TerraMeta Software, Inc.

CloudGraph HBase Easy Wide Rows

1 Introduction

The term “wide row” is used in relation to columnar key-value stores such as HBase and others and refers to rows that have a large (and variable) number of columns. Such rows can be returned in a single ‘GET’ operation but ideally with only the desired subset or “slice” of columns. Such row “slice” GET operations are typically much faster than SCAN operations, and HBase data models that facilitate slice operations can be very advantageous. This step-by-step guide shows how to build a Maven project which generates a simple HBase data model with 1 “wide row” table with example row slice queries. It uses only annotated Java (POJO) objects as the source of schema or metadata. It requires basic knowledge of the Java programing language, Apache Maven, HBase Server administration and assumes the following software install prerequisites.

- Java JDK 1.7 or Above
- Maven 3.x or Above
- HBase 1.0 or Above

For more information on wide column stores, see https://en.wikipedia.org/wiki/Wide_column_store. See https://github.com/cloudgraph/cloudgraph-examples-quickstart for working examples which accomplany this guide.

2 CloudGraph HBase Easy Wide Rows

2.1 Add Dependencies

Add the following dependency to your Maven project to get started.

```
<dependency>
	<groupId>org.terrameta</groupId>
	<artifactId>plasma-core</artifactId>
	<version>2.0.1</version>
</dependency>
<dependency>
	<groupId>org.cloudgraph</groupId>
	<artifactId>cloudgraph-hbase</artifactId>
	<version>1.0.8</version>
</dependency>
```

2.2 Create Entity POJOs

Next we create a “Banking” data model using just Java POJO’s. Create 3 Java enumeration classes annotated as below in a Java package called **examples.quickstart.types**. *(Note: Enumerations rather than Java classes are annotated to facilitate reuse across multiple code generation and metadata integration contexts. Your metadata is too valuable to relegate to a single context)*
The annotations capture typical structural metadata elements.

- Types and Aliases profiding logical / physical name isolation. See Type, Alias.
- Data Types. See DataProperty, ReferenceProperty.
- Cardinalities, Nullability and Visibility. See DataProperty, ReferenceProperty.
- Constraints. See ValueConstraint, EnumerationConstraint.
- Inheritance relationships (multiple inheritance is supported). See Type.
- Associations between entities. See ReferenceProperty.
- Enumerations as Domain Value Lists. See Enumeration.

In addition for HBase add the Table annotation and RowKeyField annotations in order to map specific entities to HBase tables and map specific entity fields to row key fields.

Enumeration 1 – `Issuer`.java

```
package examples.quickstart.types;

import org.plasma.sdo.annotation.Alias;
import org.plasma.sdo.annotation.Enumeration;

@Enumeration(name = "Issuer")
public enum Issuer {
  @Alias(physicalName = "V")
  visa,
  @Alias(physicalName = "M")
  mastercard,
  @Alias(physicalName = "A")
  americanexpress,
  @Alias(physicalName = "O")
  other
}
```

Entity 1 – Card.java

```
package examples.quickstart.types;

import org.cloudgraph.store.mapping.annotation.RowKeyField;
import org.cloudgraph.store.mapping.annotation.Table;
import org.plasma.sdo.DataType;
import org.plasma.sdo.annotation.Alias;
import org.plasma.sdo.annotation.Comment;
import org.plasma.sdo.annotation.DataProperty;
import org.plasma.sdo.annotation.EnumConstraint;
import org.plasma.sdo.annotation.ReferenceProperty;
import org.plasma.sdo.annotation.Type;
import org.plasma.sdo.annotation.ValueConstraint;

@Comment(body = "A simple example bank card entity with associated
transactions")
@Table(name = "CARD")
@Alias(physicalName = "CD")
@Type
public enum Card {
  @Comment(body = "The card issuer")
  @RowKeyField
  @ValueConstraint(maxLength = "1")
  @EnumConstraint(targetEnum = Issuer.class)
  @Alias(physicalName = "ISU")
  @DataProperty(dataType = DataType.String, isNullable = false)
  issuer,

  @Comment(body = "The card 16 digit (plastic) number")
  @RowKeyField
  @ValueConstraint(maxLength = "16")
  @Alias(physicalName = "NUM")
  @DataProperty(dataType = DataType.String, isNullable = false)
  number,

  @Comment(body = "Links the card to any number of transaction entities")
  @Alias(physicalName = "TNS")
  @ReferenceProperty(isNullable = true, isMany = true, targetClass =
Transaction.class, targetProperty = "")
  transaction;
}
```

Entity 2 – Transaction.java

```
package examples.quickstart.types;

import org.cloudgraph.store.mapping.annotation.Table;
import org.plasma.sdo.DataType;
import org.plasma.sdo.annotation.Alias;
import org.plasma.sdo.annotation.Comment;
import org.plasma.sdo.annotation.DataProperty;
import org.plasma.sdo.annotation.Type;

/**
 * Transaction entity. Note: not bound to a {@link Table} but
 * linked to a {@link Card}, forming a wide row with {@link Card}
 * as the root entity.
 */
@Comment(body = "An example banking transaction entity")
@Alias(physicalName = "T")
@Type
public enum Transaction {

  @Comment(body = "The dollar amount component for the transaction")
  @Alias(physicalName = "D")
  @DataProperty(dataType = DataType.Int)
  dollars,

  @Comment(body = "The cents amount component for the transaction")
  @Alias(physicalName = "C")
  @DataProperty(dataType = DataType.Short)
  cents,

  @Comment(body = "The date the transaction occurred")
  @Alias(physicalName = "TD")
  @DataProperty(dataType = DataType.Date)
  transactionDate,

}
```

2.3 Create Namespace POJO

In the same package as the above POJOs, create a file called package_info.java with the below annotates. These annotations associate the entities we created previously with a common namespace and data access context. For more information on applying annotations to package_into.java see https://www.intertech.com/Blog/whats-package-info-java-for

```
@Alias(physicalName = "BNK")
@Namespace(uri = "http://cloudgraph-easy-wide-rows/banking")
@NamespaceProvisioning(rootPackageName = "examples.quickstart.model")
@NamespaceService(storeType = DataStoreType.NOSQL, providerName =
DataAccessProviderName.HBASE, properties = {
    "hbase.zookeeper.quorum=zookeeper_host1:2181, zookeeper_host2:2181,
zookeeper_host3:2181",
    "hbase.zookeeper.property.clientPort=2181",
    "org.plasma.sdo.access.provider.hbase.ConnectionPoolMinSize=1",
    "org.plasma.sdo.access.provider.hbase.ConnectionPoolMaxSize=80" })
package examples.quickstart.types;
```

```
import org.plasma.runtime.DataAccessProviderName;
import org.plasma.runtime.DataStoreType;
import org.plasma.runtime.annotation.NamespaceProvisioning;
import org.plasma.runtime.annotation.NamespaceService;
import org.plasma.sdo.annotation.Alias;
import org.plasma.sdo.annotation.Namespace;
```

Namespace 1 – package_info.java

2.4 Add Plasma Maven Plugin

Add the CloudGraph Maven Plugin with 2 executions which generate data access and query (DSL) classes. See below Plasma Maven Plugin Configuration for complete listing.
See https://github.com/cloudgraph/cloudgraph-examples-quickstart for working examples which accomplany this guide.

2.5 Generate Source

After adding the plugin and 2 executions type:

```
maven generate-sources
```

The generated data access source code should appear under `target/generated-sources/quickstart.pojo.model` which is the package we specified in @NamespaceProvisioning on the namespace.

2.6 Add Run Time Dependencies

Next, add the following additional dependencies to your Maven project, including an HBase data access service provider (CloudGraph HBase).

```
<dependency>
    <groupId>org.cloudgraph</groupId>
    <artifactId>cloudgraph-hbase</artifactId>
    <version>1.0.8</version>
</dependency>
```

2.7 Insert and Query HBase Data

And finally create a class as below which inserts a single Card with random card number and several child Transaction entities. This will result ina single (wide) HBase row. The example then queries for a “slice” of the transactions within a specific dollar amount range. Then finaly the example prints the serialized result graphs as formatted XML for easy visualization and debugging. The final output should look like the below XML example. See https://github.com/cloudgraph/cloudgraph-examples-quickstart for working examples which accomplany this guide.

Figure 3 – Insert/Query HBase Data

```
package examples.quickstart;

import java.io.IOException;
import java.util.Date;
import java.util.Random;

import org.plasma.query.Expression;
import org.plasma.runtime.*;
import org.plasma.sdo.*;
import org.plasma.sdo.access.client.*;
import org.plasma.sdo.helper.*;
```

```
import commonj.sdo.*;
import examples.quickstart.model.Card;
import examples.quickstart.model.Issuer;
import examples.quickstart.model.Transaction;
import examples.quickstart.model.query.QCard;
import examples.quickstart.model.query.QTransaction;

public class ExampleRunner {
  public static final int MIN_DOLLARS = 500;
  public static final int MAX_DOLLARS = 600;
  static Random random = new Random();

  public static DataGraph[] runExample() throws IOException {
    SDODataAccessClient client = new SDODataAccessClient(
        new PojoDataAccessClient(DataAccessProviderName.HBASE));

    DataGraph dataGraph = PlasmaDataFactory.INSTANCE.createDataGraph();
    dataGraph.getChangeSummary().beginLogging();
    Type rootType = PlasmaTypeHelper.INSTANCE.getType(Card.class);

    Card card = (Card) dataGraph.createRootObject(rootType);
    card.setNumber(randomCard());
    card.setIssuer(Issuer.VISA.getInstanceName());

    for (int i = 0; i < 5; i++) {
      Transaction trans = card.createTransaction();
      trans.setTransactionDate(new Date());
      trans.setDollars(random.nextInt(1000));
      trans.setCents((short) random.nextInt(99));
    }

    client.commit(dataGraph, ExampleRunner.class.getSimpleName());

    QCard query = QCard.newQuery();
    QTransaction transaction = QTransaction.newQuery();
    Expression slice = transaction.dollars().between(MIN_DOLLARS, MAX_DOLLARS);
    query.select(query.wildcard()).select(query.transaction(slice).wildcard());
    query.where(query.issuer().eq(Issuer.VISA.getInstanceName()));

    DataGraph[] results = client.find(query);
    return results;
  }

  private static String randomCard() {
    StringBuilder buf = new StringBuilder();
    buf.append(String.format("%04d", random.nextInt(9999)));
    buf.append(String.format("%04d", random.nextInt(9999)));
    buf.append(String.format("%04d", random.nextInt(9999)));
    buf.append(String.format("%04d", random.nextInt(9999)));
    return buf.toString();
  }

  public static void main(String[] args) {
    try {
      DataGraph[] results = runExample();
      for (DataGraph graph : results)
        System.out.println(((PlasmaDataGraph) graph).asXml());
    } catch (IOException e) {
      e.printStackTrace();
    }
```

```
  }
}
```

Figure 4 – Result Graph, Serialized as XML

```
<ns1:Card xmlns:ns1="http://cloudgraph-easy-wide-rows/banking"
    issuer="V" number="9770236769916284">
  <transaction dollars="584" cents="67" transactionDate="2017-11-
26T22:40:14"></transaction>
  <transaction dollars="577" cents="26" transactionDate="2017-11-
26T22:40:15"></transaction>
  <transaction dollars="594" cents="82" transactionDate="2017-11-
26T22:40:15"></transaction>
  <transaction dollars="546" cents="50" transactionDate="2017-11-
26T22:40:15"></transaction>
  <transaction dollars="562" cents="25" transactionDate="2017-11-
26T22:40:15"></transaction>
  <transaction dollars="541" cents="98" transactionDate="2017-11-
26T22:40:15"></transaction>
  <transaction dollars="563" cents="57" transactionDate="2017-11-
26T22:40:15"></transaction>
  <transaction dollars="582" cents="38" transactionDate="2017-11-
26T22:40:14"></transaction>
</ns1:Card>
```

See https://github.com/cloudgraph/cloudgraph-examples-quickstart for working examples which accomplany this guide.

3 Plasma Maven Plugin Configuration

Below is the Maven plugin listing referenced about which is needed for generation of data access source code and DDL. See https://github.com/cloudgraph/cloudgraph-examples-quickstart for working examples which accomplany this guide.

```
<plugin>
	<groupId>org.terrameta</groupId>
	<artifactId>plasma-maven-plugin</artifactId>
	<version>2.0.0</version>
	<dependencies>
		<dependency>
			<groupId>org.terrameta</groupId>
			<artifactId>plasma-core</artifactId>
			<version>2.0.1</version>
		</dependency>
		<dependency>
			<groupId>org.cloudgraph</groupId>
			<artifactId>cloudgraph-hbase</artifactId>
			<version>1.0.8</version>
		</dependency>
	</dependencies>
	<executions>
		<execution>
			<id>sdo-create</id>
			<configuration>
```

```
                        <action>create</action>
                        <dialect>java</dialect>
                        <additionalClasspathElements>
                              <param>${basedir}/target/classes</param>
                        </additionalClasspathElements>
                        <outputDirectory>${basedir}/target/generated-
sources/java</outputDirectory>
                  </configuration>
                  <goals>
                        <goal>sdo</goal>
                  </goals>
            </execution>
            <execution>
                  <id>dsl-create</id>
                  <configuration>
                        <action>create</action>
                        <dialect>java</dialect>
                        <additionalClasspathElements>
                              <param>${basedir}/target/classes</param>
                        </additionalClasspathElements>
                        <outputDirectory>${basedir}/target/generated-
sources/java</outputDirectory>
                  </configuration>
                  <goals>
                        <goal>dsl</goal>
                  </goals>
            </execution>
      </executions>
</plugin>
.
```

4 Maven Compiler Plugin Configuration

We use 2 executions in the compiler plugin because the annotation discovery for your annotated Java requires COMPILED classes. The compiled annotated classes are first used at generate-sources phase, then for several later Maven phases. An alternative to this "trick" is to isolated your annotated classes in a separate compiled Maven module, then perform the code generation in a second module which depends on the first. See https://github.com/cloudgraph/cloudgraph-examples-quickstart for working examples which accomplany this guide.

```
<plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
      <configuration>
            <source>1.7</source>
            <target>1.7</target>
            <encoding>UTF-8</encoding>
      </configuration>
      <executions>
            <execution>
                  <id>default-compile</id>
                  <phase>generate-sources</phase>
                  <configuration>
                        <excludes>
                              <exclude>**/generated-sources/*</exclude>
                              <exclude>**/examples/quickstart/*</exclude>
```

```
                        </excludes>
                  </configuration>
            </execution>
            <execution>
                  <id>compile-generated</id>
                  <phase>compile</phase>
                  <goals>
                        <goal>compile</goal>
                  </goals>
                  <configuration>
                  </configuration>
            </execution>
      </executions>
</plugin>
```