May 2023

Making PyArrow Datasets a Protocol

TLDR: PyArrow datasets have been useful for integrating into query engines (e.g. DuckDB), but it's difficult for new table formats (Delta Lake, Lance, Iceberg) to extend them. Could we standardize a protocol or ABC for datasets instead?

Also see GitHub issues discussion: https://github.com/apache/arrow/issues/33986

In this document, we'll call libraries that want to expose datasets **producers** and query engines that want to query them **consumers**.

Consumers are able to read PyArrow datasets by calling into Python methods and exporting one or more record batch readers. The way they are implemented, they don't actually require that the dataset is implemented in PyArrow's C++ code. Thus, producers can write their own implementations that expose the same API as PyArrow's datasets, and consumers will still be able to read them.

The goal of this proposal is to formalize that API. What Python methods do the producers need to expose so that consumers understand them? What methods should consumers consider generally safe to call across implementations, as opposed to ones that are specific to PyArrow?

Non-goals

- **Establishing extension points or utilities for *implementing* datasets.** This isn't mutually exclusive with this proposal, but is something that will likely take more time. Lance and Delta Lake are implemented in Rust, so they may use utilities in arrow-rs and datafusion to implement their datasets. Meanwhile PyIceberg may build upon utilities exposed in PyArrow or Arrow C++.
- **Adding new functionality to the datasets API.** This proposal focuses on defining which parts of the existing PyArrow dataset API should be considered standard. Anything not yet implemented in that API can be considered in follow up work.

This proposal is about formalizing a contract that is already in use, rather than making anything new. These non-goals may be done as follow ups. Discussion of them should continue in Arrow Datasets Ecosystem Improvements.

Current state

First, we examine who are known producers and consumers of PyArrow datasets (other than the PyArrow library itself). How are they using them? What are their different needs?

May 2023

Producers

Besides the built-in factories in PyArrow, various table formats are interested in producing PyArrow datasets:

- Python Delta Lake (part of delta-rs)
  - Implemented in Rust.
  - Currently building datasets with PyArrow, but in order to support more advanced features will need to move to being purely implemented in Rust.
  - See PyArrow Datasets and their role in Python deltalake
- Lance
  - Dataset is entirely implemented in Rust and subclasses pyarrow.dataset.Dataset. See https://github.com/eto-ai/lance/blob/main/python/python/lance/dataset.py
  - To make this implementation work with DuckDB, they asked the DuckDB developers to loosen their type checks: https://github.com/duckdb/duckdb/pull/5170
- (potentially) PyIceberg

Consumers

These datasets can be consumed by a few query engines, and more may be interested:

- Acero (through PyArrow and the arrow R package)
- DuckDB
- Python DataFusion
- Polars?
  - Not yet properly implemented. See: https://github.com/pola-rs/polars/issues/7750#issuecomment-1541013054

Different engines have their own strategies for using datasets, depending on their execution model.

For DuckDB, a single scanner is created, and then a record batch reader is exported over the c stream interface.

For DataFusion, the scanner is created with the pushed down filter and projection, then fragments are collected from the scanner into a list. Each fragment is assigned a partition, and when each partition executes, it creates a scanner from its assigned fragment and exports all the record batches to Rust. (This could probably be improved to use the C Stream interface, and use get_fragments in the beginning rather than creating a scanner.)

In Dask, a dataset can be distributed by taking the fragments and making each one a partition, as in this example:

```
>>> import pyarrow.dataset as ds
>>> dataset = ds.dataset("hive_data_path", format="orc",
partitioning="hive")
>>> fragments = dataset.get_fragments()
>>> func = lambda frag: frag.to_table().to_pandas()
>>> df = dd.from_map(func, fragments)
```

Datasets as a protocol

Rather than providing extension points within the datasets implementation, we can define a protocol for “duck typing” the datasets interface.

Producers have the choice of building their dataset through PyArrow or creating their own classes that implement the protocol.

Consumers should **only** use the methods on datasets that are part of the protocol.

This protocol is drafted in https://github.com/apache/arrow/pull/35568

Notes on design:

- It is compatible with the existing PyArrow Dataset API.
- The API doesn’t require that fragments are eagerly loaded into memory. To support large tables, where the metadata itself may consume a significant amount of RAM, fragments can be loaded *after* pushdown filters are provided, so that only the metadata that is relevant to the query must be loaded. The get_fragments() method already has a filter argument, so the current API is acceptable.
- The design still uses PyArrow expressions for passing predicates and projections. For backwards-compatibility, this will remain. However, in the future PyArrow could offer an API to build its expressions from Substrait messages, allowing consumers to just pass down Substrait rather than maintain bespoke conversions between their expression types and PyArrow’s.
- It is designed to support returning a single stream (like how DuckDB uses it) or multiple streams (like how DataFusion and Dask do). Fragments are intended to be pickleable so they can be used in a distributed scan.

Future extensions to this protocol are discussed in this document:
Arrow Datasets Ecosystem Improvements

May 2023

May 2023

Previous discussion (not part of proposal)

Positives

There are several great things about Datasets that have made them useful:

1. The API provides a way to get a RecordBatchReader, which can be exported over the C stream interface to multi-threaded engines. This works either at the scanner level (one single stream) or at the fragment level (one stream per file). Both DuckDB and DataFusion have been able to use this to produce performant integrations without having to deeply integrate with the internals of PyArrow / libarrow.
2. Guarantees (the partition_expression in fragments) are a good abstraction. This has proven very powerful for Python deltalake, since we put file-level statistics in the guarantee in addition to the partition values.
3. Certain hidden columns are useful (__filename and maybe one day __row_index?)

Challenges / Potential improvements

There are places for improvement over the current API. Some of these challenges involve the interface itself while others are related to extending implementation. These are marked with [Interface] and [Implementation] respectively. The implementation ones aren't necessarily important when designing a protocol for datasets, but the information is provided to explain why extending the implementation isn't a viable route.

These are ordered from most to least important.

1. [Implementation] Table formats need support for projections and deletion vectors.
    a. Projections are probably easy to add to the existing system, but some formats (such as Iceberg) might have different projections needed per file, which might be more complicated to add.
    b. Deletion vectors - it's unclear how to implement, given we want to read the serialized deletion vectors lazily.
2. [Interface][Implementation] To scale to very large datasets, it would be better to put discovery after filtering. For example, we should be able to say we have a dataset at a certain path partitioned by date, and then if we ask to load the dataset for a particular date partition, it will only attempt to list the files in that partition. This is important for tables where the metadata about all the files in the full table could be gigabytes in size, and readers often don't need all that data. (Though for speed, implementation should cache this data.)
3. [Implementation] New file formats like lance need to implement their own file scanners, but doing so right now means adding code in the Apache Arrow C++ codebase.

4. [Interface] Exposing a datasets API means PyArrow expressions become part of the producer library's interface. In addition, each consumer needs to map their own expressions into PyArrow expressions for filter/projection pushdown. Is there an easier way? Hard to say if Substrait would make it easier or harder.
    a. If expressions are provided as part of the protocol interface, expressions could also be more modern. For example, they might provide Python type hints.
5. [Interface] TBD: does our datasets API provide useful abstractions for multi-process or distributed engines, such as Dask? What API changes would make it more useful? For example, can fragments be serialized/deserialized?
6. [Interface] Some datasets may be able to provide their data in a particular sort order, but we don't yet have a standard for communicating sort order with Record Batch Readers or otherwise.
7. [Implementation] Extending the Arrow C++ classes means somehow integrating with or replacing file system implementations.
    a. Filesystems need extensions for table formats for commit purposes. Deltalake relies on atomic rename and similar operations so it can handle concurrent writers. Other table formats will have similar needs.
    b. Filesystems need to be accessed by scan implementations, but don't want to be limited by the GIL.
8. Able to handle residuals. Based on the statistics the (Py)Iceberg is able to know if there are residuals in the Parquet file. For example, if you query datetime >= now() - 1 day, then you know that all the files that have data from today don't need additional filtering. This is more advanced handling of guarantees.

-