Arrow C++ Data Frame Project

Wes McKinney
May 2019

This document discusses some design requirements and use cases for a higher level semantic API / interface (called a “data frame”) for interacting with in-memory Arrow datasets. This is to be developed on top of the existing low-level Array, RecordBatch, and Table data structures and is intended to offer a layer of usability and productivity in similar style to popular “data frame” libraries in languages like Python (pandas) and R (data.table, dplyr, base R).

This project is a parallel, complementary effort to the previously discussed “Datasets” and “Query Engine” projects, and is not intended to compete with, conflict with, or replace any of the work that has been discussed there. In fact, both the Datasets and Query Engine components would be intended to be used in tandem with the Data Frame interface. To make a useful piece of software, a number of *opinionated* design choices around data structure and function execution semantics will have to be made, so I think it best to develop a data frame library as an add-on component for the Arrow core platform.

References

- C++ Datasets
  https://docs.google.com/document/d/1bVhzifD38qDypnSjtf8exvpP3sSB5x_Kw9m-n66FB2c/edit?usp=sharing
- C++ Query Engine
  https://docs.google.com/document/d/10RoUZmiMQRi_J1FcPeVAUAMJ6d_ZuiEbaM2Y33sNPu4/edit?usp=sharing

Table of contents

Background and Motivation

In popular use, “data frames” offer an alternative kind of API user interface for manipulating structured data compared with SQL queries used by relational and analytic databases. These interfaces offer numerous programming conveniences for data that is entirely memory-resident (or memory-mapped).

In most cases, analytical operations are evaluated in an eager-like fashion (where each operation materializes its results entirely in memory) and presumes to have random access to the entire dataset at any given time

Some example data frame projects includes

- pandas https://conference.scipy.org/proceedings/scipy2010/pdfs/mckinney.pdf
    - See also *Python for Data Analysis* http://amzn.to/2vvBijB
- R base data.frame
- data.table (R language) https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html
- dplyr (R language) https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html
- SFrame https://github.com/turi-code/SFrame
- pydatatable https://github.com/h2oai/datatable
- Vaex https://github.com/vaexio/vaex
- DataFrames.jl (Julia) https://juliadata.github.io/DataFrames.jl/stable/man/getting_started/

As the Apache Arrow community has developed, we have expanded into many application domains including those (“data science” or “machine learning”-focused) where data frame style work is the predominant mode. By virtue of our high performance IO utilities (e.g. for reading Parquet files), an increasing amount of data is passing through Arrow data structures, but is being converted immediately to other data structures, like pandas DataFrame objects. This has multiple problems:

- Conversions are not free, even though the conversions have been optimized (https://wesmckinney.com/blog/high-perf-arrow-to-pandas/)
- Many users are converting to pandas, performing some operations using pandas, then converting right back to Arrow, such as to write back to a Parquet file. Some of these

operations are relatively trivial like performing a comparison on a column and filtering out rows not matching the result of the comparison.
- Memory is wasted due to data duplication in memory as a result of serialization. This is especially onerous because other tools may use a lot more memory to represent the same Arrow dataset

In C++ we have begun developing “kernel functions” (https://github.com/apache/arrow/tree/master/cpp/src/arrow/compute/kernels) which perform units of computation. The interface for these kernel functions is relatively low-level, and details such as efficient evaluation on chunked arrays or parallel execution are not straightforward in all cases. Additionally, we have struggled a bit with having public APIs for Array-level execution versus ChunkedArray-level execution.

The kernel functions we are developing can be used as the building blocks for a dataflow-style query engine (see link above), but they can also be used to build a “data frame”-style data manipulation interface. A data frame interface can relatively easily abstract away a number of tedious details associated with kernel invocation to provide simple high-level, easy-to-use C++ APIs, which can also be wrapped in downstream binding languages (Python, R, Ruby).

Thus, this document proposes general scope for the semantics of a data frame interface in C++ inside Apache Arrow that can bring high-level usability to our kernel functions and enable users to manipulate fully in-memory datasets in the same way that they currently use other data frame libraries that use implementation-specific columnar formats instead of the Arrow standard.

Goals and Non-goals

Since in-memory and memory-mapped datasets are interchangeable in algorithms (due to the zero-copy nature of Arrow), any reference to interacting with “in-memory” data also includes memory-mapped on-disk data.

Goals include

- Internal data representation is the Arrow columnar format. Data frame columns can be chunked
- Trivial support for memory-mapped data frames
- Stack data frames vertically (“concatenate”) without copying memory
- Semantically mutable with copy-on-write semantics: i.e. do not copy any memory unless you have the sole reference to it (as verified by shared_ptr reference counts)
- Eagerly evaluated operations with rudimentary operator combinations (for example, filtering can be combined with aggregation for better performance)
- Multi-threaded execution of kernels against in-memory Arrow columnar data through easy-to-use, high level APIs

- Optimized data-frame-specific implementations of certain key operations
- Caching of column statistics (min, max, sum, mean, etc.) usable to hint algorithm selections
- Reading data frames into memory streamed from the forthcoming datasets API
- Support different indexing and in-memory partitioning strategies to accelerate aggregations or similar
- Evaluate user-defined functions having arrow::Array input and output

Non-goals (things we are **not** doing) include

- Cloning the API or semantics of other data frame libraries like pandas
- …

More detailed concepts and features

In this section I will jot down some ideas about various parts of the project and also provide pre-emptive answers to common questions/confusions/misconceptions.

Data frames vs. the Arrow columnar format

Data frames use the Arrow columnar format to represent data in-memory while providing higher-level computational semantics that operate against Arrow data. The data frame interface handles two key data structure issues: **chunking** and **mutability.**

Many datasets are chunked in memory for various reasons:

- Parts of the dataset came from different files, or were produced by different threads of execution (e.g. multithreaded CSV parsing)
- Record batches or tables can be concatenated without copying memory

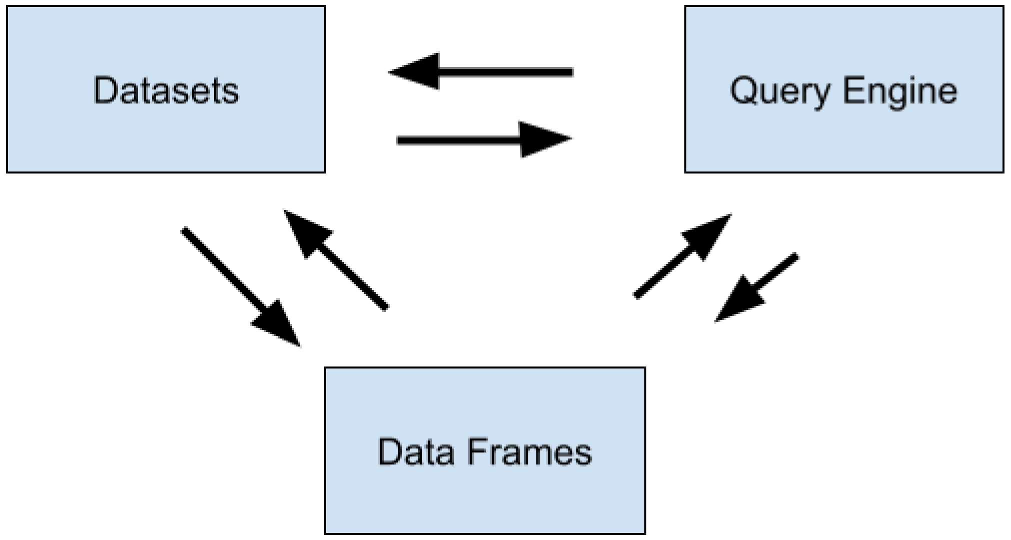
We already have the `arrow::ChunkedArray` and `arrow::Table` data structures -- which are synthetic, and not part of the Arrow columnar format -- which provide conveniences for working with chunked datasets. The data frame object will be composed from these existing lower-level APIs.

The Arrow columnar data structures are not intended to be mutable. However, we can provide APIs that perform mutations, with those mutations being performed by in-place mutation of buffers (in certain limited circumstances, see next section)

Interactions with Datasets, Query Engine components

Broadly speaking, we have the following characteristics

- **Datasets** component is responsible for creating a stream of RecordBatches
- **Query engine** component acts on one or more *input* streams of RecordBatches to produce an *output* stream. Data can be input from a **Dataset**.
- **Data frame** component manipulates a materialized in-memory collection of one or more RecordBatches. A data frame **can also be used as input to the query engine**
- **Kernels** provide units of analytical functionality usable by either the data frame or query engine



Interactions with ChunkedArray, RecordBatch, Table, Schema data structures

A data frame can be

- Constructed from a RecordBatch or Table
- Dumped to a Table

The internal representation of a materialized data frame column is a ChunkedArray (which may have only one chunk)

Mutation and copy-on-write semantics

There are two kinds of mutation that are of interest for us: mutation of the data frame data structure and mutation of internal memory.

First, on **mutation of data structures**, my inclination is that in-place mutation of a data frame (example, adding, overwriting, or removing columns in-place) is an important feature since producing a new data frame object may involve copying a relatively complex internal data structure. For these operations, it may make sense to have in-place versions and functional versions that emit new objects, so the user can choose as they please which makes sense.

There are relatively few operations that require mutation of memory. These are such operations as:

- Assigning a single value in a single column (e.g. `a[i] = val`)
- Assigning multiple values in a column (by indices, boolean selector, etc.)

I think it's acceptable to provide such mutation operations using copy-on-write. The idea is that if we infer that memory to be mutated comes from a MutableBuffer and we are the sole reference to that memory (i.e. there is no known `parent` member -- e.g. we have not sliced from a larger buffer), then we can mutate the memory in-place. If we cannot make such an inference (more on this below), then we must create a mutable, single-reference copy and mutate that.

Reasoning about memory sharing

We have a few mechanisms to determine whether it is safe to mutate a Buffer

- If and only if is_mutable_ is true
- If and only if the use_count() of the shared_ptr around a Buffer is 1
- If and only if the parent_ field is non-null

If any of these three conditions is not true, then the memory must be copied prior to mutation. Subsequent mutations, then, will not require copying because the newly-allocated Buffer copy will meet these requirements.

Copying data structures

A data frame as well as its column data structures will require a **Copy** method to clone the data structure, so that it can be mutated (example: adding a column). This Copy/Clone step will not copy any memory.

I'm not sure if it would be useful to have a "deep copy" operation that also copies underlying data, so at least in the short term it is probably not needed.

Mutation threadsafety

In-place data frame mutations, such as adding, overwriting, or removing columns (which may alter the schema), should hold a mutex to prevent concurrent mutations.

Determining whether it is safe to mutate memory in place can be costly; we should assess through benchmarks the cost of determining whether a copy is needed. Consider some example scenarios:

- Multiple threads mutating a column in-place concurrently
- A Buffer with use_count=2, where the other shared_ptr copy is destructed at some later point

My guess is that we should perform the copy-on-write check once while holding a mutex and then cached. Other mutators will have to block while awaiting for the copy-on-write ("CanMutate") check to complete.

Another complexity around mutation is if the data frame provides any APIs that provide public access to its data. Such APIs will have to invalidate the "can mutate" flag so that subsequent mutation operations will have to check again and possibly copy memory.

Lazy filtering and filter-fusion

In data frame libraries it is very common to write such code as

```
df.filter(boolean_condition)
  .get_column(col_name)
  .aggregate(func)
```

Here `filter` selects rows based on the true/false values of the boolean condition.

If there are many columns, and only one is of interest, this can generally also be written as:

```
df.get_column(col_name)
  .filter(boolean_condition)
  .aggregate(func)
```

This avoids unnecessary filtering and memory allocation.

Some libraries eagerly materialize the filtered data. This can be less memory- and performance-efficient that passing the boolean condition to a function such as `aggregate.` For example:

```
df.get_column(col_name)
  .aggregate(func, filter=boolean_condition)
```

I suggest deferring the materialization of filtered results in the data structure until materialization is required, while also allowing some functions to use the filter condition in execution for better memory use and performance.

Zero-copy optimized column selection and filtering

With the mutation-safety protections of copy-on-write, it is not necessary to allocate new memory when selecting columns.

Certain operations, such a boolean filtering (row selection by a condition), can be optimized for better performance and memory utilization over using `Slice` operations on the underlying arrays for contiguous row selections. Because of the minor overhead associated with `Array::Slice`, for small selected chunks it will be preferable to invoke a boolean selection kernel (see ARROW-1558)

Column statistics

Certain statistics are useful to assist with algorithm choice, such as:

- Monotonicity / sortedness
- Min and max value
- Number of null values

It would make sense to compute these once and cache them (in a threadsafe way). If any mutation operations are performed, then any existing statistics must be invalidated.

Indexing and lookups

Some libraries have found that different indexing strategies can assist with performance and interactivity

- Inverted index, for looking up rows for which a column matches a particular value
- Binary/B-tree index, for data frames sorted by keys

While these may not be implemented in the short term, I believe we should consider in the data frame API where they might fit in and how they can accelerate different operations.