Hive Vectorized Query Execution Design

Jitendra Pandey†, Eric Hanson‡, Owen O'Malley†,
Remus Rusanu‡, Sarvesh Sakalanaga‡, Teddy Choi

†*Hortonworks*, ‡*Microsoft*

9/17/13

Contents

Introduction

The Hive query execution engine currently processes one row at a time. A single row of data goes through all the operators before the next row can be processed. This mode of processing is very inefficient in terms of CPU usage. Research has demonstrated that row-at-a-time processing yields high instruction counts and poor processor pipeline utilization (low instructions per cycle) [Boncz 2005]. Also currently Hive heavily relies on lazy deserialization and data columns go through a layer of object inspectors that identify column type, de-serialize data and determine appropriate expression routines in the inner loop. These layers of virtual method calls further slow down the processing.

The Hive vectorization project aims to remove the above-mentioned inefficiencies by modifying the execution engine to work on vectors of columns. The data rows will be batched together and represented as a set of column vectors and the processor inner loop will work on one vector at a time. This approach has been proved to be not only cache friendly but also achieves high instructions per cycle by effective use of pipelining in modern superscalar CPUs.

This project will also closely integrate the Hive execution engine with the ORC file format removing the layers of deserialization and type inference. The cost of eager deserialization is expected to be small because filter push down will be supported and the vectorized representation of data will consist of arrays of primitive types as much as possible. We will design a new generic vectorized iterator interface to get a batch of rows at a time, with a column vector for each batch, from the storage layer. This will be implemented first for ORC, but we envision that it will later be implemented for other storage formats like text, sequence file, and other columnar formats. This will enable the CPU-saving benefits of vectorized query execution on these other formats as well.

The following sections describe the scope of changes and key design choices in this project.

Basic Idea

The basic idea is to process a batch of rows as an array of column vectors. A basic representation of vectorized batch of rows can be as follows:

```
class VectorizedRowBatch {
  boolean selecetedInUse;
  int [] selected;
  int size;
```

```
    ColumnVector [] columns;
  }
```

The selected array contains the indexes of the valid rows in the batch and size represents the number of valid rows. An example of a ColumnVector is as following:

```
class LongColumnVector extends ColumnVector {
   long [] vector;
}
```

Now addition of a long column with a constant can be implemented as following:

```
class LongColumnAddLongScalarExpression {
   int inputColumn;
   int outputColumn;
   long scalar;
   void evaluate(VectorizedRowBatch batch) {
     long [] inVector =
          ((LongColumnVector)batch.columns[inputColumn]).vector;
     long [] outVector =
          ((LongColumnVector) batch.columns[outputColumn]).vector;

     if (batch.selectedInUse) {
       for (int j = 0; j < batch.size; j++) {
         int i = batch.selected[j];
         outVector[i] = inVector[i] + scalar;
       }
     } else {
       for (int i = 0; i < batch.size; i++) {
         outVector[i] = inVector[i] + scalar;
       }
     }
   }
}
```

This approach of processing facilitates much better utilization of instruction pipelines in modern CPUs, and thus achieving high instructions per cycle. Also, processing of the whole column vector at a time leads to better cache behavior. It can also be noticed that there is no method call in the inner loop. The evaluate method will, however, be a virtual method call but that cost will be amortized over the size of the batch.

Incremental modifications to Hive execution engine.

It is important that we can make incremental changes to Hive with intermediate releases where the system works partially with the vectorized engine. Therefore, we will start with a small set of operators, expressions and data types, which will suffice to support some simple queries. The set will be increased over subsequent releases to support more queries.

Currently there is no plan to mix vectorized and non-vectorized operators for a single query. The code path will bifurcate one way or the other depending on whether entire query can be supported by new vectorized operators or not. In the future we may add an adapter layer to convert from vectorized execution back to row-at-a-time execution to further broaden the set of queries that can benefit from execution. E.g. an adapter above a vectorized group-by operator could convert to row mode before a sort operator.

Pre-compiled Expressions using templates vs Dynamic Code Generation

In the first release of the project we will implement expression templates that will be used to generate expression code for each expression and operand type combination. The alternative approach is to generate code for expressions and compile it at query runtime. This approach trades off time spent in compilation but promises to generate more efficient operator code.

The static template-based code generation gives most of the benefits of compilation because the inner loop will normally have no method calls and will compile to very efficient code. We will consider dynamic code generation approach in later releases of the project, where we would generate more complex expressions that can be evaluated in a single pass through the inner loop. For example, in our initial approach, (1-x)>0 would require two passes over vectors, one for 1-x, and another for Expr > 0, where Expr is the result of 1-x. In the future, the expression (1-x)>0 could be compiled into the inner loop to apply this expression in a single pass over the input vector. We expect vectorization and template-based static compilation to give most of the benefit. Compiling more complex expression in the inner loop should give less dramatic but noticeable gains.

The expressions in an operator are represented in a tree which the execution engine walks down for each vectorized batch of rows. An alternative model is to flatten the tree into a sequence of expressions using a stack. We will go with the expression tree approach. In the long run we will go with dynamic compilation approach, which will inline all the expressions for an operator and thus alleviating the need for walking the tree or a sequence. Although the tree walk is more overhead than a more tightly-written p-code interpreter with a stack machine, because the inner loop processes a vector of on the order of 1,000 rows at a time, we expect the tree walk overhead will be amortized to a very small portion of total execution time and won't be significant.

Boolean/Filter expressions

The vectorized row batch object will consist of an array of column vectors and also an additional selection vector that will identify the remaining rows in the object that have not been filtered out (refer to the data structure). The filter operators can be very efficiently implemented by updating the selection vector to identify the positions of only the selected rows. This approach doesn't require any additional output column. This also has a benefit of short circuit evaluation of conditions for example a row already filtered out need not be processed by any subsequent expression.

However, the expressions that evaluate to Boolean can be used in a where clause or also in projection. In a where clause they are used as filters to accept rows that satisfy the Boolean condition, while in a projection their actual value (true or false) is needed. Therefore, we will implement two versions of these expressions: a filter version that just updates the selection vector and a projection version that will generate the output in a separate column.

AND, OR implementation (short circuit optimization)

In this section we cover some of the interesting expressions. The goal is to use in-place filtering of the selection vector as much as possible, and also use short circuit evaluation. In place filtering, in this context, means to modify the selection vector in the Vectorized Row Batch and there is no need to store any intermediate output.

'And' expression in filter is very simple. It will always have two child expressions each evaluating to Boolean. A filter 'and' can be implemented as a sequential evaluation of its two child expressions where each is a filter in itself. There is no logic needed for the 'and' expression itself apart from evaluating its children.

'Or' expression can be implemented as follows. Assume it has a left child c1 and a right child as c2 and the selection vector in the input is S. First c1 is evaluated that will give a selection vector S1. Then we calculate S2=S-S1 (set operation). S2 represents the rejections of c1. We pass S2 as the selected vector for c2 so that c2 only processes rows in S2. Now c2 returns S3. The resulting selection vector of the 'Or' expression is S1 U S3, where S1 represents rows selected by left child expression and S3 are the rows selected by right child expression.

Special handling of intermediate selection vectors is planned. They will be data members of the OR operator in the tree, allocated once, at tree creation time. This will minimize memory allocation/deallocation overhead. If we ever go to a multi-threaded approach for query execution, the predicate trees will need to be copied so there is one per thread, to avoid incorrect concurrent access to these vectors. Initially, the code will be single-threaded, so this is not an issue. An alternative design would be to put these intermediate selection vectors in the vectorized row batch itself, like we will for arithmetic expressions (as described below), although we don't plan to do that at this time.

Storage of intermediate results of arithmetic expressions

Arithmetic operations, and logical operations used to produce results to return (as opposed to just as filters), must store their results for later use. Columns will be added to the vectorized row batch data structure for all results that must be operated on later (e.g. aggregated), or output in the query result. Expression results will be saved directly to these columns. Some expressions also need temporary data to store, for example OR expression with short circuit optimization (section 5). This intermediate data will be local to the expression and is not used beyond the scope of the expression. Hence, if we later go to a multi-threaded execution implementation in the same process address space, the expression trees will need to be copied so each thread has a dedicated tree.

As an example, consider this query:

Select sum(b+c+d) from t where a = 1;

The expression b+c+d will be identified, say as Expr0, at compile time, and the vectorized row batch object will have in it the columns (a, b, c, d, Expr0).

As a second example, consider this query:

Select sum(d) where a+b+c>100;

Compile time analysis will determine that the vectorized row batch object must also contain columns (a, b, c, d, Expr0). During execution, (a+b) will be computed as an intermediated result in vector Expr0 in the vectorized row batch. Then (a+b)+c will be calculated from Expr0 and c and also stored in intermediate vector Expr0 in the expression tree. Then, this intermediate vector Expr0 will be compared to 100 to filter the batch.

An open issue is whether we should re-use intermediate result vectors, as we did in the above example, or use a specific column for each intermediate result. Reusing intermediate result column vectors will save memory and give better cache behavior, compared with using another intermediate vector to hold the final result. This becomes more significant for long chains of operators like (a+b+…+z). We will need an optimization algorithm to cause the same intermediate vectors to be re-used in multiple places in the operator tree.

One basic approach is to re-use the output vector from the child expression. For example, if expression E1 has two child expressions E2 and E3. Any output of E2 and E3 will be used only in the evaluation of E1 and never again, therefore the output column for E2 can be re-used as the output column for E1, and output column for E3 can be marked as available for subsequent expressions. Assuming that expressions are evaluated in depth first manner (post order) i.e. first children expressions are evaluated and then the expression itself based on the outcome of children, we can devise the following algorithm.

Assume output columns are already allocated up to index k and available output columns are k+1, k+2 and so on. Suppose E1 is an expression with E2 and E3 as

children where E2 and E3 are leaf expressions of the tree. Our column allocator routine will traverse the expression tree in post order manner. It will first allocate for E2 and then for E3 and then for E1 itself. Column k+1 will be allocated for E2 and column k+2 will be allocated for E3. But now column k+1 will be allocated for the parent E1 as well and k+2, k+3 onwards will be marked as available. Thus the sibling of E1 will get k+2. This algorithm easily handles any number of children. This approach will result in O(logn) number of intermediate columns where n is the number of nodes in the expression tree.

One limitation of this algorithm is that it assumes a tree of expressions and will not work if expressions are re-used (e.g. common sub-expression optimization) resulting in a DAG like expression graph.

Data type handling

In the first phase of the project we support only TINYINT, SMALLINT, INT, BIGINT, DOUBLE, FLOAT, BOOLEAN, STRING and TIMESTAMP data types. We can use LongColumnVector for all integer types, Boolean and TimeStamp as well. LongColumnVector uses an array of long internally. This approach lets us re-use lots of code and reduces the number of classes generated from the template code. Similarly we use DoubleColumnVector for both double and float data types. BytesColumnVector will be used for STRING, and later, for BINARY when that type is added.

BOOLEAN shall be implemented internally within a LongColumnVector with a long integer value of 0 for false and 1 for true.

TIMESTAMP shall be implemented within a LongColumnVector with a long integer equal to the number of nanoseconds since the epoch, midnight Coordinated Universal Time (UTC), 1 January 1970. The maximum value that can be represented with this scheme is 2^63-1 nanoseconds, or 292 years, after the epoch, giving plenty of range. If dates before 1970 are supported, as proposed in HIVE-4525, a negative number of nanoseconds will be used to represent the number of nanoseconds before the epoch. This gives a date range of 1970 ± 292 years, more than adequate for almost all use cases. For dates outside the maximum range, vectorized query execution will fail, throwing an exception in the storage layer vectorized iterator. The value Long.MIN_VALUE (-2^63) shall be reserved for future use to indicate data outside the standard range. In the future, if desired, it will then be possible to store out-of-range date/time data in another place rather than the standard long column vector, and be able to fall back on another mechanism for processing it.

STRING types shall be represented as sequences of UTF-8 characters in a BytesColumnVector.

Null Handling

Each column vector will also contain an array of Boolean that will mark the index of rows with null values for that column. The expression code will also have to handle the null values in the columns. There are many useful optimizations possible for nulls.

- If it is known that column doesn't contain any null value, we can avoid a null check in the inner loop.
- Consider a binary operation between column c1 and column c2, and if any operand is null the output is null. If it is known that c2 is never null then the null vector of the output column will be same as the null vector of the input column c1. In such a situation, it is possible to just copy the null vector from c1 to output and skip the null check in the inner loop. It may also be possible to get away with just the shallow copy of the null vector.

Vectorized operators

We will implement vectorized versions for the current set of operators. The vectorized operators will contain vectorized expressions; they will take vectorized row batch as an input and will not use object inspectors to access columns.

Filter operator

The filter operator will consist of just the filter condition expression, which will be an in-place filter expression. Therefore, once this condition is evaluated the data would already be filtered and will be passed to the next operator.

Select operator

The select operator will consist of list of expressions that need to be projected. The operator will be initialized with the index of the output column for each child expression. The vectorized input will be passed to each expression in the list, which will populate the output columns appropriately. The select operator will output a different vectorized row batch object, which will consist of the projected columns. The projected columns would just refer the appropriate output columns in the initial vectorized row batch object. The select operator will have a pre-allocated vectorized row object that will be used as output.

Optimization of handling of non-null and repeating data

To reduce overhead for column vector handling during query execution, optimizations are supported for the case of all data in a vector being non null or repeating (i.e. all the same).

noNulls

The noNulls flag setting equal to true is used to optimize the common case where a column has no nulls, or at least nulls are very rare. It enables query execution to reduce overhead for NULL value handling to a minimal amount, spending O(1) time per vector on NULL handling.

If noNulls is true for a column vector, then it is guaranteed that all values in the vector are not null. In this case, the isNull[] array should never be examined in search of meaningful data. However, the isNull array is always allocated and available, so examining it will not fail.

If noNulls is false for a column vector, there may or may not be nulls in the column vector. noNulls == false *does not imply that there is at least one NULL in the vector.*

isRepeating

The isRepeating flag, when set to true, optimizes handling of repeating data, such as for partitioning columns or columns with long runs of values that are all equal. It enables such columns to be handled in O(1) time per vector.

If isRepeating is true for a column vector, then the entry 0 in the value vector shall be set. In this case, the entire vector logically contains the same value that is in position 0. In this case, the entries other than 0 should not be examined during query execution.

If isRepeating is set and noNulls is false, then the entry 0 in the isNull array may be true or false. isRepeating == true and noNulls == false *does not imply that all values are NULL in the vector.* In this calse, if isNull[0] is false, the vector is logically a repeating sequence of the entry in position 0 of the value array.

A repeating value of NULL for the whole vector is signified by setting isRepeating to true, noNulls to false, and isNull[0] to true.

Expression and Filter Evaluation Design and Implementation Notes

Here we provide detailed discussion of some specific designs and implementations of filter and expression handling.

Filter condition expressions

LIKE and REGEXP expressions:

LIKE and REGEXP expressions find any strings fitting a pattern. They compile a pattern on creation, and find matching strings on evaluation.

Both kinds of expression use the Java regular expression package. REGEXP expressions use the package as it is. But LIKE expressions have a different grammar, so they need conversion. “%” is converted to “.*” and “_” is converted to “.”. The class AbstractFilterStringColLikeStringScalar defines common behaviors.

FilterStringColLikeStringScalar class and FilterStringColRegExpStringScalar class implement differences.

There are simple and frequently used patterns; such as prefix match, suffix match, middle match, exact match, and phone numbers. There are optimized implementations for them. They evaluate using byte arrays directly to avoid UTF-8 decoding load.

Partition Support

The current implementation of Vectorized Execution supports partitions. Partition support is added into the input formatter layer as eager deserialization is done in this layer. VectorizedRowBatchCtx is the class that handles the creation of the Vectorized batch, deserialization, and the addition and update of partition columns. Below is the code flow of how the partition columns are added for ORC file format.

1) VectorizedOrcInputFormat is fed the input split to process.
2) VectorizedOrcInputFormat creates the VectorizedOrcRecordReader to read data from this spilt.
3) During the creation of the VectorizedOrcRecordReader, VectorizedRowBatchCtx is created on the input split.
4) Based on the input split, VectorizedRowBatchCtx infers from Hive the configuration the partition specification for that split and creates a Hashmap of partition columns, and two object inspectors, as follows(VectorizedRowBatchCtx::init()).
    - Raw row object inspector - an object inspector for just the data in the row. This is the data that is read from the input split (partition columns are not part of this data).
    - Row object inspector – a union object inspector the combines the raw row object inspector and the partition columns object inspector.
5) The first time when the batch is populated, a call to VectorizedOrcRecordReader::next() adds partitions columns to the batch. Since partitions are constant for a given split and for a given split the record reader does not change, partition columns are added only once to the batch for the whole split, saving time.

Vectorized Iterator

Loading data in batches

The vectorized iterator requires each VectorizedRowBatch object (batch) be loaded with data in the following way by the storage layer iterator when the “next()”

operation is invoked. The current reader for ORC is enhanced to do this. All the tree readers in ORC RecordReaderImpl have an additional method called nextVector() that reads the designated column. StructTreeReader calls these column readers one at a time and populates the row batch which is then returned to the input formatter.

Implementations of a vectorized iterator follows these guidelines.

- Implement the transfer of data from the storage layer to vectorized row batches as efficiently as possible, since otherwise it may dominate total query CPU usage. Avoid excessive function calls or unnecessary data copying and translation. Avoid allocating new objects as much as possible.
- Set the size field size for the batch to the number of rows in the batch.
- Each batch shall be loaded completely with values (with VectorizedRowBatch.defaultBatchSize elements) as much as possible. At the tail end of a storage unit, there may be partially loaded batches with the remainder of values that don't fill a full batch.
- If all the all the values in a column are known in advance to be not null (e.g. because the schema or file header says there are no nulls), set noNulls for that column vector to true. Otherwise set it to false.
- The isNull array value in a column vector will be set to true if a value is null, and otherwise false. If the noNulls value for the vector is set to true, then the isNull array does not need to be changed – the values will be ignored. To improve performance in this situation, do not set individual isNull entries.
- If an individual value is set to null, the corresponding actual value in the vector shall be set to 1 for non-floating-point integer-based values (including all integer types, boolean and timestamp), the minimum positive non-zero value possible for decimal(p,s) (e.g. 0.01 for decimal(18,2)), 1 for floating-point or unlimited-precision decimal, and Double.NaN for floating point values. This is because arithmetic operations may be performed (and results ignored) even if values are null, and this will minimize the chance of a zero-divide error or overflow. For all other data types, do not set the value entry if the value is null. The query execution system will always check if the value is NULL before operating on it, and will not examine it if it is NULL.
- If it can be efficiently determined that every value in a column vector will be the same, set isRepeating to true for the column and put the repeating value in vector position 0. This is used to speed up query execution for repeating values. Use this when feasible, e.g. if the column is a partitioning column and thus every value in the batch column must be the same, or if run-length encoding is used and it is efficient to determine that the whole vector is thus the same. If isRepeating is true for a column vector then all entries other than 0 will be ignored (both isNull and value entries), and for efficiency, they should not be set.
- Expression columns may exist in the batch and they are to be filled later during query execution. They do not need to be modified at all by the storage iterator. This means no fields should be changed, including data value vector entries, null vector entries, isRepeating, and noNulls. Only columns holding data from the

storage layer must be filled. Query execution operators are solely responsible for setting output column contents.
- Set selectedInUse to false. Do not modify the selected array.
- Set EOF to false (unless this is an empty batch and there is no data left).
- SARGs (storage-layer filters) shall be applied to input rows before loading data into the batch. Rows that don't qualify against a SARG shall never be added to a batch.
- numCols shall be set by the caller to the next() method of the iterator and is not to be modified.
- The VectorizedRowBatch.cols array will be allocated and the column vectors created before the first call to next() and the objects shall be directly modified. Don't create new column vector objects and assign them to this array. Reuse (re-loading) of the same objects gives significant performance improvement.

Vectorized Aggregates

The VectorGroupByOperator implements the vectorized variant of the row-mode GroupByOperator. Similar, to the row-mode operator, the vectorized operator separates the algorithm for computing the aggregates from the state needed by these aggregates. Classes derived from VectorAggregateExpression handle the algorithms for evaluating the input batch and computing the aggregate values. The AggregationBuffer classes nested inside subclasses of VectorAggregationExspression store the state needed by the aggregates to accumulate the value. Only hash aggregates are implemented, and the VectorHashKeyWrapper class is used as a hash key in looking up the appropriate aggregation buffer for each key. The VectorHashKeyWrapperBatch class is responsible for efficient vectorized mode evaluation of GROUP BY keys. The VectorAggregationBufferRow class stores all the aggregation buffers for a particular row (key combination). VectorAggregationBufferBatch class has a reference to each individual VectorAggregationBufferRow (and hence individual key value) present in an input batch. The role of VectorAggregationBufferBatch is to detect and optimize the case when a batch contains only one key for all rows and also to isolate/streamline the hash probe operations.

The VectorGroupByOperator class orchestrates the interaction between all these pieces.

Code Flow

During query initialization the VectorGroupByOperator compiles a VectorHashKeyWrapperBatch instance. This pre-allocates a vectorized batch-size array of individual VectorHashKeyWrapper objects. These will be reused for the whole duration of the query.

For each input vectorized row batch the operations are:

- The VectorHashKeyWrapperBatch evaluates the batch. This calls evaluate on every individual vectorized key expression follow by a copy-out of the key values from the vector columns into fields of the objects in the VectorHashKeyWrapper array. String values are copied by reference. The VectorHashKeyWrapperBatch class has specialized optimized implementations of this copy-out according to the input vector row batch characteristics: has selection vector, columns have nulls, columns are repeating.
- Hash codes are computed for each individual VectorHashKeyWrapper.
- Each VectorHashKeyWrapper is probed in the GROUP BY hash map (implemented with java.util.Map). If the key is not found then the VectorHashKeyWrapper is cloned into an immutable copy and this immutable copy is added to the hash map. The VectorAggregationBufferRow for this key (a newly allocated one if the key was just added to the hash) is added to the VectorAggregationBufferBatch . This is when VectorAggregationBufferBatch uses a versioning scheme to stamp the VectorAggregationBufferRow with the current version (basically the row batch number). This allows the VectorAggregationBufferBatch to detect if the same VectorAggregationBufferRow is added again within the same input batch and thus be able to count the number of distinct keys in the batch without resorting to a separate per-batch hash map.
- The vectorized aggregates are evaluated. If the VectorAggregationBufferBatch counted only one distinct key in the input batch then an optimized path is used by calling VectorAggregateExpression.aggregateInput. Otherwise the more expensive VectorAggregateExpression.aggregateInputSelection method is invoked.

The aggregate values are collected when the query is closed. At the time of writing, the VectorGroupByOperator emits row mode output identical to the row-mode GroupByOperator. This is because certain aggregates (AVG, STD, VARIANCE) output structures and we do not have yet support for structures in the VectorizedRowBatch.

The row-mode GroupByOperator is capable of spilling if the hash grows too big by simply emitting the collected aggregates and removing them from the hash. Due to the M/R split of the GROUP BY implementation in Hive this intermediate spill is correctly aggregated by the reduce side of the operation into a final correct result. The equivalent functionality for the VectorGroupByOperator is tracked in HIVE-4612.

Future Considerations

This implementation is expected to be efficient because it minimizes storage allocation costs and takes advantage of regular structure in the input vectors. However, partly to minimize work in the initial implementation, the code does rely

on the existing Java hash map for hash function calculation and hash table management. Prior work on hash aggregate and join [Zukowski 2006] has recommended vectorized hash function calculation and use of Cuckoo hashing to limit CPU instruction count and improve cache hit rate and cycles per instruction. A performance measurement is recommended in the future to see if it is worthwhile to use these techniques in the vectorized query execution code in Hive for both hash aggregate and hash join.

Vectorized User-Defined Function Adaptor

UDFs are commonly used in Hive and it's important that using UDFs does not prevent the ability to benefit from vectorized query execution. An adaptor has been implemented to support custom user-defined scalar functions in vectorized execution mode. There are two types of custom UDFs, standard (legacy) UDFs, and generic UDFs [Capriolo 2012]. Both types of UDFs are supported by the adaptor. All UDFs are supported that operate only on the set of scalar types supported by vectorization, and return only one of those types. The user doesn't have to take any special action for custom UDFS to work in vectorized mode. They just work.

The vectorized UDF adaptor operates on a vectorized row batch, so that for each row in the batch that still qualifies, the necessary values are extracted from a row in the batch, and assembled into an array of objects. This may also include preparation of constant argument objects if some of the function's arguments are constant. The UDF is then invoked on the array of objects via the standard mechanism, using appropriate object inspectors prepared once at plan generation time. An object is then retrieved containing the function result. This object is then converted back into a primitive type (long, double, or byte array) and copied into the output column of the VectorizedRowBatch object in the current row position.

If the function result is a null object reference, then by convention the function result is a SQL NULL value. NULLs are handled correctly by setting the isNull array Boolean value in the output column vector to true for the row when the result is NULL, and ensuring that noNulls is false for the output vector.

This adaptor implementation will usually not perform nearly as fast as a hand-coded implementation of a function or operator with a tuned inner loop that avoids data conversion and function call overhead. Nevertheless, it still can allow queries to run faster than the traditionally row mode. Some ad hoc testing indicated that roughly 30% speedups are still possible compared to row mode execution over ORC files even if the UDF must be called for every row. In cases where streamlined, built-in vectorized filters are called first, and then UDFs are called, nearly the full benefit of vectorization can be obtained. This is because the UDF adaptor will only be invoked for the small percentage of rows that pass the initial filter.

For example, for best performance, the user should normally write this:

```
select …
from mytable
where c1 = 1 and myfunc(c2) > 0
```

Instead of this:

```
select …
from mytable
where myfunc(c2) > 0 and c1 = 1;
```

This assumes that the filter c1 = 1 is reasonably selective. Since vectorized execution uses short-circuit evaluation of AND, myfunc() will only be evaluated using the adaptor if the row passes the test c1 = 1.

The vectorized UDF adaptor is implemented primarily in package org.apache.hadoop.hive.ql.exec.vector.udf in the classes VectorUDFAdaptor and VectorUDFArgDesc, as well as changes to VectorizationContext and a few other miscellaneous small changes.

In the future, we'll consider adding support that allows users to add their own high-performance vectorized UDFs by hand-coding them as subclasses of VectorExpression. For now, users must use legacy or generic UDFs that will be invoked by the vectorized UDF adaptor.

References

[Boncz 2005] Peter Boncz et al., *MonetDB/X100: Hyper-Pipelining Query Execution,* Proceedings of the CIDR Conference, 2005.

[Capriolo 2012] Edward Capriolo et al., *Programming Hive,* Ch. 13, Functions. O'Reilly, 2012.

[Zukowski 2006] Marcin Zukowski et al., *Architecture-Conscious Hashing,* Proceedings of the 2nd Intl. Workshop on Data Management on New Hardware, June 25, 2006.