DeveloperGuide

Developer Guide

- Developer Guide
  - Code Organization and a Brief Architecture
    - Introduction
    - Hive SerDe
      - How to Write Your Own SerDe
      - ObjectInspector
      - Registration of Native SerDes
    - MetaStore
    - Query Processor
      - Compiler
      - Parser
      - TypeChecking
      - Semantic Analysis
      - Plan generation
      - Task generation
      - Execution Engine
      - Plan
      - Operators
      - UDFs and UDAFs
  - Compiling and Running Hive
    - Default Mode
    - Advanced Mode
    - Running Hive Without a Hadoop Cluster
  - Unit tests and debugging
    - Layout of the unit tests
    - Running unit tests
    - Adding new unit tests
    - Debugging Hive Code
      - Debugging Client-Side Code
      - Debugging Server-Side Code
      - Debugging without Ant (Client and Server Side)
  - Pluggable interfaces
    - File Formats
    - SerDe - how to add a new SerDe
    - Map-Reduce Scripts
    - UDFs and UDAFs - how to add new UDFs and UDAFs

Code Organization and a Brief Architecture

Introduction

Hive has 3 main components:

- Serializers/Deserializers (trunk/serde) - This component has the framework libraries that allow users to develop serializers and deserializers for their own data formats. This component also contains some builtin serialization/deserialization families.
- MetaStore (trunk/metastore) - This component implements the metadata server, which is used to hold all the information about the tables and partitions that are in the warehouse.
- Query Processor (trunk/ql) - This component implements the processing framework for converting SQL to a graph of map/reduce jobs and the execution time framework to run those jobs in the order of dependencies.

Apart from these major components, Hive also contains a number of other components. These are as follows:

- Command Line Interface (trunk/cli) - This component has all the java code used by the Hive command line interface.
- Hive Server (trunk/service) - This component implements all the APIs that can be used by other clients (such as JDBC drivers) to talk to Hive.
- Common (trunk/common) - This component contains common infrastructure needed by the rest of the code. Currently, this contains all the java sources for managing and passing Hive configurations(HiveConf) to all the other code components.
- Ant Utilities (trunk/ant) - This component contains the implementation of some ant tasks that are used by the build infrastructure.
- Scripts (trunk/bin) - This component contains all the scripts provided in the distribution including the scripts to run the Hive CLI (bin/hive).

The following top level directories contain helper libraries, packaged configuration files etc..:

- trunk/conf - This directory contains the packaged hive-default.xml and hive-site.xml.
- trunk/data - This directory contains some data sets and configurations used in the Hive tests.
- trunk/ivy - This directory contains the Ivy files used by the build infrastructure to manage dependencies on different Hadoop versions.
- trunk/lib - This directory contains the run time libraries needed by Hive.
- trunk/testlibs - This directory contains the junit.jar used by the JUnit target in the build infrastructure.
- trunk/testutils (Deprecated)

Hive SerDe

What is a SerDe?

- SerDe is a short name for "Serializer and Deserializer."
- Hive uses SerDe (and FileFormat) to read and write table rows.
- HDFS files --> InputFileFormat --> <key, value> --> Deserializer --> Row object
- Row object --> Serializer --> <key, value> --> OutputFileFormat --> HDFS files

Note that the "key" part is ignored when reading, and is always a constant when writing. Basically row object is stored into the "value".

One principle of Hive is that Hive does not own the HDFS file format. Users should be able to directly read the HDFS files in the Hive tables using other tools or use other tools to directly write to HDFS files that can be loaded into Hive through "CREATE EXTERNAL TABLE" or can be loaded into Hive through "LOAD DATA INPATH," which just move the file into Hive's table directory.

Note that org.apache.hadoop.hive.serde is the deprecated old SerDe library. Please look at org.apache.hadoop.hive.serde2 for the latest version.

Hive currently uses these FileFormat classes to read and write HDFS files:

- TextInputFormat/HiveIgnoreKeyTextOutputFormat: These 2 classes read/write data in plain text file format.
- SequenceFileInputFormat/SequenceFileOutputFormat: These 2 classes read/write data in Hadoop SequenceFile format.

Hive currently uses these SerDe classes to serialize and deserialize data:

- MetadataTypedColumnsetSerDe: This SerDe is used to read/write delimited records like CSV, tab-separated control-A separated records (sorry, quote is not supported yet).
- LazySimpleSerDe: This SerDe can be used to read the same data format as MetadataTypedColumnsetSerDe and TCTLSeparatedProtocol, however, it creates Objects in a lazy way which provides better performance. Starting in Hive 0.14.0 it also supports read/write data with a specified encode charset, for example:

  ```
  ALTER TABLE person SET SERDEPROPERTIES ('serialization.encoding'='GBK');
  ```

  LazySimpleSerDe can treat 'T', 't', 'F', 'f', '1', and '0' as extended, legal boolean literals if the configuration property hive.lazysimple.extended_boolean_literal is set to `true` (Hive 0.14.0 and later). The default is `false`, which means only 'TRUE' and 'FALSE' are treated as legal boolean literals.
- ThriftSerDe: This SerDe is used to read/write Thrift serialized objects. The class file for the Thrift object must be loaded first.
- DynamicSerDe: This SerDe also read/write Thrift serialized objects, but it understands Thrift DDL so the schema of the object can be provided at runtime. Also it supports a lot of different protocols, including TBinaryProtocol, TJSONProtocol, TCTLSeparatedProtocol (which writes data in delimited records).

Also:

- For JSON files, JsonSerDe was added in Hive 0.12.0. An Amazon SerDe is available at `s3://elasticmapreduce/samples/hive-ads/libs/jsonserde.jar` for releases prior to 0.12.0.
- An Avro SerDe was added in Hive 0.9.1. Starting in Hive 0.14.0 its specification is implicit with the STORED AS AVRO clause.
- A SerDe for the ORC file format was added in Hive 0.11.0.
- A SerDe for Parquet was added via plug-in in Hive 0.10 and natively in Hive 0.13.0.
- A SerDe for CSV was added in Hive 0.14.

See SerDe for detailed information about input and output processing. Also see Storage Formats in the HCatalog manual, including CTAS Issue with JSON SerDe. For information about how to create a table with a custom or native SerDe, see Row Format, Storage Format, and SerDe.

How to Write Your Own SerDe

- In most cases, users want to write a Deserializer instead of a SerDe, because users just want to read their own data format instead of writing to it.
- For example, the RegexDeserializer will deserialize the data using the configuration parameter 'regex', and possibly a list of column names (see serde2.MetadataTypedColumnsetSerDe). Please see serde2/Deserializer.java for details.
- If your SerDe supports DDL (basically, SerDe with parameterized columns and column types), you probably want to implement a Protocol based on DynamicSerDe, instead of writing a SerDe from scratch. The reason is that the framework passes DDL to SerDe through "Thrift DDL" format, and it's non-trivial to write a "Thrift DDL" parser.
- For examples, see SerDe - how to add a new SerDe below.

Some important points about SerDe:

- SerDe, not the DDL, defines the table schema. Some SerDe implementations use the DDL for configuration, but the SerDe can also override that.
- Column types can be arbitrarily nested arrays, maps, and structures.
- The callback design of ObjectInspector allows lazy deserialization with CASE/IF or when using complex or nested types.

ObjectInspector

Hive uses ObjectInspector to analyze the internal structure of the row object and also the structure of the individual columns.

ObjectInspector provides a uniform way to access complex objects that can be stored in multiple formats in the memory, including:

- Instance of a Java class (Thrift or native Java)
- A standard Java object (we use java.util.List to represent Struct and Array, and use java.util.Map to represent Map)
- A lazily-initialized object (for example, a Struct of string fields stored in a single Java string object with starting offset for each field)

A complex object can be represented by a pair of ObjectInspector and Java Object. The ObjectInspector not only tells us the structure of the Object, but also gives us ways to access the internal fields inside the Object.

NOTE: Apache Hive recommends that custom ObjectInspectors created for use with custom SerDes have a no-argument constructor in addition to their normal constructors for serialization purposes. See ~~HIVE-5380~~ for more details.

Registration of Native SerDes

As of Hive 0.14 a registration mechanism has been introduced for native Hive SerDes. This allows dynamic binding between a "STORED AS" keyword in place of a triplet of {SerDe, InputFormat, and OutputFormat} specification, in CreateTable statements.

The following mappings have been added through this registration mechanism:

Syntax	Equivalent

STORED AS AVRO /
 	
STORED AS AVROFILE
	`ROW FORMAT SERDE`
	`  'org.apache.hadoop.hive.serde2.avro.AvroSerDe'`
	`  STORED AS INPUTFORMAT`
	`  'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'`
	`  OUTPUTFORMAT`
	`  'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'`

STORED AS ORC /
 	
STORED AS ORCFILE
	`ROW FORMAT SERDE`
	`  'org.apache.hadoop.hive.ql.io.orc.OrcSerde'`
	`  STORED AS INPUTFORMAT`
	`  'org.apache.hadoop.hive.ql.io.orc.OrcInputFormat'`
	`  OUTPUTFORMAT`
	`  'org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat'`

STORED AS PARQUET /
 	
STORED AS PARQUETFILE
	`ROW FORMAT SERDE`
	`  'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'`
	`  STORED AS INPUTFORMAT`
	`  'org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat'`
	`  OUTPUTFORMAT`
	`  'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'`

STORED AS RCFILE	`STORED AS INPUTFORMAT`
	`  'org.apache.hadoop.hive.ql.io.RCFileInputFormat'`
	`  OUTPUTFORMAT`
	`  'org.apache.hadoop.hive.ql.io.RCFileOutputFormat'`

STORED AS SEQUENCEFILE	`STORED AS INPUTFORMAT`
	`  'org.apache.hadoop.mapred.SequenceFileInputFormat'`
	`  OUTPUTFORMAT`
	`  'org.apache.hadoop.mapred.SequenceFileOutputFormat'`

STORED AS TEXTFILE	`STORED AS INPUTFORMAT`
	`  'org.apache.hadoop.mapred.TextInputFormat'`
	`  OUTPUTFORMAT`
	`  'org.apache.hadoop.hive.ql.io.IgnoreKeyTextOutputFormat'`

To add a new native SerDe with STORED AS keyword, follow these steps:

1. Create a storage format descriptor class extending from AbstractStorageFormatDescriptor.java that returns a "stored as" keyword and the names of InputFormat, OutputFormat, and SerDe classes.
2. Add the name of the storage format descriptor class to the StorageFormatDescriptor registration file.

MetaStore

MetaStore contains metadata regarding tables, partitions and databases. This is used by Query Processor during plan generation.

- Metastore Server - This is the Thrift server (interface defined in metastore/if/hive_metastore.if) that services metadata requests from clients. It delegates most of the requests underlying meta data store and the Hadoop file system which contains data.
- Object Store - ObjectStore class handles access to the actual metadata is stored in the SQL store. The current implementation uses JPOX ORM solution which is based of JDA specification. It can be used with any database that is supported by JPOX. New meta stores (file based or xml based) can added by implementing the interface MetaStore. FileStore is a partial implementation of an older version of metastore which may be deprecated soon.
- Metastore Client - There are python, java, php Thrift clients in metastore/src. Java generated client is extended with HiveMetaStoreClient which is used by Query Processor (ql/metadta). This is the main interface to all other Hive components.

Query Processor

The following are the main components of the Hive Query Processor:

- Parse and SemanticAnalysis (ql/parse) - This component contains the code for parsing SQL, converting it into Abstract Syntax Trees, converting the Abstract Syntax Trees into Operator Plans and finally converting the operator plans into a directed graph of tasks which are executed by Driver.java.
- Optimizer (ql/optimizer) - This component contains some simple rule based optimizations like pruning non referenced columns from table scans (column pruning) that the Hive Query Processor does while converting SQL to a series of map/reduce tasks.
- Plan Components (ql/plan) - This component contains the classes (which are called descriptors), that are used by the compiler (Parser, SemanticAnalysis and Optimizer) to pass the information to operator trees that is used by the execution code.
- MetaData Layer (ql/metadata) - This component is used by the query processor to interface with the MetaStore in order to retrieve information about tables, partitions and the columns of the table. This information is used by the compiler to compile SQL to a series of map/reduce tasks.
- Map/Reduce Execution Engine (ql/exec) - This component contains all the query operators and the framework that is used to invoke those operators from within the map/reduces tasks.
- Hadoop Record Readers, Input and Output Formatters for Hive (ql/io) - This component contains the record readers and the input, output formatters that Hive registers with a Hadoop Job.
- Sessions (ql/session) - A rudimentary session implementation for Hive.
- Type interfaces (ql/typeinfo) - This component provides all the type information for table columns that is retrieved from the MetaStore and the SerDes.
- Hive Function Framework (ql/udf) - Framework and implementation of Hive operators, Functions and Aggregate Functions. This component also contains the interfaces that a user can implement to create user defined functions.
- Tools (ql/tools) - Some simple tools provided by the query processing framework. Currently, this component contains the implementation of the lineage tool that can parse the query and show the source and destination tables of the query.

Compiler

Parser

TypeChecking

Semantic Analysis

Plan generation

Task generation

Execution Engine

Plan

Operators

UDFs and UDAFs

A helpful overview of the Hive query processor can be found in this Hive Anatomy slide deck.

Compiling and Running Hive

Ant to Maven

As of version 0.13 Hive uses Maven instead of Ant for its build. The following instructions are not up to date.

See the Hive Developer FAQ for updated instructions.

Hive can be made to compile against different versions of Hadoop.

Default Mode

From the root of the source tree:

```
ant package
```

will make Hive compile against Hadoop version 0.19.0. Note that:

- Hive uses Ivy to download the hadoop-0.19.0 distribution. However once downloaded, it's cached and not downloaded multiple times.
- This will create a distribution directory in build/dist (relative to the source root) from where one can launch Hive. This distribution should only be used to execute queries against Hadoop branch 0.19. (Hive is not sensitive to minor revisions of Hadoop versions).

Advanced Mode

- One can specify a custom distribution directory by using:

```
ant -Dtarget.dir=<my-install-dir> package
```

- One can specify a version of Hadoop other than 0.19.0 by using (using 0.17.1 as an example):

```
ant -Dhadoop.version=0.17.1 package
```

- One can also compile against a custom version of the Hadoop tree (only release 0.4 and above). This is also useful if running Ivy is problematic (in disconnected mode for example) - but a Hadoop tree is available. This can be done by specifying the root of the Hadoop source tree to be used, for example:

```
ant -Dhadoop.root=~/src/hadoop-19/build/hadoop-0.19.2-dev -Dhadoop.version=0.19.2-dev
```

note that:

- Hive's build script assumes that `hadoop.root` is pointing to a distribution tree for Hadoop created by running ant package in Hadoop.
- `hadoop.version` must match the version used in building Hadoop.

In this particular example - `~/src/hadoop-19` is a checkout of the Hadoop 19 branch that uses `0.19.2-dev` as default version and creates a distribution directory in `build/hadoop-0.19.2-dev` by default.

Run Hive from the command line with '`$HIVE_HOME/bin/hive`', where `$HIVE_HOME` is typically `build/dist` under your Hive repository top-level directory.

```
$ build/dist/bin/hive
```

If Hive fails at runtime, try '`ant very-clean package`' to delete the Ivy cache before rebuilding.

Running Hive Without a Hadoop Cluster

From Thejas:

```
export HIVE_OPTS='--hiveconf mapred.job.tracker=local --hiveconf fs.default.name=file:///tmp \
    --hiveconf hive.metastore.warehouse.dir=file:///tmp/warehouse \
    --hiveconf javax.jdo.option.ConnectionURL=jdbc:derby:;databaseName=/tmp/metastore_db;create=true'
```

Then you can run '`build/dist/bin/hive`' and it will work against your local file system.

Unit tests and debugging

Layout of the unit tests

Hive uses JUnit for unit tests. Each of the 3 main components of Hive have their unit test implementations in the corresponding src/test directory e.g. trunk /metastore/src/test has all the unit tests for metastore, trunk/serde/src/test has all the unit tests for serde and trunk/ql/src/test has all the unit tests for the query processor. The metastore and serde unit tests provide the TestCase implementations for JUnit. The query processor tests on the other hand are generated using Velocity. The main directories under trunk/ql/src/test that contain these tests and the corresponding results are as follows:

- Test Queries:
  - queries/clientnegative - This directory contains the query files (.q files) for the negative test cases. These are run through the CLI classes and therefore test the entire query processor stack.
  - queries/clientpositive - This directory contains the query files (.q files) for the positive test cases. Thesre are run through the CLI classes and therefore test the entire query processor stack.
  - qureies/positive (Will be deprecated) - This directory contains the query files (.q files) for the positive test cases for the compiler. These only test the compiler and do not run the execution code.
  - queries/negative (Will be deprecated) - This directory contains the query files (.q files) for the negative test cases for the compiler. These only test the compiler and do not run the execution code.
- Test Results:
  - results/clientnegative - The expected results from the queries in queries/clientnegative.
  - results/clientpositive - The expected results from the queries in queries/clientpositive.
  - results/compiler/errors - The expected results from the queries in queries/negative.
  - results/compiler/parse - The expected Abstract Syntax Tree output for the queries in queries/positive.
  - results/compiler/plan - The expected query plans for the queries in queries/positive.
- Velocity Templates to Generate the Tests:
  - templates/TestCliDriver.vm - Generates the tests from queries/clientpositive.
  - templates/TestNegativeCliDriver.vm - Generates the tests from queries/clientnegative.
  - templates/TestParse.vm - Generates the tests from queries/positive.

- templates/TestParseNegative.vm - Generates the tests from queries/negative.

Running unit tests

> **Ant to Maven**
>
> As of version 0.13 Hive uses Maven instead of Ant for its build. The following instructions are not up to date.
>
> See the Hive Developer FAQ for updated instructions.

Run all tests:

```
ant package test
```

Run all positive test queries:

```
ant test -Dtestcase=TestCliDriver
```

Run a specific positive test query:

```
ant test -Dtestcase=TestCliDriver -Dqfile=groupby1.q
```

The above test produces the following files:

- `build/ql/test/TEST-org.apache.hadoop.hive.cli.TestCliDriver.txt` - Log output for the test. This can be helpful when examining test failures.
- `build/ql/test/logs/groupby1.q.out` - Actual query result for the test. This result is compared to the expected result as part of the test.

Run the set of unit tests matching a regex, e.g. partition_wise_fileformat tests 10-16:

```
ant test -Dtestcase=TestCliDriver -Dqfile_regex=partition_wise_fileformat1[0-6]
```

Note that this option matches against the basename of the test without the .q suffix.

Apparently the Hive tests do not run successfully after a clean unless you run `ant package` first. Not sure why build.xml doesn't encode this dependency.

Adding new unit tests

> **Ant to Maven**
>
> As of version 0.13 Hive uses Maven instead of Ant for its build. The following instructions are not up to date.
>
> See the Hive Developer FAQ for updated instructions. See also Tips for Adding New Tests in Hive and How to Contribute: Add a Unit Test.

First, write a new myname.q in ql/src/test/queries/clientpositive.

Then, run the test with the query and overwrite the result (useful when you add a new test).

```
ant test -Dtestcase=TestCliDriver -Dqfile=myname.q -Doverwrite=true
```

Then we can create a patch by:

```
svn add ql/src/test/queries/clientpositive/myname.q ql/src/test/results/clientpositive/myname.q.out
svn diff > patch.txt
```

Similarly, to add negative client tests, write a new query input file in ql/src/test/queries/clientnegative and run the same command, this time specifying the testcase name as TestNegativeCliDriver instead of TestCliDriver. Note that for negative client tests, the output file if created using the overwrite flag can be be found in the directory ql/src/test/results/clientnegative.

Debugging Hive Code

Hive code includes both client-side code (e.g., compiler, semantic analyzer, and optimizer of HiveQL) and server-side code (e.g., operator/task/SerDe implementations). Debugging is different for client-side and server-side code, as described below.

Debugging Client-Side Code

The client-side code runs on your local machine so you can easily debug it using Eclipse the same way you debug any regular local Java code. Here are the steps to debug code within a unit test.

- Make sure that you have run `ant model-jar` in hive/metastore and `ant gen-test` in hive since the last time you ran `ant clean`.
- To run all of the unit tests for the CLI:
  - Open up TestCliDriver.java
  - Click Run->Debug Configurations, select TestCliDriver, and click Debug.
- To run a single test within TestCliDriver.java:
  - Begin running the whole TestCli suite as before.
  - Once it finishes the setup and starts executing the JUnit tests, stop the test execution.
  - Find the desired test in the JUnit pane,
  - Right click on that test and select Debug.

Debugging Server-Side Code

The server-side code is distributed and runs on the Hadoop cluster, so debugging server-side Hive code is a little bit complicated. In addition to printing to log files using log4j, you can also attach the debugger to a different JVM under unit test (single machine mode). Below are the steps on how to debug on server-side code.

- Compile Hive code with javac.debug=on. Under Hive checkout directory:

```
    > ant -Djavac.debug=on package
```

  If you have already built Hive without javac.debug=on, you can clean the build and then run the above command.

```
    > ant clean  # not necessary if the first time to compile
    > ant -Djavac.debug=on package
```

- Run ant test with additional options to tell the Java VM that is running Hive server-side code to wait for the debugger to attach. First define some convenient macros for debugging. You can put it in your .bashrc or .cshrc.

```
    > export HIVE_DEBUG_PORT=8000
    > export HIVE_DEBUG="-Xdebug -Xrunjdwp:transport=dt_socket,address=${HIVE_DEBUG_PORT},server=y,
suspend=y"
```

  In particular HIVE_DEBUG_PORT is the port number that the JVM is listening on and the debugger will attach to. Then run the unit test as follows:

```
    > export HADOOP_OPTS=$HIVE_DEBUG
    > ant test -Dtestcase=TestCliDriver -Dqfile=<mytest>.q
```

  The unit test will run until it shows:

```
     [junit] Listening for transport dt_socket at address: 8000
```

- Now, you can use jdb to attach to port 8000 to debug

```
    > jdb -attach 8000
```

  or if you are running Eclipse and the Hive projects are already imported, you can debug with Eclipse. Under Eclipse Run -> Debug Configurations, find "Remote Java Application" at the bottom of the left panel. There should be a MapRedTask configuration already. If there is no such configuration, you can create one with the following property:

  - Name: any task such as MapRedTask
  - Project: the Hive project that you imported.
  - Connection Type: Standard (Socket Attach)
  - Connection Properties:
    - Host: localhost

- Port: 8000
  Then hit the "Debug" button and Eclipse will attach to the JVM listening on port 8000 and continue running till the end. If you define breakpoints in the source code before hitting the "Debug" button, it will stop there. The rest is the same as debugging client-side Hive.

Debugging without Ant (Client and Server Side)

There is another way of debugging Hive code without going through Ant.
You need to install Hadoop and set the environment variable HADOOP_HOME to that.

```
> export HADOOP_HOME=<your hadoop home>
```

Then, start Hive:

```
>  ./build/dist/bin/hive --debug
```

It will then act similar to the debugging steps outlines in Debugging Hive code. It is faster since there is no need to compile Hive code,
and go through Ant. It can be used to debug both client side and server side Hive.

If you want to debug a particular query, start Hive and perform the steps needed before that query. Then start Hive again in debug to debug that query.

```
>  ./build/dist/bin/hive
>  perform steps before the query
```

```
>  ./build/dist/bin/hive --debug
>  run the query
```

Note that the local file system will be used, so the space on your machine will not be released automatically (unlike debugging via Ant, where the tables created in test are automatically dropped at the end of the test). Make sure to either drop the tables explicitly, or drop the data from /User/hive/warehouse.

Pluggable interfaces

File Formats

Please refer to Hive User Group Meeting August 2009 Page 59-63.

SerDe - how to add a new SerDe

Please refer to Hive User Group Meeting August 2009 Page 64-70.

Map-Reduce Scripts

Please refer to Hive User Group Meeting August 2009 Page 71-73.

UDFs and UDAFs - how to add new UDFs and UDAFs

Please refer to Hive User Group Meeting August 2009 Page 74-87.