HBaseIntegration

Hive HBase Integration

- Hive HBase Integration
    - Avro Data Stored in HBase Columns
  - Introduction
  - Storage Handlers
  - Usage
  - Column Mapping
    - Multiple Columns and Families
    - Hive MAP to HBase Column Family
    - Hive MAP to HBase Column Prefix
      - Hiding Column Prefixes
    - Illegal: Hive Primitive to HBase Column Family
    - Example with Binary Columns
    - Simple Composite Row Keys
    - Complex Composite Row Keys and HBaseKeyFactory
    - Avro Data Stored in HBase Columns
  - Put Timestamps
  - Key Uniqueness
  - Overwrite
  - Potential Followups
  - Build
  - Tests
  - Links
  - Acknowledgements
  - Open Issues (JIRA)

Version information

Avro Data Stored in HBase Columns

As of Hive 0.9.0 the HBase integration requires at least HBase 0.92, earlier versions of Hive were working with HBase 0.89/0.90

Version information

Hive 1.x will remain compatible with HBase 0.98.x and lower versions. Hive 2.x will be compatible with HBase 1.x and higher. (See HIVE-10990 for details.) Consumers wanting to work with HBase 1.x using Hive 1.x will need to compile Hive 1.x stream code themselves.

Introduction

This page documents the Hive/HBase integration support originally introduced in HIVE-705. This feature allows Hive QL statements to access HBase tables for both read (SELECT) and write (INSERT). It is even possible to combine access to HBase tables with native Hive tables via joins and unions.

A presentation is available from the HBase HUG10 Meetup

This feature is a work in progress, and suggestions for its improvement are very welcome.

Storage Handlers

Before proceeding, please read StorageHandlers for an overview of the generic storage handler framework on which HBase integration depends.

Usage

The storage handler is built as an independent module, `hive-hbase-handler-x.y.z.jar`, which must be available on the Hive client auxpath, along with HBase, Guava and ZooKeeper jars. It also requires the correct configuration property to be set in order to connect to the right HBase master. See the HBase documentation for how to set up an HBase cluster.

Here's an example using CLI from a source build environment, targeting a single-node HBase server. (Note that the jar locations and names have changed in Hive 0.9.0, so for earlier releases, some changes are needed.)

```
$HIVE_SRC/build/dist/bin/hive --auxpath $HIVE_SRC/build/dist/lib/hive-hbase-handler-0.9.0.jar,$HIVE_SRC/build
/dist/lib/hbase-0.92.0.jar,$HIVE_SRC/build/dist/lib/zookeeper-3.3.4.jar,$HIVE_SRC/build/dist/lib/guava-r09.jar
--hiveconf hbase.master=hbase.yoyodyne.com:60000
```

Here's an example which instead targets a distributed HBase cluster where a quorum of 3 zookeepers is used to elect the HBase master:

```
$HIVE_SRC/build/dist/bin/hive --auxpath $HIVE_SRC/build/dist/lib/hive-hbase-handler-0.9.0.jar,$HIVE_SRC/build
/dist/lib/hbase-0.92.0.jar,$HIVE_SRC/build/dist/lib/zookeeper-3.3.4.jar,$HIVE_SRC/build/dist/lib/guava-r09.jar
--hiveconf hbase.zookeeper.quorum=zk1.yoyodyne.com,zk2.yoyodyne.com,zk3.yoyodyne.com
```

The handler requires Hadoop 0.20 or higher, and has only been tested with dependency versions hadoop-0.20.x, hbase-0.92.0 and zookeeper-3.3.4. If you are not using hbase-0.92.0, you will need to rebuild the handler with the HBase jar matching your version, and change the `--auxpath` above accordingly. Failure to use matching versions will lead to misleading connection failures such as MasterNotRunningException since the HBase RPC protocol changes often.

In order to create a new HBase table which is to be managed by Hive, use the `STORED BY` clause on `CREATE TABLE`:

```
CREATE TABLE hbase_table_1(key int, value string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,cf1:val")
TBLPROPERTIES ("hbase.table.name" = "xyz", "hbase.mapred.output.outputtable" = "xyz");
```

The `hbase.columns.mapping` property is required and will be explained in the next section. The `hbase.table.name` property is optional; it controls the name of the table as known by HBase, and allows the Hive table to have a different name. In this example, the table is known as `hbase_table_1` within Hive, and as `xyz` within HBase. If not specified, then the Hive and HBase table names will be identical. The `hbase.mapred.output.outputtable` property is optional; it's needed if you plan to insert data to the table (the property is used by `hbase.mapreduce.TableOutputFormat`)

After executing the command above, you should be able to see the new (empty) table in the HBase shell:

```
$ hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Version: 0.20.3, r902334, Mon Jan 25 13:13:08 PST 2010
hbase(main):001:0> list
xyz
1 row(s) in 0.0530 seconds
hbase(main):002:0> describe "xyz"
DESCRIPTION                                                          ENABLED
 {NAME => 'xyz', FAMILIES => [{NAME => 'cf1', COMPRESSION => 'NONE', VE true
 RSIONS => '3', TTL => '2147483647', BLOCKSIZE => '65536', IN_MEMORY =>
  'false', BLOCKCACHE => 'true'}]}
1 row(s) in 0.0220 seconds
hbase(main):003:0> scan "xyz"
ROW                          COLUMN+CELL
0 row(s) in 0.0060 seconds
```

Notice that even though a column name "val" is specified in the mapping, only the column family name "cf1" appears in the DESCRIBE output in the HBase shell. This is because in HBase, only column families (not columns) are known in the table-level metadata; column names within a column family are only present at the per-row level.

Here's how to move data from Hive into the HBase table (see GettingStarted for how to create the example table `pokes` in Hive first):

```
INSERT OVERWRITE TABLE hbase_table_1 SELECT * FROM pokes WHERE foo=98;
```

Use HBase shell to verify that the data actually got loaded:

```
hbase(main):009:0> scan "xyz"
ROW                          COLUMN+CELL
 98                          column=cf1:val, timestamp=1267737987733, value=val_98
1 row(s) in 0.0110 seconds
```

And then query it back via Hive:

```
hive> select * from hbase_table_1;
Total MapReduce jobs = 1
Launching Job 1 out of 1
...
OK
98        val_98
Time taken: 4.582 seconds
```

Inserting large amounts of data may be slow due to WAL overhead; if you would like to disable this, make sure you have HIVE-1383 (as of Hive 0.6), and then issue this command before the INSERT:

```
set hive.hbase.wal.enabled=false;
```

Warning: disabling WAL may lead to data loss if an HBase failure occurs, so only use this if you have some other recovery strategy available.

If you want to give Hive access to an existing HBase table, use CREATE EXTERNAL TABLE:

```
CREATE EXTERNAL TABLE hbase_table_2(key int, value string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" = "cf1:val")
TBLPROPERTIES("hbase.table.name" = "some_existing_table", "hbase.mapred.output.outputtable" =
"some_existing_table");
```

Again, `hbase.columns.mapping` is required (and will be validated against the existing HBase table's column families), whereas `hbase.table.name` is optional. The `hbase.mapred.output.outputtable` is optional.

Column Mapping

There are two `SERDEPROPERTIES` that control the mapping of HBase columns to Hive:

- `hbase.columns.mapping`
- `hbase.table.default.storage.type`: Can have a value of either `string` (the default) or `binary`, this option is only available as of Hive 0.9 and the `string` behavior is the only one available in earlier versions

The column mapping support currently available is somewhat cumbersome and restrictive:

- for each Hive column, the table creator must specify a corresponding entry in the comma-delimited `hbase.columns.mapping` string (so for a Hive table with *n* columns, the string should have *n* entries); whitespace should **not** be used in between entries since these will be interperted as part of the column name, which is almost certainly not what you want
- a mapping entry must be either `:key`, `:timestamp` or of the form `column-family-name:[column-name][#(binary|string)` (the type specification that delimited by # was added in Hive 0.9.0, earlier versions interpreted everything as strings)
    - If no type specification is given the value from `hbase.table.default.storage.type` will be used
    - Any prefixes of the valid values are valid too (i.e. #b instead of #`binary`)
    - If you specify a column as `binary` the bytes in the corresponding HBase cells are expected to be of the form that HBase's `Bytes` class yields.
- there must be exactly one `:key` mapping (this can be mapped either to a string or struct column–see Simple Composite Keys and Complex Composite Keys)
- (note that before HIVE-1228 in Hive 0.6, `:key` was not supported, and the first Hive column implicitly mapped to the key; as of Hive 0.6, it is now strongly recommended that you always specify the key explictly; we will drop support for implicit key mapping in the future)
- if no column-name is given, then the Hive column will map to all columns in the corresponding HBase column family, and the Hive MAP datatype must be used to allow access to these (possibly sparse) columns
- Since HBase 1.1 (HBASE-2828) there is a way to access the HBase timestamp attribute using the special `:timestamp` mapping. It needs to be either `bigint` or `timestamp`.
- it is not necessary to reference every HBase column family, but those that are not mapped will be inaccessible via the Hive table; it's possible to map multiple Hive tables to the same HBase table

The next few sections provide detailed examples of the kinds of column mappings currently possible.

Multiple Columns and Families

Here's an example with three Hive columns and two HBase column families, with two of the Hive columns (`value1` and `value2`) corresponding to one of the column families (a, with HBase column names b and c), and the other Hive column corresponding to a single column (e) in its own column family (d).

```
CREATE TABLE hbase_table_1(key int, value1 string, value2 int, value3 int)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
"hbase.columns.mapping" = ":key,a:b,a:c,d:e"
);
INSERT OVERWRITE TABLE hbase_table_1 SELECT foo, bar, foo+1, foo+2
FROM pokes WHERE foo=98 OR foo=100;
```

Here's how this looks in HBase:

```
hbase(main):014:0> describe "hbase_table_1"
DESCRIPTION                                                          ENABLED
 {NAME => 'hbase_table_1', FAMILIES => [{NAME => 'a', COMPRESSION => 'N true
 ONE', VERSIONS => '3', TTL => '2147483647', BLOCKSIZE => '65536', IN_M
 EMORY => 'false', BLOCKCACHE => 'true'}, {NAME => 'd', COMPRESSION =>
 'NONE', VERSIONS => '3', TTL => '2147483647', BLOCKSIZE => '65536', IN
 _MEMORY => 'false', BLOCKCACHE => 'true'}]}
1 row(s) in 0.0170 seconds
hbase(main):015:0> scan "hbase_table_1"
ROW                          COLUMN+CELL
 100                         column=a:b, timestamp=1267740457648, value=val_100
 100                         column=a:c, timestamp=1267740457648, value=101
 100                         column=d:e, timestamp=1267740457648, value=102
 98                          column=a:b, timestamp=1267740457648, value=val_98
 98                          column=a:c, timestamp=1267740457648, value=99
 98                          column=d:e, timestamp=1267740457648, value=100
2 row(s) in 0.0240 seconds
```

And when queried back into Hive:

```
hive> select * from hbase_table_1;
Total MapReduce jobs = 1
Launching Job 1 out of 1
...
OK
100        val_100        101        102
98        val_98        99        100
Time taken: 4.054 seconds
```

Hive MAP to HBase Column Family

Here's how a Hive MAP datatype can be used to access an entire column family. Each row can have a different set of columns, where the column names correspond to the map keys and the column values correspond to the map values.

```
CREATE TABLE hbase_table_1(value map<string,int>, row_key int)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
"hbase.columns.mapping" = "cf:,:key"
);
INSERT OVERWRITE TABLE hbase_table_1 SELECT map(bar, foo), foo FROM pokes
WHERE foo=98 OR foo=100;
```

(This example also demonstrates using a Hive column other than the first as the HBase row key.)

Here's how this looks in HBase (with different column names in different rows):

```
hbase(main):012:0> scan "hbase_table_1"
ROW                          COLUMN+CELL
 100                         column=cf:val_100, timestamp=1267739509194, value=100
 98                          column=cf:val_98, timestamp=1267739509194, value=98
2 row(s) in 0.0080 seconds
```

And when queried back into Hive:

```
hive> select * from hbase_table_1;
Total MapReduce jobs = 1
Launching Job 1 out of 1
...
OK
{"val_100":100}        100
{"val_98":98}        98
Time taken: 3.808 seconds
```

Note that the key of the MAP must have datatype string, since it is used for naming the HBase column, so the following table definition will fail:

```
CREATE TABLE hbase_table_1(key int, value map<int,int>)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
"hbase.columns.mapping" = ":key,cf:"
);
FAILED: Error in metadata: java.lang.RuntimeException: MetaException(message:org.apache.hadoop.hive.serde2.
SerDeException org.apache.hadoop.hive.hbase.HBaseSerDe: hbase column family 'cf:' should be mapped to
map<string,?> but is mapped to map<int,int>)
```

Hive MAP to HBase Column Prefix

Also note that starting with Hive 0.12, wildcards can also be used to retrieve columns. For instance, if you want to retrieve all columns in HBase that start with the prefix "col_prefix", a query like the following should work:

```
CREATE TABLE hbase_table_1(value map<string,int>, row_key int)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
"hbase.columns.mapping" = "cf:col_prefix.*,:key"
);
```

The same restrictions apply though. That is, the key of the map should be a string as it maps to the HBase column name and the value can be the type of value that you are retrieving. One other restriction is that all the values under the given prefix should be of the same type. That is, all of them should be of type "int" or type "string" and so on.

Hiding Column Prefixes

Starting with Hive 1.3.0, it is possible to hide column prefixes in select query results. There is the SerDe boolean property hbase.columns.mapping.prefix.hide (false by default), which defines if the prefix should be hidden in keys of Hive map:

```
CREATE TABLE hbase_table_1(tags map<string,int>, row_key string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
"hbase.columns.mapping" = "cf:tag_.*,:key",
"hbase.columns.mapping.prefix.hide" = "true"
);
```

Then a value of the column "tags" (`select tags from hbase_table_1`) will be:

```
"x" : 1
```

instead of:

```
"tag_x" : 1
```

Illegal: Hive Primitive to HBase Column Family

Table definitions such as the following are illegal because a
Hive column mapped to an entire column family must have MAP type:

```
CREATE TABLE hbase_table_1(key int, value string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
"hbase.columns.mapping" = ":key,cf:"
);
FAILED: Error in metadata: java.lang.RuntimeException: MetaException(message:org.apache.hadoop.hive.serde2.
SerDeException org.apache.hadoop.hive.hbase.HBaseSerDe: hbase column family 'cf:' should be mapped to
map<string,?> but is mapped to string)
```

Example with Binary Columns

Relying on the default value of `hbase.table.default.storage.type`:

```
CREATE TABLE hbase_table_1 (key int, value string, foobar double)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
"hbase.columns.mapping" = ":key#b,cf:val,cf:foo#b"
);
```

Specifying `hbase.table.default.storage.type`:

```
CREATE TABLE hbase_table_1 (key int, value string, foobar double)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
"hbase.columns.mapping" = ":key,cf:val#s,cf:foo",
"hbase.table.default.storage.type" = "binary"
);
```

Simple Composite Row Keys

Version information

As of Hive 0.13.0

Hive can read and write delimited composite keys to HBase by mapping the HBase row key to a Hive struct, and using ROW FORMAT DELIMITED... COLLECTION ITEMS TERMINATED BY. Example:

```
-- Create a table with a composite row key consisting of two string fields, delimited by '~'
CREATE EXTERNAL TABLE delimited_example(key struct<f1:string, f2:string>, value string)
ROW FORMAT DELIMITED
COLLECTION ITEMS TERMINATED BY '~'
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
  'hbase.columns.mapping'=':key,f:c1');
```

Complex Composite Row Keys and HBaseKeyFactory

As of Hive 0.14.0 with HIVE-6411 (0.13.0 also supports complex composite keys, but using a different interface–see HIVE-2599 for that interface)

For more complex use cases, Hive allows users to specify an HBaseKeyFactory which defines the mapping of a key to fields in a Hive struct. This can be configured using the property "hbase.composite.key.factory" in the SERDEPROPERTIES option:

```
-- Parse a row key with 3 fixed width fields each of width 10
-- Example taken from: https://svn.apache.org/repos/asf/hive/trunk/hbase-handler/src/test/queries/positive
/hbase_custom_key2.q
CREATE TABLE hbase_ck_4(key struct<col1:string,col2:string,col3:string>, value string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
    "hbase.table.name" = "hbase_custom2",
    "hbase.mapred.output.outputtable" = "hbase_custom2",
    "hbase.columns.mapping" = ":key,cf:string",
    "hbase.composite.key.factory"="org.apache.hadoop.hive.hbase.SampleHBaseKeyFactory2");
```

"hbase.composite.key.factory" should be the fully qualified class name of a class implementing HBaseKeyFactory. See SampleHBaseKeyFactory2 for a fixed length example in the same package. This class must be on your classpath in order for the above example to work. TODO: place these in an accessible place; they're currently only in test code.

Avro Data Stored in HBase Columns

As of Hive 0.14.0 with HIVE-6147

Hive 0.14.0 onward supports storing and querying Avro objects in HBase columns by making them visible as structs to Hive. This allows Hive to perform ad hoc analysis of HBase data which can be deeply structured. Prior to 0.14.0, the HBase Hive integration only supported querying primitive data types in columns.

An example HiveQL statement where `test_col_fam` is the column family and `test_col` is the column name:

```
CREATE EXTERNAL TABLE test_hbase_avro
ROW FORMAT SERDE 'org.apache.hadoop.hive.hbase.HBaseSerDe'
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
        "hbase.columns.mapping" = ":key,test_col_fam:test_col",
        "test_col_fam.test_col.serialization.type" = "avro",
        "test_col_fam.test_col.avro.schema.url" = "hdfs://testcluster/tmp/schema.avsc")
TBLPROPERTIES (
    "hbase.table.name" = "hbase_avro_table",
    "hbase.mapred.output.outputtable" = "hbase_avro_table",
    "hbase.struct.autogenerate"="true");
```

The important properties to note are the following three:

```
"test_col_fam.test_col.serialization.type" = "avro"
```

This property tells Hive that the given column under the given column family is an Avro column, so Hive needs to deserialize it accordingly.

```
"test_col_fam.test_col.avro.schema.url" = "hdfs://testcluster/tmp/schema.avsc"
```

Using this property you specify where the reader schema is for the column that will be used to deserialize. This can be on HDFS like mentioned here, or provided inline using something like `"test_col_fam.test_col.avro.schema.literal"` property. If you have a custom store where you store this schema, you can write a custom implementation of AvroSchemaRetriever and plug that in using the `"avro.schema.retriever property"` using a property like `"test_col_fam.test_col.avro.schema.retriever"`. You would need to ensure that the jar with this custom class is on the Hive classpath. For a usage discussion and links to other resources, see HIVE-6147.

```
"hbase.struct.autogenerate" = "true"
```

Specifying this property lets Hive auto-deduce the columns and types using the schema that was provided. This allows you to avoid manually creating the columns and types for Avro schemas, which can be complicated and deeply nested.

Put Timestamps

Version information

As of Hive 0.9.0

If inserting into a HBase table using Hive the HBase default timestamp is added which is usually the current timestamp. This can be overridden on a per-table basis using the `SERDEPROPERTIES` option `hbase.put.timestamp` which must be a valid timestamp or `-1` to reenable the default strategy.

Key Uniqueness

One subtle difference between HBase tables and Hive tables is that HBase tables have a unique key, whereas Hive tables do not. When multiple rows with the same key are inserted into HBase, only one of them is stored (the choice is arbitrary, so do not rely on HBase to pick the right one). This is in contrast to Hive, which is happy to store multiple rows with the same key and different values.

For example, the pokes table contains rows with duplicate keys. If it is copied into another Hive table, the duplicates are preserved:

```
CREATE TABLE pokes2(foo INT, bar STRING);
INSERT OVERWRITE TABLE pokes2 SELECT * FROM pokes;
-- this will return 3
SELECT COUNT(1) FROM POKES WHERE foo=498;
-- this will also return 3
SELECT COUNT(1) FROM pokes2 WHERE foo=498;
```

But in HBase, the duplicates are silently eliminated:

```
CREATE TABLE pokes3(foo INT, bar STRING)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
"hbase.columns.mapping" = ":key,cf:bar"
);
INSERT OVERWRITE TABLE pokes3 SELECT * FROM pokes;
-- this will return 1 instead of 3
SELECT COUNT(1) FROM pokes3 WHERE foo=498;
```

Overwrite

Another difference to note between HBase tables and other Hive tables is that when INSERT OVERWRITE is used, existing rows are not deleted from the table. However, existing rows **are** overwritten if they have keys which match new rows.

Potential Followups

There are a number of areas where Hive/HBase integration could definitely use more love:

- more flexible column mapping (HIVE-806, HIVE-1245)
- default column mapping in cases where no mapping spec is given
- filter pushdown and indexing (see FilterPushdownDev and IndexDev)
- expose timestamp attribute, possibly also with support for treating it as a partition key
- allow per-table hbase.master configuration
- run profiler and minimize any per-row overhead in column mapping
- user defined routines for lookups and data loads via HBase client API (HIVE-758 and HIVE-791)
- logging is very noisy, with a lot of spurious exceptions; investigate these and either fix their cause or squelch them

Build

Code for the storage handler is located under `hive/trunk/hbase-handler`.

HBase and Zookeeper dependencies are fetched via ivy.

Tests

Class-level unit tests are provided under `hbase-handler/src/test/org/apache/hadoop/hive/hbase`.

Positive QL tests are under `hbase-handler/src/test/queries`. These use a HBase+Zookeeper mini-cluster for hosting the fixture tables in-process, so no free-standing HBase installation is needed in order to run them. To avoid failures due to port conflicts, don't try to run these tests on the same machine where a real HBase master or zookeeper is running.

The QL tests can be executed via ant like this:

```
ant test -Dtestcase=TestHBaseCliDriver -Dqfile=hbase_queries.q
```

An Eclipse launch template remains to be defined.

Links

- For information on how to bulk load data from Hive into HBase, see HBaseBulkLoad.
- For another project which adds SQL-like query language support on top of HBase, see HBQL (unrelated to Hive).

Acknowledgements

- Primary credit for this feature goes to Samuel Guo, who did most of the development work in the early drafts of the patch

Open Issues (JIRA)

type	key	summary	assignee	reporter	priority	status	resolution	created	updated	due

Can't show details. Ask your admin to add this Jira URL to the allowlist.

View these issues in Jira