Unit Testing Hive SQL

Motivations

Hive is widely applied as a solution to numerous distinct problem types in the domain of big data. Quite clearly it is often used for the ad hoc querying of large datasets. However it is also used to implement ETL type processes. Unlike ad hoc queries, the Hive SQL written for ETLs has some distinct attributes:

- It is executed repeatedly, usually on a schedule.
- It is frequently a large, complex body of code.
- It has much greater longevity, persisting in an organisation’s code base for long periods of time.
- It is frequently modified over time, often in an organic manner to meet changing business requirements.
- It is critical to the operation of the organisation in that it generates valuable data.

Code exhibiting such properties is a strong candidate for unit test coverage as it is very prone to bugs, errors, and accidental breakage, all of which can represent risk to an organisation.

Challenges

There are a number of challenges posed by both Hive and Hive SQL that can make it difficult to construct suites of unit tests for Hive based ETL systems. These can be broadly described as follows:

- **Defining boundaries between components:** How can and how should a problem be decomposed into smaller, testable units. The ability to do this is limited by the set of language features provided by Hive SQL.
- **Harness provision:** Providing a local execution environment that seamlessly supports Hive’s features in a local IDE setting (UDFs etc). Ideally the harness should have no environmental dependencies such as a local Hive or Hadoop installation. Developers should be able to simply check out a project and run the tests.
- **Speed of execution:** The goal is to have large numbers of isolated, small tests. Test isolation requires frequent setup and teardown and the costs incurred are multiplied the number of tests. The Hive CLI is fairly heavy process to repeatedly start and stop and so some Hive test frameworks attempt to optimise this aspect of test execution.

Modularisation

By modularising processes implemented using Hive they become easier to test effectively and more resilient to change. Although Hive provides a number of vectors for modularisation it is not always clear how a large process can be decomposed. Features for encapsulation of query logic into components is separated into two perpendicular concerns: column level logic, and set level logic. Column level logic refers to the expressions applied to individual columns or groups of columns in the query, commonly described as ‘functions’. Set level logic concerns Hive SQL constructs that manipulate groupings of data such as: column projection with `SELECT`, `GROUP BY` aggregates, `JOIN`s, `ORDER BY` sorting, etc. In either case we expect individual components to live in their own source file or deployable artifact and imported as needed by the composition. For Hive SQL-based components, the `SOURCE` command provides this functionality.

Encapsulation of column level logic

In the case of column level logic Hive provides both UDFs and macros that allow the user to extract and reuse the expressions applied to columns. Once defined, UDFs and Macros can be readily isolated for testing. UDFs can be simply tested with existing Java/Python unit test tools such as JUnit whereas Macros require a Hive command line interface to execute the macro declaration and then exercise it with some sample `SELECT` statements.

Encapsulation of set level logic

Unlike column level logic, it is much less obvious how best to encapsulate and compose collections of set based logic. Consider the following example of a single complex query comprising joins, groupings, and column projections:

Monolithic query

```
SELECT ... FROM (                  -- Query 1
  SELECT ... FROM (                --  Query 2
    SELECT ... FROM (              --   Query 3
      SELECT ... FROM a WHERE ...  --    Query 4
    ) A LEFT JOIN (                --   Query 3
      SELECT ... FROM b            --    Query 5
    ) B ON (...)                   --   Query 3
  ) ab FULL OUTER JOIN (           --  Query 2
    SELECT ... FROM c WHERE ...    --   Query 6
  ) C ON (...)                     --  Query 2
) abc LEFT JOIN (                  -- Query 1
  SELECT ... FROM d WHERE ...      --  Query 7
) D ON (...)                       -- Query 1
GROUP BY ...;                      -- Query 1
```

This query has a very broad set of responsibilities which cannot be easily verified in isolation. On closer inspection it appears that it is in fact formed of at least 7 distinct queries. To effectively unit test this process we must encapsulate each of the subqueries into separate components so that they can be tested independently. To achieve this there are a number of approaches are open to us including:

- Sequential execution of components with intermediate tables.
- Views.
- Variable substitution of query fragments.
- Functional/procedural SQL approaches.

Limited testing suggests that the `VIEW` approach is more effective than using sequential execution of components with intermediate tables, both in terms of elegance and performance. Intermediate table solutions (including `TEMPORARY` tables) take longer to run, generate more I/O, and restrict query optimization opportunities. It should also be noted that views do not appear to suffer from performance issues as often prophesied; in fact the execution plans and times for views and monolithic queries were comparable.

Variable substitution was also suggested as an approach for modularizing large queries, but upon inspection it was found to be unsuitable as an additional bash file is required which would make testing more complex. HPL/SQL was also considered, however it does not have the necessary pipelined function feature required for query modularization.

Tools and frameworks

When constructing tests it is helpful to have a framework that simplifies the declaration and execution of tests. Typically these tools allow the specification of many of the following:

- Execution environment configuration: usually `hiveconf` and `hivevar` parameters.
- Declaring input test data: creating or selecting files that back some source tables.
- Definition of the executable component of the test: normally the SQL script under test.
- Expectations: These can be in the form of a reference data file or alternatively fine grained assertions can be made with further queries.

The precise details are of course framework specific, but generally speaking tools manage the full lifecycle of tests by composing the artifacts provided by the developer into a sequence such as:

1. Configure Hive execution environment.
2. Setup test input data.
3. Execute SQL script under test.
4. Extract data written by the executed script.
5. Make assertions on the data extracted.

At this time there are are a number of concrete approaches to choose from:

- HiveRunner: Test cases are declared using Java, Hive SQL and JUnit and can execute locally in your IDE. This library focuses on ease of use and execution speed. No local Hive/Hadoop installation required. Provides full test isolation, fine grained assertions, and seamless UDF integration (they need only be on the project classpath). The metastore is backed by an in-memory database to increase test performance.
- beetest: Test cases are declared using Hive SQL and 'expected' data files. Test suites are executed using a script on the command line. Apparently requires HDFS to be installed in the environment in which the tests are executed.
- hive_test: Test cases are declared using Java, Hive SQL and JUnit and can execute locally in your IDE.
- HiveQLUnit: Test your Hive scripts inside your favourite IDE. Appears to use Spark to execute the tests.
- How to utilise the Hive project's internal test framework.

Useful practices

The following Hive specific practices can be used to make processes more amenable to unit testing and assist in the simplification of individual tests.

- Modularise large or complex queries into multiple smaller components. These are easier to comprehend, maintain, and test.

- Use macros or UDFs to encapsulate repeated or complex column expressions.
- Use Hive variables to decouple SQL scripts from specific environments. For example it might be wise to use `LOCATION ${myTableLocation}` in preference to `LOCATION /hard/coded/path`.
- Keep the scope of tests small. Making coarse assertions on the entire contents of a table is brittle and has a high maintenance requirement.
- Use the `SOURCE` command to combine multiple smaller SQL scripts.
- Test macros and the integration of UDFs by creating simple test tables and applying the functions to columns in those tables.
- Test UDFs by invoking the lifecycle methods directly (`initialize`, `evaluate`, etc.) in a standard testing framework such as JUnit.

Relevant issues

- HIVE-12703: CLI agnostic HQL import command implementation

Other Hive unit testing concerns

Although not specifically related to Hive SQL, tooling exists for the testing of other aspects of the Hive ecosystem. In particular the BeeJU project provides JUnit rules to simplify the testing of integrations with the Hive Metastore and HiveServer2 services. These are useful, if for example, you are developing alternative data processing frameworks or tools that aim to leverage Hive's metadata features.