Parquet Filter Pushdown in Drill

1 Motivation

Filter pushdown is one of the most critical factors that might impact Drill performance. As of today, Drill does not have the ability to push filters into the parquet reader (Drill has partition pruning which works similarly in that it also prunes data. However, the targeted use case is different). The lack of parquet filter pushdown means Drill's parquet reader has to read more data than necessary, and pass all the unnecessary data along the operator pipeline, until a filter operator is able to prune those data. That could incur significant I/O, CPU, and network overhead and put Drill's query performance at a significant disadvantage , when Drill is compared with with other query systems, using well known performance benchmark such as TPC-H, TPC-DS.

This proposal tries to fill in the gap with respect to parquet filter pushdown support in Drill. In particular, we try to answer the following questions:

- What kind of filter expression is qualified to be pushed down?
- What level of pruning do we want to have? Row group pruning, data page pruning or row level pruning?
- When should parquet filter evaluation happen? Planning-time vs run-time evaluation.
- How should filter expression be evaluated?
- What kind of statistics can we leverage?

It is expected that with parquet filter pushdown, Drill would see improved query performance, especially when input data is somehow sorted, and the filter could prune a large percentage of data.

2 Requirements

1) **Query result correctness**

Drill should produce the same query result[1] with or without filter pushdown.

2) **Push down as much of the filter expression as possible**.

In general, applying a filter earlier would improve query performance, especially when teh filter is applied at the parquet row group level. As such, it's desired to push down as much as of the filter expression as possible.

Drill uses Apache Calcite as its SQL planner. As a result, the filter expression in Drill execution engine is the one that is validated / transformed by Calcite. For instance, Calcite will transform

[1] It's possible to see different behaviors in scenarios when expressions in the query may hit error without filter pushdown. Applying the filtering earlier may skip such error. That's similar to what happens when short circuit evaluation hit error input.

an IN list predicate into either an OR-ed equal comparison predicate, or a JOIN operator. Drill's execution will not see an IN list predicate.

Case1: Simple predicate connected by 'and'/'or'/'not' operators

SimplePredicate ::= SimpleCompPredicate
| Column 'is' 'null'
| Column 'is' 'not' 'null'

SimpleCompPredicate := Column Comp_Op Constant_Literal

Comp_Op ::= '=' | '<>' | '<' | '>' | '<=' | '>='

Operand ::= Column

Constant_Literal ::= Int_Literal
| BigInt_Literal
| Float_Literal
| Double_Literal
| Date_Literal
| TimeStamp_Literal
| Time_Literal
| Char_Literal

Two conditions for a qualified predicate:

- 'Column' expression has to be a top-level column. It's not allowed to have a complex field[2], such as *colA.b.c = 10* or colA.b[2].c = 20.
- Column expression and Constant_Literal in simple compare predicate have to be same types.

Case2: Extend simple comparison predicate with implicit/explicit cast function.
This case is done by extending case 1 by adding implicit/explicit cast function, due the type difference between LHS and RHS of comparison predicate. In this case, there might be cast() function on either 'Column' side, or 'Constant_Literal'. The cast function might be explicitly specified in the query, or inserted by Drill, to resolve the type difference between two sides. For instance,

BigInt_Column = Int_Literal ⇒ BigInt_Column = cast (Int_Literal as bigint)

Int_Column = BigInt_Literal ⇒ cast(Int_Column as bigint) = BigInt_Literal

[2] Drill currently does not push filter passed ITEM operator, which is used for complex field.

For the second case, where cast function is applied to column, we have to make sure that the min/max range for cast() could be properly calculated from column's min/max.

Case3: Extended simple comparison predicate with functions:
Extend case 2 with a certain set of functions applied to column expression, and allow both sides of comparison operator to contain column expressions.
The following list has some examples of

```
Col1 + Col2  = 100;
Col1 - Col2 > 100;
Col1 * 1000 = 3000;
Col1 / 100.0  = 30.0;
Col + Col2  = Col 3 + 100;
```

min/max calculation rule: A function *f()* in the comparison predicate might be qualified for pushdown, if it meets the requirement of min/max calculation rule.

Given function f(a1, a2, ,,, a_n). Assume a_i is within [min_i, max_i]. Function f(a1, a2, ,,, a_n) meets min/max calculation rule if there exists functions g1, g2, such that

min(f(a1, a2, …, a_n)) = g1 (min_0, max_0, min_1, max_1, ,.., min_n, max_n)
max(f(a1, a2, …, a_n)) = g2 (min_0, max_0, min_1, max_1, ,.., min_n, max_n)

For instance, for '+' function applied on numerical values, we could simply add min/max of its two operands to get the min/max of the "+" function.

3) Filter pushdown overhead

The logic of parquet filter pushdown would certainly introduce additional overhead.

- When the filter prunes large amount of data, the additional overhead normally would be compensated by the benefit from reduced overhead in parquet reader and/or downstream operators. The additional overhead is not a big concern.
- When the filter does not have any pruning at all or only pruning minimum data, the additional overhead is a concern.

For both cases, we should target to get the filter pushdown overhead as minimum as possible.

2.1 Use Cases

Many data has column(s) which shows certain degree of order. For instance, Drill may query data coming from a streaming application, which may have a date_column naturally sorted. For the following query, filter pushdown, especially rowgroup filter pushdown, will significantly prune the data in parquet scanner.

*SELECT * from table_t1*
WHERE date_column between date '2016-01-01' and date '2016-01-31'

If the data is converted into parquet format with the min/max statistics in parquet footer, then Drill can safely discard one parquet row group if the min/max does not have any overlap with the period of '2016-01-01' and '2016-01-31'.

3 Background and Research

1) Partition pruning

Parquet filter pushdown bears similarity to partition pruning in Drill, in that both of them are trying to prune data in advance, before data is flowing through Drill's operator pipeline. There are differences.

- **Value uniqueness**.

Drill partition pruning, whether the partition column is the directory name or the partition column created with CTAS partition statement, requires the partition column has to be unique across the *entire* file, for all the files across the *entire* dataset.

  - If the partition column is the directory name, this condition is naturally meet.
  - For regular column to act as partition column, it has unique value in each parquet file (each file may have multiple row groups). This means if one parquet file, or even one row group within a certain parquet file does not meet this uniqueness requirement, partition pruning will not happen.

Parquet filter pushdown does not have the uniqueness requirement. All it requires is the min/max statistics in row group level, if pruning happens at row group level. In that sense, parquet filter pushdown might be applied in wider range of use cases, than Drill partition pruning.

- **Pruning granularity**

Drill partition pruning for parquet data happens at parquet file level. Parquet filter pushdown, will pruning data at least at parquet row group level.

- **Filter expression complexity**.

Drill partition pruning can support a wider range of functions in the filter expression, as a benefit of imposing the value uniqueness of partition columns. In contrast, parquet filter pushdown will not be able to support all the functions that are supported in Drill partition pruning. We will start with Case 1 in Section 2, where only comparison operator/is (not) null operator. Then we will add cast() or certain functions which meet the *min/max calculation rule* to the filter expression.

2) Prior prototype of parquet filter pushdown.

DRILL-1950 was opened and it has a prototype of parquet filter pushdown support. Couple of enhancements have to be made, before we can merge it into Drill code.

- The pruning happens at planning time, and requires the parquet meta format from footer. Unfortunately, the parquet metadata caching feature, which was merged into Drill code base already, changed the meta format. This breaks the prototype.
- The prototype implicitly assumes the column type and the constant literal type are the same. It may not work correctly when two operands are of different type.
- It may also need handle other types like date/timestamp/time, and a few set of functions in the filter expressions.

This proposal tries to leverage part of code in DRILL-1950 prototype, and goes beyond what had been implemented in DRILL-1950 prototype.

4 Design Overview

1) Row group filter

As a first step, this proposal tries to improve Drill parquet performance through row group level pruning, for the following reasons:

- Row group level filtering is more effective than row-level filtering in many cases, as other SQL query engine systems have experienced. Parquet file is organized in row groups. Naturally, row group filtering makes a lot of sense.
- Some logic implemented in this proposal, such as the planner rule to push down filter expression into Scan, or the filter evaluation logic, could be leveraged later on, if we want to apply page-level filtering or row-level filtering.

2) Execution time pruning

We plan to have planner rule to push the filter from Filter operator to Scan operator, similarly to what happens to Drill partition pruning. However, the filter evaluation and pruning happening in execution time, when parquet reader is being set up.

- This will make filter evaluation executed in distributed mode, and hence reduce the filter evaluation time. This is different if we apply the filter evaluation in planning time, where only foreman node will be part of the evaluation process. The filter evaluation overhead is in particular of a concern, when the filter expression contains more complex functions. In the past, we have seen that the filter evaluation cause long planning-time, especially in case there are ~100k, or even millions of parquet files.
- The parquet meta cache feature puts only min/max statistics when they are equal. Unless we make change to that, for now, it's not feasible to apply the filter evaluation logic in planning time. At execution time, Drill will read the parquet meta data from footer anyway. This will enable Drill to start evaluate the filter expression, and apply the pruning.

3) Filter expression evaluation and expression materialization.

Parquet library has its own API to evaluate a filter expression. The API takes a *FilterPredicate* instance and a list of *ColumnChunkMetaData*. The prototype in DRILL-1950 adopts this strategy. Unless parquet library enhances its API, it seems to lack of support for the following cases:

- Type mismatch between comparison operands. The API assumes that column and constant are of identical types.
- Supports functions in filter expression.

Instead of leveraging parquet library API to evaluate filter expression, we plan to implement a ParquetFilterEvaluator in Drill, which will

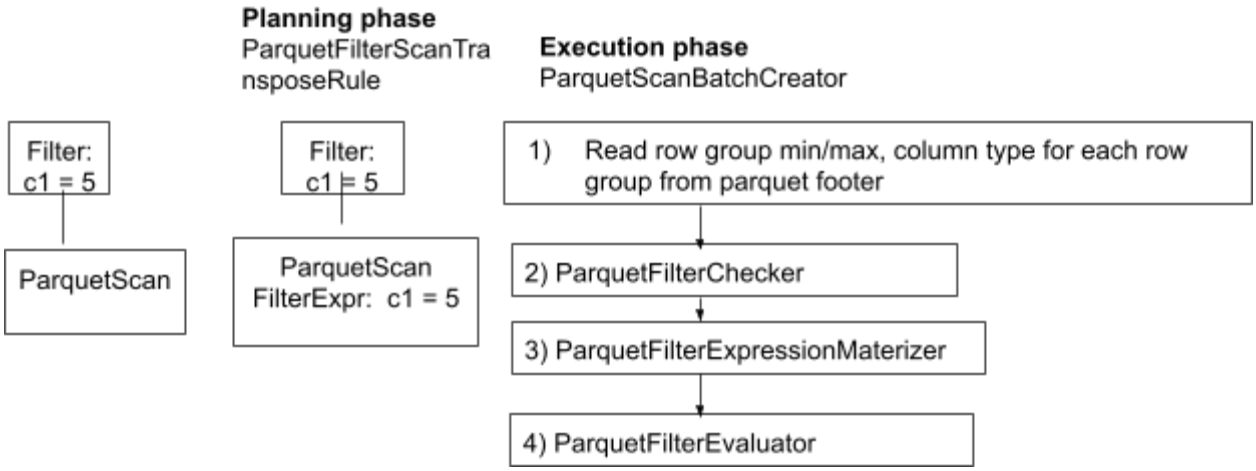
- Support type mismatch by applying implicit type cast.
- Support certain set of functions in filter expression.

Implicit cast normally happens in Drill's expression materialization logic, for operator such as Project or Filter. It's important to apply the same logic of expression materialization for parquet filter evaluation, as what happens in Filter operator. Otherwise, it's very likely that we may see different query results, with or without filter pushdown.

The initial idea is to extend ExpressionTreeMaterializer, to have it accept RangedExpression.

5 Implementation Details

5.1 Algorithms (and Data Structures)



5.2 APIs and Protocols

We will add one new type of logical expression : *RangedExpression*, which keeps the major type, min/max, and number of nulls. During filter expression materiazation, a SchemaPath expression will be replaced by a RangedExpression.

We will add *ParquetFilterChecker,* which is a Visitor over the expression tree, and will check if the filter expression is qualified for be evaluated based on min/max statistics.

5.3 Performance

It's expected that with parquet row group level filter pushdown, we may see noticeable performance improvement for typical workload, such TPC-H, or TPC-DS queries.

It's important to understand that the prerequisite for parquet filter pushdown to be applied is to have the min/max statistics in the row group metadata. As of now, some sample TPC-H dataset is created without having the min/max statistics. We have to convert those parquet dataset, if they do not have the min/max statistics.

We may see some performance degradation, in cases parquet filter pushdown does not lead to any pruning at all; the filter evaluation essentially just brings overhead without any benefit.

We should measure the impact of performance with this feature turned on, if the dataset does not have min/max statistics. This is to make sure the checking the availability of statistics does not cause any noticeable performance degradation if the existing dataset does not have min/max statistics.

5.4 Error and Failure handling

In case filter evaluation hits error, the idea is to skip filter pruning in Parquet Scan operator, and fall back the behavior without filter pruning. This will ensure that parquet filter pushdown feature will not change the query behavior, in case of hitting errors.

5.5 Memory management

ParquetFilterEvaluator may allocate memory within Direct memory pool. However, supposedly the memory footprint should be much smaller than what are required in the real Filter operator.

5.6 Scalability Issues

The planner rule for Parquet filter pushdown should incur minimal overhead, since it's logic is straightforward. One thing we have to make sure is that it will not hit a loop for planner rules, combining with all the existing rules.

We have to measure the overhead of filter evaluation in execution time. If we are able to pruning large percentage of parquet data with filter pushdown, we should see quite significant of improvement in terms of Drill's scalability.

5.7 Debugging

We will add logging information about whether parquet filter pushdown is applied, and how many row groups have been pruned, if any, to help troubleshooting performance issues.

5.8 Testing implications

QA needs to understand what kind of filter expression could be pushed down, and probably also how to check the parquet meta data using parquet metadata tool.

5.15 Tradeoffs and Limitations

As a first step, this proposal tries to apply pruning at row group level. To apply filter pruning at page or row level will not only introduce code complexity, but also filter evaluation overhead. We will have a better idea whether row group level will meet the desired performance goal, once we implement this feature and run performance measurement. If row group level filtering does not meet the goal, we will further consider extending to page or row level filtering.

We may consider moving the filter evaluation and pruning to planning time, if

- The parquet meta from cache also include row group level min/max statistics

5.16 Other Items

Apache Parquet project has on-going work for vectorized parquet reader, which may support filter. We should keep an eye on that effort to see if Drill could leverage that work.

6 Implementation Plan

1. Initial prototype and experiments.
2. Implement Filter Expression materialization logic. Currently, ExpressionTreeMateriazer requires VectorAccessible, we probably have to add another API without requiring VectorAccssible.
3. Implement ParquetFilterEvaluator. This class is similar to InterpreterEvaluator, except for that it operates on inputs of RangedExpression.
4. Add implicit/explicit cast function to filter expressions
5. Add certain set of functions to filter expression.
6. Performance evaluation.

9 References

[1] https://issues.apache.org/jira/browse/DRILL-1950 : Parquet filter pushdown

[2] https://issues.apache.org/jira/browse/DRILL-2353 : Partition pruing.

10 Document History

Date	Author	Version	Description
2016-07-25	Jinfeng Ni	0.1	Initial Draft