Design Doc

Nested Column Pruning in Hive

Background

Hive supports multiple columnar formats, such as ORC and Parquet. These formats store data for different columns, including those nested, separately. Therefore, if only one such column is needed, they can skip all the rest and only scan that particular column. This improves the scanning speed dramatically.

However, at the moment if a Hive user has the following query:

```
SELECT s.a FROM tbl
```

And suppose the column `s` is a struct type with multiple fields, including `a`. Then, at runtime Hive still needs to scan the whole struct for `s`. In other words, it needs to scan all other fields besides `a`, which is unnecessary and inefficient.

Currently, Hive already implemented the projection pushdown for columns. The nested column pruning proposed in this doc adds further granularity to that optimization.

The upstream work is tracked by HIVE-15055, and also PR.

Implementation

The implementation consists of two parts: **compile time optimization and runtime optimization**. We shall talk about each in the following.

Compile time Optimization

This part is basically inline with Hive's current projection pushdown implementation, which only operate on the whole column level.

For instance, in this query:

```
SELECT a.f FROM tbl WHERE a.g = 42
```

suppose the table `tbl` contains columns `a`, `b`, `c`, while `a` is of struct type with fields `f`, `g`, `h`. At the moment, Hive filters out columns `b` and `c`, and only scan data in column `a`. However, it will still read all fields in column `a`, including the unused columns `h`.

On the implementation side, all the work is done in class `ColumnPruner`, `ColumnPrunerProcCtx` and `ColumnPrunerProcCtxFactory`. The algorithm essentially traverse the operator tree in a bottom-up fashion, and collects the needed columns for each operator. While traversing up, it combines the used columns in children operators with the current operator, and update a map called `prunedColLists` inside `ColumnPrunerProcCtx`. This maps each operator to a list of strings (column names). When the algorithm reaches the top-level table scan operator, it gathers the columns from all its children and populate the info to the conf for the operator, which is then **used later at the runtime for reading different file formats.**

For the nested column pruning enhancement, the basic algorithm is the same as above. However, instead of maintaining a map like above, we can change it to a map from operator to a list of **FieldNode**. The definition:

```
Class FieldNode {
  String fieldName;
  List<FieldNode> children;
}
```

Essentially, a FieldNode can be roughly treat as a *subtree* in a struct type. For instance, for the type:

```
s:struct<
  f1:struct<f2:int>,
  f3:struct<f4:boolean, f5:string, f6: bigint>,
  f7:double
>
```

An instance of FieldNode could represent the paths `s.{f1, f3.f4}`, which means in this struct `s`, only the field `f1` and the subfield `f4` inside field `f3` are used.

FieldNodes for the same column need to be merged. For example, if both `s.f3.f4` and `s.f3` are used, then we should merge and produce `s.f3`, which covers both. In another case, if both `s.f3.f4` and `s.f3.f6` are used, the merged result should be `s.f3.{f4,f6}`.

The new algorithm is very similar to the old one. It works from bottom up and collect the needed columns at each operator. One exception is when processing a `ExprNodeDesc`. In the old algorithm we only generate a column name from this, but in the new algorithm we need to a case analysis, and in case that is a `ExprNodeFieldDesc`, generate a entire path from the

column name to the eventual field name. The info is then populated to the configuration, same as before.

Run time Optimization

During runtime, the above saved nested column information is used in two places:

1. **When initializing RecordReader for a particular file format**. This is done in the same way as the old projection pushdown. Before calling `getRecordReader`, Hive first applies projection pushdown, which read the needed columns from table scan operators, serialized them into strings, and then populated into job conf. For nested columns, we create a separate property in the job conf and do the same above. These properties are then used in the `getRecordReader` function for the specific file format, to generate a pruned-version of schema that can be passed to the underlying storage. Note, here the assumption is that given a pruned schema, the storage can scan the needed data efficiently. This should apply to columnar formats such as ORC or Parquet.
2. **When initializing object inspectors for a particular file format** (*This is an awkward design decision as will be explained later*). In Hive, all the object inspectors (OIs) are initialized in `MapOperator`, during the initialization of SerDe for the file format. At this time, same as case 1), we populate the nested column info into the job conf, which then is used to initialize the corresponding SerDe and OI.

Difficulties

There are two difficulties in this implementation which are covered below.

Generated Select Operator

Hive sometimes generate extra select operators and insert them into the existing operator tree. These select operators are equipped with the whole row schema, i.e., all the columns in the row are selected. This makes it hard to differentiate these kind of operators from the “natural” select operators. Imaging a fragment of operator tree like below:

```
  ...
   |
SEL (use a)
   |
FIL (use a.f)
   |
  ...
```

As we go up the tree, we first process the filter operator and record `a.f` for it. Then, we are at the select operator, and trying to merge a with `a.f`. Now what should we do?

It depends on whether the select operator is generated or it is natural. For the latter, the column `a` is specified in the input query and therefore the merged result should be `a`. However, for the former case, `a` is generated by Hive and isn't really needed. In this case the merge result should be `a.f`.

Currently, we add a flag in `ExprNodeColumnDesc` , specifying whether it is generated or not. The merge process is then done according to the value of this field.

Shifting Struct Index

As mentioned above, we need to populate the nested column information to SerDe and use that when initializing OIs for struct. This is due to an issue with shifted struct index in Parquet format. Suppose we are dealing with a simple query:

```
SELECT s.g FROM tbl
```

Where `s` is a struct with 3 fields: `f`, `g` and `h`:

```
s:<struct<f:boolean, g:int, h:string>>
```

The nested column pruning will generate a pruned struct type `s:<struct<g:int>>` and pass that to the underlying Parquet reader. The reader will return writables containing only `g`'s data, which essentially is a `ArrayWritable` with size 1. The data for field `g` is at index 0.

However, on the OI side it is still initialized with the original type info, and so the field `g` still has index 1. Therefore, at runtime the OI will fail when try to read the actual data, as the index doesn't match.

The essential problem here is that SerDe/OI in Hive is a "static" notion, i.e., they are initialized with table and type information, and is not expected to vary with queries. But in this case we need to consider the actual nested columns used in each query and initialize accordingly.

The current solution add an extra parameter to `ArrayWritableObjectInspector` for the Parquet format, which takes a separate pruned type info constructed from the nested columns.