Arrow Flight RPC/Flight SQL Proposals

Flight RPC and Flight SQL are being evaluated for use in distributed systems and databases. This proposal aims to address some common edge cases that arise there.

For context, in Flight RPC and Flight SQL, the query flow is as follows:

1. **Run the query:** the client calls GetFlightInfo with the query.
   Response: a FlightInfo describing the result partitions (FlightEndpoints).
2. **Fetch the result partitions:** for each result partition, call DoGet with the Ticket inside the FlightEndpoint to get part of the result set. This can be done in parallel or sequentially; it can be done from one client machine or several.

This raises some questions:

- What should the client do if GetFlightInfo is interrupted?
- What should the client do if DoGet is interrupted? Can it fetch the data again?
- What if part of the query completes more quickly than the rest? Can a client get partial results while the rest of the query finishes?
- Is there any order to the result partitions?

Proposals

All proposals here are backwards-compatible: new fields and new RPC endpoints will be ignored by clients, and new behaviors are opt-in for the client.

Flight RPC: Ordered Data

Currently, the endpoints within a FlightInfo explicitly have no ordering:

```
   * There is no ordering defined on endpoints. Hence, if the returned
   * data has an ordering, it should be returned in a single endpoint.
```

This is unnecessarily limiting. Systems can and do implement distributed sorts, but they can't reflect this in Flight RPC. And users are asking; see this Stack Overflow question.

Proposal

Add a flag to FlightInfo:

```
message FlightInfo {
  // FlightEndpoints are in the same order as the data.
```

```
  bool endpoints_ordered = 6;
}
```

If set, the client may assume that the data is sorted in the same order as the endpoints. Otherwise, the client cannot make any assumptions (as before).

Flight RPC: Result Set Expiration

Currently, it is undefined whether a client can call DoGet more than once. Clients may want to retry requests, and servers may not want to persist a query result forever.

Proposal

Add an expiration time to FlightEndpoint. If present, clients may assume they can retry DoGet requests. Otherwise, clients should avoid retrying DoGet requests.

```
message FlightEndpoint {
  // In UTC.
  google.protobuf.Timestamp expiration_time = 3;
}
```

This proposal is *not* a full retry protocol.

Also, add "pre-defined" actions to Flight RPC for working with result sets. These are pre-defined Protobuf messages with standardized encodings for use with DoAction:

- CancelQuery: Asynchronously cancel the execution of a distributed query. (Replaces the equivalent Flight SQL action.)
- RefreshQuery: Request an extension of the expiration of a FlightEndpoint.
- CloseQuery: Close a FlightInfo so that the server can clean up resources early.

This lets the ADBC/JDBC/ODBC drivers for Flight SQL explicitly manage result set lifetimes. These can be used with Flight SQL as regular actions. (Also see "Query Execution Hints" below.)

Prior Art

- Dremio: job results are cleaned up after a set time.
- Google BigQuery Storage: read sessions include an explicit expiration time.
- Snowflake: result sets persist for 24 hours.

Flight RPC: Long-Running Queries

In Flight RPC, FlightInfo includes addresses of workers alongside result partition info. This lets clients fetch data directly from workers[1], in parallel or even distributed across multiple machines. But this also comes with tradeoffs.

Queries generally don't complete instantly (as much as we would like them to). So where can we put the 'query evaluation time'?

- In GetFlightInfo: block and wait for the query to complete.
  - Con: this is a long-running blocking call, which may fail or time out. Then when the client retries, the server has to redo all the work.
  - Con: parts of the result may be ready before others, but the client can't do anything until everything is ready.
- In DoGet: return a fixed number of partitions
  - Con: this makes handling worker failures hard. Systems like Trino support fault-tolerant execution by replacing workers at runtime. But GetFlightInfo has already passed, so we can't notify the client of new workers[2].
  - Con: we have to know or fix the partitioning up front.

Neither solution is optimal.

Proposal

We can address this by adding a retryable version of GetFlightInfo. First, we add a new RPC call and result message:

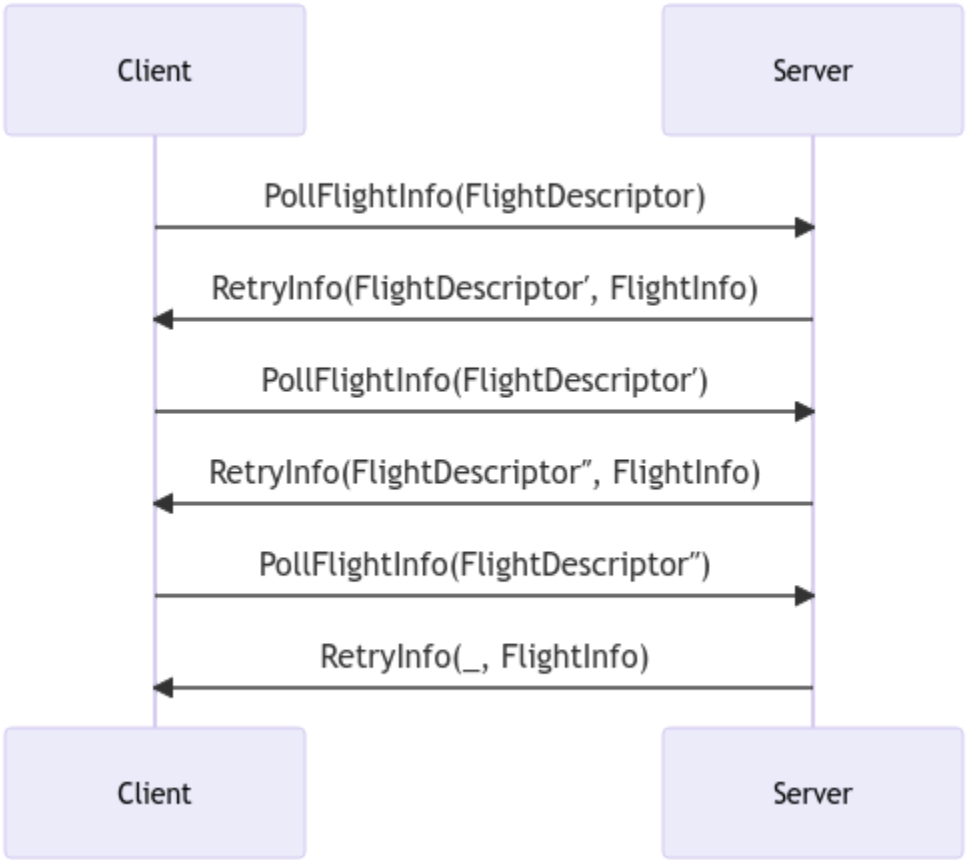
```
service FlightService {
  // ...
  rpc PollFlightInfo(FlightDescriptor) returns (RetryInfo);
}

message RetryInfo {
  // The currently available results so far.
  FlightInfo info = 1;
  // The descriptor the client should use on the next try.
  // If unset, the query is complete.
```

[1] Of course, servers are free to return the location of a proxy/load balancer/etc., or omit locations and have the client fetch results from the same server that they issued the query to. Flight RPC offers this flexibility to servers; clients don't have to know or care.

[2] Again, the server could proxy workers, or depend on Kubernetes DNS routing, or configure gRPC XDS. But this somewhat defeats the point of returning worker locations in the first place, and is much more complicated (operationally, implementation-wise).

```
  FlightDescriptor retry_descriptor = 2;
  // The descriptor the client should use to cancel the query.
  // If the query is complete or the server does not support query
  // cancellation, this is not set.
  FlightDescriptor cancel_descriptor = 3;
  // Query progress. Must be in [0.0, 1.0] but need not be
  // monotonic or nondecreasing. If unknown, do not set.
  optional double progress = 4;
  // Expiration time for this request. After this passes, the server
  // might not accept the retry_descriptor anymore (and the query may
  // be cancelled). This may be updated on a call to PollFlightInfo.
  google.protobuf.Timestamp expiration_time = 5;
}
```

A client executes a query and polls for result completion. The server returns a FlightInfo representing the state of the query execution up to that point.



The server:

- Must respond with the complete FlightInfo each time, *not* just the delta between the previous and current FlightInfo.
- Should respond as quickly as possible on the first call.
- Should not respond until the result would be different from last time. (That way, the client can "long poll" for updates without constantly making requests. Clients can set a short timeout to avoid blocking calls if desired.)
- May respond by only updating the "progress" value (though it shouldn't spam the client with updates).
- Should recognize a retry_descriptor that is not necessarily the latest (in case the client misses an update in between).
- Should only append to the endpoints in FlightInfo each time. (Otherwise the client has to do extra work to identify what endpoints it has and hasn't seen.)

  When endpoints_ordered is set, this means the server returns endpoints in order.
- Should return an error status instead of a response if the query fails. The client should not retry the request (except for TIMED_OUT and UNAVAILABLE, which may not originate from the server).

Prior Art

- Amazon Redshift: executing a query gives an ID that can be used to check the query status and fetch results.
- Google BigQuery Storage: you explicitly create a "read session", after which you can read subsets of the response with further requests. There is no "query execution time" since BigQuery Storage only queries tables. Instead, running a query (with the base BigQuery API) will cache the result in a table that can be read via BigQuery Storage.
- Snowflake: short queries return synchronously. Longer queries require polling for completion of the query. You cannot retrieve any results until the query is complete.

Flight RPC: More Application Metadata

FlightInfo and FlightEndpoint lack application-defined fields that can be shared between clients and servers. This becomes necessary if we want things like CancelQuery to work, since otherwise, the server may not have a way to identify which query corresponds to the FlightInfo given by the client. (There may not yet be any FlightEndpoints, or a schema to attach metadata to, which would be the normal places to put such info.)

Proposal

Add the following fields:

```
message FlightInfo {
  bytes app_metadata = 7;
}

message FlightEndpoint {
  bytes app_metadata = 4;
}
```

These are intended to be parsed by clients. They have no standard semantics otherwise (as in FlightData).

Flight SQL: Query Execution Hints

Add new hints to execution requests in Flight SQL:

```
// Hints for query execution. The server is not obligated
// to fulfill any of them.
message QueryHints {
  // Request at least a certain result set lifetime.
  google.protobuf.Duration desired_ttl = 1;
  // Request at least a certain number of result partitions.
  optional uint32 desired_min_endpoints = 2;
  // Request at most a certain number of result partitions.
  optional uint32 desired_max_endpoints = 3;
  // Custom, application-specific hints.
  map<string, string> app_hints = 100;
}
```

These would apply to the following messages:

- CommandStatementQuery
- CommandStatementSubstraitPlan
- CommandPreparedStatementQuery
- CommandStatementUpdate
- CommandPreparedStatementUpdate

There is no need for a SqlInfo value, since there are hints.

Flight SQL: Transaction Options

Currently ActionBeginTransactionRequest is an empty message, but some systems allow requesting specific transaction options such as requesting a particular Isolation Level. The SqlInfo enum values already have values for requesting the default transaction isolation level and checking which levels are supported. So we could simply add a value to the transaction request and change the protobuf definition to:

```
message ActionBeginTransactionRequest {
  option (experimental) = true;
  // if unset or set to SQL_TRANSACTION_NONE then it will use
  // the database's default isolation level.
  // if the requested level is not supported, the BeginTransaction
  // request should fail with a NotImplemented error.
  optional SqlTransactionIsolationLevel isolation_level = 1;
}
```

We would then add a SqlInfo value to indicate support for transaction levels in requests.

Open Questions

- How should cancellation be handled in PollFlightInfo? Is that redundant with Flight SQL's CancelQuery/should this be left up to the application layer (e.g. Flight SQL)?
    - **I will prototype defining 'standard' Protobuf messages that can be used in Flight RPC and Flight SQL.**
- Is Flight SQL's CancelQuery redundant with CloseQuery/should the two be merged?
- Should any of this be added to Flight RPC, or should it be kept in Flight SQL only (leaving Flight RPC to be more minimal)? This would mean adding more app_metadata fields to structures like FlightInfo/FlightEndpoint for Flight SQL (and other applications) to encode the necessary fields.
    - Or should we go the other way, and try to make Flight RPC more full-featured?