JDBCStorageHandler

The JDBCStorageHandler allows the creation of tables in Hive that are backed by a JDBC compliant database. Please see Storage Handlers on the Hive wiki for more background on the StorageHandler interface.

The JDBCStorageHandler is implemented as a wrapper around Hadoop's DBInputFormat & DBOutputFormat API .

Data Definition Statements (DDL)

To interact with a JDBC table in Hive you must add the table to the Hive MetaStore via a create table statement. This provides Hive with the metadata that it needs to read and write to the table.

An Example Create Table Statement

```
CREATE EXTERNAL TABLE domains (
        id INT,
        account_id INT,
        domain_name STRING,
        created STRING,
        modified STRING,
        link_metric_id INT,
        config_domain_id INT
)
STORED BY `org.apache.hadoop.hive.jdbc.storagehandler.JDBCStorageHandler`
TBLPROPERTIES (
        “mapred.jdbc.driver.class” = “com.mysql.jdbc.Driver”,
        “mapred.jdbc.url” = “jdbc:mysql://${jdbc.reporting.host}/${jdbc.reporting.databaseName}”,
“mapred.jdbc.input.table.name” = “domains”,
        “mapred.jdbc.username” =  “${jdbc.reporting.username}”,
        “mapred.jdbc.password” = “${jdbc.reporting.password}”,
        “hive.jdbc.update.on.duplicate” = “true”
);
```

The keyword EXTERNAL tells Hive that the table should not be managed by Hive. Dropping the table in Hive will not effect the underlying table. If the underlying table cannot be found Hive will not create it.

If you do not include the EXTERNAL keyword then the storage handler will attempt to execute a CREATE TABLE statement against the underlying database. Likewise, when you drop the table in Hive, the storage handler will attempt to drop the table in the underlying database.

The keyword STORED BY tells Hive to delegate to the JDBCStorageHandler for access to the actual data. This class needs to be available on Hive's classpath at runtime. The table properties that are supported are described in the table below:

Property Name	Required	Details
mapred.jdbc.driver.class	Yes	The classname for the JDBC Driver to use. This must be available on Hive's classpath.
mapred.jdbc.url	Yes	The connection url that will be used by the JDBC driver.
mapred.jdbc.input.table.name	Yes	The name of the table in the database. It does not have to be the same as the name of the table in Hive.
mapred.jdbc.username	No	The DB username, if it's required.
mapred.jdbc.password	No	The DB Password. To avoid having this stored in Hive's MetaStore it can also be injected into the Hadoop Job Configuration at runtime.
mapred.jdbc.input.field.name	No	By default the handler assumes the column names and number in Hive match the underlying table. Setting this allows the column names to be different. It also allows the Hive table to provide limited visibility onto a wider table.
hive.jdbc.vendor.bridge.class	No	The name of a class implementing HiveJDBCVendorBridge. The storage handler will delegate to this class to provide all database vendor specific functionality. If it is not specified the storage handler defaults to MySql and uses HiveMySqlVendorBridge.
hive.jdbc.update.on.duplicate	No	Expected values are either "true" or "false". If "true" then the storage handler will call HiveJDBCVendorBridge.getUpsertStatement() for an appropriate SQL statement

hive.jdbc.delete.on.insert	No	Expected values are either "true" or "false". This turns the table into a deletion target. Any INSERT statements against a table when this is true will actually generate DELETE sql statements. The storage handler will delegate to HiveJDBCVendorBridge.getDeleteStatement() to generate vendor appropriate SQL.

hive.jdbc.update.on.duplicate and hive.jdbc.delete.on.insert cannot both be true. Setting both these values to true will raise an IllegalStateException.

Select Statements & Predicate Pushdown

Once the table is defined it can be used like any other Hive table. E.g.

```
SELECT * FROM domains WHERE id = 23;
```

The storage handler provides support for predicate pushdown. This means that any WHERE clause expression that can be evaluated by the underlying database will be handled there. In many cases this will materially decrease the number of records that ever get emitted by a RecordReader.

Handling Vendor Specific Behavior

The storage handler expects all vendor specific behavior to be handled by implementations of the HiveJDBCVendorBridge interface, which is reproduced below:

```
import java.sql.PreparedStatement;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

import org.apache.hadoop.mapreduce.lib.db.DBConfiguration;

/**
 * HiveJDBCVendorBridge encapsulates the vendor specific differences between database products.
 * <P>
 * Implementations are responsible for generating SQL template strings that can be used in
 * PreparedStatements to write records back to the database. Implementations are also responsible
 * for providing concrete implementations of {@link HiveJDBCTypeBridge}. This class handles coercing
 * values between Hive types and JDBC types.
 *
 */
public interface HiveJDBCVendorBridge {

    /**
    * Return an instance of ResultSetMetaData that the storage handler can use to determine
    * the column names and types for the underlying database table.
    * <p>
    * The value of {@link ResultSetMetaData#getColumnCount()} must match the number of columns
    * on the Hive table.
    *
    * @param conf a DBConfiguration instance configured with the mapred.jdbc.* values from
    * the TBLPROPERTIES
    */
    public ResultSetMetaData getResultSetMetaData(DBConfiguration conf);

    /**
     * Return an implementation of {@link HiveJDBCTypeBridge} that will coerce values between
     * the types defined in the Hive CREATE TABLE statement and the types returned by calls to
     * {@link ResultSetMetaData#getColumnClassName(int)} on the ResultSetMetaData instance returned
     * by {@link HiveJDBCVendorBridge#getResultSetMetaData(DBConfiguration)}.
     *
     */
    public HiveJDBCTypeBridge getTypeBridge();

    /**
```

```
     * Whether the table backed by this vendor bridge can be used as the target of INSERT statements.
     */
    public boolean supportsInsert();

    /**
     * Provide a PreparedStatement that can be used to INSERT records into the database.
     * <p>
     * The storage handler will call {@link PreparedStatement#setObject(int, Object)} once
     * for each column described by the object returned by
     * {@link HiveJDBCVendorBridge#getResultSetMetaData(DBConfiguration)}.
     *
     * @param conf A DBConfiguration instance configured with the mapred.jdbc.* values from
     * the TBLPROPERTIES
     * @throws SQLException
     */
    public PreparedStatement getInsertStatement(DBConfiguration conf) throws SQLException;

    /**
     * Whether the table backed by this vendor bridge can be used as the target of UPSERT statements.
     */
    public boolean supportsUpsert();

    /**
     * Provide a PreparedStatement that can be used to UPSERT records into the database.
     * <p>
     * The storage handler will call {@link PreparedStatement#setObject(int, Object)} once
     * for each column described by the object returned by
     * {@link HiveJDBCVendorBridge#getResultSetMetaData(DBConfiguration)}.
     *
     * @param conf A DBConfiguration instance configured with the mapred.jdbc.* values from
     * the TBLPROPERTIES
     * @throws SQLException
     */
    public PreparedStatement getUpsertStatement(DBConfiguration conf) throws SQLException;

    /**
     * Whether the table backed by this vendor bridge can be used as the target of DELETE statements.
     */
    public boolean supportsDelete();

    /**
     * Provide a PreparedStatement that can be used to DELETE records in the database.
     * <p>
     * The storage handler will call {@link PreparedStatement#setObject(int, Object)} once
     * for each column described by the object returned by
     * {@link HiveJDBCVendorBridge#getResultSetMetaData(DBConfiguration)}.
     *
     * @param conf A DBConfiguration instance configured with the mapred.jdbc.* values from
     * the TBLPROPERTIES
     * @throws SQLException
     */
    public PreparedStatement getDeleteStatement(DBConfiguration conf) throws SQLException;
```

```
}
```

Implementors of this interface have three responsibilities:

1. Provide an instance of JDBC's ResultSetMetaData that can be used to describe the underlying database table.
2. Provide vendor appropriate PrepareStatements for INSERT, UPSERT, and DELETE operations.
3. Provide an appropriate implementation of HiveJDBCTypeBridge to handle type conversion.

Hive's type system is not as rich as most database type systems. During query execution the storage handler will attempt to coerce each JDBC type into the target type that is declared in the Hive table definition. The actual type coercion operation is delegated to a concrete implementation of the HiveJDBCTypeBridge interface that is provided by the HiveJDBCVendorBridge. The HiveJDBCTypeBridge is reproduced below:

```
/**
 * HiveJDBCTypeBridge provides coercion from JDBC types to Hive types.
 * <p>
 * The storage handler expects a database specific implementation to be
 * provided by the {@link HiveJDBCVendorBridge}.
 */
public interface HiveJDBCTypeBridge {

 /**
  * Coerces a sqlValue returned from a ResultSet to the type expected by Hive.
  *
  * @param sqlValue an Object returned from {@link ResultSet#getObject}
  * @param sqlType the Class object returned from {@link ResultSetMetaData#getColumnClassName}
  * @param hiveType the Class object returned from
        * {@link PrimitiveObjectInspector#getJavaPrimitiveClass} for the target Hive column
  *
  * @return the sqlValue as an instance of hiveType
  */
    public <F, T> T toHiveType(Object sqlValue, Class<F> sqlType, Class<T> hiveType);

 /**
  * Coerces a hiveValue to the type expected by the PreparedStatement that will
        * write it to the database.
  *
  * @param hiveValue an Object that is part of a hive row
  * @param sqlType the Class object returned from {@link ResultSetMetaData#getColumnClassName}
  * @param hiveType the Class object returned from
        * {@link PrimitiveObjectInspector#getJavaPrimitiveClass}
  *
  * @return the hiveValue as an instance of sqlType
  */
    public <F, T> T toSqlType(Object hiveValue, Class<T> sqlType, Class<F> hiveType);

}
```

Insert / Update Statements

An example Hive insert statement is included below.

```
INSERT OVERWRITE TABLE domains
   SELECT   id
           ,cast(null as string) AS account_id
           ,domain_name
           ,now() AS created
           ,now() AS modified
      FROM discovered_domains;
```

The storage handler will split up the workload and write to the underlying database in parallel. The DBRecordWriters used by each TaskRunner will batch up the statements for efficiency.

Similarly to select statements the storage handler will attempt to coerce the Hive data types back into JDBC types. These coercions are the inverse functions of those used during querying. This coercion is also handled by implementations of the HiveJDBCTypeBridge.

Each TaskRunner only does one commit at the end. If a TaskRunners fails midway it can be safely tried again. However, there is no global coordination between TaskRunners so it is possible that even in situations where the job fails or is killed globally some of the TaskRunners will have already committed.

Upsert Behavior With MySql

This section only relates to using the JDBCStorageHandler with Mysql. The MySql vendor bridge is the default vendor bridge that will be used when an alternative is not provided.

If the table property "hive.jdbc.update.on.duplicate" is set to "true" then the storage handler will use MySql's INSERT ... ON DUPLICATE KEY UPDATE ... syntax.

This means that one can INSERT mixed record sets into tables backed by the JDBCStorageHandler. MySql will handle each record according to the following rules:

- If the record's primary key column is null then MySql will assign a new id to it during insertion. The column will be populated the next time the table is queried so that it can be used as a foreign key column in downstream Hive insert statements to other tables.
- If the record's primary key column matches an existing primary key (or the record's values conflict with a unique index on the table), then MySql will update the existing record in place.

Deleting Records

Hive does not provide native syntax for delete statements yet. To work around this, the JDBCStorageHandler allows you to define a table that acts as a deletion target. That is, any hive insert statements actually generate SQL delete statements.

To create a deletion target table you must set the table property "hive.jdbc.delete.on.insert". When this value is set to true the columns defined for the Hive table will be used as the predicate in the deletion clause. For example,

An Example Deletion Target for Urls

```
CREATE EXTERNAL TABLE delete_urls (
        domain STRING,
        url STRING
)
STORED BY `org.apache.hadoop.hive.jdbc.storagehandler.JDBCStorageHandler`
TBLPROPERTIES (
        “mapred.jdbc.driver.class” = “com.mysql.jdbc.Driver”,
        “mapred.jdbc.url” = “jdbc:mysql://${jdbc.reporting.host}/${jdbc.reporting.databaseName}”,
“mapred.jdbc.input.table.name” = “urls”,
        “mapred.jdbc.username” =  “${jdbc.reporting.username}”,
        “mapred.jdbc.password” = “${jdbc.reporting.password}”,
        “hive.jdbc.delete.on.insert” = “true”
);
```

Imagine there was a table in Hive called stale_urls, which represented all the urls that we wanted to delete from our relational database. To have Hive delete them, we would execute the following statement:

```
INSERT OVERWRITE delete_urls SELECT domain, url FROM stale_urls;
```

If we were using the HiveMysqlVendorBridge then the call to getDeleteStatement() would return a statement backed by the following SQL template:

```
DELETE FROM urls WHERE domain = ? AND url = ?;
```

Delete operations behave similarly to inserts. Each TaskRunner will batch up the statements and execute a single commit, however there will be no global coordination between TaskRunners.