

- **Indexing** how data within a single partition is sorted
  - for time series, data should typically indexed by (series, time):
Partitioning vs Indexing

- **Partitioning** how datapoints are distributed among partitions
  - Kudu: tablet
  - HBase: region
  - Cassandra: VNode

- **Indexing** how data within a single partition is sorted
  - for time series, data should typically indexed by (series, time):

<table>
<thead>
<tr>
<th>(series, time)</th>
<th>(time, series)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(us-east.appserver01.loadavg, 2016-05-09T15:14:00Z)</td>
<td>(2016-05-09T15:14:00Z, us-east.appserver01.loadavg)</td>
</tr>
<tr>
<td>(us-east.appserver01.loadavg, 2016-05-09T15:15:00Z)</td>
<td>(2016-05-09T15:14:30Z, us-west.dbserver03.rss)</td>
</tr>
<tr>
<td>(us-west.dbserver03.rss, 2016-05-09T15:14:30Z)</td>
<td>(2016-05-09T15:15:00Z, us-east.appserver01.loadavg)</td>
</tr>
<tr>
<td>(us-west.dbserver03.rss, 2016-05-09T15:14:30Z)</td>
<td>(2016-05-09T15:14:30Z, us-west.dbserver03.rss)</td>
</tr>
</tbody>
</table>
Partitioning vs Indexing

- **Partitioning** how datapoints are distributed among partitions
  - Kudu: tablet
  - HBase: region
  - Cassandra: VNode

- **Indexing** how data within a single partition is sorted
  - for time series, data should typically indexed by (series, time):

```
SELECT * WHERE series = 'us-east.appserver01.loadavg';
```
Partitioning

• Kudu has flexible policies for distributing data among partitions

• Indexing is independent of partitioning

• Hash partitioning is built in, and can be combined with range partitioning
Partitioning — By Time Range

Partition 1
start: 2016-05-07T00:00:00Z
end: 2016-05-08T00:00:00Z

Partition 2
start: 2016-05-08T00:00:00Z
end: 2016-05-09T00:00:00Z

Partition 3
start: 2016-05-09T00:00:00Z
end: 2016-05-10T00:00:00Z
Partitioning — By Time Range (inserts)

Partition 1
start: 2016-05-07T00:00:00Z
deadline: 2016-05-08T00:00:00Z

Partition 2
start: 2016-05-08T00:00:00Z
deadline: 2016-05-09T00:00:00Z

Partition 3
start: 2016-05-09T00:00:00Z
deadline: 2016-05-10T00:00:00Z

All Inserts go to Latest Partition
Partitioning — By Time Range (scans)

Partition 1
start: 2016-05-07T00:00:00Z
end: 2016-05-08T00:00:00Z

Partition 2
start: 2016-05-08T00:00:00Z
end: 2016-05-09T00:00:00Z

Partition 3
start: 2016-05-09T00:00:00Z
end: 2016-05-10T00:00:00Z

Big scans (across large time intervals) can be parallelized across many partitions
Partitioning — By Series Range

- **Partition 1**
  - start: `<start>`
  - end: 'us-east'

- **Partition 2**
  - start: 'us-east'
  - end: 'us-west'

- **Partition 3**
  - start: 'us-west'
  - end: `<end>`
Partitioning — By Series Range (inserts)

Inserts are spread among all partitions
Partitioning — By Series Range (scans)

Partition 1
start: <start>
end: 'us-east'

Partition 2
start: 'us-east'
end: 'us-west'

Partition 3
start: 'us-west'
end: <end>

Scans are over a single partition
Partitioning — By Series Range

Partitions can become unbalanced, resulting in hot spotting
Partitioning — By Series Hash

Partition 1
series bucket: 0

Partition 2
series bucket: 1

Partition 3
series bucket: 2
Partitioning — By Series Hash (inserts)

Inserts are spread among all partitions
Partitioning — By Series Hash (scans)

Partition 1
series bucket: 0

Partition 2
series bucket: 1

Partition 3
series bucket: 2

Scans are over a single partition
Partitioning — By Series Hash

Partitions grow overtime, eventually becoming too big for a single server
Partitioning — By Series Hash + Time Range
Partitioning — By Series Hash + Time Range (inserts)

Inserts are spread among all partitions in the latest time range.
Partitioning — By Series Hash + Time Range (scans)

Big scans (across large time intervals) can be parallelized across partitions
Time Series Storage in Kudu

• Columnar Storage

• Flexible Partitioning
Kudu — Roadmap

• Current version - 0.8
  • Beta quality

• 6 week release cycle

• 1.0 tentatively planned for late summer
  • First ‘production ready’ release
More Kudu

• Using Kafka and Kudu for Fast, Low Latency SQL Analytics on Streaming Data
  • Mike Percy & Ashish Singh
  • Tuesday at 3:00pm

• Kudu and Spark for Fast Analytics on Streaming Data
  • Mike Percy & Dan Burkert
  • Vancouver Spark Meetup (in the Hyatt)
  • Tuesday, 6:00pm

• getkudu.io/ | @getkudu | @danburkert

• Slack room – getkudu-slack.herokuapp.com/

• github.com/danburkert/kudu-ts - early stage metrics storage API built on Kudu