FilterPushdownDev

Filter Pushdown

- Filter Pushdown
  - Introduction
  - Use Cases
  - Components Involved
  - Primary Filter Representation
  - Other Filter Representations
  - Filter Passing
  - Filter Collection
  - Filter Decomposition

Introduction

This document explains how we are planning to add support in Hive's optimizer for pushing filters down into physical access methods. This is an important optimization for minimizing the amount of data scanned and processed by an access method (e.g. for an indexed key lookup), as well as reducing the amount of data passed into Hive for further query evaluation.

Use Cases

Below are the main use cases we are targeting.

- Pushing filters down into Hive's builtin storage formats such as RCFile
- Pushing filters down into storage handlers such as the HBase handler (http://issues.apache.org/jira/browse/HIVE-1226)
- Pushing filters down into index access plans once an indexing framework is added to Hive (http://issues.apache.org/jira/browse/HIVE-417)

Components Involved

There are a number of different parts to the overall effort.

1. Propagating the result of Hive's existing predicate pushdown. Hive's optimizer already takes care of the hard work of pushing predicates down through the query plan (controlled via configuration parameter **hive.optimize.ppd=true/false**). The "last mile" remaining is to send the table-level filters down into the corresponding input formats.
2. Selection of a primary filter representation to be passed to input formats. This representation needs to be neutral (independent of the access plans which will use it) and loosely coupled with Hive (so that storage handlers can choose to minimize their dependencies on Hive internals).
3. Helper classes for interpreting the primary representation. Many access plans will need to analyze filters in a similar fashion, e.g. decomposing conjunctions and detecting supported column comparison patterns. Hive should provide sharable utilities for such cases so that they don't need to be duplicated in each access method's code.
4. Converting filters into a form specific to the access method. This part is dependent on the particular access method; e.g. for HBase, it involves converting the filter condition into corresponding calls to set up an HBase scan object.

Primary Filter Representation

To achieve the loosest possible coupling, we are going to use a string as the primary representation for the filter. In particular, the string will be in the form produced when Hive unparses an `ExprNodeDesc`, e.g.

```
((key >= 100) and (key < 200))
```

In general, this comes out as valid SQL, although it may not always match the original SQL exactly, e.g.

```
cast(x as int)
```

becomes

```
UDFToInteger(x)
```

Column names in this string are unqualified references to the columns of the table over which the filter operates, as they are known in the Hive metastore. These column names may be different from those known to the underlying storage; for example, the HBase storage handler maps Hive column names to HBase column names (qualified by column family). Mapping from Hive column names is the responsibility of the code interpreting the filter string.

Other Filter Representations

As mentioned above, we want to avoid duplication in code which interprets the filter string (e.g. parsing). As a first cut, we will provide access to the `ExprNodeDesc` tree by passing it along in serialized form as an optional companion to the filter string. In followups, we will provide parsing utilities for the string form.

We will also provide an IndexPredicateAnalyzer class capable of detecting simple sargable subexpressions in an `ExprNodeDesc` tree. In followups, we will provide support for discriminating and combining more complex indexable subexpressions.

```
public class IndexPredicateAnalyzer
{
  public IndexPredicateAnalyzer();

  /**
 * Registers a comparison operator as one which can be satisfied
 * by an index search.  Unless this is called, analyzePredicate
 * will never find any indexable conditions.
   *
 * @param udfName name of comparison operator as returned
 * by either {@link GenericUDFBridge#getUdfName} (for simple UDF's)
 * or udf.getClass().getName() (for generic UDF's).
   */
  public void addComparisonOp(String udfName);

  /**
 * Clears the set of column names allowed in comparisons.  (Initially, all
 * column names are allowed.)
   */
  public void clearAllowedColumnNames();

  /**
 * Adds a column name to the set of column names allowed.
   *
 * @param columnName name of column to be allowed
   */
  public void allowColumnName(String columnName);

  /**
 * Analyzes a predicate.
   *
 * @param predicate predicate to be analyzed
   *
 * @param searchConditions receives conditions produced by analysis
   *
 * @return residual predicate which could not be translated to
 * searchConditions
   */
  public ExprNodeDesc analyzePredicate(
    ExprNodeDesc predicate,
    final List<IndexSearchCondition> searchConditions);

  /**
 * Translates search conditions back to ExprNodeDesc form (as
 * a left-deep conjunction).
   *
 * @param searchConditions (typically produced by analyzePredicate)
   *
 * @return ExprNodeDesc form of search conditions
   */
  public ExprNodeDesc translateSearchConditions(
    List<IndexSearchCondition> searchConditions);
}

public class IndexSearchCondition
{
  /**
 * Constructs a search condition, which takes the form
 * <pre>column-ref comparison-op constant-value</pre>.
   *
 * @param columnDesc column being compared
   *
 * @param comparisonOp comparison operator, e.g. "="
```

```
  * (taken from GenericUDFBridge.getUdfName())
    *
  * @param constantDesc constant value to search for
    *
  * @Param comparisonExpr the original comparison expression
    */
   public IndexSearchCondition(
     ExprNodeColumnDesc columnDesc,
     String comparisonOp,
     ExprNodeConstantDesc constantDesc,
     ExprNodeDesc comparisonExpr);
}
```

Filter Passing

The approach for passing the filter down to the input format will follow a pattern similar to what is already in place for pushing column projections down.

- `org.apache.hadoop.hive.serde2.ColumnProjectionUtils` encapsulates the pushdown communication
- classes such as `HiveInputFormat` call `ColumnProjectionUtils` to set the projection pushdown property (READ_COLUMN_IDS_CONF_STR) on a jobConf before instantiating a `RecordReader`
- the factory method for the `RecordReader` calls `ColumnProjectionUtils` to access this property

For filter pushdown:

- `HiveInputFormat` sets properties `hive.io.filter.text` (string form) and `hive.io.filter.expr.serialized` (serialized form of ExprNodeDesc) in the job conf before calling getSplits as well as before instantiating a record reader
- the storage handler's input format reads these properties and processes the filter expression
- there is a separate optimizer interaction for negotiation of filter decomposition (described in a later section)

Note that getSplits needs to be involved since the selectivity of the filter may prune away some of the splits which would otherwise be accessed. (In theory column projection could also influence the split boundaries, but we'll leave that for a followup.)

Filter Collection

So, where will `HiveInputFormat` get the filter expression to be passed down? Again, we can start with the pattern for column projections:

- during optimization, `org.apache.hadoop.hive.ql.optimizer.ColumnPrunerProcFactory's` `ColumnPrunerTableScanProc` populates the pushdown information in `TableScanOperator`
- later, `HiveInputFormat.initColumnsNeeded` retrieves this information from the `TableScanOperator`

For filter pushdown, the equivalent is `TableScanPPD` in `org.apache.hadoop.hive.ql.ppd.OpProcFactory`. Currently, it calls `createFilter`, which collapsed expressions into a single expression called condn, and then sticks that on a new `FilterOperator`. We can call condn.getExprString() and store the result on `TableScanOperator`.

Hive configuration parameter `hive.optimize.ppd.storage` can be used to enable or disable pushing filters down to the storage handler. This will be enabled by default. However, if `hive.optimize.ppd` is disabled, then this implicitly prevents pushdown to storage handlers as well.

We are starting with non-native tables only; we'll revisit this for pushing filters down to indexes and builtin storage formats such as RCFile.

Filter Decomposition

Consider a filter like

```
x > 3 AND upper(y) = 'XYZ'
```

Suppose a storage handler is capable of implementing the range scanfor `x > 3`, but does not have a facility for evaluating {{upper(y) = 'XYZ'}}. In this case, the optimal plan would involve decomposing the filter, pushing just the first part down into the storage handler, and leaving only the remainder for Hive to evaluate via its own executor.

In order for this to be possible, the storage handler needs to be able to negotiate the decomposition with Hive. This means that Hive gives the storage handler the entire filter, and the storage handler passes back a "residual": the portion that needs to be evaluated by Hive. A null residual indicates that the storage handler was able to deal with the entire filter on its own (in which case no `FilterOperator` is needed).

In order to support this interaction, we will introduce a new (optional) interface to be implemented by storage handlers:

```
public interface HiveStoragePredicateHandler {
  public DecomposedPredicate decomposePredicate(
    JobConf jobConf,
    Deserializer deserializer,
    ExprNodeDesc predicate);

  public static class DecomposedPredicate {
    public ExprNodeDesc pushedPredicate;
    public ExprNodeDesc residualPredicate;
  }
}
```

Hive's optimizer (during predicate pushdown) calls the decomposePredicate method, passing in the full expression and receiving back the decomposition (or null to indicate that no pushdown was possible). The `pushedPredicate` gets passed back to the storage handler's input format later, and the `residualPredicate` is attached to the `FilterOperator`.

It is assumed that storage handlers which are sophisticated enough to implement this interface are suitable for tight coupling to the `ExprNodeDesc` representation.

Again, this interface is optional, and pushdown is still possible even without it. If the storage handler does not implement this interface, Hive will always implement the entire expression in the `FilterOperator`, but it will still provide the expression to the storage handler's input format; the storage handler is free to implement as much or as little as it wants.