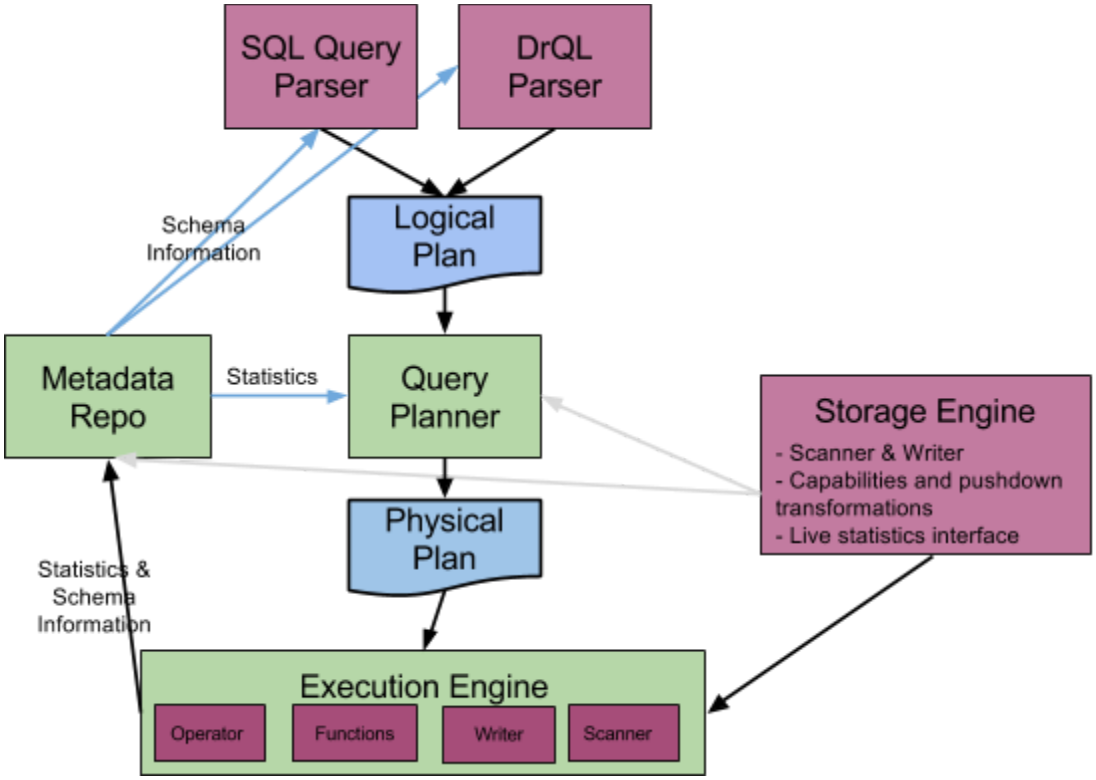
Overview

One of the goals of Drill is to define clear interfaces at various levels of functionality to allow future expansion. To this end, a general overview of Drill components and a common vocabulary is necessary to understand key integration points. The basic execution flow of drill is that a user or machine generated query is created and handed to a query parser. From there, the query is converted into a logical plan. The logical plan describes the basic dataflow of the query operation. The optimizer is then responsible for reading this logical plan and converting it into a physical (or execution) plan. From there, the execution engine is responsible for executing the plan. The execution engine relies on a number of components including various types of operators, scanners (readers) and writers. A metadata repository is responsible for providing information to various components of the Drill query execution framework. The key pluggable components for the first version of Drill are:

- Query Parser: Can request schema information from a metadata API and produces a standard logical plan for optimizer consumption.
- Storage Engine: Responsible for interacting with a particular source of data. Provides scanner and/or writer capabilities. Informs the metadata repo of any known characteristics of the data (schema, file size, data ordering, secondary indices, number of blocks, etc). Informs the execution engine of any native capabilities (predicate pushdown, joins, SQL, etc).
- Operator: Responsible for transforming a stream of data.
- Functions:
    - Aggregate Functions: Converts a stream of values into a single value.
    - Scalar Functions: Converts one or more scalar values into a single value.



Data Structures

At its core, Drill operates on sequences of trees. This is in contrast to traditional row-based systems. In Drill, the primary unit of data is a data tree. A data tree is composed of labeled values. The value of each node can either be a subtree, a scalar or a list of values. "Document" and "row" are often used interchangeably with the word "tree" for this concept within this document. Whether schemaless or not, abstractly you can think of each data tree as a JSON object. Each tree is composed of a key and a root node.

```
{
      "key": "123",
      "value": {
            "foo": { "x": "y"},
            "blue" : [
              "red",
              "blue"
            ]
      }
}
```

In the case that Drill is consuming a more traditional relational datasource, the unit of data can abstractly be thought of as a flat tree.

```
{
      "key": "123",
      "value": {
            "foo": "bar",
            "blue" : "orange"
      }
}
```

General Considerations

The abstract structures of both the Logical Plan and the Physical Plan are JSON representations of a directed acyclic graph (DAG).

Data Typing

The key to plan description is the set of operators and their definitions, notably including the types of things that they can accept as inputs and outputs. Some of the typing that the query optimizer and execution engine will need to know about can be known from the plan itself, but most of the type information will be dynamic and would be difficult to know except in the context of the actual data being processed. Exactly how we will determine types and exactly which types we will need to think about isn't clear yet.

Regardless of the lack of clarity on types, we can probably lock down most of the set of operators required for Drill.

Logical Plan

Purpose

The logical plan describes the abstract dataflow of a language-agnostic query. It generally tries to work with primitive operations without focus on optimization. This makes it more verbose than traditional query languages. This is to allow a substantial level of flexibility in defining higher-level query language features. Typically, the logical plan will be produced by a query parser. When schemas are available, the query parser will also be able to leverage this information to further validate the query and construct a logical plan.

The logical plan is a DAG of data stream operators. The edges of the DAG represent the flow of data. (Note that, as opposed to traditional programming SSAs, our SSA representation of the DAG allows higher-order scalar functional expressions to be provided as arguments as opposed to including the expressions directly as part of the SSA. This is to simplify human understanding and reduce verbosity.) Because the logical plan is an abstract representation of dataflow, each operator can only have one output. (This is in contrast to the physical plan where many operators will have multiple outputs.) On the flipside, a particular operator could

subscribe to the outputs of multiple other operators.

Because hierarchical data is supported, Drill provides a number of explosion and implosion operations. These allow manipulation of nested values to be managed with sub-DAGs. Because all operators are independent, any re-implosion of data must be provided in stream.

Logical Plan Vocabulary

Arguments

A logical plan describes the collection and connection of logical operators. Each operator has a number of arguments. The values that these arguments may have is described in this section using the following abbreviations for classes of values:

- <name> A name to be used as the definition of **or** reference to a particular output field in the record stream.
- <string> Quoted string.
- <expr> Value expression that may contain one or more value functions, values and field references. Note that a value expression can return a scalar value or a complex value (map or array).
- <aggexpr> Expression that contains one or more scalar expressions as well as one or more aggregate expressions (SUM, COUNT, etc). Note that an aggregate expression must never nest one aggregate within another but can have a scalar expression applied on top of two separate aggregates.
- <runaggexpr> Expression that contains Aggregate that supports running operation.
- <opref> Integer defining or referencing a particular operator.
- <json> An opaque JSON option string.
- <operator> The definition of an operator or a <opref>

In the cases where a particular value is optional, the value is marked with an asterisk. In the context of the *do* argument of the *sequence* operator, some values are optional. Values that are optional in that context will be marked with †.

Key Concepts

- **Value**: A value in Apache Drill is either a scalar value (int32, int64, uint32, float32, etc), an array of values, or a map of values.
- **Record:** A record is a collection of zero or more values, typically called a row in relational databases.
- **Field**: The particular position of a field within a record.
- **Stream**: A stream is the set of records arriving at one operator from its producing operator.
- **Segments**: One important concept discussed within the logical plan is the concept of a

segment. A segment defines a subcollection of input records that are collected together for some purpose. In many cases, an operator can be provided a segment key and will then contextually manage its operation separately on each segment and avoid carrying state between segment boundaries.

Logical Plan Operators

Operators fall into a number of overlapping classes. Each operator identifies its classes. The current operator classes are:

- 0: An operator that generates data without relying on any input operators (source).
- 1: An operator that consumes a single input source
- M: An operator that consumes more than one source of input data
- K: An operator that produces no output stream (sink).

Property Properties

- †: This property is required outside a sequence and not allowed within a sequence.
- *: This property is optional.

Scan (0)

The Scan operator outputs a stream of records. The "`storageengine`" argument must refer by name to a storage engine defined in the engines clause of the logical plan. The "selection" argument accepts a JSON object that is used by the data source itself to limit the amount of data actually retrieved. The format and content of this object is specific to the actual input source being used. Examples might include an HBase table name, a MongoDB collection, an HDFS path or a partition of a Hive table. Data sources will use the the selection argument in an implementation-specific way. The provided “ref” argument ensures that all records within the scanned source are held in the provided namespace.

```
{ @id†: <opref>, op: “scan”,
  storageengine: <string>,
  selection*: <json>,
  ref: <name>
}
```

Constant (0)

The `constant` operator returns a constant result. This is necessary to be able to introduce values, 1) in cases where you don't know what tables exist, or 2) for evaluating expressions that don't belong in a table. The operator is analogous to the VALUES operator in SQL.

```
{ @id†: <opref>, op: “constant”,
  content: <json>
}
```

With the `constant` operator it is for example possible to implement the SQL VALUES clause as

shown in the following example:

SQL `VALUES (1, 'iamastr') AS t(c1, c2)`

DLP `{ @id: 1, op: “constant”, content: { [ { c1: 1, c2: “iamastr” } ] } }`

The implementation described here has been completed using implicit typing. More information can be found on the JIRA page of Issue #57. At some point in the future we will be adding support for explicit typing in some form. We have not decided on a concrete format for specifying types but have discussed a number of options that are covered in detail here: https://docs.google.com/document/d/1nn8sxcuBvpAHm-BoreCWELPIQ8QhhsAUSENX0s5sSys/edit

Join (M)

The Join operator joins two inputs based on one or more join conditions. The output of this operator is the combination of the two inputs. This is done by providing a combination record for each set of input records that matches all provided join conditions. In the case that no conditions are provided, a cartesian join is generated. The combination record is a single record that contains a merged map of values from both provided input records. For example, if the left record is {donuts: [data]} and the right is {purchases: [data]}, the combination record would be {donuts: [data], purchases: [data]}. Join also requires a condition variable “type” which describes what happens when a record does not match the join conditions. “Inner” means that only records that match the join conditions should be included. “Outer” means that unmatched records from either input are included. “Left” means that only unmatched records from the left input are included. When an unmatched record is included, the resulting record contains only contains known values from the noted side and no values from the unmatched side.

Available relationship types “reltype” include: >, >=, <=, <, !=, ==

```
{ @id†: <opref>,  op: “join”,
  left: <input>,
  right: <input>,
  type: <inner|outer|left>,
  conditions*: [
    {relationship: <reltype>, left: <expr>, right: <expr>}, ...
  ]
}
```

Union (M)

The Union operator combines two or more data inputs into a single stream. No kind of ordering is implied or necessarily maintained. If “distinct” is true, duplicate rows are removed from the stream. Duplicate rows are defined as both sets of records containing exactly the same fields with each field having exactly the same value.

```
{ @id†: <opref>, op: “union”,
  distinct: <true|false>,
  inputs: [
    <input>, ... <input>
  ]
}
```

Store (1 K)

The Store operator stores the stream output to a storage engine. The `storageengine` parameter refers to a storage engine defined within the logical plan. The “target” parameter describes the storage-engine specific parameters (e.g., storage type, filename, etc). Each individual storage engine’s capabilities can include what level of parallelization is possible so this isn’t represented at the logical plan level.

```
{ @id†: <opref>, op: “store”,
  input†: <input>,
  storageengine: <string>,
  target: <json>
}
```

Project (1)

The Project operator returns a particular field subset of the incoming stream of data corresponding to the expressions provided. The "projections" parameter specifies the values to be retained in the output records. For each of the projections, the name given to the value in the output record is specified by the “ref” parameter and the expression to be evaluated on the input record to determine the output value is given by the “expr” parameter. To enable clarity and support upleveling a given set of outputs, the projection ref names must begin with “output”. An example might be {ref: “output” expr: “sum(donuts.sales)}”. This enables output of values that have simple types. If multiple projections overlap, they will be merged with the later projections overriding the values of earlier projections.

```
{ @id†: <opref>, op: “project”,
  input†: <input>,
  projections: [
    {ref: <name>, expr: <expr>}, ...
  ]
}
```

Order (1)

The Order operator orders an input stream by one or more ordering expressions. Ordering can be constrained within particular segments by providing the optional “within” parameter. Segments are defined by other operators such as the group operator. In the case that a segment is provided, ordering must only be done within the boundary of each segment.

The option of “nullCollation” defines where in the ordering null values should be positioned. When not provided, the default is that null values are the first in the ordering.

```
{ @id†: <opref>, op: “order”,
  input†: <input>,
  within*: <name>,
  orderings: [
    {order: <desc|asc>, expr: <expr>, nullCollation: <first|last> }, ...
  ]
}
```

Filter (1)

The Filter operator removes certain values from the stream based on the provided expression. For each input record where the provided expression evaluates to true, that record is passed to the output. Where the expression evaluates to false or any other value, the record is not passed to the output.

```
{ @id†: <opref>, op: “filter”,
  input†: <input>,
  expr: <expr>
}
```

Transform (1)

The Transform operator transforms data to allow for data reference and data flow clarity. While anonymous expressions can be used within various operators, unless the operator specifically outputs the value of the expression, it will not be available within that operator’s output stream.

For example, when using the Segment operator, if the provided grouping expressions are not simple field references, they will be evaluated for segmentation purposes and then thrown away. If the query requires those values to be maintained, they need to be generated by a transform clause before or after the segment operation.

A relabeling (an AS clause) can be written as an identity transform where the “expr” is an existing field reference and the “ref” is a new field reference.

```
{ @id†: <opref>, op: “transform”,
  input†: <input>,
  transforms: [
    {ref: <name>, expr: <expr>}, ...
  ]
}
```

Limit (1)

The Limit operator limits the output of the operator to only include a prefix of the input. Input records are implicitly numbered starting with 0. The "first" and "last" parameters specify the

half-open range of records that are to be included in the output. Last = 0 returns no objects. First = 0, Last = 1 returns the first object. Limit operates over an entire dataset.

```
{ @id†: <opref>, op: “limit”,
  input†: <input>,
  first: <number>,
  last: <number>
}
```

Segment (1)

The Segment operator is responsible for collecting all records that share the same values of the provided expressions and outputting them together as a single segment (or group) of data. Each record of the output segment will have the same value for all provided expressions. The segment operator will also output a segment key in the provided ref.

The segment operator is stable. This means that all records within a segment will appear in the order that they appeared in the input. There is no guarantee, however, about the order in which inner segments will be output.

```
{ @id†: <opref>, op: “segment”,
  input†: <input>,
  ref: <name>,
  exprs: [<expr>,..., <expr>]
}
```

Window Frame (1)

For each record of an incoming stream (let’s call this the target record), the window operator will create a segment containing the records in a sliding window surrounding the target record. The sliding window will include a specified number of records before and/or after the target record. The Window Frame operator can operate across an entire input stream or can be told to only allow operations within each incoming segment.

A simple example: a window range of -2 start and 0 end is applied to a 5-record single-segment input [0,1,2,3,4]. This would result in an output of 12 rows (brackets denote each output segment): [0] [0,1] [0,1,2] [1,2,3] [2,3,4]. Each output record would contain two additional fields: ref.segment, which holds the current window segment, and ref.position, which will the negative, positive or zero position within the window. For the example data, the values to each of these would be: ref.segment: [0,1,1,2,2,2,3,3,3,4,4,4], ref.position: [0,-1,0,-2,-1,0,-2,-1,0,-2,-1,0]

A window operator can also optionally take a segment key as input. This key is used to restrict the windows so that they do not cross segment boundaries. The window operator takes as inputs a “start” and an “end” value. These values define the range of the window. The window range is defined based on thinking of the target record as zero, and each record after incrementing positively and each record before incrementing negatively. In the case that start or end is not defined, that portion of the range will be unbounded. In all cases, start must be

greater than or equal to end. At least one of start or end must be defined.

```
{ @id†: <opref>, op: “windowframe”,
  input†: <input>,
  within*: <name>,
  start*: <number>,
  end*: <number>
  ref: {
    segment: <name>,
    position: <name>
  },

}
```

CollapsingAggregate (1)

The Collapse aggregate operator collapses a segment of records into a single record. In the case that a segment (within) is undefined, the collapse aggregate will collapse all input records into a single output record.

The only outputs of the collapse aggregate operator are the provided aggregations and the values defined as “`carryovers`”. The collapse aggregate operator also can be provided a target field reference with which to select the record used for carryover values.

In the case that a target reference is undefined, the collapse aggregate operation will be free to choose which record the carryover values are drawn from (typically, this is because all records share the same value). In the case that a target field reference is provided, will draw the carryover variables from a record where the target field references has a true value. If more than one record has a target field value that is true, the carryover values will be drawn from one of those records. In the case that no record within the target segment has a target value of true, no record will be emitted from that target segment. In no case will more than one record be emitted per segment.

```
 { @id†: <opref>, op: “collapsingaggregate”,
  input†: <input>,
  within*: <name>,
  target*: <name>,
  carryovers: [<name>, … , <name>],
  aggregations: [
    {ref: <name>, expr: <aggexpr> },...
  ]
 }
```

See the Arguments section in the first part of this document for a discussion of aggregating expressions suitable for *aggexpr*.

RunningAggregate (1)

The Running Aaggregate operator takes an input record and adds appends a set of running aggregations and outputs the resulting record. The aggregations are re-evaluated on each record within the incoming segment in the order in which they are provided. Segment focus can be defined with the "within" value such that aggregations are reset at each segment boundary.

```
{ @id†: <opref>, op: "runningaggregate",
  input†: <input>,
  within*: <name>,
  aggregations: [
    {ref: <name>, expr: <aggexpr> },...
  ]
}
```

Flatten (1)

The Flatten operator generates one or more output records based on each input record. The output record will be the input record along with an appended additional appended field(s) for the flattening. The specific behavior depends upon the type of the flatten expression provided. In all cases, the "drop" parameter defines whether or not all fields referenced in the target expression are maintained in the output record or removed.

- Scalar/Map: In the case that the provided expression returns a scalar or map, Flatten will return a single record for each input record with one additional field at the "ref" location defined with the same value as the input expression.
- Array: If the provided expression returns an array, flatten will return multiple copies of the original input record where each is augmented with the value within the array at the "ref" position.

```
{ @id†: <opref>,  op: "flatten",
  input†: <input>,
  ref: <name>,
  expr: <expr>,
  drop: <boolean>
}
```

Sequence (1)

The Sequence operator is a syntactic operator to ease the definition of simple flows. Since the vast majority of operators are single-input operators, sequence allows one to define a non-branching flow that simply transforms elements (possibly deleting or adding them) without having to specifically define input and definition references. All elements beyond the first element in a sequence must be single-input operators. The first element must either be a source operator or a single-input operator. The last element of a sequence can be a sink. The sequence will be a sink if and only if the last element of the sequence is a sink.

```
{ @id: <opref>, op: “sequence”,
  input: <input>, do: [
      <operator>, <operator>, ... <operator>
  ]
}
```

Physical Plan

Purpose

The physical plan (often called the execution plan) is a description of the physical operations the execution engine will undertake to get the desired result. It is the output of the query planner and is a transformation of the logical plan. Typically, the physical and execution plans will be represented using the same JSON format as the logical plan.

Physical Plan Operators

Storage Operators

The storage operators are: scan-json, scan-hbase, scan-trevni, scan-cassandra, scan-rcfile, scan-pb, scan-pbcol, scan-mongo, scan-text, scan-sequencefile, scan-odbc, and scan-jdbc (and corresponding store-* for each).

Normal Operators

The normal operators are: limit, sort, hash-join, merge-join, streaming-aggregate, partial-aggregate, hash-aggregate, partition, and union.

Appendix A: Filtering nested values

Because Drill operates on data trees instead of traditional rows, the use of filtering is different than traditional database systems. During filtering, an object node is retained if one or more of its children satisfy the filter being applied. An object node thus retained is retained in its entirety, including all children. Those children in a retained object node are recursively filtered in the same way.

An array node, on the other hand, is retained if at least one of its children satisfies the filter and only those children that satisfy the filter are retained. Those surviving children are filtered recursively in the same way.

As an example, suppose that this sequence of JSON structures is filtered by the expression `a.x > 3`

```
[{a:{b:1, x:5}, c:1},
  {a:[{b:1}, {z:{x:4}}, {q:3, x:5}], c:2},
  {a:{b:2, x:[1,2,3,4,5]}, c:3},
  {a:{b:3, x:[1,2]}, c:4}]
```

The results will be as follows

```
[{a:{b:1, x:5}, c:1},
  {a:[{b:1}, {z:{x:4}}, {q:3, x:5}], c:2},
  {a:{b:2, x:[1,2,3,4,5]}, c:3},
  {a:{b:3, x:[1,2]}, c:4}]
```

The first structure that contains `c:1` is retained because the first sub-tree labelled a has a member labelled x that has the scalar value 5. The `a.x=5` sub-tree is retained, which causes the a sub-tree to be retained, which causes the `c` sub-tree to be retained as well.

The second structure has a vector value for `a`. The first value in the vector doesn't have an `x` field, so it is discarded. The second value in the vector has an x value, but the `x` value is one level too deep to match. The third value has a value of x at the right level and that x field has a value greater than three so it is retained. This causes the value `a.q=3` value to be retained as well. Since a has at least one retained value, it and the `c=2` field are both retained. The `{q:3, x:5}.`

In order to be able to rebuild a correct structure, we have to retain some notion of where arrays are found in the data structures. This is most clear when dealing with nested arrays. For instance, with this structure, we would like filtering with `a.x > 3`

```
{a:[[{x:1},{x:4}], [{x:5}]]}
```

to recognize and preserve the array nesting and produce this

```
{a:[[{x:4}], [{x:5}]]}
```

The problem here is the notation `x.a` has no way to record the extra levels of array nesting. Essentially, we need for the `a.x` path to be modified as it is carried through the system as something more like `a[0][0].x` or `a[1][0].x`. These array indicators can then be used to rebuild the correct nesting when the data is marshalled for output. In the bind operator, the name given to a variable is of the form a.x, but this is just used for matching to the actual data elements. Each of the data elements remembers its detailed location in the data tree to allow reconstruction of the tree. Of course, one way for data elements to "remember" their location in the tree is based on a reference to a schema. In that case, explicit retention of a name is not necessary.

These difficulties with nesting only appear with data without a strong schema. This means that another alternative is for Drill to require that the nesting notations be inserted either by the

parser by referring to the schema of the data or explicitly inserted by the user. Array indexes still have to be inserted by the execution engine. Allowing the user to insert array nesting indicators in the original query would be an incompatible change relative to Dremel's syntax that should require discussion. It is very desirable that a program that runs using data encoded using protobufs produce the same results if the data is encoded using a schema-less format like JSON. Requiring that the user supply nesting hints would violate that expectation.