*Author: Kirill Kozlov*
*Status:* Implemented
*Last updated: 12/02/2019*

# Objective

Propose an interface for Beam SQL IO APIs to support predicate and/or project push-down functionality for queries.

# Background

Users want more efficient (in terms of network bandwidth and compute resources) SQL pipelines when using IO sources with support for filter and/or project push-down, which are not currently being leveraged in Beam SQL.

# Detailed Design

We should introduce an interface responsible for determining whether a filter push-down is supported, and if so, return any expressions that are not supported to be preserved. Since core IO APIs support this feature via vastly different means (ex: BigQuery uses a SQL String, MongoDB - own Filter class), an implementation for this interface can transform a calcite RexNode condition into any appropriate format used by the API. Calcite RexProgram has a condition, which is a local reference to an expression, which has to be met (ex: boolean true) in order for the project to happen.

```
public interface BeamSqlTableFilter {
      List<RexNode> getNotSupported();
      int numSupported();
}
```

Implementation of a *BeamSqlTableFilter* interface should do all necessary processing of the RexNode and transformation to the acceptable data format. A list of expressions in a conjunctive normal form will be passed to a constructor.

Method `numSupported` should return the number of filters that can be pushed-down, used for calculating the benefit of project push-down by the cost model.

Method `getNotSupported` should return parts of a condition that are not supported. Return a list of RexNode if filter push-down is partially/completely not supported, containing all unsupported nodes. Return an empty list if an entire condition can be pushed-down to IO layer.

Let's say we have an API, which is only capable of performing a predicate push-down for expressions comparing a column value to a literal (ex: "*where col3>5*"). Imagine a scenario, a user executes a query looking something like this: "*where col3>5 and col1=col2*". Our hypothetical IO API does not support "*col1=col2*", but we can still push-down "*col3>5*". #getNotSupported method should return a list with the following condition "*col1=col2*", to be preserved in a Calc.

Several new methods need to be added to *BeamSqlTable* interface:

- `BeamSqlTableFilter constructFilter(List<RexNode> filter)`
  - Constructs an appropriate implementation of an interface described above
  - List should contain nodes, AND of which should make up a condition (in CNF)

- `ProjectSupport supportsProjects()`
  - Whether or not API supports simple projects (no nested tables for now)
  - `ProjectSupport` is an enum with the following options:
    - NONE
    - WITHOUT_FIELD_REORDERING
    - WITH_FIELD_REORDERING

- `PCollection<Row> buildIOReader(PBegin begin,`
  `BeamSqlTableFilter filters,`
  `List<String> fieldNames)`
  - Build an IO reader with projects and filters pushed-down when applicable

## Changes to the cost model

Main objective to be achieved by modifying the cost model is to ensure that the IO with push-down is chosen over the one without. Currently, cost model aims to improve efficiency by performing join reordering. The difficulty in modifying the cost model is to ensure that join reordering in unaffected. In order to make that happen change to the price of Rel node should be very small, no more than `BeamCostModel.FACTORY#makeTinyCost`.

The benefit of using a pushed-down IO is calculated using the number of fields and the number of filters pushed-down. The benefit value is normalized to be between 0.0 and 1.0. Then, the normalized value is multiplied by the `TinyCost` and subtracted from the cost of the IO without push-down.

The following formula is used for calculating the benefit:

$$normalizedBenefit = \frac{benefit}{max(benefit, originalIOCost) + 1}$$

$$benefit = projectBenefit + filterBenefit$$

$$projectBenefit = \#\ of\ fields\ pushed\ down$$

$$filterBenefit = scale(\#\ of\ filters\ pushed\ down) * ((projectBenefit + 1) * 0.1)$$

$$scale(x) = 1 - \frac{1}{x + 1}$$

Since the benefit of filter push-down is not as significant as the benefit of project push-down, the filterBenefit is scaled down to be between 0.0 and 1.0. Then, multiplied by the projectBenefit and 0.1, that way filterBenefit will make up at most 10% of the benefit of the project push-down.

An alternative approach that has been considered is to utilize a `Planner#setImportance` method to set the importance of the original `Calc` and

`BeamIOSourceRel` to 0 in hopes to discourage the planner from using them (Rel nodes w/o push-down) in further optimization. Such approach showed to be inconsistent, due to `Calc` importance not propagating to physical `BeamCalcRel` nodes.

## Example

Adding project push-down support to MongoDbTable would look something like this:

```
@Override
public PCollection<Row> buildIOReader(
    PBegin begin, BeamSqlTableFilter filters, List<String> fieldNames) {
  MongoDbIO.Read readBuilder =
    MongoDbIO.read()
             .withUri(dbUri)
             .withDatabase(dbName)
             .withCollection(dbCollection);

  // Resolve fields needed after push-down.
  final FieldAccessDescriptor resolved =
      FieldAccessDescriptor.withFieldNames(fieldNames)
          .withOrderByFieldInsertionOrder()
          .resolve(getSchema());

  final Schema newSchema = SelectHelpers.getOutputSchema(getSchema(), resolved);

  if (!fieldNames.isEmpty()) {
    // Create a find query with projection.
    readBuilder.withQueryFn(FindQuery.create().withProjection(fieldNames));
  }

  return readBuilder
    .expand(begin)
    // Transform MongoDb Documents to Beam rows.
    .apply(DocumentToRow.withSchema(newSchema));
}

@Override
public ProjectSupport supportsProjects() {
  // MongoDB supports project push-down with field reordering.
  return ProjectSupport.WITH_FIELD_REORDERING;
}
```

Predicate push-down example for MongoDbTable: [PR#10417].

MongoDbTable supports push-down via specifying a list of projects and filters in a FindQuery. For BigQuery `buildIOReader` method would involve calling `withSelectedFields` and `withRowRestrictions` setters in *BigQueryIO#Read* builder.

# Alternatives Considered

## Utilizing Flink implementation

We could also reuse an existing Flink implementation for project and filter push-down. That would require transforming Calcite data types to Flink supported types, thus introducing extra complexity.