Storing Data

OpenTSDB stores data in two tables. Before a row can be inserted into the tsdb table, all UIDs must first be generated. Let’s start at the beginning.

Creating a UID

Before a measurement can be written to the tsdb table, all of its tags must first be written to tsdb-uid. In pseudo-code, that process is handled by the UniqueId.getOrCreateId() method, which looks like this:

Listing 1 Pseudo-code for inserting a tag into the tsdb-uid table

class UniqueId:
    MAXID_ROW = 0x0
    ID_FAMILY = toBytes("id")
    NAME_FAMILY = toBytes("name")

    def UniqueId(this, table, kind): #A
        this.table = table
        this.kind = toBytes(kind) #B

    def getOrCreateId(this, name):
        uid = HBase.get(this.table, toBytes(name), #C
                        ID_FAMILY, this.kind)      #C
        if 0x0 != uid:                             #C
            return uid
        #C
        uid = HBase.incrementColumnValue(MAXID_ROW, #D
                                         ID_FAMILY, #D
                                         this.kind) #D

        HBase.put(this.table, toBytes(uid),          #E
                  NAME_FAMILY, this.kind, toBytes(name)) #E
        HBase.put(this.table, toBytes(name), ID_FAMILY, #F
                  this.kind, toBytes(uid))              #F
        return uid

#A A single UniqueId instance is instantiated for each kind of tag
#B Kind can be one of "metric", "tag name", or "tag value"
#C Returns the uid for name if it already exists
#D Otherwise, generates and stores a new uid
#E Writes the uid => name mapping
#F Writes the name => uid mapping

One UniqueId class is instantiated by a tsd process for each kind of UID stored in the table. In this case, "metric", "tag name", and "tag value" are the three kinds of UIDs in the system. The local variable kind will be set appropriately in the constructor, along with the variable table for the table name, “tsdb-uid” by default. The very
first thing the `UniqueId.getOrCreateId()` method does is look in the table to see if there’s already a UID of this kind with this name. If it’s there, we’re done. Return it and move on. Otherwise, a new UID will need to be generated and registered for this mapping.

HBase exposes an atomic increment method that will increment a cell value without risk of interruption by another process. New UIDs are generated by way of a counter stored in this table, increased using this increment feature. A new UID is generated and the two mappings are stored to the table. Once a mapping is written, it’s expected to never change. For this reason the UID-to-name mapping is written before the name-to-UID mapping. A failure here results in a wasted UID but no further harm. A name-to-UID mapping without its reciprocal means the name is available in the system but can never be resolved from a measurement record. That’s a bad thing. Finally, having assigned the bidirectional mapping, the UID is returned.

The Java code for this method contains additional complexity due to error handling and a workaround for a feature not yet implemented in the client API. It’s included here but with these additional concerns removed for the sake of brevity.

Listing 2 Reduced Java code for `UniqueId.getOrCreateId()`

```java
public byte[] getOrCreateId(String name) throws HBaseException {
    short attempt = MAX_ATTEMPTS_ASSIGN_ID;
    HBaseException hbe = null;

    while (attempt-- > 0) {
        try {  // A
            return getId(name);
        } catch (...) {
            RowLock lock = ... getLock();  // B
            try {  // C
                final byte[] id = getId(name);
                return id;
            } catch (...) {
                try {  // D
                    final byte[] id = getId(name);
                    return id;
                } catch (...) {
                    long id;
                    byte[] row;  // E
                    try {  // F
                        row = hbaseICV[MAXID_ROW, ID_FAMILY, lock)
                        if (row == null) {
                            id = 1;
                            row = Bytes.fromLong(id);
                        } else {
                            id = Bytes.getLong(row);
                        }
                        row = Arrays.copyOfRange(row, row.length + idWidth, row.length);
                    } catch (...) {
                        ...  // G
                    }
                    try {  // H
                        final PutRequest reverse_mapping = new PutRequest(  // H
                            table, row, NAME_FAMILY, kind, toBytes(name));          // H
                        hbasePutWithRetry(reverse_mapping, MAX_ATTEMPTS_PUT, INITIAL_EXP_BACKOFF_DELAY);  // H
                    } catch (...) {
                        ...  // I
                    }
                    try {  // J
                        final PutRequest forward_mapping = new PutRequest(  // J
                            table, toBytes(name), ID_FAMILY, kind, row);          // J
                        hbasePutWithRetry(forward_mapping, MAX_ATTEMPTS_PUT, INITIAL_EXP_BACKOFF_DELAY);  // J
                    }
                }
            }
        }
    }
}
```

For Source Code, Sample Chapters, the Author Forum and other resources, go to [http://www.manning.com/dimidukkhurana/](http://www.manning.com/dimidukkhurana/)
Having registered our tags, we can move on to generating a rowkey for an entry in the tsdb table.

### Generating a partial rowkey

Every rowkey in the tsdb table for the same metric and tag name value pairs looks the same except for the timestamp. OpenTSDB implements this functionality in a helper method for partial construction of these rowkeys in `IncomingDataPoints.rowKeyTemplate()`. That method implemented in pseudo-code looks like this.

```python
Listing 3 Pseudo-code for generating a rowkey template

class IncomingDataPoints:
    TIMESTAMP_BYTES = 4

    def static getOrCreateTags(tsdb, tags):
        tag_ids = []
        for(name, value in tags.sort()): #A
            tag_ids += tsdb.tag_names.getOrCreateId(name)   #B
            tag_ids += tsdb.tag_values.getOrCreateId(value) #C
        return ByteArray(tag_ids)

    def static rowKeyTemplate(tsdb, metric, tags):
        metric_width = tsdb.metrics.width()       #D
        tag_name_width = tsdb.tag_names.width()   #D
        tag_value_width = tsdb.tag_values.width() #D
        num_tags = tags.size()
        row_size = (metric_width + TIMESTAMP_BYTES    #E
                                   + tag_name_width * num_tags   #E
                                   + tag_value_width * num_tags) #E
        row = ByteArray(row_size)
        row[0 .. metric_width] = tsdb.metrics.getOrCreateId(metric) #F
        row[metric_width + TIMESTAMP_BYTES ..] = #G
         getOrCreateTags(tsdb, tags) #G

        return row

#A Tags is a map from tag name => tag value
#B tsdb.tag_names is a UniqueID instance
#C So is tsdb.tag_values
#D Requires a tsdb instance for relevant context
#E Row width is variable based on the number of tags
#F tsdb.metrics is a UniqueID instance
```

For Source Code, Sample Chapters, the Author Forum and other resources, go to [http://www.manning.com/dimidukkhurana/](http://www.manning.com/dimidukkhurana/)
The primary concern of this method is correct placement of the rowkey components. As this order is metric UID, partial timestamp, and tag pair UIDs. Notice the tags are sorted before insertion. This is simply to avoid ambiguity in iteration over an unordered collection.

The Java implementation in listing 4 is almost identical to our pseudo-code. The biggest difference is organization of helper methods.

**Listing 4 Java code for IncomingDataPoints.rowKeyTemplate().**

```java
static byte[] rowKeyTemplate(final TSDB tsdb,
                             final String metric,
                             final Map<String, String> tags) {
    final short metric_width = tsdb.metrics.width();
    final short tag_name_width = tsdb.tag_names.width();
    final short tag_value_width = tsdb.tag_values.width();
    final short num_tags = (short) tags.size();

    int row_size = (metric_width + Const.TIMESTAMP_BYTES
                   + tag_name_width * num_tags
                   + tag_value_width * num_tags);
    final byte[] row = new byte[row_size];

    short pos = 0;
    copyInRowKey(row, pos, (AUTO_METRIC ? tsdb.metrics.getOrCreateId(metric)
                          : tsdb.metrics.getId(metric)));
    pos += metric_width;
    pos += Const.TIMESTAMP_BYTES;
    for (final byte[] tag : Tags.resolveOrCreateAll(tsdb, tags)) {
        copyInRowKey(row, pos, tag);
        pos += tag.length;
    }
    return row;
}
```

That’s all there is to it! You finally have all the missing pieces.

**Writing a measurement**

With all the necessary helper methods in place, it’s time to write a record to the tsdb table. The process looks like this:

1. Build the rowkey.
2. Decide on the column family and qualifier.
3. Identify the bytes to store in the cell.
4. Write the record.

This logic is encapsulated behind the TSDB.addPoint() method. Those tsdb instances in the previous code listings are instances of this class. We’ll start once more with pseudo-code before diving into the Java implementation.

**Listing 5 Pseudo-code for inserting a tsdb record**

```java
class TSDB:
    FAMILY = toBytes("t")
    MAX_TIMESPAN = 3600  #A
    FLAG_BITS    = 4     #B
    FLOAT_FLAGS  = 1011b #C
    LONG_FLAGS   = 0111b #C

def addPoint(this, metric, timestamp, value, tags):
    row =                                                    #D
        IncomingDataPoints.rowKeyTemplate(this, metric, tags) #D
```

For Source Code, Sample Chapters, the Author Forum and other resources, go to 
base_time = (timestamp - (timestamp % MAX_TIMESPAN))  #D
row[metrics.width()] = base_time  #D
flags = value.isFloat? ? FLOAT_FLAGS : LONG_FLAGS  #E
qualifier = (timestamp - basetime) << FLAG_BITS | flags  #E
qualifier = toBytes(qualifier)  #E
HBase.put(this.table, row, FAMILY, qualifier, toBytes(value))  
#A 60 seconds * 60 minutes = 1 hour resolution per row
#B 4 bits of the 2-byte column qualifier are reserved for a mask
#C Flag masks in binary
#D Assembles the rowkey
#E Assembles the column qualifier

Writing the value is as simple as that! Now let’s look at the same thing in Java. The code for writing Longs and Floats is almost identical. We’ll look at writing Longs.

Listing 6 Java code for TSDB.addPoint()

```java
public Deferred<Object> addPoint(final String metric,
final long timestamp,
final long value,
final Map<String, String> tags) {
    final short flags = 0x7;
    return addPointInternal(metric, timestamp, Bytes.fromLong(value),
                             tags, flags);
}
```

```java
private Deferred<Object> addPointInternal(final String metric,
final long timestamp,
final byte[] value,
final Map<String, String> tags,
final short flags) {
    if ((timestamp & 0xFFFFFFFF00000000L) != 0) {  #A
        throw ... #A
    }

    IncomingDataPoints.checkMetricAndTags(metric, tags);
    final byte[] row = IncomingDataPoints.rowKeyTemplate(this, metric, tags);
    final long base_time = (timestamp - (timestamp % Const.MAX_TIMESPAN));
    Bytes.setInt(row, (int) base_time, metrics.width());
    final short qualifier = (short) ((timestamp - base_time) <<
        Const.FLAG_BITS | flags);
    final PutRequest point = new PutRequest(table, row, FAMILY,
        Bytes.fromShort(qualifier), value);
    return client.put(point);
}
```

#A Verifies timestamp < 0 || timestamp > Integer.MAX_VALUE

Looking back over these listings, both pseudo-code and Java, there’s not much interaction with HBase going on. The most complicated part of writing a row to HBase is assembling the values you want to write. Actually writing the record is the easy part!

**Summary**

HBase is built in Java; therefore, its native client library is also Java. However, you can access HBase by running a REST service or in your favorite language via thrift bindings. OpenTSDB’s tsd is implemented in Java using a nonstandard client library called asynchbase for all interaction. In this short article, you learned how OpenTSDB stores data.

For Source Code, Sample Chapters, the Author Forum and other resources, go to [http://www.manning.com/dimidukkhurana/](http://www.manning.com/dimidukkhurana/)
Here are some other Manning titles you might be interested in:

**Hadoop in Practice**
Alex Holmes

**Big Data**
Nathan Marz and Sam Ritchie

**Machine Learning in Action**
Peter Harrington

Last updated: February 16, 2012