Partitioned Table Functions and Windowing in Hive

Harish Butani, Prajakta Kalmegh

December 23, 2012

1 Overview

In this document we describe an extension to HQL: the introduction of **Partition table Functions(PTF)**. A related feature is **Windowing clauses** in SQL(as described in [Hive 896]). Our solution provides Windowing clause support by using the PTF mechanism support we have built(with a built-in PTF: WindowingTableFunction). For a description of why these features are interesting and the kinds of Analysis this enables, we refer you to our Hadoop Summit and Hive User's Group presentations ([Hdp Sum, Hive UGrp]).

The solution let's you invoke a Partitioned Table Function anywhere a Table/SubQuery can appear in HQL. PTFs can be chained in one invocation. The Hive FunctionRegistry has been extended to support PTFs; so PTF lifecycle management will be similar to how other kinds of functions are managed in Hive.

The Windowing clause support in HQL matches standard SQL as much as possible: ability to define windows with the Query or individual Function; ability to specify a range or value based window with any UDAF. But since Windowing is handled as a PTF invocation, all Window specification must have the same Partition and Order specification. In a subsequent patch we can easily relax the Order restriction (ie allow different orderings on Window functions), but multiple Partitions is a bigger change. We also introduce 2 new concepts: the ability to do inter-row calculations using **Lead/Lag** functions, and the ability to apply filters on the Window/Partition after it has been processed.

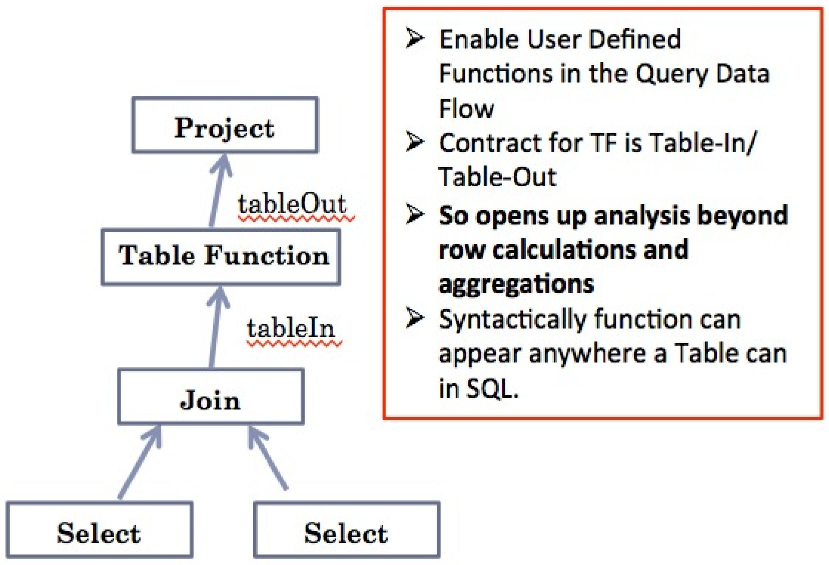
The rest of the document has the following structure: in *Introduction* we describe the key concepts of our solution; in *Language changes* we describe the extensions to HQL to enable PTF invocations and Windowing; in *Query Semantics* we describe how Windowing and PTF queries are interpreted and executed; in *Implementation Details* we describe some of the details of our solution; in *Function Library* we describe Windowing and PTF functions we have implemented (or planning to migrate from our prototype); and finally in *Future Work* we talk about some of the possible paths that a PTF mechanism enable.

2 Introduction

In this section we describe the concepts of PTFs and Windowing; relating them to HQL.

2.1 Table Function

- Enable User Defined Functions in the Query Data Flow
- Contract for Function is Table-In/Table-Out
- **So opens up analysis beyond row calculations and aggregations**
- Syntactically function can appear anywhere a Table can in SQL.



➢ Enable User Defined Functions in the Query Data Flow
➢ Contract for TF is Table-In/ Table-Out
➢ **So opens up analysis beyond row calculations and aggregations**
➢ Syntactically function can appear anywhere a Table can in SQL.

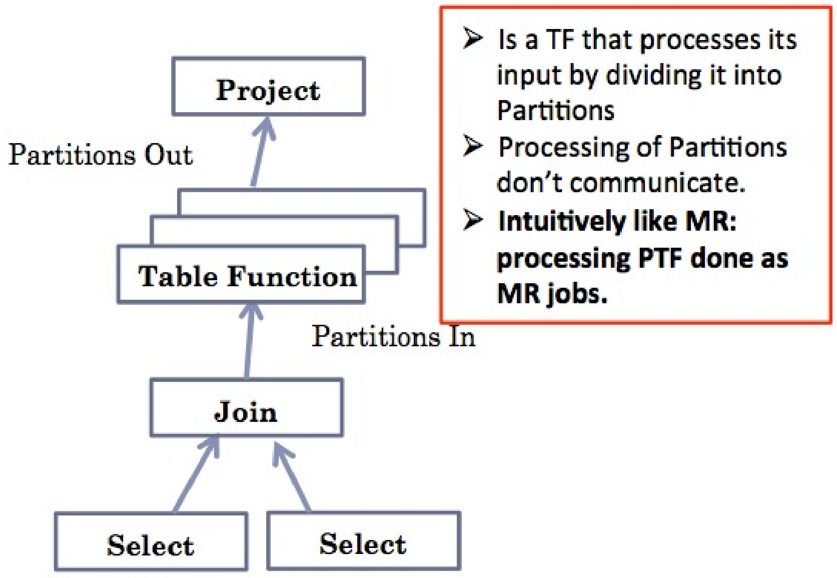
2.2 Partition Specification

Is used to specify how an input dataset is to be partitioned and how partitions should be ordered. In HQL this can be specified in multiple ways:

- With distribute by, sort by clause combination.
- With a cluster by clause
- **Note:** For Table Functions and Windowing clauses a Query with just a distribute by clause is treated as a Partition Specification where the Input rows are partitioned by the columns in the distribute by clause and rows in a Partition are sorted on the partition columns. So the cluster by clause is redundant.
- With a window specification. See below

2.3 Partitioned Table Function

Is a TF that processes its input by dividing it into Partitions Processing of Partitions don't communicate. Intuitively like MR: processing PTF done as MR jobs.


Project
Partitions Out
Table Function
Partitions In
Join
Select
Select
➢ Is a TF that processes its input by dividing it into Partitions
➢ Processing of Partitions don't communicate.
➢ Intuitively like MR: processing PTF done as MR jobs.

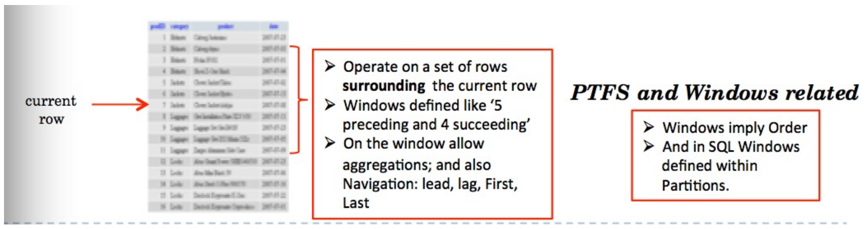
So a PTF is:

- A Function whose contract is to operate on an Input Dataset and return an Output Dataset. *So interface is Table In, Table Out.*

- A PTF operates on the Input in Partitions; the Partitions are ordered.
- A PTF invocation must specify the Input Dataset and how to partition and order the Input. The partitioning may not be specified explicitly, as PTFs can be chained; so a PTF can inherit the partition specification from a PTF up the function chain.
- A PTF may also operate on the *raw input* before it is partitioned.
- Operating on data implies:
  - it can be reshaped (the schema can be altered)
  - the contract is always (both on raw data and on partitioned data) Table in/Table out.

2.4 Windowing Specification

- Each row in a Partition can be operated on in the context of a Window of surrounding rows.
- A window can be specified as a Range of Rows, or as a Range of Values.
- *The window clause can specify everything: the distribute by, sort by and the window frame.*
- See Grammar details on how to specify a Window.
- A Window maybe specified for each UDAF invocation, or globally for the Query and then referenced by a UDAF invocation.



- Operate on a set of rows **surrounding** the current row
- Windows defined like '5 preceding and 4 succeeding'
- On the window allow aggregations; and also Navigation: lead, lag, First, Last

PTFS and Windows related

- Windows imply Order
- And in SQL Windows defined within Partitions.

3 Language changes

3.1 PTF invocations

- We extend the **from** Clause to have a new form: *a PTF invocation*

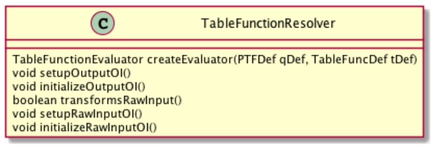
- A PTF invocation has the following form:
  - the function name
  - The Input on which this function operates: which can be a Table, a SubQuery or another PTF invocation.
  - A *Partitioning Specification*, which specifies how the input should be operated on.
  - Followed by Arguments to the function, which can be any HQL expression.
  - Finally a PTF invocation may have an alias.
- See [**?**] for grammar rules.

3.2 Windowing specifications

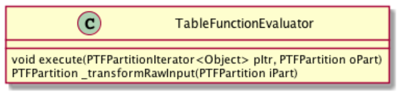
- We introduce a *Window frame*. This is either *Range* or *Value* based. A Range based Window is specified by the number of rows Preceding or Succeeding the Current Row. A Value based range is specified by an Amount that is Less or More than the Value of the Current Row.
- A *Window specification* is a combination of a Partitioning Specification and a Window Frame. A Window Specification may also inherit properties from a Window Definition. If a Window Specification has no Partitioning Specification, it inherits the default Specification of the Query.
- A *Window Definition* is a named *Window Specification*. It is specified at the Query level and can be referenced from any UDAF's Window specification.
- A query can how contain a Window Clause. The Window Clause contains one or more Window Definitions. All forms of the Query body support the Window Clause. We only show the SelectStatement here.
- Finally a Select Expression can have a window specified using the OVER sub clause.
- See [**?**] for grammar rules.

4 Query Semantics

4.1 The PTF interface



```
TableFunctionResolver
TableFunctionEvaluator createEvaluator(PTFDef qDef, TableFuncDef tDef)
void setupOutputOI()
void initializeOutputOI()
boolean transformsRawInput()
void setupRawInputOI()
void initializeRawInputOI()
```



```
TableFunctionEvaluator
void execute(PTFPartitionIterator<Object> pItr, PTFPartition oPart)
PTFPartition _transformRawInput(PTFPartition iPart)
```

4.1.1 Table Function Resolver

Based on Hive GenericUDAFResolver. The Resolver is responsible for:

- creating the Table Function Evaluator during translation.
- Setting up the The Raw and Output ObjectInspectors of the Evaluator.
- The Evaluator holds onto the TableFunctionDef. This provides information about the arguments to the function, the shape of the Input partition and the Partitioning details.

The Resolver for a function is obtained from the FunctionRegistry. The Resolver is initialized by the following 4 step process:

- The initialize method is called; which is passed the PTFDef and the TableFunctionDef.
- The resolver is then asked to setup the Raw ObjectInspector. This is only required if the Function reshapes the raw input.
- Once the Resolver has had a chance to compute the shape of the Raw Input that is fed to the partitioning machinery; the translator sets up the partitioning details on the tableFuncDef.
- finally the resolver is asked to setup the output ObjectInspector.

During runtime, when the PTF chain is initialized:

- initializeOutputOI is invoked. At this point the TableFunction can assume that the Expression Nodes exist for all the Def (ArgDef, ColumnDef, WindowDef..). It is the responsibility of the TableFunction to construct the ExprNodeEvaluator and setup the OI.
- initializeRawInputOI has the same responsibility as initializeOI, but for the RawInput.

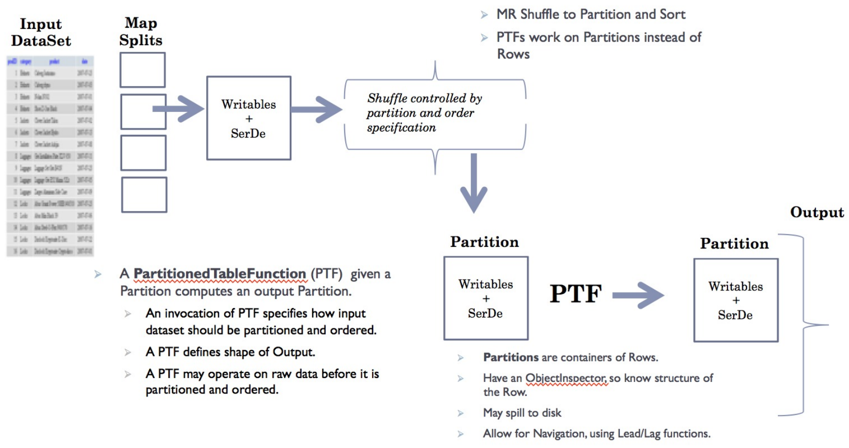
4.1.2 Table Function Evaluator

Based on GenericUDAFEvaluator. The Evaluator also holds onto the TableFunctionDef. This provides information about the arguments to the function, the shape of the Input partition and the Partitioning details. The Evaluator is responsible for providing the 2 execute methods:

execute which is invoked after the input is partitioned; the contract is, it is given an input Partition and must return an output Partition. The shape of the output Partition is obtained from the getOutputOI call.

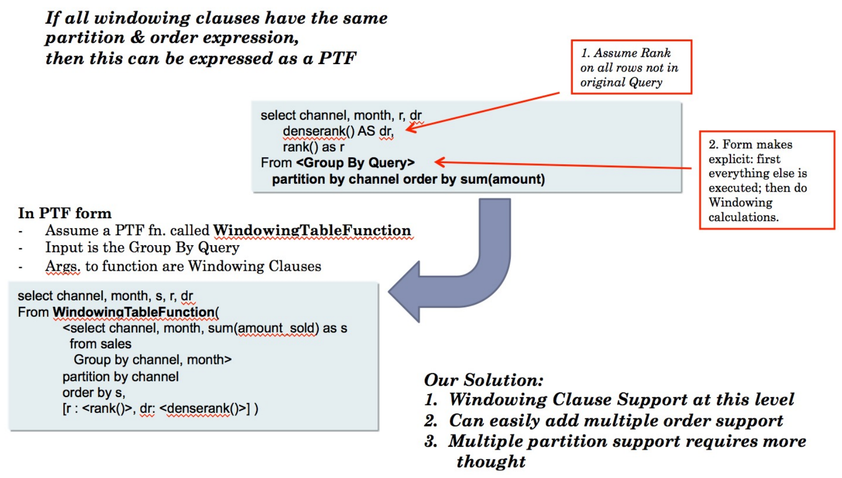
transform Raw Input In the case where this function indicates that it will transform the raw input before it is fed through the partitioning mechanics, this function is called. Again the contract is it is given an input Partition and must return an Partition. The shape of the output Partition is obtained from getRawInputOI() call.

4.2 The PTF Invocation



Input DataSet
Map Splits
Writables + SerDe
Shuffle controlled by partition and order specification
MR Shuffle to Partition and Sort
PTFs work on Partitions instead of Rows
Output
Partition
Writables + SerDe
PTF
Partition
Writables + SerDe
A PartitionedTableFunction (PTF) given a Partition computes an output Partition.
An invocation of PTF specifies how input dataset should be partitioned and ordered.
A PTF defines shape of Output.
A PTF may operate on raw data before it is partitioned and ordered.
Partitions are containers of Rows.
Have an ObjectInspector, so know structure of the Row.
May spill to disk
Allow for Navigation, using Lead/Lag functions.

4.3 Processing Windowing Clauses with PTF mechanism



If all windowing clauses have the same partition & order expression, then this can be expressed as a PTF

```
select channel, month, r, dr
      denserank() AS dr,
      rank() as r
From <Group By Query>
   partition by channel order by sum(amount)
```

1. Assume Rank on all rows not in original Query

2. Form makes explicit: first everything else is executed; then do Windowing calculations.

In PTF form

- Assume a PTF fn. called **WindowingTableFunction**
- Input is the Group By Query
- Args. to function are Windowing Clauses

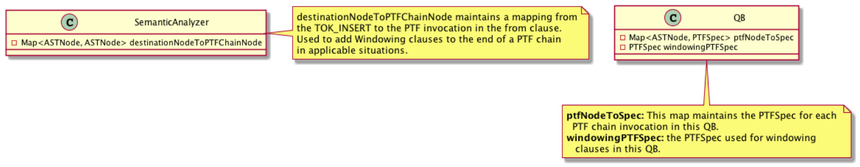
```
select channel, month, s, r, dr
From WindowingTableFunction(
            <select channel, month, sum(amount_sold) as s
              from sales
               Group by channel, month>
            partition by channel
            order by s,
            [r : <rank()>, dr: <denserank()>] )
```

Our Solution:

1. ***Windowing Clause Support at this level***
2. ***Can easily add multiple order support***
3. ***Multiple partition support requires more thought***

5 Implementation Details

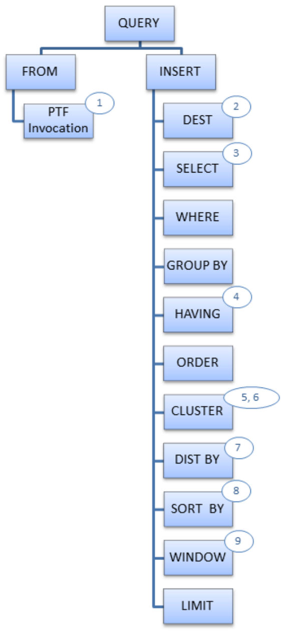
5.1 Data Structures

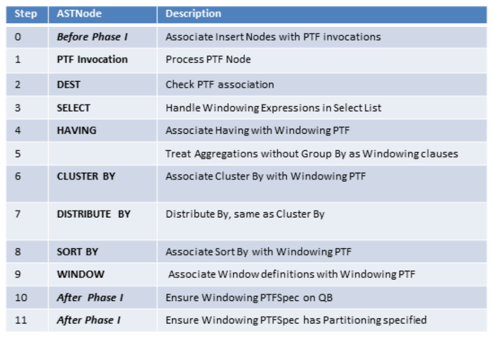

SemanticAnalyzer
Map<ASTNode, ASTNode> destinationNodeToPTFChainNode
QB
Map<ASTNode, PTFSpec> ptfNodeToSpec
PTFSpec windowingPTFSpec

destinationNodeToPTFChainNode maintains a mapping from the TOK_INSERT to the PTF invocation in the from clause. Used to add Windowing clauses to the end of a PTF chain in applicable situations.

ptfNodeToSpec: This map maintains the PTFSpec for each PTF chain invocation in this QB.
windowingPTFSpec: the PTFSpec used for windowing clauses in this QB.

5.2 Phase 1: AST to Query Specification





Step	ASTNode	Description
0	***Before Phase I***	Associate Insert Nodes with PTF invocations
1	**PTF Invocation**	Process PTF Node
2	**DEST**	Check PTF association
3	**SELECT**	Handle Windowing Expressions in Select List
4	**HAVING**	Associate Having with Windowing PTF
5		Treat Aggregations without Group By as Windowing clauses
6	**CLUSTER BY**	Associate Cluster By with Windowing PTF
7	**DISTRIBUTE BY**	Distribute By, same as Cluster By
8	**SORT BY**	Associate Sort By with Windowing PTF
9	**WINDOW**	Associate Window definitions with Windowing PTF
10	***After Phase I***	Ensure Windowing PTFSpec on QB
11	***After Phase I***	Ensure Windowing PTFSpec has Partitioning specified

Step 0, associate Insert Nodes with PTF invocations

Typically a PTF Chain invocation and Windowing processing are handled independently. But in one case Windowing processing can happen at the end of an existing PTF Chain. If the from Clause:

- contains only a PTF invocation
- there is no GBy or where clause
- and there is only 1 insert clause

then:

- we can process any Window Expressions in the SelectList as part of the same PTF chain as the chain in the from clause.

Any Windowing specifications on the Select list will be handled as an invocation on the internal WindowingTableFunction PTF added to the end of the PTF chain.

Step 1, Process PTF Node

Invoked during FROM AST tree processing, on encountering a PTF invocation.

- tree form is (TOK_PTBLFUNCTION name partitionTableFunctionSource partitioningSpec? arguments*)
- a partitionTableFunctionSource can be a tableReference, a SubQuery or another PTF invocation.
  - For a TABLEREF: set the tableName to the alias returned by processTable
  - For a SubQuery: set the tableName to the alias returned by processSubQuery
  - For a PTF invocation: recursively create a TableFunctionSpec.
- setup a PTFSpec for this top level PTF invocation.
- if the Query needs Windowing handling, then when encountering the associated DESTINATION AST node, this QuerySpec will be associated with the QB.
- Step 2, Check PTF association ::

If this destination is associated with a PTFNode, then set its PTFSpec as the Windowing PTFSpec of this Query Block.

Step 3, Handle Windowing Expressions in Select List

Invoked during Phase1 when a TOK_SELECT is encountered.

- Select tree form is: (TOK_SELECT (TOK_SELECTEXPR...) (TOK_SELECTEXPR...) ...)
- A Select Item form is: (TOK_SELEXPR selectExpression Identifier* window_specification?)

We need to extract from the SelectList any SelectItems that must be handled during Windowing processing. These are:

- SelectItems that have a window_specification
- SelectItems that invoke row navigation functions: Lead/Lag.

Do we need to change the SelectList in any way?

- initially we thought of replacing the selectExpressions handled by Windowing with a ASTNode that is of type Identfier and references the alias to the orginal expression. Why?
    - the output of processing the PTF Operator that handles windowing will contain the values of the Windowing expressions.
    - the final Select Op that is a child of the above PTF Op can get these values from its input by referring to the computed windowing expression via its alias.
- but this is not needed. Why?
    - When transforming a AST tree to an ExprNodeDesc the TypeCheckFactory checks if there is a mapping from an AST tree to an output column in the InputResolver; if there is, it uses the alias for the Output column
    - This is how values get handed from a GBy Op to the next Select Op;

We need the same thing to happen here.

Step 4, Associate Having with Windowing PTF

if QB has a Windowing PTFSpec and no GroupBy clause associate Having with it.

Step 5, Treat Aggregations w/o Group By as Windowing clauses

If the Query has no Group by, but there are Agg Expressions on this dest, we move these to the Windowing PTFSpec. If necessary we create a Windowing PTFSpec.

Step 6, Associate Cluster By with Windowing PTF

In the case when the Query has Windowing Clauses and a Distribute/Cluster By we associate the Distribute/Cluster By with the Windowing PTFSpec on the QB. If the Query has no Group by, but there are Agg Expressions on this dest, we move these to the Windowing PTFSpec.

Step 7, Distribute By, same as Cluster By

Do Steps 5, 6 when encountering a DistributeBy AST Node.

Step 8, Associate Sort By with Windowing PTF

if QB has a Windowing PTFSpec associate SortBy with it.

Step 9, Associate Window definitions with Windowing PTF

Associate windowing definitions with the Windowing PTFSpec on the QB.

Step 10, Ensure Windowing PTFSpec on QB

This is the same as Step 5. Moves any Agg Expressions to Windowing PTFSpec.

Step 11, Ensure Windowing PTFSpec has Partitioning specified

ensure that the PTF chain has a partitioning specification associated. This method should be called when:

- a PTF chain is encountered as a fromSource. (from the processPTF method)
- at the end of Phase1 to check that the WindowingTableFunction is driven from a partitioning specification (specified in the distribute by or cluster by clauses).

5.3 PTF Specification to Definitiion translation

5.3.1 Input Translation

- Translation of a Table Function Specification
  - Get the *TableFunctionResolver* for this Function from the FunctionRegistry.
  - Create the TableFuncDef object.
  - Get the InputInfo for the input to this function.
  - Translate the Arguments to this Function in the Context of the InputInfo.
  - ask the TableFunctionResolver to create a TableFunctionEvaluator based on the Args passed in.
  - ask the TableFunctionEvaluator to setup the Map-side ObjectInspector. Gives a chance to functions that reshape the Input before it is partitioned to define the Shape after raw data is transformed.

  - Setup the Window Definition for this Function. The Window Definition is resolved wrt to the InputDef's Shape or the RawInputOI, for Functions that reshape the raw input.
  - ask the TableFunctionEvaluator to setup the Output ObjectInspector for this Function.
  - setup a Serde for the Output partition based on the OutputOI.

- Setting up the SerDe and output OI of a TableFunction
  - For NOOP table functions, the serde is the same as that on the input; for other table functions it is the lazy binary serde.
  - If the query has a map-phase, the map oi is set to be the oi on the lazy binary serde unless the table function is a NOOP_MAP_TABLE_FUNCTION (in which case it is set to the oi on the serde of the input hive table definition).

- Constructing a Row Resolver from a OI
  - Construct a Row Resolver from a PTF OI.
  - For WindowTablFunction and Noop functions the PTF's Input RowResolver is also passed in. This way for any columns that are in the Input Row Resolver we carry forward their internalname, alias and table Alias.
  - for the Virtual columns:
    - the internalName is UPPER Case and the alias is lower case
    - since we put them in an OI, the fieldnames became lower cased.
    - so we look in the inputRR for the fieldName as an alias.

5.3.2 Output Translation

- setup Select RR and OI
  - Setup based on the OI of the final PTF + expressions in the SelectList that are handled by the Partition mechanism.
  - For expressions that are handled by the PTFOp : navigation expressions in SelectList or windowing clauses
  - add the mapping from ASTNode to ColumnInfo so that the SelectOp doesn't try to evaluate these.

5.4 Plan generation

A Plan is constructed for every PTF Specification in the Query. So this happens from:

Any PTF invocation in the from clause PTF invocations are handled analogous to how Sub Queries are handled. In Plan Generation right after plans are created for Sub Queries; we invoke *GenPTFPlan* for each PTF invocation. Note the one caveat is if the PTF invocation has been associated with the Window clause processing. In that case it is not processed here. This has already been taken care of in Phase 1; such an invocation is pulled out of the list of PTF invocations of the Query.

The Windowing PTF invocation of the Query This is handled in Body Plan Generation. This happens after Group By and Having plans are generated. Again *GenPTFPlan* is invoked to generate the Plan for Windowing processing.

5.4.1 Generate PTF plan

1. **Create the PTF Definition** The PTF Definition is constructed based on the Input(Parent) Operator's RowResolver and the PTF Specification.

2. **Build Reduce Sink Details** build the data structures (keyCols, valueCols, outColNames etc.) used for constructing a ReduceSinkDesc and ReduceSinkOperator.

   - Use the PartitionDef and OrderDef on the window of the first table function in the chain to base these structures of.
   - Order columns are used as key columns for constructing the ReduceSinkOperator. Since we do not explicitly add these to outputColumnNames, we need to set includeKeyCols = false while creating the ReduceSinkDesc.

3. **Build Map-Side Operator Graph** Template is either Input -> PTF_map -> ReduceSink **or** Input -> ReduceSink. This is based on whether the first table function in the query reshapes the Raw

4. **Build Reduce-side Operator Graph** Construct RowResolver for ExtractOperator using the ReduceSinkOperator's RowResolver. Next we

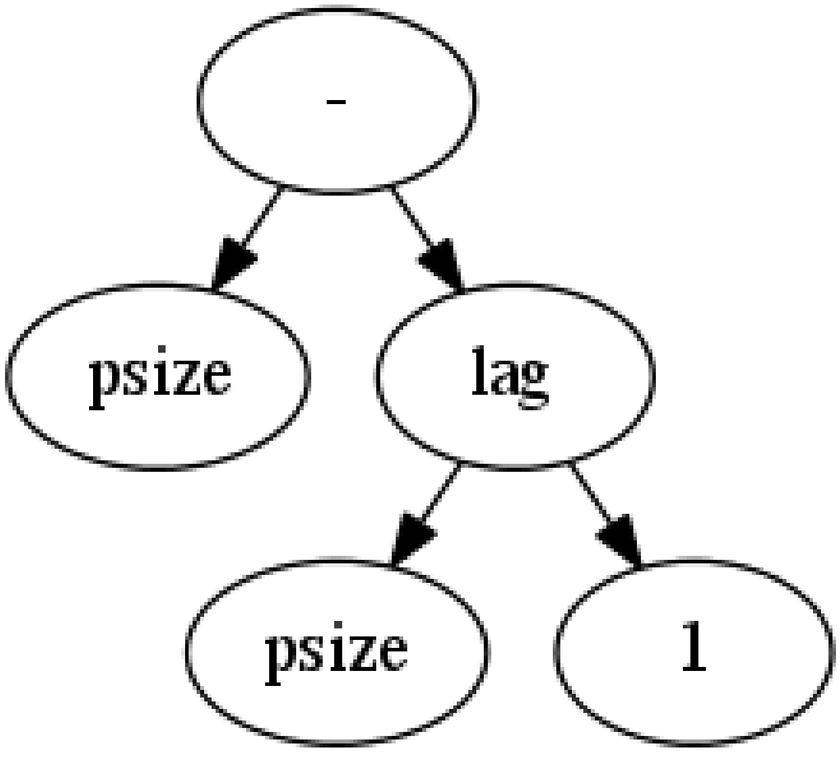
reconstruct the PTFDef using the RowResolver from the Extract Operator; so that the OIs in the Definition classes are based of the RowResolver from the Extract Op.

5. **Construct the PTF Operator** We serialize the PTF Definition to construct and add it to the PTFDesc of the PTF Operator. This way of setting up the PTF Operator is a leftover from our original implementation. We plan to support directly setting the PTF Definition on the Descriptor.

5.5 Lead/Lag Processing

Consider the following expression $psize - lag(psize, 1)$. This computes the difference in psize between the *current* row and the previous row.

In Hive the expression is translated int to the following Evaluator Tree



5.5.1 Translation

We setup the LeadLagUDF with an ExprEvaluator that is a duplicate of the first argument. Since the ExprEvaluator classes don't expose their fields with public getters, setting this up is superfluously contrived:

- We introduced a **WindowingExprNodeEvaluatorFactory**. Whenever it is called upon to create an ExprEvaluator from an ExprNode, before handing off to Hive's EvalFactory it records any Lead/Lag expressions in the given expression tree.

- Now when the ExprEvaluator is initialized based on a RowResolver. A check is made if its Expression Tree had any lead/lag functions. If yes:

  - a duplicate Evaluator is made for the of the 1st child (argument) of the lead/lag expression.
  - The duplicate is initialized using the RowResolver.
  - Finally it is attached with the Lead/Lag UDF.
  - For details see *TranslateUtils:initExprNodeEvaluator*

5.5.2 Execution

- The context of processing is always a **Partition**

- There is now a **PartitionIterator** which extends the Iterator with the following functions:

  - *lead(i)* or *lag(i) :* returns a row that precedes or succeeds the current row.
  - *getIndex() :* returns the index of the current row.
  - *resetToIndex(i) :* this sets the Iterator back to the given index.

- The utility **RuntimeUtils.connectLeadLagFunctionsToPartition(qDef, pItr)** sets the PartitionIterator in any lead/lag UDF in the current Query. This is how a lead/lag UDF gets access to surrounding rows in the Partition.

- During the execution of the row:

  - The function navigates to the corresponding row using lead/lag
  - Evaluates the expression for the first Arg on this row

    - Before returning this value it resets the Partition back to the index of the input row.

- Handling SerDe and Partition state
    - The Partition and SerDe object are designed to stream rows.
    - So the same java Object is returned on each call to **Partition:getAt** and **SerDe:deserialize**
    - Combine this with the *Lazy* model of some of the SerDes even though we can get to a different row in the partition **when the actual computation happens all subexpressions access the same row.** To get around this problem we have the lead/lag UDF return a StandardOI and during execution we copy the result of executing the first Arg to Standard Object.
    - There is one last thing that needs to be done:
        * If lead / lag is the last expression in the first Arg, then merely resetting the Partition back to the current row is not enough. The reason being this is not enough to trigger recomputation of the field level caches. Resetting the index causes the Struct:parsed to be false; but direct field level access doesn't check this flag. So a LazyInteger on this Struct if accessed directly will return the value already parsed. So in the current e.g. since the psize expression was evaluated before the lag the LazyInteger contains the value from the previous row (triggered by p_size expression in lag). To have it point to the current row; after the resetIndex we evaluate the first Arg in the context of the current row.

6 Function Library

7 Future Work

8 Appendix

8.1 PTF invocations

```
fromSource
@init { msgs.push("from source"); }
@after { msgs.pop(); }
```

```
    :
    ((Identifier LPAREN)=> partitionedTableFunction |
     tableSource | subQuerySource ) (lateralView^)*
    ;

partitioningSpec
@init { msgs.push("partitioningSpec clause"); }
@after { msgs.pop(); }
   :
   clusterByClause -> ^(TOK_PARTITIONINGSPEC clusterByClause) |
   distributeByClause sortByClause? ->
      ^(TOK_PARTITIONINGSPEC distributeByClause sortByClause?)
   ;

partitionTableFunctionSource
@init { msgs.push("partitionTableFunctionSource clause"); }
@after { msgs.pop(); }
   :
   subQuerySource |
   tableSource |
   partitionedTableFunction
   ;

partitionedTableFunction
@init { msgs.push("ptf clause"); }
@after { msgs.pop(); }
   :
   name=Identifier
   LPAREN
     ptfsrc=partitionTableFunctionSource partitioningSpec?
     (COMMA expression)*
   RPAREN alias=Identifier?
   ->   ^(TOK_PTBLFUNCTION $name $alias?
               partitionTableFunctionSource
               partitioningSpec? expression*)
   ;
```

8.2 Windowing specifications

```
selectItem
```

```
@init { msgs.push("selection target"); }
@after { msgs.pop(); }
    :
    ( selectExpression
      ((KW_AS? Identifier) | (KW_AS LPAREN Identifier (COMMA Identifier)* RPAREN))?
      (KW_OVER LPAREN ws=window_specification RPAREN )?
    ) -> ^(TOK_SELEXPR selectExpression Identifier* $ws?)
    ;

window_clause
@init { msgs.push("window_clause"); }
@after { msgs.pop(); }
:
  KW_WINDOW window_defn (COMMA window_defn)* -> ^(KW_WINDOW window_defn+)
;

window_defn
@init { msgs.push("window_defn"); }
@after { msgs.pop(); }
:
  Identifier KW_AS window_specification ->
    ^(TOK_WINDOWDEF Identifier window_specification)
;

window_specification
@init { msgs.push("window_specification"); }
@after { msgs.pop(); }
:
  Identifier? partitioningSpec? window_frame? ->
     ^(TOK_WINDOWSPEC Identifier? partitioningSpec? window_frame?)
;

window_frame :
 window_range_expression |
 window_value_expression
;

window_range_expression
@init { msgs.push("window_range_expression"); }
@after { msgs.pop(); }
```

```
:
 KW_ROWS KW_BETWEEN s=rowsboundary KW_AND end=rowsboundary ->
    ^(TOK_WINDOWRANGE $s $end)
;

rowsboundary
@init { msgs.push("rowsboundary"); }
@after { msgs.pop(); }
:
  KW_UNBOUNDED (r=KW_PRECEDING|r=KW_FOLLOWING)  -> ^($r KW_UNBOUNDED) |
  KW_CURRENT KW_ROW  -> ^(KW_CURRENT) |
  Number (d=KW_PRECEDING | d=KW_FOLLOWING ) -> ^($d Number)
;

window_value_expression
@init { msgs.push("window_value_expression"); }
@after { msgs.pop(); }
:
 KW_RANGE KW_BETWEEN s=valuesboundary KW_AND end=valuesboundary ->
    ^(TOK_WINDOWVALUES $s $end)
;

valuesboundary
@init { msgs.push("valuesboundary"); }
@after { msgs.pop(); }
:
  KW_UNBOUNDED (r=KW_PRECEDING|r=KW_FOLLOWING)  -> ^($r KW_UNBOUNDED) |
  KW_CURRENT KW_ROW  -> ^(KW_CURRENT) |
  rowExp=expression rngExp=Number (d=KW_LESS | d=KW_MORE ) -> ^($d $rowExp $rngExp)
;

selectStatement
   :
   selectClause
   fromClause
   whereClause?
   groupByClause?
   havingClause?
   orderByClause?
   clusterByClause?
   distributeByClause?
   sortByClause?
   window_clause?
   limitClause?
;
```

9 Bibliography

References

[Hadoop] Hadoop][http://tiny.cc/b72fbHadoop]] website

[Hive] Hive][http://tiny.cc/559shHive]] website

[Hive ICDE] Hive][http://tiny.cc/o5q6hHive]] - A Petabyte Scale Data Warehouse Using Hadoop

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

[Hdp Sum] Hadoop][http://www.slideshare.net/Hadoop_Summit/analytical-queries-with-hiveHadoop]] Summit '12 presentation on Adv. Analysis with Hive

[Hive UGrp] Hive][https://github.com/hbutani/SQLWindowing/blob/hive-rt/docs/HiveUserGroupMeet.pptxHive]] User's Group Presentation: PTF and Windowing with Hive