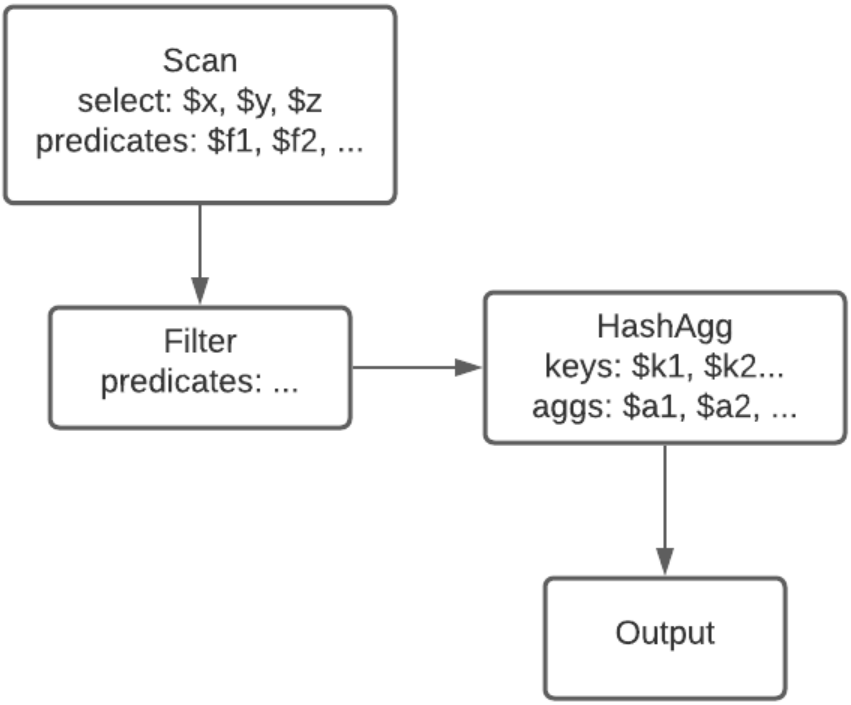
Arrow C++ streaming columnar query engine physical execution rough scaffolding

Authors: Ben Kietzman, Wes McKinney, Michal Nowakiewicz

Physical execution data flow graph

One exceedingly common structure of a query engine is a data flow graph.



Each **“node”** in the graph can be implemented as a generic data flow operator. An operator has one or more inputs and one or more outputs (though a single output ends up being the most common in practice).

For example, consider the query:

```
SELECT a, b, sum(c)
FROM my_data
WHERE d < 10, e > 0
```

Depending on the characteristics of “my_data”, the execution plan might look like:

- SCAN: my_data with partition predicate “d < 10”
    - FILTER (add selection vectors) with “e > 10”
        - HASH AGGREGATE with keys “a, b” and agg exprs “sum(c)”

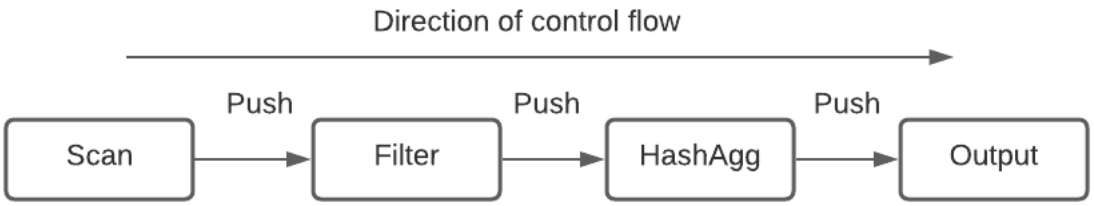
This is called a **“physical execution plan”** because it describes the exact algorithms to use and shows a deterministic data flow with the input and output data schemas known at the input and output of each operator node.

The process of coming up with a good execution plan is called “query optimization and planning”. Optimization generally refers specifically to restructuring the physical plan to be more efficient. **Optimization and planning is outside the scope of this document.**

Push and pull-based node/operator APIs

Modern query engines typically use either a “pull” based API or a “push” based API. Both models are fundamentally streaming. They operate on a sequence of relatively small data batches which are often 64K rows or fewer. The reason that the batches are small is to benefit from keeping data in CPU cache as it’s flowing through the operator graph. It also reduces memory use so that query engines can process inputs that do not fit into memory all at once, and that large intermediate outputs (e.g. very large outputs of joins) do not cause OOM.

- In a pull based API, the terminal output operator asks its parent to produce its next output, which is either accumulated in memory or streamed out via ODBC (or similar) to whoever asked the query to be executed.
- In a push based API, the root nodes “push” their outputs into the nodes that depend on them.



For various reasons, more recent systems seem to prefer the push-based model, so I propose going with that. The abstract API for an operator looks something like:

```
class ExecNode {
 public:
  // Begin executing the query pipeline. Mainly used for Scan nodes
  // to initiate the execution pipeline.
  // Propagates QueryContext to children
  // ctx: a context object for bookkeeping, profiling,
  // error reporting, etc.
  virtual Status Open(QueryContext* ctx) = 0;

  // input_index: the index of the input dependency
  // batch: the next chunk of data produced by the dependency
  virtual bool Consume(int input_index, const ExecBatch& batch) = 0;

  //
  virtual bool InputFinished(int input_index) = 0;

  ExecNode* child(int child_index);
  int num_children() const;

  ExecNode* parent(int parent_index);
  int num_parents() const;

  QueryContext* query_context() const;

 private:
  QueryContext* query_context_;

  // These need to be owned elsewhere (e.g. in QueryContext) to
  // avoid circular references
  std::vector<ExecNode*> parents_;
  std::vector<ExecNode*> children_;
};
```

When all of the raw data producing nodes are done materializing inputs, they invoke the "InputFinished" API when ultimately triggers the "InputFinished" on the terminal output node, so at that point you know the query is finished.

When a physical execution plan is "built", it is constructed from the bottom (output) up to the top (scan). To create an operator, you need

- One or more output "child" operators
- In many cases some "bound expressions", which are objects providing low-level array expression evaluation. More on this later.

Note that ExecNodes **should not scrutinize or analyze input and output schemas** — all of that analysis should be done prior to constructing the operator graph so that no field lookups or other analysis need necessarily to be done during execution. This is discussed more below in the "bound expressions" section.

Types of ExecNodes

An implementation of ExecNode generally implements a specific version of some high level algorithm in the standard relational algebra. For example, there are different ways to implement joins: hash joins, nested loop joins, etc.

Some of the most common operators include

- **Scan**: materializes in-memory data from storage (e.g. Parquet files or some other storage)
- **Projection**: selects columns from input table(s) and executes any array expressions thereof. For example, you might see the projection "SELECT a.*, sqrt(a.FOO - b.BAR) as SOMETHING FROM …", where a and b are two different tables. Here the expression **"sqrt(a.FOO - b.BAR)"** is an array expression involving selecting columns FOO and BAR originating from different operator inputs, such as the inputs to a join.
- **Filter**: evaluates a row selection filter on input batches. In columnar engines, it is common for the output of a filter operator to be the input batch with a selection vector attached containing the indices of the "selected" rows. This can result in much better execution performance in aggregations, for example
- **Group/Hash Aggregate**: compute group-based aggregates ("GROUP BY") given one or more keys. This is often called "Hash Aggregation" because the algorithm used is almost always a hash-table based bucketing.
- **Union**: "stack" two or more tables having the same schema into a single larger table
- **Join**: perform an inner, left, outer, semi, or anti join given some join predicates
- **Sort:** reorder the rows of a table by some sorting condition
- **Top-K**: retrieve a limited subset of rows from a table as though it were in sorted order. For example, find the "top 100 values for FOO" in a table with 1 billion rows. This is

different from sorting usually since fully sorting a 1 billion requires a lot of memory whereas a heap-based method can be used to compute Top-K with fixed memory use

Bound array expressions

A “logical” array expression is something like

sqrt(bar - foo) + 1

Suppose that **foo** and **bar** come from an Arrow table with schema:

apple: string
bar: int32
pear: string
foo: int16

“Binding” is one way to refer to the process of mapping the logical expression “**sqrt(bar - foo) + 1”** onto the actual physical data structure(s) that will be flowing through the graph. When binding this expression, the field reference “foo” is replaced by a bound reference **BoundRef(input=0, field_index=1)**. So the bound expression tree looks like this:

```
Add -> float64
  0: Sqrt -> float64
     0: Subtract -> int32
        0: BoundRef(0, 1) “bar”
        1: ImplicitCast BoundRef(0, 3) “foo” to int32
  1: Literal “1” as scalar[float64]
```

Given our current structures in the C++ library, during binding, we can also select the appropriate kernel function from the FunctionRegistry to utilize when evaluating the bound expression given an input ExecBatch.

If an operator has multiple input node dependencies, then the “input” part of the BoundRef refers to which input batch to select a column from (think expressions on columns of joined tables).

Development plan from here

- We must implement the **BoundExpression** concept above to allow us to build an expression tree that can compute an array expression without any schema or bounds checks given an input ExecBatch.
- I haven’t looked closely at the Datasets API lately, but I believe that **Projection** and **Filter** operations must be decoupled and turned into implementations of the **ExecNode** API above.

- Dataset scans must be wrapped in a **ScanNode** implementation of **ExecNode**
- We must implement a simple **OutputNode** for accumulating query outputs to be returned as an arrow::Table at the end. Other output node types can be implemented later, such as ones that stream results as soon as they are available over Flight
- The grouped / hash aggregation implementation in https://github.com/apache/arrow/pull/9621 must be refactored to implement the **ExecNode** API above. Hash aggregation will take both bound key expressions and bound aggregate expressions as inputs.
- Future consumption of Datasets should be planned to be routed through the query engine rather than maintaining a separate API that only knows how to scan, filter, and project. We should therefore contemplate the future of some parts of the Datasets API.