SQL Windowing Functions with Hive

Harish Butani
harish.butani@sap.com

January 16, 2012

1 Overview

This document describes a solution for issue ([Hive 896]). Hive allows you to define UDFs, UDAFs and UDTFs. But none of these fit the model of Table-In/Table-Out functions needed to implement a generic solution. Here we propose a solution that runs with Hive as opposed to enhancing Hive. SQL Analytical functions lend themselves to being processed as an add-on layer. By keeping the Windowing processing outside of Hive we will be able to provide similar functionality for Pig; we already support Native MR processing.But admittedly this architecture misses opportunities in Query Optimization and interoperability. Long term SQL Windowing functions should be integrated into the Hive Engine. For now, we try to address the interoperability issue by making it very simple syntactically to combine Hive queries with Windowing functions. We also think this is an interesting exercise in extending Hive; we plan to use this architecture to provide other OLAP functionality in the future. Finally we hope that windowing functions are useful on their own.

The Windowing Processor works with Hive in several modes: *Hive* and *MR* are the 2 major modes. The Windowing Query is fashioned on the Oracle Analytical clauses([Oracle Doc.]). Here is a simple Windowing Query:

Listing 1: A simple Windowing Query

```
1   from <select county, tract, arealand
2         from geo_header_sf1
3         where sumlev = 140>
4   partition by county
5   order by county, arealand desc
6   with rank() as r,
7        sum(arealand) over rows between
8                           unbounded preceding and
9                           current row as cum_area,
10  select county, tract, arealand, r, cum_area
11  where <r \\> 3>
12  into path='/tmp/wout'
```

- the Source of the Windowing Query can be a Hive Query or table among other things.
- The Query itself provides an area to express partitioning, ordering and Windowing expressions

- The output of the Query can be stored using any SerDe/OutputFormat combination.

In the following we first explain the Windowing Query, next we describe how a Query is Processed, then we describe the Modes of operation, and finally we provide some final thoughts.

2 The Windowing Query

Consider Query 1. The Query form is:

- the *tableinput* clause specifies the input of the Query. In this example the input is an *embedded Hive Query*. Other possibilities are a *Hive tableName* or a *tableinput* clause that contains all the details(location, SerDe, Format, columns, columnTypes etc) about the input.

- the *partition* and *order by* clauses specify what columns the rows are ordered on and what are the partition/group boundaries. The order by columns must be a superset of the partition columns.Currently the ordering is specified globally and not on a per function basis(this is possible to do; time/resource constraints is why it is not available). The Window functions are applied on each Partition.

- Next the Query contains a list of function expressions. Currently there are 16 functions available. These are loosely divided into Ranking, Aggregation and Navigation functions. Not all functions support a windowing clause. Functions are described by an annotation that specifies their details: name, description, windowing support, args with their names, types and if they are required/optional.

- The Processor uses Groovy for expression support. So arguments to functions can be columns, literals(strings, numbers or boolean) and also groovy expressions. Groovy expressions are enclosed in <>. For e.g. when requesting a *sum* one may write it as $sum(arealand)$ which is saying compute the sum over the *arealand* column; but one can also write $sum(< arealand/2 >)$, which is a sum over the arealand divided by 2.

- The window clause supports both range boundaries and value boundaries. Boundaries can both start and end at the Current Row.

- The Select List is a comma separated list of identifiers and/or groovy expressions. Examples of groovy expression usage are: expressing ratios over computed aggregations, also available are lead and lag functions to do relative comparisons/computations like compute the delta etc.

- the *where* clause can be used to filter result rows. This can be used to do TopN queries. As shown here the Top 3 tracts(by land area) for each county are listed.

- the *output* clause is used to express the location, SerDe and Format of the output file. At the time of writing this document, we don't provide the next logical step of creating a Hive table that wraps the output. Its

coming soon. In this example the defaults are used so apart from the filepath other parameters are optional.

A Hive mode query looks very similar, except that the tableinput provides the details about the Input; in this case the Windowing Process is spawned by the Hive Script Operator:

Listing 2: A Hive mode Query

```
from tableinput(
 columns = \'p_partkey,p_name,p_mfgr,p_brand,p_type,p_size,
                    p_container,p_retailprice,p_comment\',
 \'columns.types\' = \'int,string,string,string,string,int,
                                 string,double,string\' )
 partition by p_mfgr
 order by p_mfgr, p_name
with
 rank() as r,
 sum(p_size) over rows between 2 preceding
                     and 2 following as s,
 min(<p_size>) over rows between 2 preceding
                     and 2 following as m[int],
 denserank() as dr, cumedist() as cud,
 percentrank() as pr,
 ntile(<3>) as nt,
 count(<p_size>) as c,
 count(<p_size>, \'all\') as ca,
 count(<p_size>, \'distinct\') as cd,
 avg(<p_size>) as avg,
 stddev(p_size) as st,
 first_value(p_size) as fv,
 last_value(p_size) as lv,
 first_value(p_size, \'true\') over rows between 2 preceding
                                    and 2 following as fv2
 select p_mfgr,p_name, p_size, r, s, m, dr, cud, pr, nt, c,
          ca, cd, avg, st, fv,lv, fv2
```

- this Query lists more functions supported: denserank, percentrank, count (with all and distinct variations), stddev, *first_value*, *last_value*.
- In hive mode quotes must be escaped; this is because of handling by ScriptOperator and/or the shell. Additionally newlines may have to be removed. See below for a complete hive query.
- In hive mode the Processor assumes the input stream is coming in from stdin and the format is TypedBytesSerDe.

The details of the language are beyond the scope of this doc; interested readers should look at the antlr grammar and the annotations for each function.

3 Processing Details

The Windowing Processor is a generic Query Processor that can work in multiple modes. Conceptually it is fed a sorted input stream and a Windows Query

that it generates a Output Stream that contains the results of the Windowing functions. Currently the Processor works in MR and Hive mode. We describe the Modes in the next section; here we describe the how a Query is evaluated.

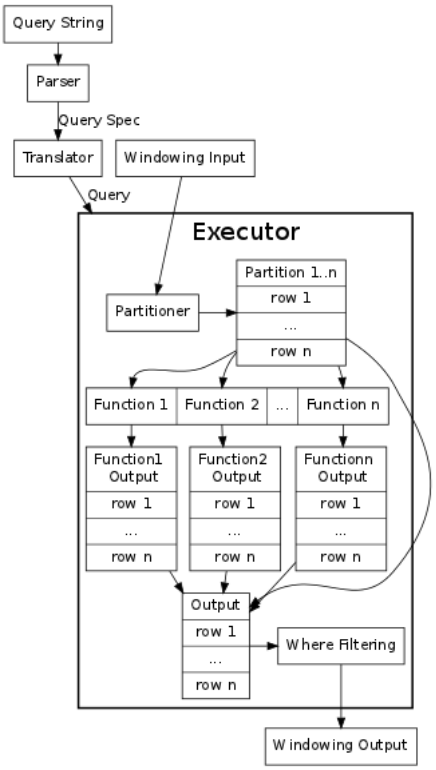

Figure 1: Execution of a Query

The internal flow of processing a Query is shown in Figure 1.

- A Query String is converted into a Query Object and passed to the Executor. A Query contains a list of Windowing Functions, the Windowing Input and the Windowing Output.
- The Executor streams the rows from the WindowingInput through the Partitioner. This produces a table for each Partition. It then passes the table to each Windowing Function which evaluates the function for the Partition (and if necessary for a specific window for each row in the partition). The function outputs an Object that provides the context in which the where Clause expression and the Select List are evaluated.
- The Executor then creates an Output Object for each input row in the partition by processing the Select List. Before this the where clause filter is applied, if present and only those rows that meet the criteria are output. The Output Object is then passed to the Windowing Output.
- during execution the Functions and the Output processing have a GroovyShell at their disposal. When functions evaluate Groovy Expressions the columns

of the Input are bound to the execution environment and can be used as variables. When the Output is processed both the input columns and the outputs of functions are available as variables.

- The *Windowing Input* is responsible for making Writables available to the processing environment. The specific process of how this happens depend on the Mode of Operation. Details are beyond this introductory doc. See [Wndw Interal Doc.] for details.

4 Modes of Operation

4.1 MapReduce Mode

- In this mode the WindowingProcessor assumes it can interact with a HiveMetaStoreServer and with a Map-Reduce cluster.
- Further it assumes it can access a service that can execute *embedded Hive Queries*. Hence there are 2 variations of this mode:

  Windowing CLI: In this mode the User interacts with a CLI client that is literally an extension of the Hive CLI. The WindowingProcessor runs inside a sandbox in this landscape but asks the Hive CLI to execute Hive Queries. We describe this mode in detail below.

  Hive ThriftServer: In this setup the WindowingProcessor connects to hive via the Thrift interface. We use this in our MRTest environment. This mode is useful for embedding the WIndowing jar in your application.
- In this mode the Windowing Processing is done in a MR job. The Processor setups up the Job with the relevant partitioning, sorting configuration. The Windowing functions are run in the Rduce Job. The details of the Job are outside the scope of this doc. See [Wndw Interal Doc.] for details.

4.1.1 MapReduce Mode using the Windowing CLI

We provide a CLI interface. This is exposed as a hive service and is invoked as '$hive --servicewindowingCli -w < pathtowindoingjar >$'. In this interface users can enter hive or windowing queries. The *wmode* settings controls how a command is interpreted. See Figure 2. for the overall architecture.

4.2 Hive Mode

The overall flow of processing is shown in Figure 3. Processing involves streaming data from a Hive Query/Table via the Script Operator:

- The Map Operation is a Noop. It is used so that the M-R shuffle mechanism gets invoked for partitioning and sorting the input rows.
- The *Readers* and *Writers* capture the transformations that have to happen to integrate with an external process. The Windowing Process uses the TypedBytesSerDe for transforming bytes to Objects and back. The readers and writers handle the transformations from the input SerDe to TypedBytesSerDe and then to the output SerDe.

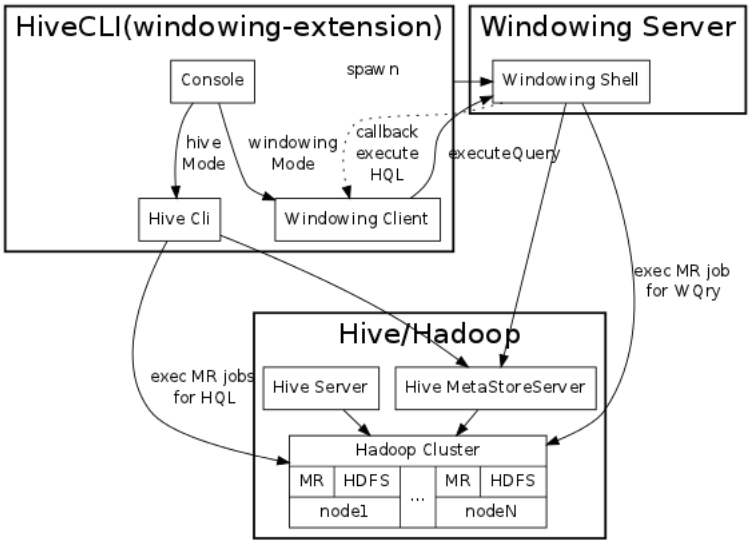

Figure 2: Overall Architecture: MR mode

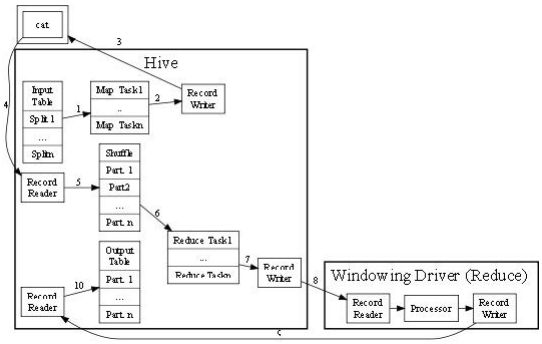

Figure 3: Hive Mode: Overall processing of a Windowing Request

- The Windowing Process is invoked during the Reduce phase. The Window process assumes that the data being streamed in is sorted. The Window process takes a *Windowing Query* that lists the operations to perform on the input stream.
- The Output from the Windowing process is streamed back to the Script Operator. After going through the RecordReader it can then flow into the next operation in the Hive Query Graph.

In Hive mode the Windowing Query is expressed as a Script Operation. Here is a complete Query in Hive mode:

Listing 3: A complete Query in Hive Mode

```
1 CREATE TABLE windowing_test as
2 select p_mfgr,p_name, p_size, r, s, m, dr, cud, pr, nt, c,
3          ca, cd, avg, st, fv,lv, fv2
```

```
4  from
5  (
6  from (
7    from part
8    select transform(p_partkey,p_name,p_mfgr,p_brand,p_type,
9                                    p_size,p_container,p_retailprice,p_comment)
10     ROW FORMAT SERDE 'org.apache.hadoop....TypedBytesSerDe'
11     RECORDWRITER 'org.apache.hadoop.hive....TypedBytesRecordWriter'
12     USING '/bin/cat'
13     as (p_partkey,p_name,p_mfgr,p_brand,p_type,p_size,
14           p_container,p_retailprice,p_comment)
15     ROW FORMAT SERDE 'org.apache.hadoop...TypedBytesSerDe'
16     RECORDREADER 'org.apache.hadoop....TypedBytesRecordReader'
17     DISTRIBUTE BY p_mfgr
18     SORT BY p_mfgr, p_name
19 ) map_output
20   select transform(p_partkey,p_name,p_mfgr,p_brand,p_type,
21                    p_size,p_container,p_retailprice,p_comment)
22     ROW FORMAT SERDE 'org.apache.hadoop....TypedBytesSerDe'
23     RECORDWRITER 'org.apache.hadoop....TypedBytesRecordWriter'
24     USING 'java
25        -cp "antlr-runtime-3.4.jar:windowing.jar:groovy-all-1.8.0.jar...."
26        windowing.WindowingDriver
27        -m hive
28         -q "from tableinput(
29         columns = \'p_partkey,p_name,p_mfgr,..\',
30         \'columns.types\' = \'int,string,string,...\' )
31         partition by p_mfgr
32         order by p_mfgr, p_name
33         with
34         rank() as r,
35          sum(p_size) over rows between unbounded preceding
36                                          and current row as s,
37          ntile(<3>) as nt,
38 select p_mfgr,p_name, p_size, r, s, nt"'
39 as (p_mfgr,p_name, p_size, r, s, nt)
40 ROW FORMAT SERDE 'org.apache.hadoop....TypedBytesSerDe'
41 RECORDREADER 'org.apache.hadoop....TypedBytesRecordReader'
42 ) reduce_output;
```

5 Summary

1. As an analogy to Oracle, look at this as a quick implementation in PL/SQL. Typically new DB features are implemented in PL/SQL, once they show value they are pushed down into the Engine and finally integrated with the Semantic layers.

2. In this regard it would have been helpful if there was an In-Process Script Operator. This would avoid the overhead of streaming data to an external process.

3. The implementation hopefully shows a technique for providing extensible

mini DSLs. We hope to use this pattern for introducing other Operations specifically for dimensional processing in OLAP.

6 Possible Next Steps

Performance In Hive Mode the data is going through several transformations: serialization through TypedBytesSerDe, inter process streaming, copy to a Standard Java Object, serialization through TypedBytesSerDe, and finally inter process streaming. Need to look at avoiding some of these conversions. Along these lines it would be nice to avoid streaming data through to an external process. In MR mode the transformation to Standard Java Object can be eliminated.

Another performance improvement is on the evaluation of the Functions. Also, currently the Executor is very simple. It can be parallelized, and it can be made smarter when computing windows. Currently for computing windows the Executor computes the raw expressions for each row in the Partition in the first pass and then processes each row w.r.t. to its window range in a second pass. In this way it avoids computing the raw expressions multiple times, but it still takes $O(n^2)$ time.

Multiple Orderings currently we only support a single Ordering; whereas in SQL each function can have its own ordering. This is fairly easy to support by reordering a partition before evaluating the function.

Multiple Partitioning In SQL each function can have its own partitioning. Instead of providing partitioning specification per function, an initial step would be to allow for multiple Partitioning clauses. This is a fairly involved process and would be done only if there is significant demand for it.

References

[Hadoop] Hadoop website *http://tiny.cc/b72fb*

[Hive] Hive website *http://tiny.cc/559sh*

[Hive ICDE] Hive - A Petabyte Scale Data Warehouse Using Hadoop *http://tiny.cc/o5q6h*

[Hive 896] Hive Issue 896 *https://issues.apache.org/jira/browse/HIVE-896*

[Wndw Interal Doc.] SQL Windowing development details: posted on website.

[Oracle Doc.] Oracle SQL Reference: SQL for Analysis and Reporting http://tinyurl.com/7s9qgjr