Support for partitioning and table formats

1) Rational

Modern analytics query engines usually work by querying open format data files laid out on some kind of storage medium. This comes with an abstraction, called the **table,** that groups files sharing a common schema and thus queryable consistently. There are multiple technologies to organize files into tables, called **table formats**. One of the key features of these formats is to further subdivide the file set contained in the table into **partitions** that allow to quickly prune files that will not be relevant in the context of a given query.

Some table formats are simply “agreed upon” file layouts: the most common example would be Hive partitioning, which structures files in folders such as `year=2021/month=08/day=30/yourfile.csv`. Others are more complex standards such as Delta or Iceberg.

The objective of this document is to propose the right abstraction to allow Datafusion to be extended to support any of these technologies on any kind of storage (file system or object store).

2) Some existing technologies

Let us start by a very brief tour of some common table formats.

a) Internal catalog

Many database systems maintain the table and partition metadata in an internal format (possibly in memory). For instance ElasticSearch keeps the information about its shards on the master node. This allows them to provide very low latency scheduling and often also ensure other write related features such as consistency.

b) Raw Hive partitioning

As mentioned above, it is usual to structure files on a filesystem or an object store by partition. Probably the most commonly used layout is what is often referred to as Hive Partitioning:

```
mybucket/myprefix/date=2019-01-01/file001.parquet
…
mybucket/myprefix/date=2019-01-01/file096.parquet
…
mybucket/myprefix/date=2021-08-27/file096.parquet
```

The “date” here can be seen as an extra column in the dataset. All rows in all files in the same folder share the same value for that field. If you specify in the query a filtering condition such as “date>=2021-08-01”, the query planning doesn’t even need to list the files in folders that don’t fulfill that condition. This is a very basic instance of partition pruning.

But the file layout might also be a bit different. One very common case is the output of AWS Firehose, where the bucket will look like this:

```
mybucket/myprefix/2019/01/01/00/68286a1e-c595-4092-9be7-efe1af153b4c.parquet
…
mybucket/myprefix/2019/01/01/23/30c196f0-2153-490e-8a08-53c4de31b596.parquet
…
mybucket/myprefix/2021/08/30/09/e9ffb9c7-7f27-417f-ae14-16548f35bfab.parquet
```

Most query systems only support the traditional Hive partitioning.

These partitionings can contain any file type (Parquet, CSV, Avro…).

This partitioning is self defining, meaning that it is encoded in the folder structure itself and can be discovered by simply using the listing capabilities of the file system / object store. But it can also be indexed by a Hive catalog, which stores the path to each partition with metadata about the content into a separate database.

Note for object stores: object stores are not fully featured file systems, and usually don’t structure files into directories. To discover the partitions dynamically, you need the capability on the object store to list the “folders” encoded in the object keys. This is required to build a list of partitions that can be pruned first and avoid listing all the files stored in the table (the number of files quickly rises to 100s of thousands for typical ETL tasks).

c) Delta / Iceberg

Advanced table formats exist that allow more efficient partitioning pruning and provide other features such as atomicity or time travel.

These standards use metadata files that contain the list of data files for the table along with metadata like statistics and partitioning structure.

Iceberg supports multiple file types for the table data (Parquet, ORC, Avro), Delta mostly focuses on Parquet.

3) Current Datafusion abstractions

Let us now review the main traits in Datafusion that are somewhat related to the notion of table.

CatalogProvider and SchemaProvider

The catalog and schema providers focus on the organization of tables and databases and are not at the partition level.

TableProvider

The TableProvider is that abstraction that is stored by the TableScan variant of the logical plan.

```
pub trait TableProvider: Sync + Send {
    fn as_any(&self) -> &dyn Any;

    fn schema(&self) -> SchemaRef;

    fn table_type(&self) -> TableType {
        TableType::Base
    }

    fn scan(
        &self,
        projection: &Option<Vec<usize>>,
        batch_size: usize,
        filters: &[Expr],
        limit: Option<usize>,
    ) -> Result<Arc<dyn ExecutionPlan>>;

    fn statistics(&self) -> Statistics;

    fn has_exact_statistics(&self) -> bool {
        false
    }

    fn supports_filter_pushdown(
        &self,
        _filter: &Expr,
```

```
    ) -> Result<TableProviderFilterPushDown> {
        Ok(TableProviderFilterPushDown::Unsupported)
    }
}
```

Currently TableProvider implementations are structured by file type (ParquetTable, CsvFile, NdJsonFile). To select the files to be processed, they take a path and a suffix and call a common util to list the files, `common::build_file_list(path,suffix).`

ObjectStore

Object store is a new abstraction to allow reading files both from the file system and remote object stores.

```
pub trait ObjectStore: Sync + Send + Debug {
    async fn list_file(
        &self,
        prefix: &str,
    ) -> Result<FileMetaStream>;

    async fn list_dir(
        &self,
        prefix: &str,
        delimiter: Option<String>,
    ) -> Result<FileMetaStream>;

    fn file_reader(&self, file: FileMeta) -> Result<Arc<dyn ObjectReader>>;
}
```

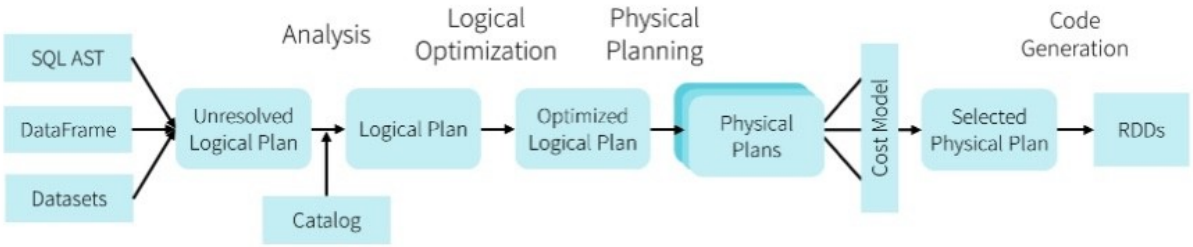
4) Core problematic

Where/when should we query the table format to get:

- **the list of files to be processed**
- **the metadata of the partitions (statistics…)**

5) Solution chosen by Spark

From this article that presents the Spark SQL optimization pipeline nicely:



Spark positions the catalog resolution at the level of the Logical Plan. It has an extra state for the query that is not present in DataFusion: the **Unresolved Logical Plan**.

It is during the *analysis* phase that Spark checks that the Unresolved Logical Plan is valid regarding the schema.

6) Proposition for Datafusion

The possible stages at which we could query the table format are:

a) Create a new abstraction like the Spark Unresolved Logical Plan and query the table format while converting it to a logical plan
b) Upon creation of the TableProvider *(not recommended)*
c) During the LogicalPlan optimization *(not recommended)*
d) Upon conversion from LogicalPlan to ExecutionPlan
e) From the ObjectStore abstraction *(not recommended)*

As explained below, most solutions are not really viable. Only **option a** and **option d** should be considered as potential candidates.

Option a: From an Unresolved Logical Plan abstraction

This introduces a new stage in the query planning before the logical plan as it is expressed today in Datafusion. The two states of the logical plan would be similar, except that the resolved one would be aware of the statistics.

Pros:

- To avoid transferring the list of files from the client to the scheduler in Ballista, we could transfer the unresolved logical plan to the scheduler instead of the resolved one.[1]
- While resolving the logical plan, we could run the pushdowns until the table providers which would thus also exist in two states: resolved and unresolved. This way the resolved table provider itself would be aware of filter expressions and limits. This implies

[1] In Ballista, the LogicalPlan is created in the client, transferred to the scheduler, converted to a PhysicalPlan on the scheduler, then the PhysicalPlan is serialized and distributed to the executors.

that the statistics in the logical plan could already contain the adjustments from the partition pruning and could be sourced from the table format itself when possible.
- We can apply cost based optimizations on the logical plan. This should be easier to implement because the logical plan represents a higher level of abstraction For instance a logical aggregation node is much simpler than its physical representation that can be splitted into multiple nodes, including repartitioning stages, as well as having multiple implementations.

Cons:

- Currently the partitioning abstractions are only defined on the physical plan (`fn output_partitioning(&self)`). Without those, shuffle boundaries cannot be determined and the plan can't be broken up to be distributed. This implies that the plan must be converted into a physical plan before being distributed (cf Ballista). In that case, the cost based optimizations (CBO) applied on the logical plan must also be implemented on the physical plan to allow adaptive query execution (AQE).
- Adds an extra plan state which implies more boilerplate.

Note:

- In this setup, rule based optimizations are applied on the unresolved logical plan first but can also be applied on the resolved plan. Cost based optimizations would be applied on the resolved plan only.
- If we already applied the filter/limit pushdown when converting the Unresolved Logical Plan into the LogicalPlan, we don't need to pass the filter/limit expression to the `TableProvider.scan(...)` method as the TableProvider will already know about the filter/limit expressions.

Option b: Upon creation of the TableProvider *(not recommended)*

Pros:

- avoids adding a new abstraction for UnresolvedLogicalPlan

Cons:

- doesn't play well with Ballista as this means that the list of files to be processed would need to be transferred from the client to the scheduler
- **You don't have access to the filter expressions**

Option c: During the LogicalPlan optimization *(not recommended)*

Pros:

- avoids adding a new abstraction for UnresolvedLogicalPlan

Cons:

- Strange because it means you cannot convert some logical plans to physical plans if they are not optimized first

Option d: Upon conversion of the LogicalPlan to a PhysicalPlan

This would likely require making the `TableProvider.scan(...)` method async.

Pros:

- This is the closest to the existing implementation where the file list is built when creating the ExecutionPlan
- The logical plan does not load to much metadata which
    - makes it simpler to reimplement in another language
    - makes it lightweight for serialization

Cons:

- We need a schema to build a valid logical plan. If the schema is sourced from the table format, this means that the table format will need to be queried once when creating the logical plan to get the schema, then another time when converting the logical plan to a physical plan to get the statistics and the file list.
- The statistics in the TableProvider don't have access to the filter expressions. To have properly prunable statistics, we need to move them and thus all cost based optimizations (CBOs) into the physical plan. This is doable (as shown in #962) but is challenging as the physical plan is much more complex than the logical one.

As shown above, table formats often support multiple data file formats. We have two possibilities to accommodate this:

- Keeping the TableProvider implementations by file format: each table provider would need to have a configurable table format provided. The file format information from the TableProvider would not be useful for all table formats that know the format of their data files.
- Making the TableProvider implementation by table format instead of file format. In many cases (Hive Catalog, Delta, Iceberg), the logical plan could rely on the table format to figure out which file format is used and generate the appropriate execution plan. For the cases where the file format cannot be figured out (e.g. raw Hive partitioning) it could be specified to the TableProvider as a parameter.

If we go for the second option, implementations of TableProvider would look like:

- ListingTableProvider, the classical table provider that gets the file list from the object store / file system itself by using its listing capability. This could also support parsing and pruning hive partitioned file layouts.
    - `ListingTableProvider::new(prefix: &str, file_format: enum, partitions: &[String])`
        - The constructor takes the file format as parameter as the table provider is not specific to a file type anymore.
        - Specify the partitions you want to parse (empty array if no partitioning)
    - `ListingTableProvider.scan(...)`
        - The returned ExecutionPlan type depends on the file_format specified

- DeltaTableProvider
    - `DeltaTableProvider::new(prefix: &str, timestamp_as_of: Option<String>, version_as_of: Option<String>)`
        - The constructor can be configured with all the supported Delta options such as time travel.
    - `ListingTableProvider.scan(...)`
        - Returns the ParquetTable implementation of ExecutionPlan
- Etc...

Option e: From the ObjectStore abstraction *(not recommended)*

This would be done by passing the filter expressions to the `ObjectStore.list(...)` method.

Cons:

- The ObjectStore abstraction is meant to be at a much lower level, only representing the basic features of opening a file reader and listing files on a path. If the object store is also in charge of the table format, that means that one specific implementation will be required per table format (hive, delta, iceberg..) per infrastructure (fs, s3…).

Option b can be eliminated right away because we need the filter expressions when resolving the Table Format to do the partition pruning.

Option c and ***option e*** are counter intuitive and break the meaning of the current abstractions. They should thus also be discarded.

Option a and ***option d*** both make sense, but the implementation for ***option d*** seems more straightforward.

Side notes:

Experience from buzz

In Buzz for now we have implemented a variant of solution A. It uses a specific abstraction (CatalogTable) that resolves a given table format into serializable descriptions of TableProviders. This workaround is specific to Buzz because it distributes a separate SQL query to the workers instead of splitting one single plan as a typical query engine would.

Implementation sketch for option A

- Add a variant called UnresolvedTableScan to the LogicalPlan enum
- Trying to convert a LogicalPlan that contains an UnresolvedTableScan to a PhysicalPlan fails
- Add a method `ExecutionPlan.analyze(LogicalPlan) -> Result<LogicalPlan>`. Note that this is pretty similar to `ExecutionPlan.create_physical_plan(...)`.